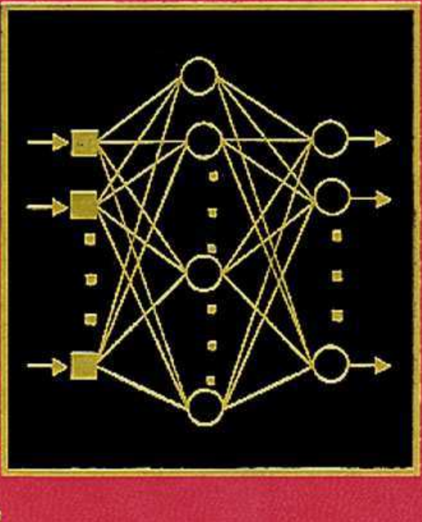
ELSEVIER
ACADEMIC PRESS

Advances in COMPUTERS

Volume 65



Edited by
MARVIN V. ZELKOWITZ

Advances

in COMPUTERS
VOLUME 65

Advances in **COMPUTERS**

EDITED BY

MARVIN V. ZELKOWITZ

Department of Computer Science
and Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland

VOLUME 65

ELSEVIER
ACADEMIC
PRESS

Amsterdam Boston Heidelberg London New York Oxford
Paris San Diego San Francisco Singapore Sydney Tokyo

ELSEVIER B.V.
Radarweg 29
P.O. Box 211, 1000 AE Amsterdam
The Netherlands

ELSEVIER Inc.
525 B Street, Suite 1900
San Diego, CA 92101-4495
USA

ELSEVIER Ltd
The Boulevard, Langford Lane
Kidlington, Oxford OX5 1GB, UK

ELSEVIER Ltd
84 Theobalds Road
London WC1X 8RR, UK

First edition 2005

Library of Congress Cataloging in Publication Data
A catalog record is available from the Library of Congress.

British Library Cataloguing in Publication Data
A catalogue record is available from the British Library.

ISBN: 0-12-012165-4
ISSN (Series): 0065-2458

⊗ The paper used in this publication meets the requirements of ANSI/NISO Z39.48-1992 (Permanence of Paper).

Printed in USA.

Contents

Early Cognitive Computer Vision

Jan-Mark Geusebroek

Contributors

David N. Card is a Fellow of the Software Productivity Consortium, a not-for-profit technology research and transition organization. During 15 years at Computer Sciences Corporation, Mr. Card spent six years as the Director of Software Process and Measurement, one year as a Resident Affiliate at the Software Engineering Institute, and seven years as a member of the research team supporting the NASA Software Engineering Laboratory. Mr. Card is the Editor-In-Chief of the *Journal of Systems and Software*. He is the author of *Measuring Software Design Quality* (Prentice Hall, 1990), co-author of *Practical Software Measurement* (Addison–Wesley, 2002), and co-editor of ISO/IEC Standard 15939:2002, *Software Measurement Process*. Mr. Card has been designated a Senior Member of the American Society for Quality.

Dr. Jan-Mark Geusebroek is an assistant professor in the Intelligent Sensory Information Systems (ISIS) group at the University of Amsterdam. He received the Ph.D. degree in computer sciences from the University of Amsterdam in 2000. His research interests are in front-end vision, especially color and texture vision. His current research concentrates on computational theories for cognitive vision, based on invariant representations and visual attention.

Dr. Peter B. Henderson is co-founder of the math thinking discussion group (www.math-in-cs.org) which advocates the importance of mathematics and mathematical reasoning in computer science and software engineering education. He holds the chair of the Department of Computer Science and Software Engineering at Butler University and is editor of two educational columns, “Software Engineering Education” in the SIGSOFT Software Engineering Notes, and “Math CountS” in SIGCSE InRoads. In addition, he has conducted workshops and given numerous presentations on the role of mathematics in computer science and software engineering education, and has been instrumental in formulating recommendations on the mathematical needs of undergraduate computer science and software engineering programs for the Mathematical Association of America’s Committee on the Undergraduate Program in Mathematics. He holds a Ph.D. in Electrical Engineering from Princeton

University and has been teaching computer science and software engineering since 1974.

Dr. Gerard J. Holzmann is the principal designer of the widely used formal verification tool SPIN. Dr. Holzmann received the Ph.D. degree from Delft University in 1979. Formerly a Director of the Computing Principles Research group at Bell Labs in Murray Hill, New Jersey, he joined NASA's Jet Propulsion Laboratory in Pasadena, California in 2003 to set up a new Laboratory for Reliable Software. In 2001 Dr. Holzmann was the recipient of the prestigious ACM Software Systems Award for the design of the SPIN system, and in 2002 he received the ACM SIGSOFT's Outstanding Researcher Award.

Adrian A. Hopgood is professor of computing and dean of the School of Computing and Informatics at Nottingham Trent University in the UK. He has previously worked for Systems Designers, Telstra, and the Open University, where he remains a visiting professor. His main research interests are in intelligent systems and their practical applications. He graduated with a B.Sc. (Hons.) in physics from the University of Bristol in 1981 and obtained a Ph.D. from the University of Oxford in 1984. He is a member of the British Computer Society and a committee member for its specialist group on artificial intelligence.

Dr. Mark Huiskes received his M.S. degree (cum laude) in Technical Mathematics from Delft University of Technology in 1996 and his Ph.D. degree in Mathematics from Wageningen University in 2002. In August 2001 he joined the Signals and Images research group at the Centre for Mathematics and Computer Science (CWI, Amsterdam). His main research interests are in machine learning and computer vision.

Chris Lokan is a Senior Lecturer with the School of Information Technology and Electrical Engineering, UNSW@ADFA, in Canberra, Australia. He received a Ph.D. in computer science from the Australian National University, and then worked at CSIRO before joining the University of New South Wales. His research interests are in software metrics, especially measuring and estimating software size, software benchmarking, and measuring object-oriented software. He has been the principal data analyst for the International Software Benchmarking Standards Group since 1996.

Prof. Tim Menzies has recently been appointed associate research professor at Portland State University in the United States, and has been working with NASA on software quality issues since 1998. He has a CS degree and a Ph.D. from the University of New South Wales. His recent research concerns modeling and learning with a particular focus on light-weight modeling methods. His doctoral research is aimed at improving the validation of possibly inconsistent, knowledge-based systems in the QMOD specification language. He also has worked as an object-oriented consultant in industry and has authored over 150 publications and served on numerous conference and workshop programs and well as guest editor of journal special issues. He can be reached at tim@menzies.us.

Dr. Eric Pauwels obtained a Ph.D. in Mathematics in 1989, after which he joined the Computer Vision Group at Leuven University (Belgium) and worked on mathematical problems related to computer vision. In 1999 he joined the Signals and Images research group at the Centre for Mathematics and Computer Science (CWI, Amsterdam) where he heads the subtheme on Image Understanding, Retrieval and Indexing. His research interests include segmentation, classification, content-based image retrieval and visual learning. He coordinates the EU project FOUNDIT and the FP6 Network of Excellence on Multimedia Understanding through Semantics, Computation and Learning (MUSCLE). He is also the current chairman of the ERCIM Working Group on Image and Video Understanding.

Prof. Charles Pecheur has recently been appointed assistant professor at Université Catholique de Louvain in Belgium, and has been a Staff Scientist for the Research Institute for Advanced Computer Science (RIACS) at NASA Ames for the last six years. He has an EE degree and a Ph.D. from the University of Liège, Belgium. His recent research at NASA concerns the formal verification and validation of NASA's critical software systems, with a particular focus on model-based autonomous controllers and diagnosis. His doctoral research aimed at improving data type definitions in the LOTOS specification language. He also performed a couple of applied formal verification projects at INRIA Rhône-Alpes in France. He has authored more than twenty publications and served on several conference and workshop programs. He can be reached at pecheur@info.ucl.ac.be.

Preface

Since 1960, "Advances in Computers," first published by Academic Press and now by Elsevier, has been at the forefront of publishing articles describing the latest new technology in the information technology field. In this present volume, the 65th in the series, we present 8 chapters that focus on the latest advances in artificial intelligence, databases and software engineering.

Artificial intelligence is one of the oldest application domains for computers, starting in the mid-1950s, and the ability to mimic biological "intelligence" is a long sought goal. What has happened to AI and what is the current state of the AI world? In the first chapter Adrian A. Hopgood in "The state of artificial intelligence" discusses the various approaches to artificial intelligence and explores some of the current AI applications: intelligent agents, genetic algorithms, neural networks and hybrid systems.

In Chapter 2, Gerard Holzmann in "Model checking with SPIN" explores the use of artificial intelligence ideas in the software engineering of correct programs. SPIN is a well-known program that uses a formal verification approach toward verifying the specification of a program written using a model-checking specification language. He describes the process of developing an automated model-checking verification system and then concludes with a brief comparison of techniques used in hardware verification and a perspective of likely developments in the near future.

Jan-Mark Geusebroek in "Early Cognitive Computer Vision" explores another application domain for artificial intelligence—computer vision. He outlines computational models for visual cognition in both biological and technical systems. He focuses on the physical and statistical constraints in the sensory input, and how this can be exploited to construct cognitive vision systems.

In Chapter 4, Tim Menzies and Charles Pecheur discuss "Verification and Validation and Artificial Intelligence." The first three chapters of this book explored different facets of artificial intelligence. One important concern is how dependable are the programs that solve the problems discussed in these chapters? AI programs typically involve searching large numbers of potential solutions. How do you know the search is choosing the correct choice? So validating an AI program is particularly difficult. This chapter discusses how one can verify and validate such programs.

"Indexing, learning and content-based retrieval for special purpose image databases" by Mark Huiskes and Eric Pauwels is the subject of Chapter 5. While there has been much written about retrieving text from a database, what are the special issues involved in retrieving non-textual information? In this chapter, the authors discuss the problems of content-based retrieval in the ability to retrieve images from a database that have specified properties.

Chapter 6 by David Card is entitled "Defect Analysis: Basic Techniques for Management and Learning." It provides an overview of defect analysis in the production of correct software. The emphasis in this chapter is on the three uses of such collected defect data: modeling and predicting software quality, monitoring process performance, and learning from past mistakes about how to improve process performance and software quality.

In Chapter 7, Christopher Lokan discusses "Function points." Function points are a method for estimating the effort involved and complexity of a software product from information available during the requirements phase of a project. While *lines of code* is a more common estimator of the effort needed to build a product, this number is not available accurately until the product is completed—too late to make a prediction. The author discusses the use of function points and the various methods that have been developed to count them.

In the final chapter, "The Role of Mathematics in Computer Science and Software Engineering Education," Peter Henderson discusses an issue often argued by various factions in the academic world. What is the appropriate background in mathematics for a software engineer and what mathematical preparation is necessary in order to design and build complex programs? If you review some of the earlier chapters in this volume, mathematics plays a heavy role in understanding some of them. In this chapter, Dr. Henderson provides his views on an appropriate foundation for a university curriculum.

I hope that you find these articles of interest. If you have any suggestions of topics for future chapters, or if you wish to contribute such a chapter, I can be reached at mvz@cs.umd.edu.

Marvin Zelkowitz
College Park, Maryland

The State of Artificial Intelligence

ADRIAN A. HOPGOOD

School of Computing & Informatics
Nottingham Trent University
Burton Street, Nottingham, NG1 4BU
UK
adrian.hopgood@ntu.ac.uk

Abstract
Artificial intelligence has been an active branch of research for computer scientists and psychologists for 50 years. The concept of mimicking human intelligence in a computer fuels the public imagination and has led to countless academic papers, news articles and fictional works. However, public expectations remain largely unfulfilled, owing to the incredible complexity of everyday human behavior. A wide range of tools and techniques have emerged from the field of artificial intelligence, many of which are reviewed here. They include rules, frames, model-based reasoning, case-based reasoning, Bayesian updating, fuzzy logic, multiagent systems, swarm intelligence, genetic algorithms, neural networks, and hybrids such as blackboard systems. These are all ingenious, practical, and useful in various contexts. Some approaches are pre-specified and structured, while others specify only low-level behavior, leaving the intelligence to emerge through complex interactions. Some approaches are based on the use of knowledge expressed in words and symbols, whereas others use only mathematical and numerical constructions. It is proposed that there exists a spectrum of intelligent behaviors from low-level reactive systems through to high-level systems that encapsulate specialist expertise. Separate branches of research have made strides at both ends of the spectrum, but difficulties remain in devising a system that spans the full spectrum of intelligent behavior, including the difficult areas in the middle that include common sense and perception. Artificial intelligence is increasingly appearing in situated systems that interact with their physical environment. As these systems become more compact they are likely to become embedded into everyday equipment. As the 50th anniversary approaches of the Dartmouth conference where the term 'artificial intelligence' was first published, it is concluded that the field is in good shape and has delivered some great results. Yet human thought processes are incredibly complex, and mimicking them convincingly remains an elusive challenge.

1. Introduction

1.1 What Is Artificial Intelligence?

Over many centuries, tools of increasing sophistication have been developed to serve the human race. Digital computers are, in many respects, just another tool. They can perform the same sort of numerical and symbolic manipulations that an ordinary person can, but faster and more reliably. A more intriguing idea is whether we can build a computer (or a computer program) that can *think*. As Penrose [1] has pointed out, most of us are quite happy with machines that enable us to do physical things more easily or more quickly, such as digging a hole or traveling along a freeway. We are also happy to use machines that enable us to do physical things that would otherwise be impossible, such as flying. However, the idea of a machine that can think for us is a huge leap forward in our ambitions, and one which raises many ethical and philosophical questions.

Research in artificial intelligence (or simply AI) is directed toward building such a machine and improving our understanding of intelligence. Most of the definitions in the standard texts are over-complex, so here is a simple one that will suffice instead:

> Artificial intelligence is the science of mimicking human mental faculties in a computer.

The ultimate achievement in this field would be to construct a machine that can mimic or exceed human mental capabilities, including reasoning, understanding, imagination, perception, recognition, creativity, and emotions. We are a long way from achieving this, but some significant successes have nevertheless been achieved.

Perhaps more importantly, in achieving these modest successes, research into artificial intelligence has resulted in the development of a family of extremely useful computing tools. These tools have enabled a range of problems to be tackled that were previously considered too difficult, and have enabled a large number of other problems to be tackled more effectively. From a pragmatic point of view, this in itself makes them interesting and useful.

The tools of AI can be roughly divided into these broad types:

- knowledge-based systems (KBSs), i.e., explicit models using words and symbols;
- computational intelligence (CI), i.e., implicit modeling with numerical techniques;
- hybrids.

The first category includes techniques such as rule-based, model-based, frame-based, and case-based reasoning. As the knowledge is explicitly modeled in words and symbols, it can be read and understood by a human. Although symbolic techniques have had undoubted success in their narrow domains, they are intrinsically limited in their ability to cope only with situations that have been explicitly modeled. Although some systems allow the model to expand with experience, symbolic models are generally poor at dealing with the unfamiliar.

Computational intelligence goes some way to overcoming these difficulties by enabling the computer to build up its own model, based on observations and experience. Here the knowledge is not explicitly stated but is represented by numbers that are adjusted as the system improves its accuracy. This category includes neural networks, genetic algorithms and other optimization algorithms, as well as techniques for handling uncertainty, such as fuzzy logic.

Pinpointing the beginning of research into artificial intelligence is tricky. George Boole (1815–1864) had plenty of ideas on the mathematical analysis of thought processes, and several of his ideas have been retained in the field of AI today. However, since he had no computer, the above definition appears to rule him out as the founder of AI. Just as historians on either side of the Atlantic have different opinions of who built the first programmable computer, the same divergence of opinion occurs over the origins of AI. British historians point to Alan Turing's article in 1950 which included the so-called Turing test to determine whether a computer displays intelligence [2]. American historians prefer to point to the Dartmouth conference of 1956, which was explicitly billed as a study of AI and is believed to be the first published use of the term 'artificial intelligence'. As the golden jubilee of that historic event approaches, a review of the field is timely.

1.2 A Spectrum of Intelligent Behavior

The definition of AI presented above leaves the notion of intelligence rather vague. To explore this further, a spectrum of intelligent behaviors can be drawn, based on the level of understanding involved [3], as shown in Figure 1. The lowest-level behaviors include instinctive reactions, such as withdrawing a hand from a hot object or dodging a projectile. High-level behaviors demand specialist expertise such as in the legal requirements of company takeovers or the interpretation of mass spectrograms. Such

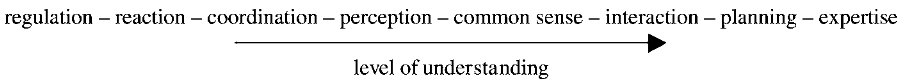

regulation – reaction – coordination – perception – common sense – interaction – planning – expertise
level of understanding

FIG. 1. A spectrum of intelligent behavior.

a spectrum of intelligent behaviors is useful for charting the progress of AI, although it has been criticized for oversimplifying the many dimensions of intelligence [4].

Conventional computing techniques have been developed to handle the low-level decision-making and control needed at the low end of the spectrum. Highly effective computer systems have been developed for monitoring and controlling a variety of equipment. An example of the close regulation and coordination that is possible is demonstrated by the ASIMO humanoid robot [5] developed by Honda. It has 16 flexible joints, requiring four high-specification computers just to control its balance and movement. ASIMO shows exceptional human-like mobility, but its behavior is nevertheless anchored at the lower end of the spectrum. Although it has some capabilities for recognizing speech, faces and gestures, it still lacks any depth of understanding. Sony's smaller QRIO robot [6] has similar capabilities, but is claimed to be distinguished by its capacity for autonomous decision-making.

Early AI research, on the other hand, focused on the problems at the high-level end of the spectrum. Two early applications, for example, concerned the specialist areas of mass spectrometry [7] and bacterial blood infections [8]. These early triumphs generated great optimism. If a computer could deal with difficult problems that are beyond the capabilities of most ordinary people, then surely more modest human reasoning would be straightforward. Unfortunately, this is not so.

The behaviors in the middle of the spectrum, that we all perform with barely a conscious thought, have proved to be the most difficult to emulate in a computer. Consider the photograph in Figure 2. Although most of us can spot the rabbit in the picture, the perception involved is an extremely complex behavior. First, recognizing the boundary between objects is difficult. Once an object has been delineated, recognition is far from straightforward. For instance, rabbits come in different shapes, sizes, and colors. They can assume different postures, and they may be partially occluded, as in Figure 2. Yet a fully sighted human can perform this perception in an instant, without considering it a particular mark of intelligence. But getting a computer to do it reveals the task's astonishing complexity.

1.3 An Integrated Approach

As with any other technique, knowledge-based systems and computational intelligence are not suitable for all types of problems. Each problem calls for the most appropriate tool, but knowledge-based systems and computational intelligence can

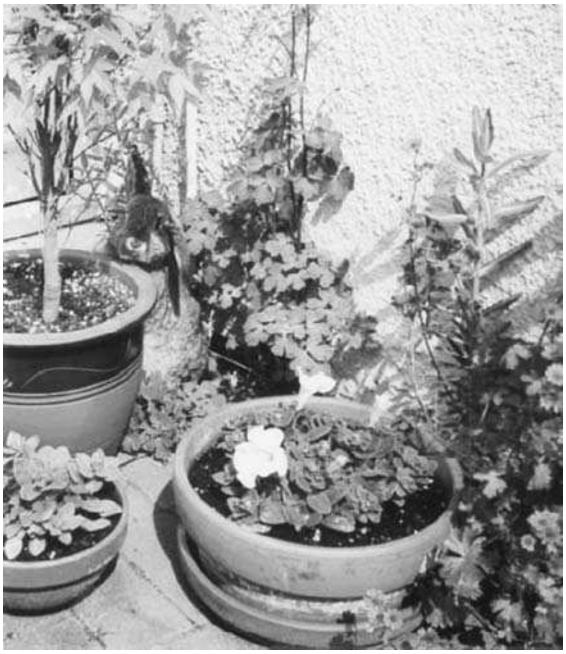


FIG. 2. Spot the rabbit. (See Color Plate Section, Plate 1.)

be used for many problems that would be impracticable by other means. In fact, this chapter will propose that no single software technique is capable of emulating the full spectrum of intelligent behaviors.

Substantial successes have been achieved in specific parts of the spectrum. It is therefore proposed that the key to achieving an AI system that is truly capable of mimicking human mental faculties in a computer is to integrate various AI techniques with each other and possibly with conventional programs (Figure 3). The various techniques do not necessarily represent exclusive alternatives, but can often be used cooperatively. The blackboard architecture will be proposed as one suitable way of achieving this.

2. Rule-Based Systems

2.1 Knowledge-Based and Expert Systems

The principal difference between a knowledge-based system and a conventional program lies in its structure. In a conventional program, domain knowledge is inti-

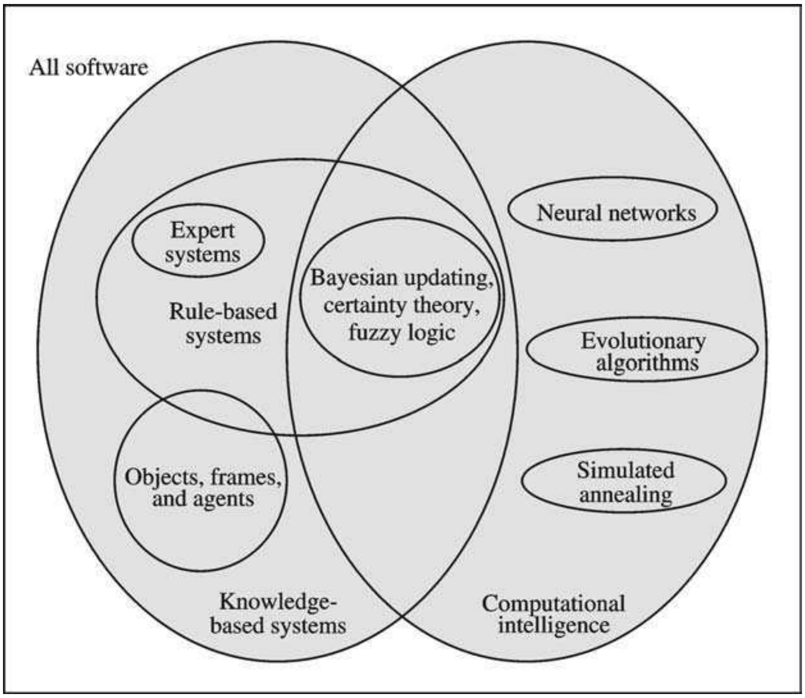


FIG. 3. Categories of intelligent system software.

mately intertwined with software for controlling the application of that knowledge. In a knowledge-based system, the two roles are explicitly separated. In the simplest case there are two modules—the knowledge module is called the knowledge base, and the control module is called the inference engine (Figure 4).

Within the knowledge base, the programmer expresses information about the problem to be solved. Often this information is declarative, i.e., the programmer states some facts, rules, or relationships without having to be concerned with the detail of *how* and *when* that information should be applied. These details are implicit in the inference engine. However, the domain expert will often wish to use metaknowledge (i.e., knowledge about knowledge) to steer the inference engine. For instance, he or she may know that a plumbing diagnostic system should examine rules about valves before rules about pipes. In the most complex case, the inference engine can become a metaknowledge-based system.

As the knowledge is represented explicitly in the knowledge base, rather than implicitly within the structure of a program, it can be entered and updated with relative

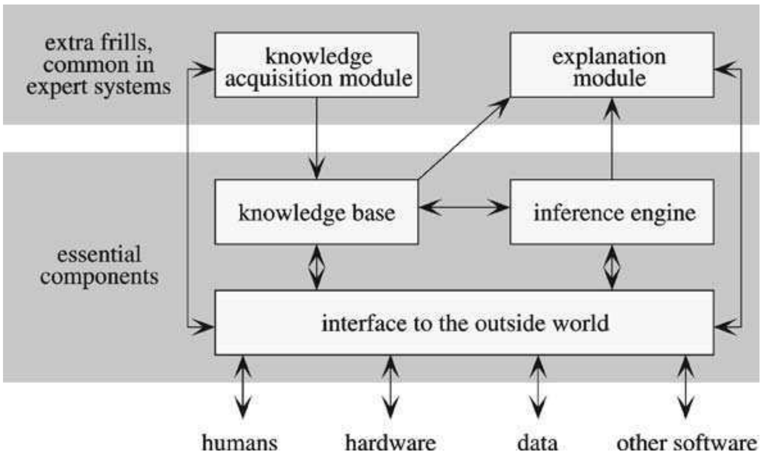


FIG. 4. The main components of a knowledge-based system.

ease by domain experts who may not have any programming expertise. The inference engine uses the knowledge base in a manner that can be likened to a conventional program using a data file. There is also an analogy with the brain, the control processes of which are approximately unchanging in their nature (like the inference engine), even though individual behavior is continually modified by new knowledge and experience (like updating the knowledge base).

Expert systems are a type of knowledge-based system designed to embody expertise in a particular specialized domain such as configuring computer networks or diagnosing faulty equipment. An expert system is intended to act as a human expert who can be consulted on a range of problems within his or her domain of expertise. Typically, the user of an expert system will enter into a dialogue in which he or she describes the problem—such as the symptoms of a fault—and the expert system offers advice, suggestions, or recommendations. It is often proposed that an expert system must offer certain capabilities that mirror those of a human consultant. In particular, it is often claimed that an expert system must be capable of justifying its current line of inquiry and explaining its reasoning in arriving at a conclusion. This is the purpose of the explanation module in Figure 4.

2.2 Rules and Inference

One of the most straightforward means of representing knowledge in a KBS is as rules. The simplest type of rule is called a production rule and takes the form:

```
if <condition> then <conclusion>
```

An example production rule about ACME Inc. might be:

```
rule r1
if the employer of Person is acme then the salary of Person is
  large.
```

Part of the attraction of using production rules is that they can often be written in a form that closely resembles natural language, as opposed to a computer language. The use of capitalization indicates that Person is a variable that can be replaced by a constant value, such as `joe_bloggs or mary_smith`. A fact in this KBS might be:

```
/* fact f1 */
the employer of joe_bloggs is acme.
```

Rules like the one above are a useful way of expressing many types of knowledge, where there is a reasonable level of certainty. This is not always the case, and in this example uncertainty may arise from three distinct sources:

- *uncertain evidence* (perhaps we are not certain that Joe Bloggs works for ACME);
- *uncertain link between evidence and conclusion* (we cannot be certain that an ACME employee earns a large salary, we just know that it is likely);
- *vague rule* (what is a “large” salary anyway?).

The first two sources of uncertainty can be handled by Bayesian updating, or variants of it, and the third source of uncertainty can be handled by fuzzy logic. Both techniques are reviewed below.

One or more given facts may satisfy the condition of a rule, resulting in the generation of a new fact, known as a derived fact. For example, by applying Rule r1 to Fact f1, we can derive:

```
/* fact f2 */
the salary of joe_bloggs is large.
```

The derived fact may satisfy, or partially satisfy, another rule, such as:

```
rule r2
if the salary of a Person is large or a Person has
  job_satisfaction then the Person has
  professional_contentment.
```

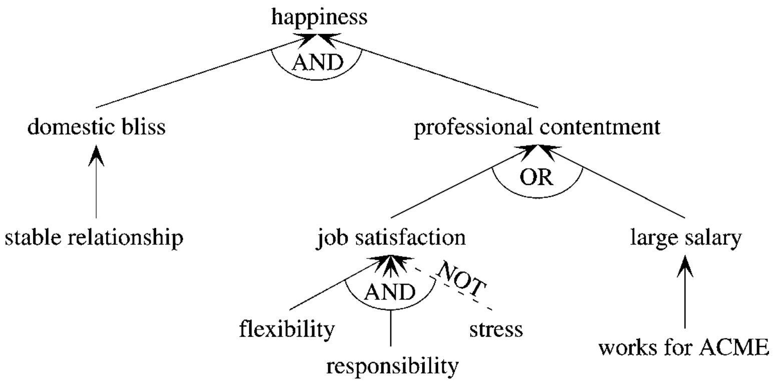


FIG. 5. An inference network.

This, in turn, may lead to the generation of a new derived fact about a person having professional contentment. Rules r1 and r2 are interdependent, since the conclusion of one can satisfy the condition of the other. The interdependencies amongst the rules define a network, as shown in Figure 5, known as an inference network.

It is the job of the inference engine to traverse the inference network to reach a conclusion. Two important types of inference engines can be distinguished: forward-chaining and backward-chaining, also known as data-driven and goal-driven, respectively. A knowledge-based system working in data-driven mode takes the available information (the "given" facts) and generates as many derived facts as it can. The output is therefore unpredictable. This may have either the advantage of leading to unexpected, novel, or innovative solutions to a problem or the disadvantage of wasting time generating irrelevant information. The data-driven approach might typically be used for problems of interpretation, where we wish to know whatever the system can tell us about some data. A goal-driven strategy is appropriate when a more tightly focused solution is required. For instance, a planning system may be required to generate a plan for manufacturing a consumer product. Any other plans are irrelevant. A backward-chaining system might be presented with the proposition: "a plan exists for manufacturing a widget". It will then attempt to ascertain the truth of this proposition by generating the plan. If it fails, the proposition is assumed to be false and no plan is possible.

2.3 Deduction, Abduction, and Induction

The rules that make up the inference network in Figure 5, and the network taken as a whole, are used to link cause and effect:

```
if <cause> then <effect>
```

Using the inference network, we can infer that if Joe Bloggs works for ACME and is in a stable relationship (the causes) then he is happy (the effect). This is the process of *deduction*. Many problems, such as diagnosis, involve reasoning in the reverse direction, i.e., we wish to ascertain a cause, given an effect. This is *abduction*. Given the observation that Joe Bloggs is happy, we can infer by abduction that Joe Bloggs enjoys domestic bliss and professional contentment. However, this is only a valid conclusion if the inference network shows *all* of the ways in which a person can find happiness. This is the closed-world assumption.

The inference network therefore represents a closed world, where nothing is known beyond its boundaries. As each node represents a possible state of some aspect of the world, a model of the current overall state of the world can be maintained. Such a model is dependent on the extent of the relationships between the nodes in the inference network. In particular, if a change occurs in one aspect of the world, many other nodes could be affected. Determining what else is changed in the world model as a consequence of changing one item is known as the frame problem. In the description of Joe Bloggs' world represented in Figure 5, this is equivalent to determining the extent of the relationships between the nodes. For example, if the flexibility of Joe Bloggs' job were to change, Figure 5 suggests that the only direct change is his job satisfaction, which could change his professional contentment and happiness. However, in a more complex model of Joe Bloggs' world, many other nodes could also be affected.

If we have many examples of cause and effect, we can infer the rule (or inference network) that links them. For instance, if every employee of ACME Inc. that we have met earns a large salary, then we might infer Rule r1 above from those examples. Inferring a rule from a set of example cases of cause and effect is termed *induction*.

Deduction, abduction, and induction can be summarized as follows:

- *deduction*: cause $+$ rule $\Rightarrow$ effect;
- *abduction*: effect $+$ rule $\Rightarrow$ cause;
- *induction*: cause $+$ effect $\Rightarrow$ rule.

2.4 Forward Chaining

Consider a rule-based system to monitor the state of a power station boiler and to advise appropriate actions. Sensors on the boiler monitor:

- the temperature of the steam;
- the voltage output from a transducer, which in turn monitors the level of water;
- the status of pressure release valve (i.e., open or closed);
- the rate of flow of water through the control valve.

Using the syntax of the Flex KBS toolkit [9], the following rules could be written for controlling the boiler:

```
rule r1
  if water_level is low
  then report('** open the control valve! **').

rule r2
  if temperature is high
  and water_level is low
  then report('** open the control valve! **')
  and report('** shut down the boiler tubes! **').

rule r3
  if steam_outlet is blocked
  then report('** outlet pipe needs replacing! **').

rule r4
  if release_valve is stuck
  then steam_outlet becomes blocked.

rule r5
  if pressure is high
  and release_valve is closed
  then release_valve becomes stuck.

rule r6
  if steam is escaping
  then steam_outlet becomes blocked.

rule r7
  if temperature is high
  and water_level is not low
  then pressure becomes high.

rule r8
  if transducer_output is low
  then water_level becomes low.

rule r9
  if release_valve is open
  and flow_rate is high
  then steam becomes escaping.

rule r10
  if flow_rate is low
  then control_valve becomes closed.
```

The conclusions of three of the above rules (r1, r2, and r3) consist of recommendations to the boiler operators. In a fully automated system, such rules would be able to perform their recommended actions rather than simply making a recommendation. The remaining rules all involve taking a low-level fact, such as a transducer reading, and deriving a higher-level fact, such as the level of water in the boiler. The input data to the system (sensor readings in our example) are low-level facts; higher-level facts are facts derived from them.

Most of the rules in this example are specific to one particular boiler arrangement and would not apply to other situations. These rules could be described as *shallow*, because they represent shallow knowledge. On the other hand, Rule r7 is a reformulation of a fundamental rule of physics, namely that the boiling temperature of a liquid increases with increasing applied pressure. This is valid under any circumstances and is not specific to a particular boiler. It is an example of a *deep* rule expressing deep knowledge.

The distinction between deep and shallow rules should not be confused with the distinction between high-level and low-level rules. Low-level rules are those that depend on low-level facts. Rule r8 is a low-level rule since it is dependent on a transducer reading. High-level rules make use of more abstract information, such as Rule r3 which relates the occurrence of a steam outlet blockage to a recommendation to replace a pipe. Higher-level rules are those which are closest to providing a solution to a problem, while lower-level rules represent the first stages toward reaching a conclusion.

The task of interpreting and applying the rules belongs to the inference engine. In a forward-chaining system, the application of rules can be broken down as follows:

(i) selecting rules to examine—these are the *available rules*;
(ii) determining which of these are applicable—these are the *triggered* rules; they make up the *conflict set*;
(iii) selecting a rule to *fire*, i.e., to apply.

Suppose the rule-based system has access to the transducer output and to the temperature readings. A sensible set of rules to examine would be r2, r7, and r8, as these rules are conditional on the boiler temperature and transducer output. If the transducer level is found to be low, then Rule r8 is applicable. If it is selected and used to make the deduction `water_level low`, then the rule is said to have fired. If the rule is examined but cannot fire (because the transducer reading is not low), the rule is said to fail.

If we do not know that a given proposition is true, most rule-based systems will assume it to be false. This assumption, known as the closed-world assumption, simplifies the logic required as all propositions are either TRUE or FALSE. If the closed-world assumption is not made, then a third category, namely UNKNOWN, has to be introduced. In the boiler control example, if the water level is unknown or has not yet been investigated, then `water_level is low` is assumed to be false and hence `water_level is not low` is assumed to be true. Thus the condition part of rule r7 could appear to be satisfied and the rule fire, even though the level of the water has not been established at all.

There are two approaches that could be taken to avoid this difficulty—to modify the rules so that they do not contain any negative conditions or to modify the inference engine so that Rule r8 is always examined before r7. A method for achieving the latter approach is described in Section 2.6 below.

2.5 Backward Chaining

Backward-chaining is an inference strategy that assumes the existence of a goal that needs to be established or refuted. In the boiler control example, the goal might be to establish whether it is appropriate to replace the outlet pipe, and we may not be interested in any other deductions that the system is capable of making.

Initially, only those rules that can lead directly to the fulfillment of the goal are selected for examination. In this example, the only rule that can achieve the goal is Rule r3. Its condition part is examined but, since no information is yet established about a steam outlet blockage, Rule r3 cannot be fired. A new goal is then produced, namely steam outlet blocked, corresponding to the condition part of Rule r3. Two rules, r4 and r6, are capable of fulfilling this goal and are described as *antecedents* of Rule r3. The backward-chaining inference engine then tries to establish the truth of the condition part of one of these rules by examining its antecedents, and so on down that branch of the search tree. Eventually it may reach a rule whose condition is satisfied, in which case the goal has been proven. Otherwise it will *backtrack* to the branch point and try to establish the truth of the other branch—see Figure 6. Ultimately, the goal will either be established, with a rule trail to prove it, or the goal fails to be proven and is assumed to be false under the closed-world assumption.

In some systems, such as Flex, the rule syntax for backward chaining is reversed compared with forward chaining. The placing of the conclusion before the condition reflects the fact that in backward-chaining systems it is the conclusion part of a rule that is assessed first, and only if the conclusion is relevant is the condition examined. In Flex, backward-chaining rules are called relations, and the boiler control rules could be re-cast as follows:

```
relation needs_opening(control_valve)
  if low(water_level).

relation needs_shutting_down(boiler_tubes)
  if high(temperature)
  and low(water_level).
```

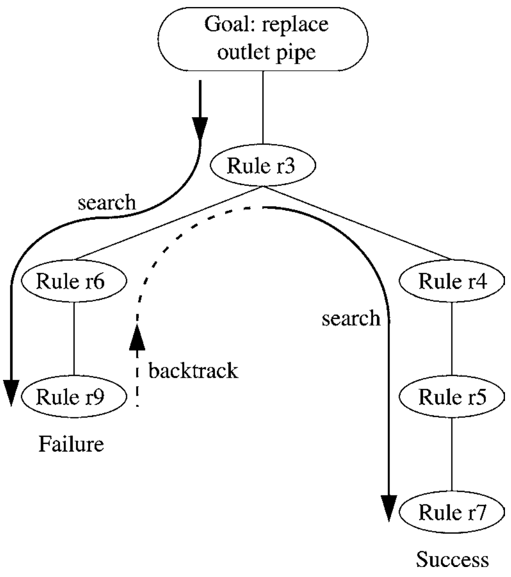


FIG. 6. Backward-chaining applied to the boiler control rules.

```
relation needs_replacing(outlet_pipe)
  if blocked(steam_outlet).
relation blocked(steam_outlet)
  if stuck(release_valve).
relation stuck(release_valve)
  if high(pressure)
  and closed(release_valve).
relation blocked(steam_outlet)
  if escaping(steam).
relation high(pressure)
  if high(temperature)
  and not low(water_level).
relation low(water_level)
  if low(transducer_output).
relation escaping(steam)
  if open(release_valve)
  and high(flow_rate).
relation closed(control_valve)
  if low(flow_rate).
```

2.6 A Hybrid Inference Engine

In all problems involving data interpretation (such as the boiler control example), the high-level rules concerning the overall goal can never fire until lower-level rules for data manipulation have been fired. For example, if Rule r8 fails, then Rule r2 need not be examined as it too will fail. The standard mechanisms for forward- or backward-chaining do not take this information into account and, therefore, involve a great deal of redundant rule examination. The hybrid strategy used in ARBS (Algorithmic and Rule-based Blackboard System) [10,11] is a means of eliminating this source of inefficiency.

Under the hybrid strategy, a *rule dependence network* is built prior to running the system. For each rule, the network shows which other rules may enable it, i.e., its antecedents, and which rules it may enable, i.e., its dependants. The rule dependencies for the boiler control knowledge base are shown in Figure 7. In its data-driven mode, known as *directed forward-chaining*, the hybrid strategy achieves improved efficiency by using the dependence network to select rules for examination. Low-level rules concerning the sensor data are initially selected for examination. As shown in Figure 7, only Rules r8, r9, and r10 need be examined initially. Then higher-level

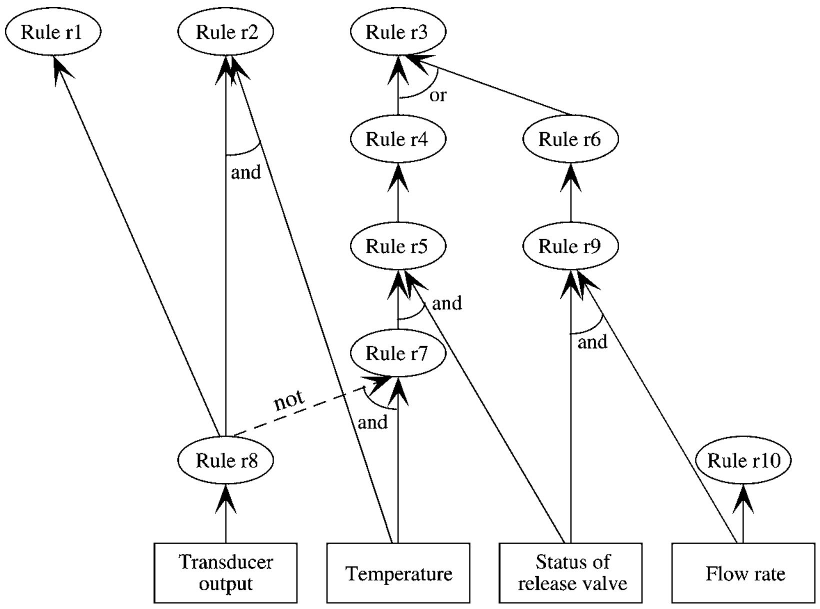


FIG. 7. A rule dependence network.

rules, leading toward a solution, are selected depending on which rules have actually fired. So, if Rules r8 and r9 fire successfully, the new set of rules to be examined becomes r1, r2, and r6. The technique is an effective way of carrying out the first step of forward chaining, i.e., selecting the rules to examine.

For a given rule base, the dependence network needs to be generated only once, and is then available to the system at run-time. The same mechanism can easily be adapted to provide an efficient goal-driven strategy. Given a particular goal, the control mechanism can select the branch of the dependence network leading to that goal and then backward-chain through the selected rules.

Rule-dependence networks help to cut down on the number of rules selected for examination. They are distinct from the more well-known Rete networks that are intended to help to avoid constantly recreating the conflict set [12]. In most applications, the firing of a single rule makes only slight changes to the currently known facts and hence to the membership of the conflict set. The Rete network is constructed dynamically to record these changes. After a rule has fired, the conflict set can be updated rather than created from scratch. In principle, the two types of network could be used together.

3. Moving Beyond Rules

3.1 The Limitations of Rules

When modeling a real system, the amount of knowledge that can be represented in rules that operate on simple variables is limited. Frames provide a flexible structure for modeling complex entities, thereby allowing the creation of more flexible and versatile rules. One key use of frames is in the construction of model-based systems, which are particularly important for fault diagnosis. The links between symptoms and diagnosis are not explicitly stated but can be inferred by comparing the characteristics of a model with those of the real system.

This section will also consider symbolic learning, in which rules can be expanded and altered in the light of experience. An important class of symbolic learning is case-based reasoning, in which previously encountered cases are stored for possible future retrieval and re-use. Finally, this section will consider some of the ways in which rules can be embellished to represent uncertainty and imprecision in the evidence, the conclusion, or the link between them.

3.2 Frame-Based Systems

Frames are data structures developed by AI researchers as a means of representing and organizing knowledge. They are similar in concept to objects, which were developed to meet the needs of software engineers. Like object-oriented systems, frame-based systems contain the ideas of classes, instances, and inheritance. For example, the class vehicle could be defined, along with subclasses car and truck. Characteristics of vehicle are inherited by car and truck classes, so that only information that is specific to the sub-class, or which overrides the inherited information, needs to be declared at the subclass level. Specific instances of classes can then be declared, e.g., my truck can be represented by an instance called `my_truck`. This instance inherits information from its class `truck`, which itself inherits from its parent class `vehicle`.

The attributes of a frame are sometimes called *slots*, into which *values* can be inserted. They allow us to put information onto a frame, such as the number of wheels on my truck. Thus `number_of_wheels` could be a slot associated with the frame instance `my_truck`. This slot could use the default value of 4 inherited from `vehicle` or it may be a locally defined value that overrides the default. The value associated with a slot can be a number, a description, a number range, a procedure, another frame, or anything allowed by the particular implementation. Some frame-based systems allow us to place multiple values in a slot. In such systems, the different pieces of information that we might want to associate with a slot are known as its *facets*. Each facet can have a value associated with it, as shown in Figure 8. For example, we may wish to specify limits on the number of wheels, provide a default, or calculate a value using a function known as *an access function*. In this example, an access function `count_wheels` could calculate the number of wheels when a value is not previously known.

Consider the following example, written using the syntax of Flex:

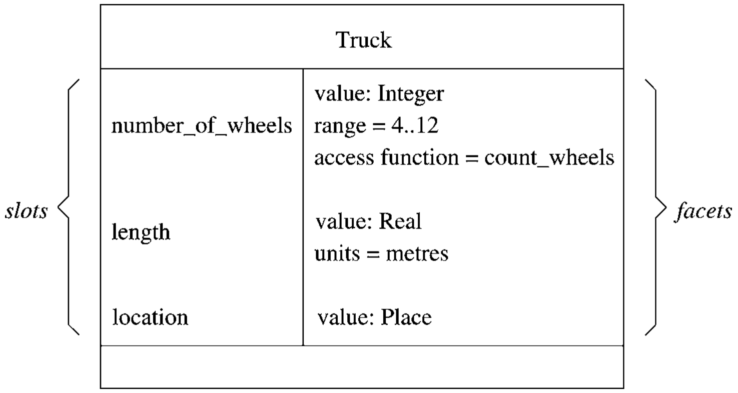


FIG. 8. An example of a frame-based representation.

```
/* define a frame */
frame vehicle;
  default location is garage and
  default number_of_wheels is 4
  and default mass_in_tonnes is 1 .

/* define another frame */
frame truck is a kind of vehicle;
  default mass_in_tonnes is 10 .

/* create an instance of frame truck */
instance my_truck is a truck.

/* create another instance of frame truck */
instance your_truck is a truck;
  number_of_wheels is my_truck's number_of_wheels + 2 .
```

In this example, the default number of wheels is inherited by `my_truck`. The number of wheels on `your_truck` is derived by an access function defined within the frame.

The term *frame* has been used so far here to imply a framework onto which information can be hung. However, frames are also analogous to the frames of a movie film or video tape. In associating values with the slots of a frame we are taking a snapshot of the world at a given instant. At one moment the slot `location` on `my_truck` might contain the value `smallville`, while sometime later it might contain the value largetown.

Frames are so endemic in rule-based systems that they were implicitly included in the first example of rules and facts introduced in Section 2:

```
/* fact f1 */
the employer of joe_bloggs is acme.
```

Here, `joe_bloggs` is an instance of a frame class of person, and employer is a slot that is set to the value acme when this fact is met.

3.3 Model-Based Reasoning

Fulton and Pepe [13] have highlighted three major inadequacies of a purely rule-based system in the context of diagnosing faults: (a) building a complete rule set is a massive task; (b) there is uncertainty arising from whether sensor readings can be believed; and (c) maintaining the rules is a complex task because of the interdependence between them. They used these arguments to justify a model-based approach to fault diagnosis.

The principle of model-based reasoning is that, rather than storing a huge collection of symptom–cause pairs in the form of rules, these pairs can be *generated* by applying underlying principles to the model. The model, which is often frame-based, may describe any kind of system, including physical [14,15], software [16], medical [17], legal [18], and behavioral [19] systems. This review will focus on fault diagnosis in physical systems, which are made up of fundamental components such as tubes, wires, batteries, and valves. As each of these components performs a fairly simple role, it also has a simple failure mode. For example, a wire may break and fail to conduct electricity, a tube can spring a leak, a battery can lose its charge, and a valve may become stuck. Given a model of how these components operate and interact to form a device, faults can be diagnosed by determining the effects of local malfunctions on the overall device.

3.3.1 Building the Model

A physical device is made up of a number of components, each of which is represented as an instance of a class of component. The function of each component is defined within its class definition. The structure of a device is defined by links between the instances of components that make up the device. The device may be in one of several states, for example a refrigerator door may be open or closed, and the thermostat may have switched the compressor on or off. These states are defined by setting the values of instance variables. Function, structure, and state are now described in more detail.

Function. The function of a component is defined by the methods and attributes of its class. Fink and Lusth [20] define four functional primitives, which are classes of components. All components are considered to be specializations of one of these four classes, although Fink and Lusth hint at the possible need to add further functional primitives in some applications. The four functional primitives are:

- *transformer*—transforms one substance into another;
- *regulator*—alters the output of substance B, based upon changes in the input of substance A;
- *reservoir*—stores a substance for later output;
- *conduit*—transports substances between other functional primitives.

A fifth class, *sensor*, that displays the value of its input, may be added to this list.

Structure. Links between component instances can be used to represent their physical associations, thereby defining the structure of a device. For example, two resistors connected in series might be represented as two instances of the class `resistor` and one instance of the class `wire`. Each instance of `resistor` would have a link to the instance of `wire`, to represent the electrical contact between them.

In a refrigerator, the compressor can be regarded either as a device made up from several components or as a component of the overall appliance. It is, therefore, an example of a functional group. In devices where functional groups exist, the device structure is hierarchical. The hierarchical relationship can be represented by means of the composition relationship between objects. It is often adequate to consider just three levels of the structural hierarchy:

$$\begin{tikzcd}\text{device} \arrow[d] \\ \text{functional group} \arrow[d] \\ \text{component}\end{tikzcd}$$

The application of a three-level hierarchy to the structure of a refrigerator is shown in Figure 9.

State. A device may be in one of many alternative states. For example, a refrigerator door may be open or closed, and the compressor may be running or stopped. A state can be represented by setting appropriate instance variables on the components or functional groups, and transitions between states can be represented by changes in these parameters.

In a functioning refrigerator, the compressor will be in the state *running* only if the thermostat is in the state *closed circuit*. The state of the thermostat will alter according to the cabinet temperature. The cabinet temperature is partially dependent on an external factor, namely, the room temperature, particularly if the refrigerator door is open. The components can be modeled as objects. If the thermostat object changes its state, it would send a message to the compressor, which in turn would change its state. A map of possible states can be drawn up, with links indicating the ways in which one state can be changed into another.

3.3.2 Using the Model

The details of how a model can assist in diagnostic tasks vary according to the specific device and the method of modeling it. In general, three potential uses can be identified:

- monitoring the device to check for malfunctions;
- finding a suspect component, thereby forming a tentative diagnosis; and
- confirming or refuting the tentative diagnosis by simulation.

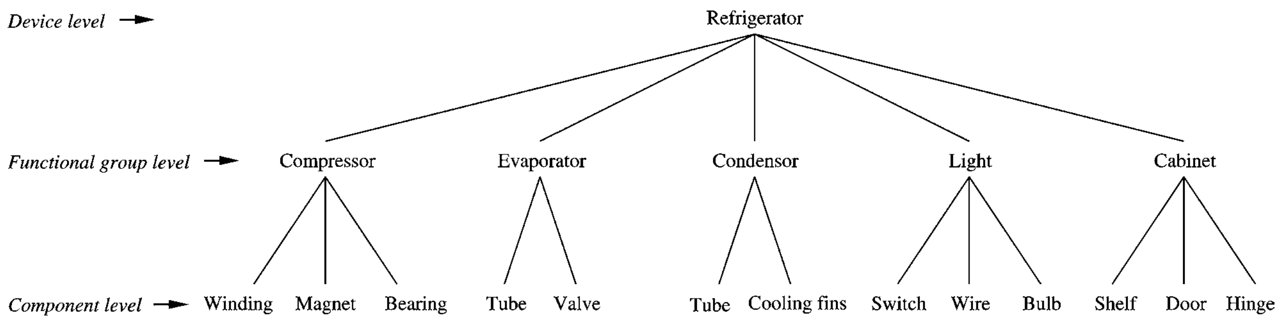


FIG. 9. Structural hierarchy for some components of a refrigerator.

The diagnostic task is to determine which nonstandard component behavior in the model could make the output values of the model match those of the physical system. When a malfunction has been detected, the *single point of failure* assumption is often made. This is the assumption that the malfunction has only one root cause. Such an approach is justified by Fulton and Pepe [13] on the basis that no two failures are truly simultaneous. They argue that one failure will always follow the other either independently or as a direct result.

In summary, the key advantages of model-based reasoning for fault diagnosis are:

- A model is less cumbersome to maintain than a rule base. Real-world changes are easily reflected in changes in the model.
- The model need not waste effort looking for sensor verification. Sensors are treated identically to other components, and therefore a faulty sensor is as likely to be detected as any other fault.
- Unusual failures are just as easy to diagnose as common ones. This is not the case in a rule-based system, which is likely to be most comprehensive in the case of common faults.
- The separation of function, structure, and state may help a diagnostic system to reason about a problem that is outside its area of expertise.
- The model can simulate a physical system, for the purpose of monitoring or for verifying a hypothesis.

3.4 Symbolic Learning

The preceding sections have discussed ways of representing knowledge and drawing inferences. It was assumed that the knowledge itself was readily available and could be expressed explicitly. However, there are many circumstances where this is not the case, such as those listed below.

- The software engineer may need to obtain the knowledge from a domain expert. This task of knowledge acquisition is extensively discussed in the literature [21], often as an exercise in psychology.
- The rules that describe a particular domain may not be known.
- The problem may not be expressible explicitly in terms of rules, facts or relationships. This category includes *skills*, such as welding or painting.

One way around these difficulties is to have the system learn for itself from a set of example solutions. Two approaches can be broadly recognized—*symbolic learning* and *numerical learning*. Symbolic learning describes systems that formulate and modify rules, facts, and relationships, explicitly expressed in words and symbols. In other words, they create and modify their own knowledge base. Numerical learning refers to systems that use numerical models—learning in this context refers to techniques for optimizing the numerical parameters. Numerical learning includes genetic algorithms (Section 5) and artificial neural networks (Section 6).

A learning system is usually given some feedback on its performance. The source of this feedback is called the *teacher* or the *oracle*. Often the teacher role is fulfilled by the environment within which the knowledge-based system is working, i.e., the reaction of the environment to a decision is sufficient to indicate whether the decision was right or wrong. Learning with a teacher is sometimes called *supervised* learning. Learning can be classified as follows, where each category involves a different level of supervision:

(i) *Rote learning*. The system receives confirmation of correct decisions. When it produces an incorrect decision it is “spoon-fed” with the correct rule or relationship that it should have used.

(ii) *Learning from advice*. Rather than being given a specific rule that should apply in a given circumstance, the system is given a piece of general advice, such as “gas is more likely to escape from a valve than from a pipe.” The system must sort out for itself how to move from this high-level abstract advice to an immediately usable rule.

(iii) *Learning by induction*. The system is presented with sets of example data and is told the correct conclusions that it should draw from each. The system continually refines its rules and relations so as to correctly handle each new example.

(iv) *Learning by analogy*. The system is told the correct response to a similar, but not identical, task. The system must adapt the previous response to generate a new rule applicable to the new circumstances.

(v) *Explanation-based learning* (*EBL*). The system analyzes a set of example solutions and their outcomes to determine *why* each one was successful or otherwise. Explanations are generated, which are used to guide future problem solving. EBL is incorporated into PRODIGY, a general-purpose problem-solver [22].

(vi) *Case-based reasoning*. Any case about which the system has reasoned is filed away, together with the outcome, whether it be successful or otherwise. Whenever a new case is encountered, the system adapts its stored behavior to fit the new circumstances. Case-based reasoning is discussed in further detail in Section 3.5 below.

(vii) *Explorative or unsupervised learning*. Rather than having an explicit goal, an explorative system continuously searches for patterns and relationships in the input data, perhaps marking some patterns as interesting and warranting further investigation. Examples of the use of unsupervised learning include:

- *data mining*, where patterns are sought among large or complex data sets;
- identifying *clusters*, possibly for compressing the data;
- learning to *recognize* fundamental features, such as edges, from pixel images;
- *designing* products, where innovation is a desirable characteristic.

In rote learning and learning from advice, the sophistication lies in the ability of the teacher rather than the learning system. If the teacher is a human expert, these two techniques can provide an interactive means of eliciting the expert's knowledge in a suitable form for addition to the knowledge base. However, most of the interest in symbolic learning has focused on case-based reasoning, described in more detail below. Reasoning by analogy is similar to case-based reasoning, while many of the problems and solutions associated with learning by induction also apply to the other categories of symbolic learning.

3.5 Case-Based Reasoning

A characteristic of human intelligence is the ability to recall previous experience whenever a similar problem arises. This is the essence of case-based reasoning (CBR). As Riesbeck and Schank [23] put it,

> A case-based reasoner solves new problems by adapting solutions that were used to solve old problems.

Consider the example of diagnosing a fault in a refrigerator. If an expert system has made a successful diagnosis of the fault, given a set of symptoms, it can file away this information for future use. If the expert system is subsequently presented with details of another faulty refrigerator of exactly the same type, displaying exactly the same symptoms in exactly the same circumstances, then the diagnosis can be completed simply by recalling the previous solution. However, a full description of the symptoms and the environment would need to be very detailed, and it is unlikely to be reproduced exactly. What we need is the ability to identify a previous case, the solution of which can be modified to reflect the slightly altered circumstances, and then saved for future use. Aamodt and Plaza [24] have therefore proposed that CBR can be described by a four-stage cycle:

- *retrieve* the most similar case(s);
- *reuse* the case(s) to attempt to solve the problem;
- *revise* the proposed solution if necessary;
- *retain* the new solution as a part of a new case.

Such an approach is arguably a good model of human reasoning. Indeed case-based reasoning is often used in a semi-automated manner, where a human can intervene at any stage in the cycle.

3.6 Dealing with Uncertainty

3.6.1 Sources of Uncertainty

The discussion of rule-based systems in Section 2 above assumed that we live in a clear-cut world, where every hypothesis is either true, false, or unknown. Furthermore, it was pointed out that many systems make use of the closed-world assumption, whereby any hypothesis that is unknown is assumed to be false. We were then left with a binary system, where everything is either true or false. While this model of reality is useful in many applications, real reasoning processes are rarely so clear-cut. Referring to the example of the control of a power station boiler, we made use of the following rule:

```
IF transducer output is low THEN water level is low
```

There are three distinct forms of uncertainty that might be associated with this rule:

- *Uncertainty in the rule itself.* A low level of water in the boiler is not the only possible explanation for a low transducer output. Another possible cause could be that the float attached to the transducer is stuck. What we really mean by this rule is that if the transducer output is low then the water level is *probably* low.
- *Uncertainty in the evidence*. The evidence upon which the rule is based may be uncertain. There are two possible reasons for this uncertainty. First, the evidence may come from a source that is not totally reliable. For instance, we may not be absolutely certain that the transducer output is low, as this information relies upon a meter to measure the voltage. Second, the evidence itself may have been derived by a rule whose conclusion was probable rather than certain.
- *Use of vague language*. The above rule is based around the notion of a "low" transducer output. Assuming that the output is a voltage, we must consider whether "low" corresponds to 1 mV, 1 V or 1 kV.

Two popular techniques for handling the first two sources of uncertainty are Bayesian updating and certainty theory. Bayesian updating has a rigorous derivation based upon probability theory, but its underlying assumptions, e.g., the statistical independence of multiple pieces of evidence, may not be true in practical situations. Certainty theory [25] does not have a rigorous mathematical basis, but has been devised as a practical and pragmatic way of overcoming some of the limitations of

Bayesian updating. It was first used in the classic MYCIN [8,26] system for diagnosing infectious diseases. Possibility theory, or fuzzy logic, allows the third form of uncertainty, i.e., vague language, to be used in a precise manner.

Bayesian updating and fuzzy logic are explained in more detail below, but many other approaches to handling uncertainty have also been proposed. Some, such as the Dempster–Shafer theory of evidence [27,28] and Inferno [29], attempt to build in greater sophistication. The assumptions and arbitrariness of some of these techniques have meant that reasoning under uncertainty remains a controversial issue. There are also other non-numerical approaches, such as rules that hypothesize and then test for supporting evidence [30].

3.6.2 Bayesian Updating

The technique of Bayesian updating provides a mechanism for updating the probability of a hypothesis $P(H)$ in the presence of evidence E. Often the evidence is a symptom and the hypothesis is a diagnosis. Each hypothesis is allocated an initial likelihood, in the absence of any evidence for or against, known as its prior probability. This probability is then updated by taking into account the available evidence, expressed as rules with an especially extended syntax.

The technique is based upon the application of Bayes' theorem. Bayes' theorem provides an expression for the conditional probability $P(H|E)$ of a hypothesis H given some evidence E, in terms of $P(E|H)$, i.e., the conditional probability of E given H:

$$P(H|E) = \frac{P(H) \times P(E|H)}{P(E)}. \tag{1}$$

A little mathematics allows this relationship to be reformulated as

$$O(H|E) = A \times O(H) \tag{2}$$

where:

$$A = \frac{P(E|H)}{P(E|{\sim}H)} \tag{3}$$

and "$\sim H$" means "not H". The odds $O(H)$ of a given hypothesis H are related to its probability $P(H)$ by the relation:

$$O(H) = \frac{P(H)}{P({\sim}H)} = \frac{P(H)}{1 - P(H)}. \tag{4}$$

$O(H|E)$ is the updated odds of H, given the presence of evidence E, and A is the *affirms* weight of evidence E. It is one of two likelihood ratios. The other is the *denies* weight D of evidence E. The *denies* weight can be obtained by considering the absence of evidence, i.e., $\sim E$:

$$O(H|\sim E) = D \times O(H) \tag{5}$$

where:

$$D = \frac{P(\sim E|H)}{P(\sim E|\sim H)} = \frac{1 - P(E|H)}{1 - P(E|\sim H)}. \tag{6}$$

Using these equations, the *affirms* and *denies* weightings can be derived from probability estimates. However, a more pragmatic approach is frequently adopted, namely, to choose values that produce the right sort of results, even though the values cannot be theoretically justified.

Rule r7 above could be replaced by the following Bayesian updating rule, shown here in Flex syntax:

```
uncertainty_rule r7b
  if    the temperature  is high (affirms 18.0; denies 0.11)
  and   the water_level  is low (affirms 0.10; denies 1.90)
  then  the pressure     is high.
```

Although Bayesian updating is soundly based on probability theory, it is typically used in an *ad hoc* and imprecise manner because:

- linear interpolation of the *affirms* and *denies* weighting is frequently used as a convenient means of compensating for uncertainty in the evidence;
- the likelihood ratios (or the probabilities from which they are derived) and prior probabilities are often based on estimates rather than statistical analysis;
- separate items of evidence that support a single assertion are assumed to be statistically independent, although this may not be the case in reality.

3.6.3 Fuzzy Logic

Fuzzy logic provides a precise way of handling vague terms such as low, medium and high. It does this by replacing simple Boolean logic, based on propositions being true or false, with varying degrees of truth. Thus a fuzzy variable such as temperature may have, perhaps, three fuzzy sets—low medium and high. For a temperature of, for example, 90 °C, the proposition “temperature is high” may have a membership value of 0.6 while the propositions “temperature is medium” and “temperature is low” may have memberships of 0.4 and 0 respectively.

One of the key features of fuzzy logic systems is that a small set of rules can produce output values that change smoothly as the input values change. They are therefore particularly well suited to control decisions, where the control actions need to be smoothly scaled as input measurements change. Control decisions can be thought of as a transformation from state variables to action variables. State variables describe the current state of the physical plant and the desired state. Action variables are those that can be directly altered by the controller, such as the electrical current sent to a furnace, or the flow rate through a gas valve. In high-level control, analytical functions to link state variables to action variables are rarely available, and this provides a strong incentive for using fuzzy logic instead.

A small number of fuzzy rules can produce smooth changes in the action variables as the state variables change. The number of rules required is dependent on the number of state variables, the number of fuzzy sets, and the ways in which the state variables are combined in rule conditions. It could be said that numerical information is explicit in crisp rules, but in fuzzy rules it becomes implicit in the chosen shape of the fuzzy membership functions. In Flex, fuzzy rules look like this

```
fuzzy_rule r1f
  if the temperature is high
  then the pressure is high.

fuzzy_rule r2f
  if the temperature is medium
  then the pressure is medium.

fuzzy_rule r3f
  if the temperature is low
  then the pressure is low.
```

The fuzzy sets associated with the fuzzy variables have to be defined. In Flex, this can be done either through a graphical user interface or by coding:

```
fuzzy_variable temperature;
  ranges from 0 to 400;
  fuzzy_set low is \ shaped and linear at 0, 200;
  fuzzy_set medium is /\ shaped and linear at 0, 200, 400;
  fuzzy_set high is / shaped and linear at 200, 400 .

fuzzy_variable pressure;
  ranges from 0.1 to 0.7;
  fuzzy_set low is \ shaped and linear at 0.1, 0.4;
  fuzzy_set medium is /\ shaped and linear at 0.1, 0.4, 0.7;
  fuzzy_set high is / shaped and linear at 0.4, 0.7;
  defuzzify using
   all memberships
   and mirror rule
   and shrinking.
```

```
relation fuzzy_boiler(Temperature, Pressure)
  if reset all fuzzy values
  and fuzzify temperature from Temperature
  and propagate boiler_rules fuzzy rules
  and defuzzify pressure to Pressure.
```

There is insufficient space in this chapter for a full review of fuzzy logic, though the subject is well reviewed elsewhere [31]. Inevitably, the effectiveness of a fuzzy logic system lies in its details, and particularly the definitions of the fuzzy sets.

4. Intelligent Agents

4.1 Characteristics of an Intelligent Agent

Agent-based technologies have been growing apace, both within the world of AI and in more general software engineering. One motivation has been the rapid escalation in the quantity of information available. Software assistants—or agents—are needed to take care of specific tasks for us. For example, much of the trading on the world's stock exchanges is performed by agents that can react quickly to minor price fluctuations.

While noting that not all agents are intelligent, Wooldridge [32] gives the following definition for an agent:

> An agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.

From this definition we can see that the three key characteristics of an agent are autonomy, persistence, and the ability to interact with its environment. Autonomy refers to an agent's ability to make its own decisions based on its own expertise and circumstances, and to control its own internal state and behavior. The definition implies that an agent functions continuously within its environment, i.e., it is persistent over time. Agents are also said to be situated, i.e., they are responsive to the demands of their environment and are capable of acting upon it. Interaction with a physical environment requires perception through sensors, and action through actuators or effectors. Interaction with a purely software environment requires only access to and manipulation of data and programs.

We might reasonably expect an *intelligent* agent to be all of the following:

- reactive,
- goal-directed,

- adaptable,
- socially capable.

Social capability refers to the ability to cooperate and negotiate with other agents (or humans). It is quite easy to envisage an agent that is purely reactive, e.g., one whose only role is to place a warning on your computer screen when the printer has run out of paper. Likewise, modules of conventional computer code can be thought of as goal-directed in the limited sense that they have been programmed to perform a specific task regardless of their environment. Since it is autonomous, an intelligent agent can decide its own goals and choose its own actions in pursuit of those goals. At the same time, it must also be able to respond to unexpected changes in its environment. It, therefore, has to balance reactive and goal-directed behavior, typically through a mixture of problem solving, planning, searching, decision making, and learning through experience.

There are at least four different schools of thought about how to achieve an appropriate balance between reactive and goal-directed behavior. These are reviewed below.

4.1.1 Logic-Based Architectures

At one extreme, the purists favor logical deduction based on a symbolic representation of the environment [33,34]. This approach is elegant and rigorous, but it relies on the environment remaining unchanged during the reasoning process. It also presents particular difficulties in symbolically representing the environment and reasoning about it.

4.1.2 Emergent Behavior Architectures

In contrast, other researchers propose that logical deduction about the environment is inappropriately detailed and time-consuming. For instance, if a heavy object is falling toward you, the priority should be to move out of the way rather than to analyze and prove the observation. These researchers suggest that agents need only a set of reactive responses to circumstances, and that intelligent behavior will emerge from the combination of such responses. This kind of architecture is based on *reactive agents*, i.e., agents that include neither a symbolic world model nor the ability to perform complex symbolic reasoning [35]. A well-known example of this approach is Brooks' *subsumption architecture* [36], containing behavior modules that link actions to observed situations without any reasoning at all. The behaviors are arranged into a *subsumption hierarchy*, where low-level behavior such as “avoid object” has precedence over higher-level goal-oriented behaviors such as “move across room.” This simple and practical approach can be highly effective. Its chief drawback is that the emphasis placed on the local environment can lead to a lack of awareness of the bigger picture.

4.1.3 Knowledge-Level Architectures

A third type of architecture, based on *knowledge-level* agents, treats each intelligent agent as a knowledge-based system in microcosm. Such agents are said to be *deliberative*. They are the antithesis of reactive agents, since they explicitly represent a symbolic model of the world and make decisions via logical reasoning based on pattern matching and symbolic manipulation [35]. A deliberative agent's knowledge determines its behavior in accordance with Newell's Principle of Rationality [37], which states that:

> if an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action.

One of the most important manifestations of this approach is known as the beliefs–desires–intentions (BDI) architecture [38]. Here, knowledge of the environment is held as "beliefs" and the overall goals are "desires." Together, these shape the "intentions," i.e., selected options that the system commits itself toward achieving (Figure 10). The intentions stay in place only so long as they remain both consistent with the desires and achievable according to the beliefs. The process of determining what to do, i.e., the desires or goals, is *deliberation* [39]. The process of determining how to do it, i.e., the plan or intentions, is *means–ends analysis*. In this architecture, the balance between reactivity and goal-directedness can be restated as one between reconsidering intentions frequently (as a cautious agent might) and infrequently (as a bold or cavalier agent might). Unsurprisingly, Kinny and Georgeff [40] found that the cautious approach works best in a rapidly changing environment and the bold approach works best in a slowly changing environment.

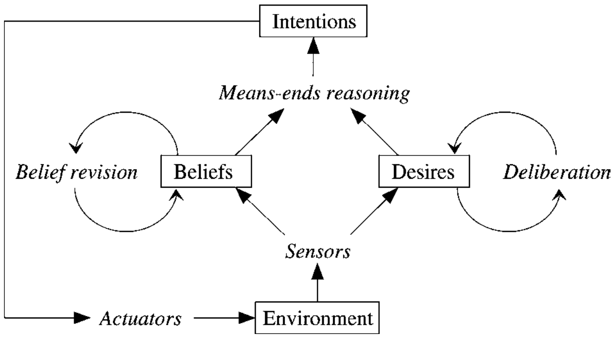


FIG. 10. BDI architecture.

4.1.4 Layered Architectures

The final approach to the balancing of reactive and goal-directed behavior is to mix modules which adopt the two different stances. This is the basis of the layered architecture of Touring Machines [41], so-called because their original application was as autonomous drivers for vehicles negotiating crowded streets. In fact, these agents contain three specific layers: a reactive layer, a planning layer for goal-directed behavior, and a modeling layer for modeling the environment. The problem of balancing the layers remains; in this case an intelligent control subsystem ensures that each layer has an appropriate share of power.

4.2 Multiagent Systems

Adding intelligence to an agent, so that it can mimic human capabilities, is interesting enough, but things get particularly exciting when we consider the possibilities of multiple interacting intelligent agents. This gives us the possibility not just of mimicking the behavior of individual humans, but of mimicking human organizations and societies containing individuals with their own personalities. The agents need not run on a single computer—a distributed AI system is one in which they run on separate networked computers.

4.2.1 Benefits of a Multiagent System

There are two broad classes of problems for which multiagent systems offer the only practicable approach:

- *Inherently complex problems*: Such problems are simply too large to be solved by a single hardware or software system. As the agents are provided with the intelligence to handle a variety of circumstances, there is some uncertainty as to exactly how an agent system will perform in a specific situation. Nevertheless, well-designed agents will ensure that every circumstance is handled in an appropriate manner even though it may not have been explicitly anticipated.
- *Inherently distributed problems*: Here the data and information may exist in different physical locations, or at different times, or may be clustered into groups requiring different processing methods or semantics. These types of problems require a distributed solution, which can be provided by agents running in parallel on distributed processors.

However, Jennings has argued that multiagent systems are not just suited to complex and distributed problems, but are also applicable to the design and construction of mainstream software [42]. His argument is that multiagent systems offer the following general benefits: a more natural view of intelligence, speed and efficiency, robustness and reliability, scalability, ease of development, and reduced cost. He also points out the benefits of granularity, i.e., that agents can be designed to operate at an appropriate level of detail. Many "fine-grained" agents may be required to work on separate small details of a problem, while a few "coarse-grained" agents can concentrate on higher-level strategy.

4.2.2 Building a Multiagent System

A multiagent system is dependent on interactions between intelligent agents. There are, therefore, some key design decisions to be made, e.g., when, how, and with whom should agents interact? In cooperative models, several agents try to combine their efforts to accomplish as a group what the individuals cannot. In competitive models, each agent tries to get what only some of them can have. In either type of model, agents are generally assumed to be honest.

Multiagent systems are often designed as computer models of human functional roles. For example, in a hierarchical control structure, one agent is the superior of other subordinate agents. Peer group relations, such as may exist in a team-based organization, are also possible. This section will address three models for managing agent interaction, known as contract nets [43], cooperative problem solving (CPS) [44,45] and shifting matrix management (SMM) [46]. The semantics of communication between agents will also be briefly addressed.

Contract Nets. Imagine that you have decided to build your own house. You are unlikely to undertake all the work yourself. You will probably employ specialists to draw up the architectural plans, obtain statutory planning permission, lay the foundations, build the walls, install the floors, build the roof, and connect the various utilities. Each of these specialists may in turn use a subcontractor for some aspect of the work. This arrangement is akin to the contract net framework [43] for agent cooperation. Here, a manager agent generates tasks and is responsible for monitoring their execution. The manager enters into explicit agreements with contractor agents willing to execute the tasks. Individual agents may take on manager or contractor roles dynamically during problem solving.

To establish a contract, the manager agent advertises the existence of the tasks to other agents. Agents that are potential contractors evaluate the task announcements and submit bids for those to which they are suited. The manager evaluates the bids and awards contracts for execution of the task to the agents it determines to be the most appropriate. The negotiation process may recur if a contractor subdivides its task and awards contracts to other agents, for which it is the manager.

CPS Framework. The cooperative problem-solving (CPS) framework is a top-down model for agent cooperation. As in the BDI model, an agent's intentions play a key role. They determine the agent's personal behavior at any instant, while joint intentions control its social behavior [47]. The framework comprises the following four stages, after which the team disbands and Stage 1 begins again:

Stage 1: *recognition*. Some agents recognize the potential for cooperation with an agent that is seeking assistance, possibly because it has a goal it cannot achieve in isolation.

Stage 2: *team formation*. An agent that recognized the potential for cooperative action at Stage 1 solicits further assistance. If successful, this stage ends with a group having a joint commitment to collective action.

Stage 3: *plan formation*. The agents attempt to negotiate a joint plan that they believe will achieve the desired goal.

Stage 4: *team action*. The newly agreed plan of joint action is executed. By adhering to an agreed social convention, the agents maintain a close-knit relationship throughout.

Shifting Matrix Management (SMM). SMM [46] is a model of agent coordination that has been inspired by Mintzberg's Shifting Matrix Management model of organizational structures [48]. Unlike the traditional management hierarchy, matrix management allows multiple lines of authority, reflecting the multiple functions expected of a flexible workforce. *Shifting* matrix management takes this idea a stage further by regarding the lines of authority as temporary, typically changing as different projects start and finish. In order to apply these ideas to agent cooperation, a cyclical six-stage framework has been devised, as outlined below [46]:

Stage 1: *goal selection*. Agents select the tasks they want to perform, based on their initial mental states.

Stage 2: *action selection*. Agents select a way to achieve their goals. In particular, an agent that recognizes its intended goal is common to other agents would have to decide whether to pursue the goal in isolation or in collaboration with other agents.

Stage 3: *team formation*. Agents that are seeking cooperation attempt to organize themselves into a team. The establishment of a team requires an agreed code of conduct, a basis for sharing resources, and a common measure of performance.

Stage 4: *team planning*. The workload is distributed among team members.

Stage 5: *team action*. The team plan is executed by the members under the team's code of conduct.

Stage 6: *shifting*. The last stage of the cooperation process, which marks the disbanding of the team, involves shifting agents' goals, positions, and roles. Each agent updates its probability of team-working with other agents, depending on whether or not the completed team-working experience with that agent was successful. This updated knowledge is important, as iteration through the six stages takes place until all the tasks are accomplished.

4.2.3 Comparison of Cooperative Models

Li et al. [46] have implemented all three cooperative models using a blackboard architecture (see Section 7.2). In a series of tests, the three different models of agent cooperation were used to control two robots that were required to work together to complete a variety of tasks. A total of 50 different tasks were generated, each of which was presented twice, creating a set of 100 tasks that were presented in random order. Figure 11 shows the number of tasks completed for the three models as a function of time. The data are an average over four rounds of tests, where the tasks were presented in a different random order for each round. For these tasks and conditions, the SMM model achieves an average task completion time of 5.0 s, compared with 6.03 s for the CPS model and 7.04 s for the Contract Net model. Not only is the task completion rate greatest for SMM, so too is the total number of completed tasks. All 100 tasks were completed under the SMM model, compared with 71 for Contract Nets and 84 for the CPS model.

As the SMM model requires predominantly deliberative behaviors from the agents rather than simple reactive behaviors, it might be argued that it is inappropriate in a time-critical environment. However, in the domain of cooperation between multiple robots, the agents' computation is much faster than the robots' physical movements. Thus Li et al. propose that the SMM model is appropriate because it emphasizes the avoidance of mistakes by reasoning before acting. If a mistake is make, actions are stopped sooner rather than later.

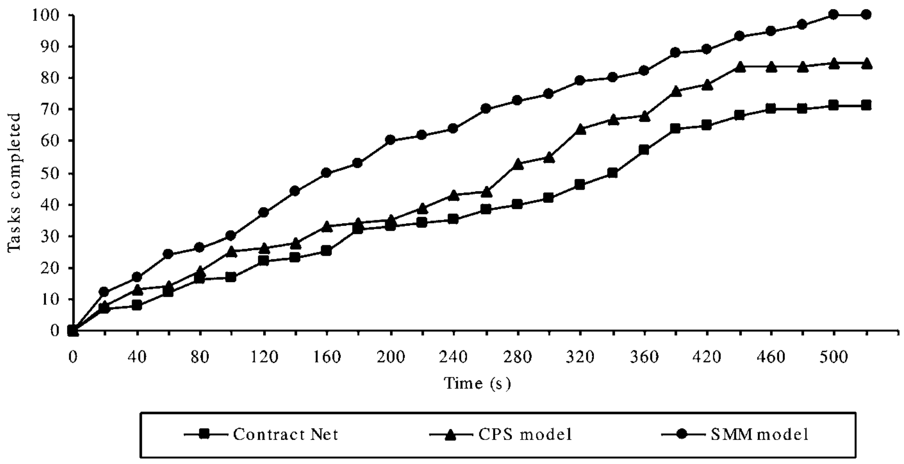


FIG. 11. Task completion, averaged over four test runs.

4.2.4 Communication Between Agents

Agents need to communicate, either synchronously or asynchronously. Synchronous communication is rather like a conversation—after sending a message, the sending agent awaits a reply from the recipient. Asynchronous communication is more akin to sending an email or a letter—although you might expect a reply at some future time, you do not expect the recipient to read or act upon the message immediately.

Agents may be implemented by different people at different times on different computers, yet still be expected to communicate with each other. Consequently, there has been a drive to standardize the structure of messages between agents, regardless of the domain in which they are operating. A generally accepted premise is that the *form* of the message should be understandable by all agents regardless of their domain, even if they do not understand its content. Thus, the structure needs to be standardized in such a way that the domain-specific content is self-contained within it. Only specialist agents need to understand the content, but all agents need to be able to understand the form of the message. Structures for achieving this are called agent communication languages (ACLs), which include Knowledge Query and Manipulation Language (KQML) [49].

To make KQML as general a communication language as possible, it is defined without reference to the content of the body of the messages it transports. However, each KQML message does say something about the meaning of the message—for instance, it may be a question, or it may be a statement of fact. In KQML, the structure contains at least the following components:

- A *performative*. This is a single word that describes the purpose of the message, e.g., tell, cancel, evaluate, advertise, ask-one, register, reply.
- The identity of the *sender* agent.
- The identity of the *receiver* agent.
- The *language* used in the content of the message. Any programming language can be used for the domain-specific content.
- The *ontology*, or vocabulary, of the message. This provides the context within which the message content is to be interpreted.
- The message *content*.

4.3 Swarm Intelligence

Brookes' subsumption architecture [36] has already been mentioned as a model that does not attempt to build intelligence into individual agents, and yet intelligent behavior emerges from the combined behavior of several agents. This is the principle behind the growing area of swarm intelligence [50,51]. It contrasts starkly with cooperative multiagent models such as SMM, in which agents deliberate at length about their individual contributions to the overall task.

Much of the work on swarm intelligence has been inspired by the behavior of ant colonies, although other analogies exist such as worker bees, birds, and spiders. One commonly-cited example is the traveling salesperson example, in which the shortest route between sites is sought. The problem is identical in principle to that encountered in optimizing the layout of printed circuit boards or routing traffic on a telecommunications network. Ants tackle this problem by leaving a trail of pheromone as they travel, which encourages other ants to follow. Ants that have found the shortest route to a food source are the first to reinforce their route with pheromone as they return to the nest. The shortest route then becomes the preferred one for other members of the colony. This behavior is easily mimicked with simple agents, and the modeled pheromone can be given a degree of volatility, allowing the pheromone to evaporate and hence the swarm to change behavior as the environment changes [52].

Other behaviors have also been successfully modeled using swarm intelligence. For instance, data-mining of bank records has been achieved by clustering customers with similar characteristics using swarm intelligence that mimics ant colonies' ability to cluster corpses and thereby clean their nest [53]. The ability to perceive a visual image has been modeled by simulating colonies of ants that swarm across a printed image. By reinforcing paths across the image, it can be transformed from a photographic image to an outline map [54].

5. Genetic Algorithms

5.1 Evolutionary Computation

Like swarm intelligence, evolutionary computation mimics nature and relies on an emergent solution from the combined effect of many individuals. Genetic algorithms (GAs) have been inspired by natural evolution, the process by which successive generations of animals and plants are modified so as to approach an optimum form. Each offspring has different features from its parents, i.e., it is not a perfect copy. If the new characteristics are favorable, the offspring is more likely to flourish and pass its characteristics to the next generation. However, an offspring with unfavorable characteristics is likely to die without reproducing.

These ideas have been applied to mathematical optimization, where a population of candidate solutions "evolves" toward an optimum [55]. Often the optimization problem is one of determining a minimum, where the function that is being minimized is referred to as a cost function. The cost function might typically be the difference, or error, between a desired output and the actual output. Alternatively, optimization is sometimes viewed as maximizing the value of a function, known then as a fitness function. In fact the two approaches are equivalent, because the fitness can simply be taken to be the negation of the cost and vice versa, with the optional addition of a constant value to keep both cost and fitness positive. Similarly, fitness and cost are sometimes taken as the reciprocals of each other.

Other simpler numerical optimization techniques, such as hill-climbing, store just one “best so far” candidate solution, and new trial solutions are generated by taking a small step in a chosen direction. Genetic algorithms are different in both respects. First, a population of several candidate solutions is maintained. Second, the members of one generation can be a considerable distance in the search space from the previous generation.

Each individual in the population of candidate solutions is graded according to its fitness. The higher the fitness of a candidate solution, the greater are its chances of reproducing and passing its characteristics to the next generation. In order to implement a GA, the following design decisions need to be made:

- how to use sequences of numbers, known as chromosomes, to represent the candidate solutions;
- the size of the population;
- how to evaluate the fitness of each member of the population;
- how to select individuals for reproduction using fitness information (conversely, how to determine which less-fit individuals will not reproduce);
- how to reproduce candidates, i.e., how to create a new generation of candidate solutions from the existing population;
- when to stop the evolutionary process.

A flow chart for the basic GA is shown in Figure 12. In the basic algorithm, the following assumptions have been made:

- The initial population is randomly generated.
- Individuals are evaluated according to the fitness function.
- Individuals are selected for reproduction on the basis of fitness; the fitter an individual, the more likely it is to be selected. Further details are given in Section 5.6 below.
- Reproduction of chromosomes to produce the next generation is achieved by “breeding” between pairs of chromosomes using the crossover operator and then applying a mutation operator to each of the offspring. The crossover and

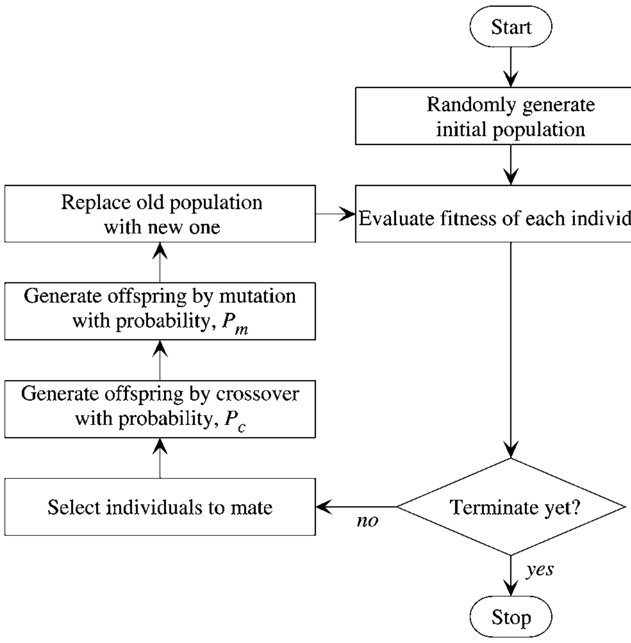

Start
Randomly generate initial population
Evaluate fitness of each individual
Terminate yet?
no
yes
Stop
Select individuals to mate
Generate offspring by crossover with probability, P_c
Generate offspring by mutation with probability, P_m
Replace old population with new one

FIG. 12. The basic genetic algorithm.

mutation operators are described below; the balance between them is yet another decision that the GA designer faces.

5.2 Chromosomes

Each point in the search space can be represented as a unique chromosome, made up of *genes*. Suppose, for example, we are trying to find the maximum value of a fitness function, $f(x, y)$. In this example, the search space variables, x and y, are constrained to the 16 integer values in the range 0–15. A chromosome corresponding to any point in the search space can be represented by two genes:

x	y

Thus the point (2, 6) in search space would be represented by the following chromosome:

2	6

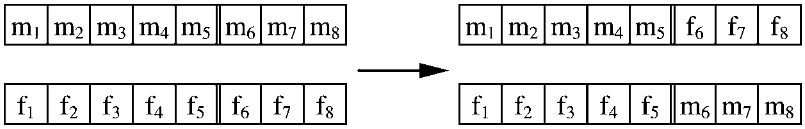
The possible values for the genes are called *alleles,* so there are 16 alleles for each gene in this example. Each position along the chromosome is known as a *locus*; there are two loci in the above example. The loci are usually constrained to hold only binary values. (The term *evolutionary algorithm* describes the more general case where this constraint is relaxed.) The chromosome could therefore be represented by eight loci comprising the binary numbers 0010 and 0110, which represent the two genes:

0	0	1	0	0	1	1	0

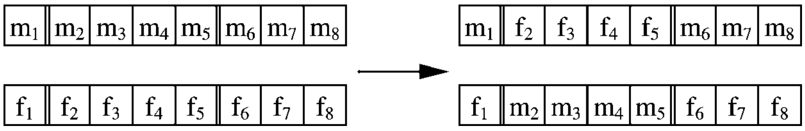
Although there are still 16 alleles for the genes, there are now only two possible values (0 and 1) for the loci. The chromosome can be made as long as necessary for problems involving many variables, or where many loci are required for a single gene. In general, there are 2^N alleles for a binary-encoded gene that is N bits wide.

5.3 Crossover

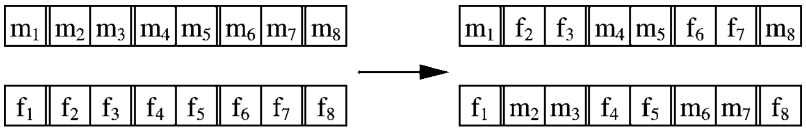
Crossover (and mutation, below) are used to introduce the diversity in the population so that the algorithm can explore the search space. Child chromosomes are produced by aligning two parents, picking a random position along their length, and swapping the tails with a probability P_c, known as the crossover probability. An example for an eight-loci chromosome, where the mother and father genes are represented by m_i and f_i respectively, would be:



This is known as single-point crossover, as only one position is specified for separating the swapped and unswapped loci. In fact this is a misnomer, as a second cross-over position is always required. In single-point crossover the second crossover position is assumed to be the end of the chromosome. This can be made clearer by considering two-point crossover, where the chromosomes are treated as though they were circular, i.e., m_1 and m_8 are neighboring loci:



In general, multipoint crossover is also possible, provided there are an even number of crossover points:


m_1 m_2 m_3 m_4 m_5 m_6 m_7 m_8
f_1 f_2 f_3 f_4 f_5 f_6 f_7 f_8
m_1 f_2 f_3 m_4 m_5 f_6 f_7 m_8
f_1 m_2 m_3 f_4 f_5 m_6 m_7 f_8

In the extreme case, each locus is considered for crossover, independently of the rest, with crossover probability P_c. This is known as *uniform crossover* [56].

5.4 Mutation

Unlike crossover, mutation involves altering the values of one or more loci. This creates new possibilities for gene combinations that can be generated by crossover. In a binary chromosome, randomly selected loci can be toggled, i.e., 1 becomes 0 and 0 becomes 1. In non-binary chromosomes, a randomly selected gene can be replaced by a randomly generated valid value. Loci are mutated randomly with a probability P_m. The main advantage of mutation is that it puts variety into the gene pool, enabling the GA to explore potentially beneficial regions of the search space that might otherwise be missed. This helps to counter premature convergence, described below.

5.5 Validity Check

Depending on the optimization problem, an additional check may be required to ensure that the chromosomes in the new generation represent valid points in the search space. Consider, for example, a chromosome comprising four genes, each of which can take three possible values: A, B, or C. The binary representation for each gene would require two bits, where each gene has redundant capacity of one extra value. In general, binary encoding of a gene with n alleles requires X bits, where X is $\log_2 n$ rounded up to the nearest integer. Thus there is redundant capacity of $2^X - n$ values per gene. Using the binary coding $A = 01$, $B = 10$, $C = 11$, a binary chromosome to represent the gene combination BACA would look like this:

1	0	0	1	1	1	0	1

A mutation that toggled the last locus would generate an invalid chromosome, since a gene value of 00 is undefined:

1	0	0	1	1	1	0	0

Similarly defective chromosomes can also be generated by crossover. In each case the problem can be avoided by using *structured* operators, i.e., requiring crossover and mutation to operate at the level of genes rather than loci. Thus, crossover points could be forced to coincide with gene boundaries and mutation could randomly select new values for whole genes.

An alternative approach is to detect and repair invalid chromosomes. Once a defective chromosome has been detected, a variety of ways exist to repair it. One approach is to generate "spare" chromosomes in each generation, which can then be randomly selected as replacements for any defective ones.

5.6 Selection

It has already been stated that individuals are selected for reproduction on the basis of their fitness, i.e., the fittest chromosomes have the highest likelihood of reproducing. Selection determines not only which individuals will reproduce, but how many offspring they will have. The selection method can have an important impact on the effectiveness of a GA.

Selection is said to be strong if the fittest individuals have a much greater probability of reproducing than less fit ones. Selection is said to be weak if the fittest individuals have only a slightly greater probability of reproducing than the less fit ones. If the selection method is too strong, the genes of the fittest individuals may dominate the next generation population even though they may be suboptimal. This is known as premature convergence, i.e., the exploitation of a small region of the search space before a thorough exploration of the whole space has been achieved.

On the other hand, if the selection method is too weak, less fit individuals are given too much opportunity to reproduce and evolution may become too slow. This can be a particular problem during the latter stages of evolution, when the whole population may have congregated within a smooth and fairly flat region of the search space. All individuals in such a region would have similar, relatively high, fitnesses and, thus, it may be difficult to select among them. This can result in stalled evolution, i.e., there is insufficient variance in fitness across the population to drive further evolution.

A range of alternative methods of selection exist. The method originally proposed by Holland [55], is roulette wheel selection with replacement. This is a random process in which individuals have a probability of selection that is proportional to their fitness. A refinement of this approach that overcomes statistical variability is stochastic universal selection. Various refinements and alternatives exist, all of which are intended to counter premature convergence by slowing evolution and maintaining diversity in the early generations, and to counter stalled evolution by spreading out the selection rates for the population in the later stages. These approaches include linear fitness scaling, Boltzmann fitness scaling, rank selection, truncation selection, and tournament selection [11,57].

A further refinement of the selection process is elitism, whereby one or more of the fittest individuals pass unchanged through to the next generation. The fittest solutions found so far are, therefore, preserved within the population. Elitism can be thought of as the allowance of cloning alongside reproduction.

5.7 Selecting GA Parameters

One of the main difficulties in building a practical GA is choosing suitable values for parameters such as population size, mutation rate, and crossover rate. De Jong's guidelines, as cited in [57], are still widely followed, namely, to start with:

- a relatively high crossover probability (0.6–0.7);
- a relatively low mutation probability (typically set to $1/l$ for chromosomes of length l);
- a moderately sized (50–500) population.

Some of the parameters can be allowed to vary. For example, the crossover rate may be started at an initially high level and then progressively reduced with each generation or in response to particular performance measures.

Given the difficulties in setting the GA parameters, it is unsurprising that many researchers have tried encoding them so that they too might evolve toward optimum values. These self-adaptive parameters can be encoded in individual chromosomes, providing values that adapt specifically to the characteristics of the chromosome. Typically, a minimal background mutation rate applies to the population as a whole, and each chromosome includes a gene that encodes a mutation rate to apply to the remaining genes on that chromosome. Self-adaptive parameters do not completely remove the difficulties in choosing parameters, but by deferring the choice to the level of metaparameters, i.e., parameters' parameters, it may become less critical.

6. Neural Networks

6.1 What Is a Neural Network?

Artificial neural networks are a family of techniques for numerical learning, like genetic algorithms, but in contrast to the symbolic techniques reviewed in the previous sections. They consist of many nonlinear computational elements which form the network nodes or neurons, linked by weighted interconnections. They are analogous in structure to the neurological system in animals, which is made up of real rather than artificial neural networks. Artificial neural networks have, perhaps, been the subject of more hype than any other aspect of artificial intelligence because their name conjures up unrealistic notions of an artificial brain. Practical artificial neural networks are much simpler than biological ones, so it is unrealistic to expect them to produce the sophisticated behavior of humans or animals. Nevertheless, they can perform certain tasks, particularly classification, most effectively. Throughout the rest of this chapter, the expression 'neural network' will be taken to mean an artificial neural network. The technique of using neural networks is described as connectionism.

Each node in a neural network may have several inputs, each of which has an associated weighting. The node performs a simple computation on its input values, which are single integers or real numbers, to produce a single numerical value as its output. The output from a node can either form an input to other nodes or be part of the output from the network as a whole. The overall effect is that a neural network generates a pattern of numbers at its outputs in response to a pattern of numbers at its inputs. These patterns are treated as vectors, e.g., (0.1, 1.0, 0.2).

The weights on the node interconnections, together with the overall topology, define the output vector that is derived by the network from a given input vector. The weights do not need to be known in advance, but can be learned by adjusting them automatically using a training algorithm. In the case of supervised learning, the weights are derived by repeatedly presenting to the network a set of example input vectors along with the corresponding desired output vector for each of them. The weights are adjusted with each iteration until the actual output for each input is close to the desired vector. In the case of unsupervised learning, the examples are presented without any corresponding desired output vectors. With a suitable training algorithm, the network adjusts its weights in accordance with naturally occurring patterns in the data. The output vector then represents the position of the input vector within the discovered patterns of the data.

Part of the appeal of neural networks is that when presented with noisy or incomplete data, they will produce an approximate answer rather than one that is incorrect. Similarly, when presented with unfamiliar data that lie within the range of its previously seen examples, the network will generally produce an output that is a reasonable interpolation between the example outputs. Neural networks are, however, unable to extrapolate reliably beyond the range of the previously seen examples. Interpolation can also be achieved by fuzzy logic (see Section 3.6). Thus, neural networks and fuzzy logic often represent alternative solutions to a particular engineering problem and may be combined in a hybrid system (see Section 7.4.1).

6.2 Neural Network Applications

Neural networks can be applied to a diversity of tasks. In general, the network associates a given input vector $(x_1, x_2, \ldots, x_n)$ with a particular output vector $(y_1, y_2, \ldots, y_m)$, although the function linking the two may be unknown and may be highly nonlinear.

6.2.1 Nonlinear Estimation

Neural networks provide a useful technique for determining the values of variables that cannot be measured easily, but which are known to depend in some complex way on other more accessible variables. The measurable variables form the network input vector and the unknown variables constitute the output vector. This is nonlinear estimation. Supervised learning is used, so each example in the training data comprises two vectors: an input vector and its corresponding desired output vector. (This assumes that some values for the less accessible variable have been obtained to form the desired outputs.) During training, the network learns to associate the example input vectors with their desired output vectors. When it is subsequently presented with a previously unseen input vector, the network is able to interpolate between similar examples in the training data to generate an output vector.

6.2.2 Classification

Often the output vector from a neural network is used to represent one of a set of known possible outcomes, i.e., the network acts as a classifier. For example, a speech recognition system could be devised to recognize three different words: *yes*, *no*, and *maybe*. The digitized sound of the words would be preprocessed in some way to form the input vector. The desired output vector would then be either (0, 0, 1), (0, 1, 0), or (1, 0, 0), representing the three classes of word.

Each example in the training data set would comprise a digitized utterance of one of the words as the input vector, using a range of different voices, together with the corresponding desired output vector. During training, the network learns to associate similar input vectors with a particular output vector. When it is subsequently presented with a previously unseen input vector, the network selects the output vector that offers the closest match. This type of classification would not be straightforward using non-connectionist techniques, as the input data rarely correspond exactly to any one example in the training data.

6.2.3 Clustering

Clustering is a form of unsupervised learning, i.e., the training data comprise a set of example input vectors without any corresponding desired output vectors. As successive input vectors are presented, they are clustered into N groups, where the integer N may be pre-specified or may be allowed to grow according to the diversity of the data. For instance, digitized preprocessed spoken words could be presented to the network. The network would learn to cluster together the examples that it considered to be in some sense similar to each other. In this example, the clusters might correspond to different words or different voices.

Once the clusters have formed, a second neural network can be trained to associate each cluster with a particular desired output. The overall system then becomes a classifier, where the first network is unsupervised and the second one is supervised. Clustering is useful for data compression and is an important aspect of data mining, i.e., finding patterns in complex data.

6.2.4 Content-Addressable Memory

The use of a neural network as a content-addressable memory involves a form of supervised learning. During training, each example input vector becomes stored in a dispersed form through the network. There are no separate desired output vectors associated with the training data, as the training data represent both the inputs and the desired outputs.

When a previously unseen vector is subsequently presented to the network, it is treated as though it were an incomplete or error-ridden version of one of the stored examples. So the network regenerates the stored example that most closely resembles the presented vector. This can be thought of as a type of classification, where each of the examples in the training data belongs to a separate class, and each represents the ideal vector for that class. It is useful when classes can be characterized by an ideal or perfect example. For example, printed text that is subsequently scanned to form a digitized image will contain noisy and imperfect examples of printed characters. For a given font, an ideal version of each character can be stored in a content-addressable memory and produced as its output whenever an imperfect version is presented as its input.

6.3 Nodes and Networks

Each node, or neuron, in a neural network is a simple computing element having an input side and an output side. Each node may have directional connections to many other nodes at both its input and output sides. Each input x_i is multiplied by its associated weight w_i. Typically, the node's role is to sum each of its weighted inputs and add a bias term w_0 to form an intermediate quantity called the activation, a. It then passes the activation through a nonlinear function f_t known as the transfer function or activation function. Figure 13 shows the function of a single neuron.

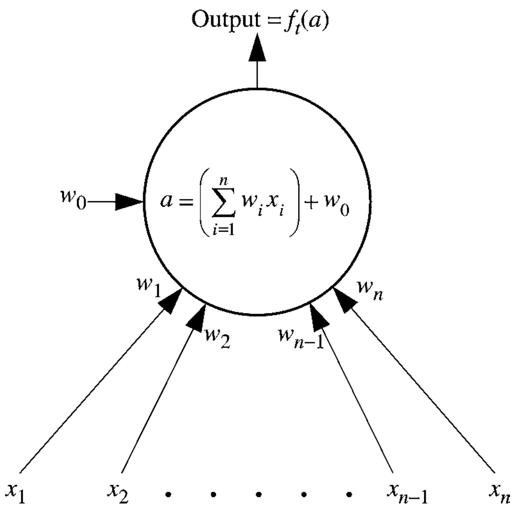


FIG. 13. A single neuron.

The behavior of a neural network depends on its topology, the weights, the bias terms, and the transfer function. The weights and biases can be learned, and the learning behavior of a network depends on the chosen training algorithm. Typically a sigmoid function is used as the transfer function, given by:

$$f_t(a) = \frac{1}{1 + e^{-a}}. \tag{7}$$

For a single neuron, the activation a is given by:

$$a = \left(\sum_{i=1}^{n} w_i x_i\right) + w_0 \tag{8}$$

where n is the number of inputs and the bias term w_0 is defined separately for each node.

Many network topologies are possible, but by far the most popular is the multi-layer perceptron (MLP), sometimes called a back-propagation network, which can be used for categorization or, more generally, for nonlinear mapping. However, many other architectures have been proposed, and refinements of them regularly appear in the literature. These include the Hopfield network [58,59], which is used as a content-addressable memory, and support-vector machines [60], which provide a means of reducing the number of training examples required for supervised learning. Popular network architectures for unsupervised clustering include the radial basis function network [61,62], Kohonen self-organizing network [63–66], and ART-2 network [67,68].

Another class of networks that are growing in importance are recurrent networks [69,70], where the input includes not only the current data, but also the output arising from previous data. In this way, recurrent networks can take account of the temporal relationships between the input data. This is particularly important for data whose interpretation is dependent on context, i.e., previous or future data. Such situations are commonplace, e.g., in trend-based diagnosis, the recognition of natural-language words, and forecasting.

6.4 Multilayer Perceptrons

6.4.1 Network Topology

The topology of a multilayer perceptron (MLP) is shown in Figure 14. The neurons are organized in layers, such that each neuron is totally connected to the neurons in the layers above and below, but not to the neurons in the same layer. These networks are also called feedforward networks, although this term could be applied more generally to any network where the direction of data flow is always "forwards," i.e., toward the output. MLPs can be used either for classification or as nonlinear estima-

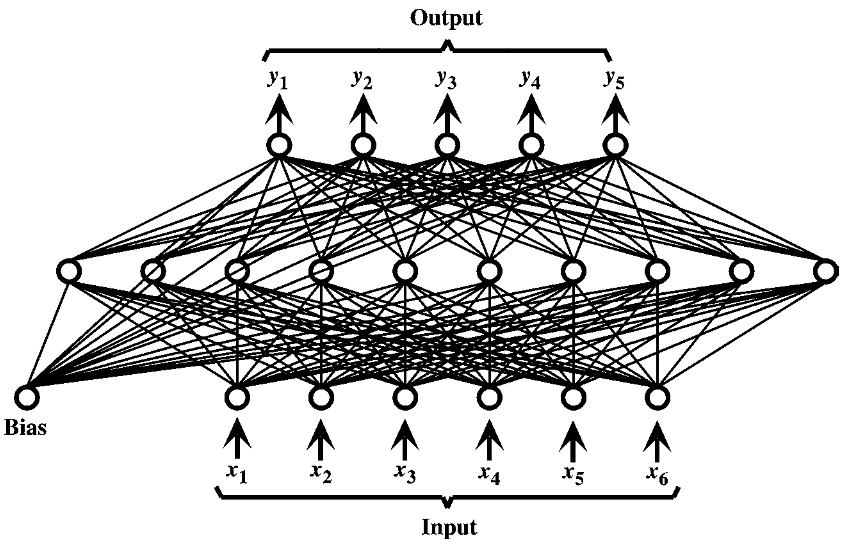


FIG. 14. A multi-layered perceptron.

tors. The number of nodes in each layer and the number of layers are determined by the network builder, often on a trial-and-error basis. There is always an input layer and an output layer; the number of nodes in each is determined by the size of the input and output vectors being considered. There may be any number of layers between these two layers. Unlike the input and output layers, the layers between often have no obvious interpretation associated with them, and they are known as hidden layers. The network shown in Figure 14 has six input nodes, one hidden layer with ten nodes, and an output layer of five nodes. It can, therefore, be described as a 6–10–5 MLP.

An MLP operates by feeding data forwards along the interconnections from the input layer, through the hidden layers, to the output layer. With the exception of the nodes in the input layer, the inputs to a node are the outputs from each node in the previous layer. At each node apart from those in the input layer, the data are weighted, summed, added to the bias, and then passed through the transfer function.

There is some inconsistency in the literature over the counting of layers, arising from the fact that the input nodes do not perform any processing, but simply feed the input data into the nodes above. Thus although the network in Figure 14 is clearly a three-layer network, it only has two processing layers. A so-called single-layered perceptron (SLP) has two layers (the input and output layers) but only one processing layer, namely the output layer. It has no hidden layers.

6.4.2 Perceptrons as Classifiers

In general, neural networks are designed so that there is one input node for each element of the input vector and one output node for each element of the output vector. Thus in a classification application, each output node would usually represent a particular class. A typical representation for a class would be for a value close to 1 to appear at the corresponding output node, with the remaining output nodes generating a value close to 0. A simple decision rule is needed in conjunction with the network, e.g., the *winner takes all* rule selects the class corresponding to the node with the highest output. If the input vector does not fall into any of the classes, none of the output values may be very high. For this reason, a more sophisticated decision rule might be used, e.g., one that specifies that the output from the winning node must also exceed a predetermined threshold such as 0.5.

Consider the practical example of an MLP for interpreting satellite images of the Earth in order to recognize different forms of land use. Figure 15 shows a region of the Mississippi Delta, imaged at six different wavebands. The MLP shown in Figure 14 was trained to associate the six waveband images with the corresponding land use. The pixels of the waveband images constitute the inputs and the five categories

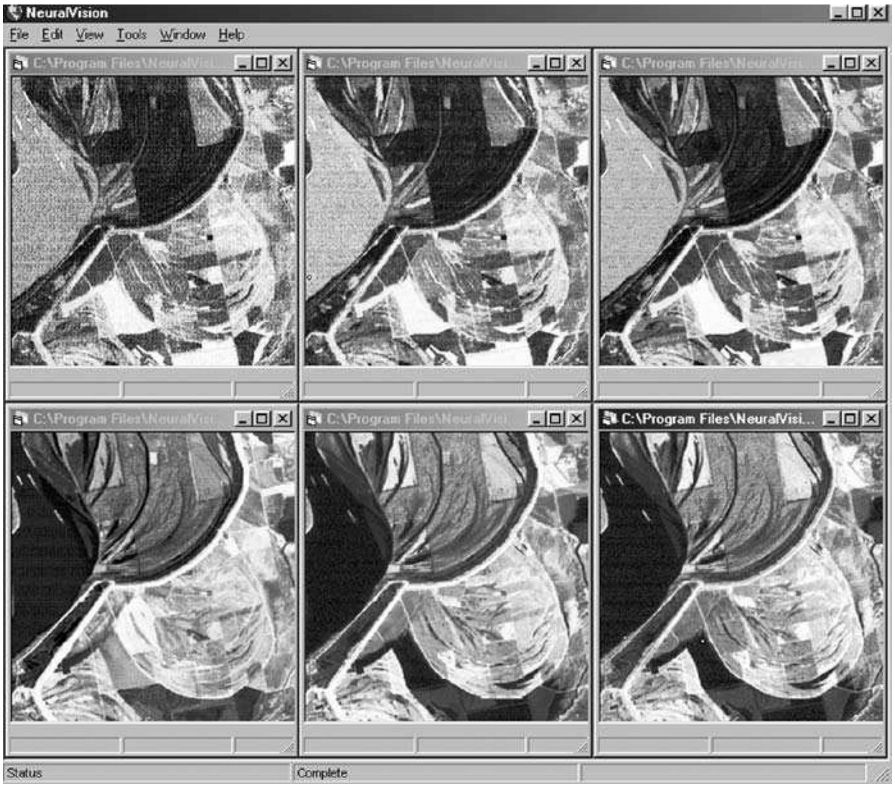

NeuralVision
File Edit View Tools Window Help
C:\Program Files\NeuralVisi...
Status
Complete

FIG. 15. Portion of a Landsat-4 TM satellite image of an area just to the south of Memphis, Tennessee, taken in six different wavebands. (Source: NASA, with permission.) (See Color Plate Section, Plate 2.)

of land use (water, trees, cultivated, soil/rock, swamp) constitute the outputs. The network was trained pixel-by-pixel on just the top 1/16 of these images and tested against the whole images, 15/16 of which were previously unseen. The results are shown in Figure 16. The classification is mostly correct, although some differences between the results and the actual land use can be seen. The neural network performance could certainly be improved with a little refinement, but it has deliberately been left unrefined so that these discrepancies can be seen. Nevertheless, the network's ability to generalize from a limited set of examples is clearly demonstrated.

Instead of having one output node per class, more compact representations are also possible. Hallam et al. [71] have used just two output nodes to represent four classes. This was achieved by treating both outputs together, so that the four possibilities corresponding to four classes are (0, 0), (0, 1), (1, 0), and (1, 1).

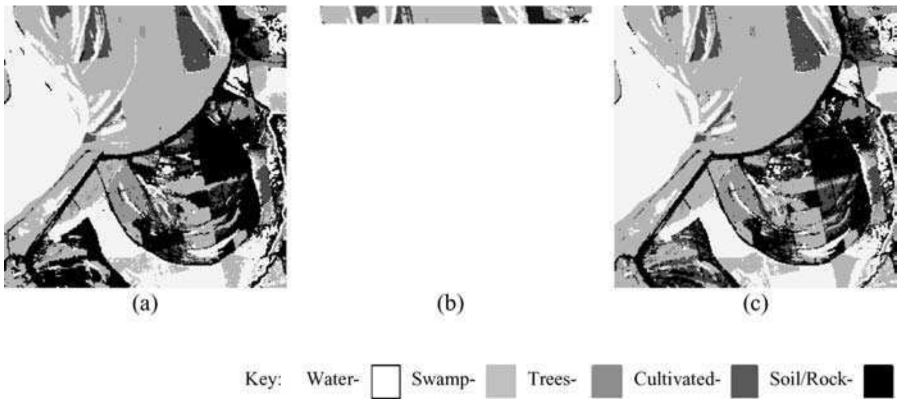


FIG. 16. (a) Actual land use map; (b) portion used for training; (c) Land use map from neural network outputs. (See Color Plate Section, Plate 3.)

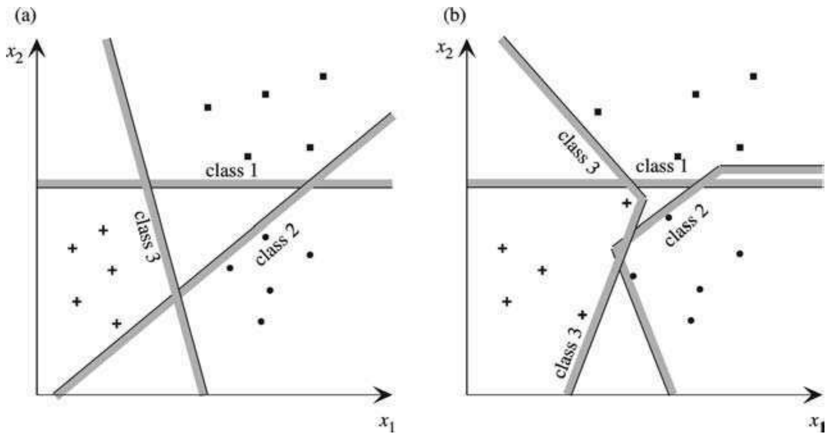


FIG. 17. Dividing up the pattern space: (a) linearly separable classes; (b) nonlinearly separable classes. Data points belonging to classes 1, 2, and 3 are respectively represented by ■, •, and +.

If the input vector has two elements, it can be represented as a point in two-dimensional state space, sometimes called the pattern space. The process of classification is then one of drawing dividing lines between regions. A single-layered perceptron, with two neurons in the input layer and the same number of neurons in the output layer as there are classes, can associate with each class a single straight dividing line, as shown in Figure 17(a). Classes that can be separated in this way are said to be linearly separable. More generally, n-dimensional input vectors are points in n-dimensional hyperspace. If the classes can be separated by $(n-1)$-dimensional hyperplanes, they are linearly separable.

To see how an SLP divides up the pattern space with hyperplanes, consider a single processing neuron of an SLP. Its output, prior to application of the transfer function, is a real number given by Equation (8). Regions of the pattern space that clearly belong to the class represented by the neuron will produce a strong positive value, and regions that clearly do not belong to the class will produce a strong negative value. The classification becomes increasingly uncertain as the activation a becomes close to zero, and the dividing criterion is usually assumed to be $a = 0$. This would correspond to an output of 0.5 after the application of the sigmoid transfer function. Thus the hyperplane that separates the two regions is given by:

$$\left(\sum_{i=1}^{n} w_i x_i\right) + w_0 = 0. \tag{9}$$

In the case of two inputs, Equation (9) becomes simply the equation of a straight line, since it can be rearranged as:

$$x_2 = \frac{-w_1}{w_2} x_1 - \frac{w_0}{w_2} \tag{10}$$

where $-w_1/w_2$ is the gradient and $-w_0/w_2$ is the intercept on the x_2 axis.

For problems that are not linearly separable, as in Figure 17(b), regions of arbitrary complexity can be drawn in the state space by a multilayer perceptron with one hidden layer and a differentiable transfer function such as the sigmoid function. The first processing layer of the MLP can be thought of as dividing up the state space with straight lines (or hyperplanes), and the second processing layer forms multifaceted regions by Boolean combinations (AND, OR, and NOT) of the linearly separated regions. It is therefore generally accepted that only one hidden layer is necessary to perform any nonlinear mapping or classification with an MLP that uses a sigmoid transfer function. This conclusion stems from *Kolmogorov's Existence Theorem* [72]. However, the ability to learn from a set of examples cannot be guaranteed and, therefore, the detailed topology of a network inevitably involves a certain amount of trial and error. A pragmatic approach to network design is to start with a small network and expand the number of nodes or layers as necessary.

6.4.3 Training a Perceptron

During training, a multilayer perceptron learns to separate the regions in state space by adjusting its weights and bias terms. Appropriate values are learned from a set of examples comprising input vectors and their corresponding desired output vectors. An input vector is applied to the input layer, and the output vector produced at the output layer is compared with the desired output. For each neuron in the output layer, the difference between the generated value and the desired value is the *error*. The overall error for the neural network is expressed as the square root of the mean of the squares of the errors. This is the root-mean-squared (RMS) value, designed to take equal account of both negative and positive errors. The RMS error is minimized by altering the weights and bias terms, which may take many passes through the training data. The search for the combination of weights and biases that produces the minimum RMS error is an optimization problem. When the RMS error has become acceptably low for each example vector, the network is said to have *converged* and the weights and bias terms are retained for application of the network to new input data.

One of the most commonly used training algorithms is the back-error propagation algorithm, sometimes called the generalized delta rule [73,74]. It relies upon the transfer function being continuous and differentiable. The sigmoid function is a particularly suitable choice since its derivative is simply given by:

$$f_t'(a) = f_t(a)\bigl(1 - f_t(a)\bigr). \tag{11}$$

At the core of the algorithm is the delta rule that determines the modifications to the weights, Δw_{Bij}:

$$\Delta w_{Bij} = \eta\delta_{Bi}y_{Aj} + \alpha(\Delta w_{Bij}) \tag{12}$$

for all nodes j in layer A and all nodes i in layer B, where $A = B - 1$. Neurons in the output layer and in the hidden layers have an associated error term, δ. When the sigmoid transfer function is used, δ_{Ai} for the output layer is given by:

$$\delta_{Ai} = f_t'(y_{Ai})(d_i - y_{Ai}) = y_{Ai}(1 - y_{Ai})(d_i - y_{Ai}) \tag{13}$$

while δ_{Aj} for the hidden layers is given by:

$$\delta_{Aj} = f_t'(y_{Aj})\sum_i \delta_{Bi}w_{Bij} = y_{Aj}(1 - y_{Aj})\sum_i \delta_{Bi}w_{Bij}. \tag{14}$$

The learning rate, η, is applied to the calculated values for δ_{Aj}. Knight [75] suggests a value for η of about 0.35. As written in Equation (12), the delta rule includes a momentum coefficient, α, although this term is sometimes omitted, i.e., α is sometimes set to zero. Gradient-proportional descent techniques can be inefficient, especially close to a minimum in the cost function, which in this case is the RMS error of the output. To address this, a momentum term forces changes in weight to be dependent on previous weight changes. The value of the momentum coefficient must be in the range 0–1. Knight [75] suggests that α be set to 0.0 for the first few training passes and then increased to 0.9.

6.4.4 Some Practical Considerations

Sometimes it is appropriate to stop the training process before the point where no further reductions in the RMS error are possible. This is because it is possible to *over-train* the network, so that it becomes expert at giving the correct output for the training data, but less expert at dealing with new data. This is likely to be a problem if the network has been trained for too many cycles or if the network is over-complex for the task in hand. For instance, the inclusion of additional hidden layers or large numbers of neurons within the hidden layers will tend to promote over-training. The effect of over-training is shown in Figure 18(a) for a nonlinear mapping of a single input parameter onto a single output parameter, and Figure 18(b) shows the effect of over-training using the nonlinearly separable classification data from Figure 17(b).

One way of avoiding over-training is to divide the data into three sets, known as the training, testing, and validation data. Training takes place using the training data, and the RMS error with these data is monitored. However, at predetermined intervals the training is paused and the current weights saved. At these points, before training resumes, the network is presented with the test data and an RMS error calculated. The RMS error for the training data decreases steadily until it stabilizes. However, the RMS error for the test data may pass through a minimum and then start to increase again because of the effect of over-training, as shown in Figure 19. As soon as the RMS error for the test data starts to increase, the network is over-trained, but the previously stored set of weights would be close to the optimum. Finally, the performance of the network can be evaluated by testing it using the previously unseen validation data.

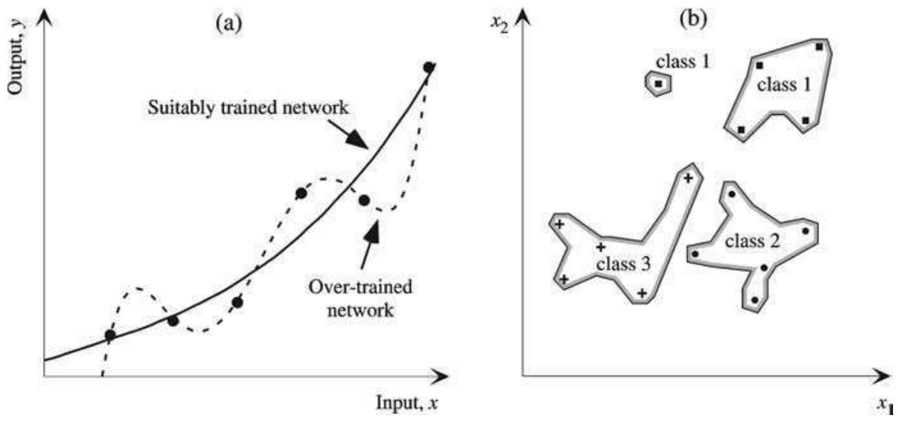


FIG. 18. The effect of over-training: (a) nonlinear estimation; (b) classification (●, ■, and + are data points used for training).

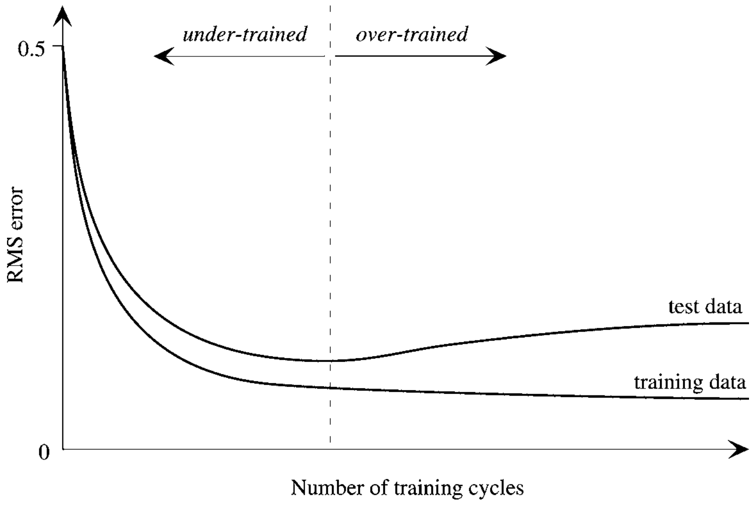


FIG. 19. RMS error during training.

A problem that is frequently encountered in real applications is a shortage of suitable data for training and testing a neural network. In one classification problem [30], there were only 20 suitable examples, which needed to be shared between the training and testing data. The authors used a technique called *leave-one-out* as a way of reducing the effect of this problem. The technique involves repeatedly training on all but one of the examples and testing on the missing one. So, in this case, the network would initially be trained on 19 of the examples and tested on the remaining one. This procedure is repeated a further 19 times: omitting a different example each time from the training data, resetting the weights to random values, retraining, and then testing on the omitted example. The leave-one-out technique is clearly time-consuming as it involves resetting the weights, training, testing, and scoring the network many times—20 times in this example. Its advantage is that the performance of the network can be evaluated using every available example as though it were previously unseen test data.

Neural networks that accept real numbers are only effective if the input values are constrained to suitable ranges, typically between 0 and 1 or between -1 and 1. The range of the outputs depends on the chosen transfer function, e.g., the output range is between 0 and 1 if the sigmoid function is used. In real applications, the actual input and output values may fall outside these ranges or may be constrained to a narrow band within them. In either case the data will need to be scaled before being presented to the neural network.

7. Hybrid Systems

7.1 Convergence of Techniques

One of the aims of this chapter is to demonstrate that there are a wide variety of computing techniques that can be applied to particular problems. These include conventional programs as well as symbolic representations, such as knowledge-based systems, and computational intelligence methods. In many cases the techniques need not be exclusive alternatives to each other but can be seen as complementary tools that can be brought together within a hybrid system. There are several ways in which different computational techniques can be complementary: dealing with multifaceted problems, parameter setting, capability enhancement, and clarification and verification.

7.2 Blackboard Systems for Multifaceted Problems

Most real-life problems are complex and have many facets, where each facet or sub-task may be best suited to a different technique. Therefore, many practical systems are designed as hybrids, incorporating several specialized modules, each of which uses the most suitable tools for its specific task. The blackboard system, shown in Figure 20, provides a software structure that is well-suited to multifaceted tasks. The first published application of a blackboard system was Hearsay-II for speech understanding in 1975 [76,77]. Further developments took place during the 1980s [78] and the blackboard model is now seen as a key technology for the burgeoning area of multiagent systems [79].

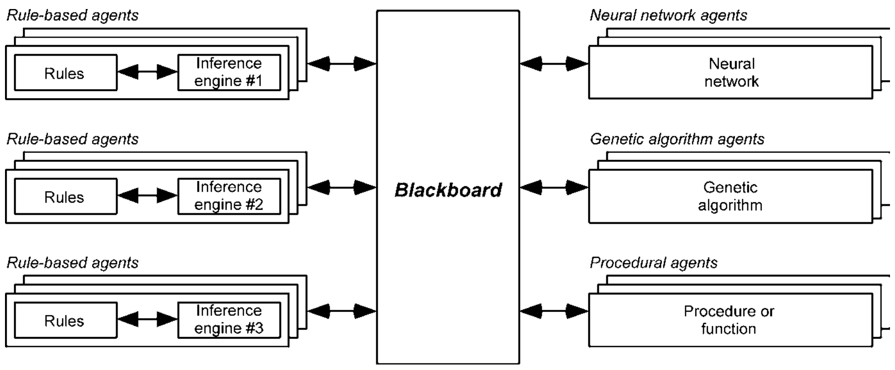


FIG. 20. A blackboard system.

In a blackboard system, knowledge of the application domain is divided into modules. These modules were formerly referred to as knowledge sources but, as they are independent and autonomous, they are now commonly regarded as agents. Each agent is designed to tackle a particular subtask. Agents can communicate only by reading from or writing to the blackboard, a globally accessible working memory where the current state of understanding is represented. Agents can also delete unwanted information from the blackboard.

A blackboard system is analogous to a team of experts who communicate their ideas via a physical blackboard, by adding or deleting items in response to the information that they find there. Each agent represents such an expert having a specialized area of knowledge. As each agent can be encoded in the most suitable form for its particular task, blackboard systems offer a mechanism for the collaborative use of different computational techniques such as rules, neural networks, genetic algorithms, and fuzzy logic. Each rule-based agent can use a suitable reasoning strategy for its particular task, e.g., backward- or forward-chaining, and can be thought of as a rule-based system in microcosm.

In the late 1980s, the Algorithmic and Rule-Based Blackboard System (ARBS) was developed and subsequently applied to diverse problems including the interpretation of ultrasonic images [30], the management of telecommunications networks [10], and the control of plasma deposition processes [80]. More recently, ARBS was redesigned as a distributed system, DARBS, in which the software modules run in parallel, possibly on separate computers connected via the Internet [81].

Agents are applied in response to information on the blackboard, when they have some contribution to make. This leads to increased efficiency since the detailed knowledge within an agent is only applied when that agent becomes relevant. The agents are said to be opportunistic, activating themselves whenever they can contribute to the global solution. In early, non-distributed backboard systems, true opportunism was difficult to achieve as it required explicit scheduling and could involve interrupting an agent that was currently active. In modern blackboard systems, each agent is its own process, either in parallel on multiple processors or concurrently on a single processor, so no explicit scheduling is required.

In the interests of efficiency and clarity, some degree of structure is usually imposed on the blackboard by dividing it into partitions. An agent then only needs to look at a partition rather than the whole blackboard. Typically the blackboard partitions correspond to different levels of analysis of the problem, progressing from detailed information to more abstract concepts. In the Hearsay-II blackboard system for computerized understanding of natural speech, the levels of analysis include those of syllable, word, and phrase [77]. In ultrasonic image interpretation using ARBS, the levels progress from raw signal data, via a description of the significant image features, to a description of the defects in the component [30].

The key advantages of the blackboard architecture, adapted from Feigenbaum [82] can be summarized as follows:

(i) Many and varied sources of knowledge can participate in the development of a solution to a problem.

(ii) Since each agent has access to the blackboard, each can be applied as soon as it becomes appropriate. This is opportunism, i.e., application of the right knowledge at the right time.

(iii) For many types of problem, especially those involving large amounts of numerical processing, the characteristic style of incremental solution development is particularly efficient.

(iv) Different types of reasoning strategy (e.g., data- and goal-driven) can be mixed as appropriate in order to reach a solution.

(v) Hypotheses can be posted onto the blackboard for testing by other agents. A complete test solution does not have to be built before deciding to modify or abandon the underlying hypothesis.

(vi) In the event that the system is unable to arrive at a complete solution to a problem, the partial solutions appearing on the blackboard are available and may be of some value.

7.3 Parameter Setting

Designing a suitable computational intelligence solution for a given application can involve a large amount of trial and error. Although the “knowledge acquisition bottleneck” associated with knowledge-based systems is avoided, a “parameter-setting bottleneck” may be introduced instead. The techniques described here are intended to avoid this bottleneck by using one computational intelligence technique to set the parameters of another.

7.3.1 Genetic-Neural Systems

Neural networks can suffer from the parameter-setting bottleneck as the developer struggles to configure a network for a particular problem. Whether a network will converge, i.e., learn suitable weightings, will depend on the topology, the transfer function of the nodes, the values of the parameters in the training algorithm, and the training data. It may even depend on the order in which the training data are presented. Although Kolmogorov’s Existence Theorem leads to the conclusion that a three-layered perceptron, with the sigmoid transfer function, can perform any mapping from a set of inputs to the desired outputs, the theorem tells us nothing about the learning parameters, the necessary number of neurons, or whether additional layers would be beneficial. It is, however, possible to use a genetic algorithm to optimize the network configuration [83]. A suitable cost function might combine the RMS error with duration of training.

Supervised training of a neural network involves adjusting its weights until the output patterns obtained for a range of input patterns are as close as possible to the desired patterns. The different network topologies use different training algorithms for achieving this weight adjustment, typically through back-propagation of errors. Just as it is possible for a genetic algorithm to configure a network, it is also possible to use a genetic algorithm to train the network [83]. This can be achieved by letting each gene represent a network weight, so that a complete set of network weights is mapped onto an individual chromosome. Each chromosome can be evaluated by testing a neural network with the corresponding weights against a series of test patterns. A fitness value can be assigned according to the error, so that the weights represented by the fittest generated individual correspond to a trained neural network.

7.3.2 Genetic-Fuzzy Systems

The performance of a fuzzy system depends on the definitions of the fuzzy sets and on the fuzzy rules. As these parameters can all be expressed numerically, it is possible to devise a system whereby they are learned automatically using genetic algorithms. A chromosome can be devised that represents the complete set of parameters for a given fuzzy system. The cost function could then be defined as the total error when the fuzzy system is presented with a number of different inputs with known desired outputs.

Often, a set of fuzzy rules for a given problem can be drawn up fairly easily, but defining the most suitable membership functions remains a difficult task. Karr [84,85] has performed a series of experiments to demonstrate the viability of using genetic algorithms to specify the membership functions. In Karr's scheme, all membership functions are triangular. The variables are constrained to lie within a fixed range, so the fuzzy sets low and high are both right-angle triangles (Figure 21). The

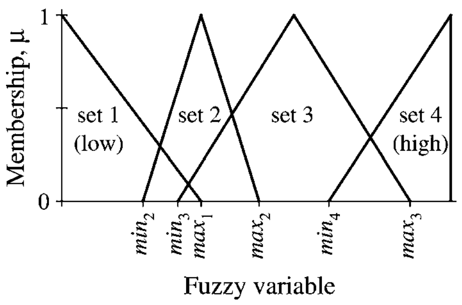


FIG. 21. Defining triangular membership functions by their intercepts on the abscissa.

slope of these triangles can be altered by moving their intercepts on the abscissa, marked max_1 and min_4 in Figure 21. All intermediate fuzzy sets are assumed to have membership functions that are isosceles triangles. Each is defined by two points, max_i and min_i, where i labels the fuzzy set. The chromosome is then a list of all the points max_i and min_i that determine the complete set of membership functions. In several demonstrator systems, Karr's GA-modified fuzzy controller outperformed a fuzzy controller whose membership functions had been set manually.

7.4 Capability Enhancement

One technique may be used within another to enhance the latter's capabilities. Here, three examples are described: neuro-fuzzy systems which combine the benefits of neural networks and fuzzy logic; Lamarckian and Baldwinian inheritance for enhancing the performance of a genetic algorithm with local search around individuals in the population; and learning classifier systems that use genetic algorithms to discover rules.

7.4.1 Neuro-Fuzzy Systems

Section 7.3.2 above shows how a genetic algorithm can be used to optimize the parameters of a fuzzy system. In such a scheme, the genetic algorithm for parameter setting and the fuzzy system that uses those parameters are distinct and separate. The parameters for a fuzzy system can also be learned using neural networks, but here much closer integration is possible between the neural network and the fuzzy system that it represents. A neuro-fuzzy system is a fuzzy system, the parameters of which are derived by a neural network learning technique. It can equally be viewed as a neural network that represents a fuzzy system. The two views are equivalent and it is possible to express a neuro-fuzzy system in either form.

Consider the following fuzzy rules, in Flex format:

```
fuzzy_rule r4f
  if the temperature is high or the water level is high
  then the pressure is high.
fuzzy_rule r5f
  if the temperature is medium or the water level is medium
  then the pressure is medium.
fuzzy_rule r6f
  if the temperature is low or the water level is low
  then the pressure is low.
```

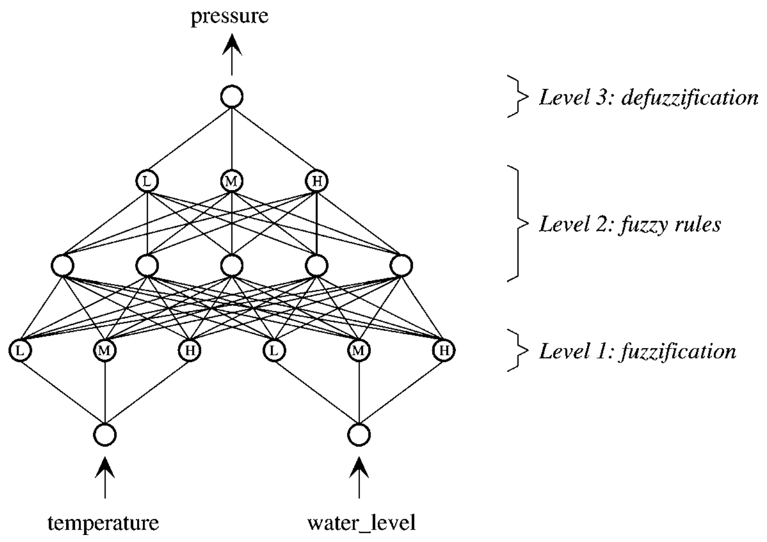


FIG. 22. A neuro-fuzzy network.

These fuzzy rules and the corresponding membership functions can be represented by the neural network shown in Figure 22. The first stage is fuzzification, in which any given input value for `temperature` is given a membership value for `low`, `medium`, and `high`. A single layer perceptron, designated level 1 in Figure 22, can achieve this because it is a linear classification task. The only difference from other classifications met previously is that the target output values are not just 0 and 1, but any value in-between. A similar network is required at level 1 for the other input variable, `water level`. The neurons whose outputs correspond to the `low`, `medium`, and `high` memberships are marked L, M, and H, respectively, in Figure 22.

Level 2 of the neuro-fuzzy network performs the role of the fuzzy rules, taking the six membership values as its inputs and generating as its outputs the memberships for `low`, `medium`, and `high` of the fuzzy variable `pressure`. The final stage, at level 3, involves combining these membership values to produce a defuzzified value for the output variable.

The definitions of the fuzzy sets and the fuzzy rules are implicit in the connections and weights of the neuro-fuzzy network. Using a suitable learning mechanism, the weights can be learned from a series of examples. The network can then be used on previously unseen inputs to generate defuzzified output values. In principle, the fuzzy sets and rules can be inferred from the network and run as a fuzzy rule-based system to produce identical results [86,87].

7.4.2 Lamarckian and Baldwinian Inheritance

Capability enhancement of a genetic algorithm can be achieved by hybridizing it with local search procedures. This can help the genetic algorithm to optimize individuals within the population, while maintaining its ability to explore the search space. The aim would be either to increase the quality of the solutions, so they are closer to the global optimum, or to increase the efficiency, i.e., the speed at which the optimum is found. One approach is to introduce an extra step involving local search around each chromosome in the population, to see whether any of its neighbors offers a fitter solution.

A commonly used, if rather artificial, example is to suppose that the task is to find the maximum denary integer represented by a 7-bit binary-encoded chromosome. The optimum is clearly 1111111. A chromosome's fitness could be taken as the highest integer represented either by itself or by any of its near neighbors found by local search. A near neighbor of an individual might typically be defined as one that has a Hamming separation of 1 from it, i.e., one bit is different. For example, the chromosome 0101100 (denary value 44) would have a fitness of 108, since it has a nearest neighbor 1101100 (denary value 108). The chromosome could be either:

(a) left unchanged while retaining the fitness value of its fittest neighbor—this is Baldwinian inheritance; or
(b) replaced by the higher scoring neighbor, in this case 1101100—this is Lamarckian inheritance.

Baldwinian inheritance is analogous to Baldwin's discredited proposal that, in nature, offspring can inherit their parents' learned characteristics. Lamarckian inheritance is analogous to Lamarck's generally accepted proposal that, in nature, offspring can inherit their parents' capacity for learning. Although only one of the two types of inheritance is considered credible biologically, both are useful in genetic algorithms.

El Mihoub et al. [88] have set out to establish the relative benefits of the two forms of inheritance. In a series of experiments, they have adjusted the relative proportions of Lamarckian and Baldwinian inheritance, and the probability of applying a local search to any individual in the population. Their previously unpublished results for one fitness landscape—a four-dimensional Shwefel function [89]—are summarized in Figure 23. They have shown that the optimal balance depends on the population size, the probability of local search and the nature of the fitness landscape. With this in mind, they have also experimented with making these parameters self-adaptive by coding them within the chromosomes. Taking this idea to its logical conclusion,

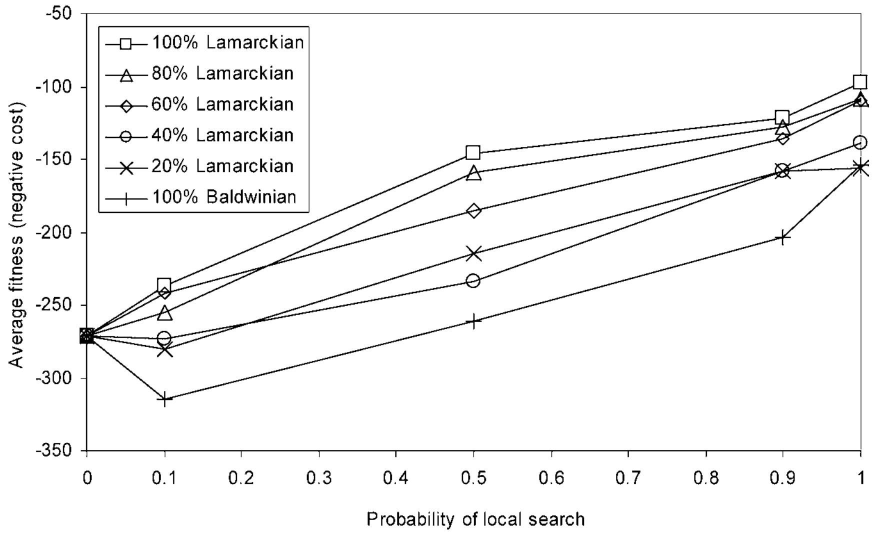


FIG. 23. Effects of probability of local search and relative proportions of Lamarckian and Baldwinian inheritance for a four-dimensional Schwefel test function. (Data courtesy of Tarek El-Mihoub.)

a completely self-adapting genetic algorithm could be envisaged where no parameters at all need to be specified by the user.

7.4.3 Learning Classifier Systems

Holland's learning classifier systems (LCSs) combine genetic algorithms with rule-based systems to provide a mechanism for rule discovery [90,91]. The rules are simple production rules, coded as a fixed-length mixture of binary numbers and wild-card, i.e., "don't care," characters. Their simple structure makes it possible to generate new rules by means of a genetic algorithm.

The overall LCS is illustrated in Figure 24. At the heart of the system is the message list, which fulfils a similar role to the blackboard in a blackboard system. Information from the environment is placed here, along with rule deductions and instructions for the actuators, which act on the environment.

A credit-apportionment system, known as the bucket-brigade algorithm, is used to maintain a credit balance for each rule. The genetic algorithm uses a rule's credit balance as the measure of its fitness. Conflict resolution between rules in the conflict set is achieved via an auction, in which the rule with the most credits is chosen to fire. In doing so, it must pay some of its credits to the rules that led to its conditions being satisfied. If the fired rule leads to some benefit in the environment, it receives additional credits.

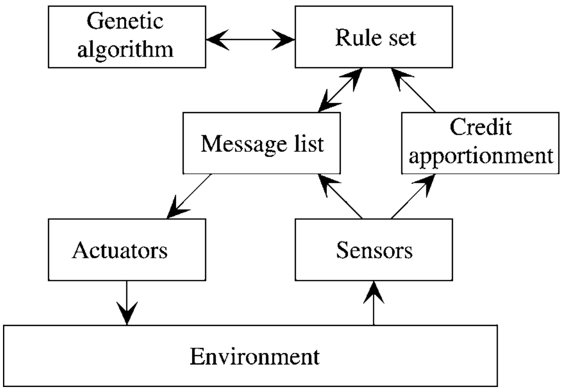


FIG. 24. Learning classifier system.

7.5 Clarification and Verification of Neural Network Outputs

Neural networks have the ability to learn associations between input vectors and their associated outputs. However, the underlying reasons for the associations may be opaque, as they are effectively encoded in the weightings on the interconnections between the neurons. Thus, neural networks are often regarded as “black boxes” that simply generate an output from a given input, but whose internal state conveys no readily useful information to an observer. This contrasts with a transparent system, such as a KBS, where the internal state, e.g., the value of a variable, does have meaning for an observer.

There has been a considerable research effort into rule extraction to automatically produce intelligible rules that are equivalent to the trained neural network from which they have been extracted [92]. A variety of methods have been reported for extracting different types of rules, including production rules and fuzzy rules.

In safety-critical systems, reliance on the output from a neural network without any means of verification is not acceptable. It has, therefore, been proposed that rules be used to verify that the neural network output is consistent with its input [93]. The use of rules for verification implies that at least some of the domain knowledge can be expressed in rule form. Johnson et al. [94] suggest that an adjudicator module be used to decide whether a set of rules or a neural network is likely to provide the more reliable output for a given input. The adjudicator would have access to information relating to the extent of the neural network’s training data and could determine whether a neural network would have to interpolate between, or extrapolate from, examples in the training set. (Neural networks are good at interpolation but poor at extrapolation.) The adjudicator may, for example, call upon rules to handle the exceptional cases which would otherwise require a neural network to extrapolate from its training data. If heuristic rules are also available for the less exceptional cases, then they could be used to provide an explanation for the neural network's findings. A supervisory rule-based module could dictate the training of a neural network, deciding how many nodes are required, adjusting the learning rate as training proceeds, and deciding when training should terminate.

8. Conclusions

8.1 Benefits

This chapter has reviewed a range of AI techniques. Whether the resultant systems display true intelligence remains questionable. Nevertheless, the following practical benefits have stemmed from the development of AI techniques:

Reliability and Consistency. An AI system makes decisions that are consistent with its input data and its knowledge base (for a knowledge-based system) or numerical parameters (for a computational intelligence technique). It may, therefore, be more reliable than a person, particularly where repetitive mundane judgments have to be made.

Automation. In many applications, such as visual inspection on a production line, judgmental decision-making has to be performed repeatedly. A well-designed AI system ought to be able to deal with the majority of such cases, while highlighting any that lie beyond the scope of its capabilities. Therefore, only the most difficult cases, which are normally the most interesting, are deferred to a person.

Speed. AI systems are designed to automatically make decisions that would otherwise require human reasoning, judgment, expertise, or common sense. Any lack of true intelligence is compensated by the system's processing speed. An AI system can make decisions informed by a wealth of data and information that a person would have insufficient time to assimilate.

Improved Domain Understanding. The process of constructing a knowledge-based system requires the decision-making criteria to be clearly identified and assessed. This process frequently leads to a better understanding of the problem being tackled. Similar benefits can be obtained by investigating the decision-making criteria used by the computational intelligence techniques.

Knowledge Archiving. The knowledge base is a repository for the knowledge of one or more people. When these people move on to new jobs, some of their expert knowledge is saved in the knowledge base, which continues to evolve after their departure.

New Approaches to Software Engineering. Since AI systems are supposed to be flexible and adaptable, development is usually based upon continuous refinements of an initial prototype. This is the *prototype–test–refine* cycle, which applies to both knowledge-based systems and computational intelligence techniques. The key stages in the development of a system are:

- decide the requirements;
- design and implement a prototype;
- continuously test and refine the prototype.

Rapid prototyping and iterative development have gained respectability across most areas of software engineering in recent years, replacing the traditional linear "waterfall process" of meticulous specification, analysis, and design phases prior to implementation and testing.

8.2 Networked AI

The widespread availability of the Internet is helping to shape the development of modern AI. This chapter has highlighted multiagent systems, and the blackboard architecture for dividing problems into subtasks that can be shared among specialized agents so that the right software tool can be used for each job. The growth in the use of the Internet is likely to see AI become increasingly distributed as agents reside on separate computers, and mobile agents travel over the net in search of information.

Paradoxically, there is also a sense in which the Internet is making AI more integrated. Watson and Gardingen describe a sales support application that has become integrated by use of the World Wide Web, as a single definitive copy of the software accessible via the web has replaced distributed copies [95]. In commercial decision-making, separate distributed functions are becoming integrated by the need for communication between them. For example, production decisions need to take into account and influence design, marketing, personnel, sales, materials stocks, and product stocks.

8.3 Ubiquitous AI

Early AI systems were mostly consultative in nature, e.g., diagnostic expert systems that reached a conclusion following dialogue with a human, or neural networks

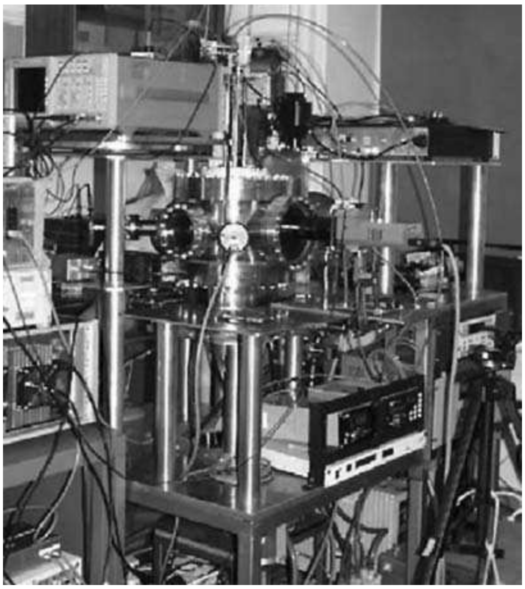


FIG. 25. Interaction with plasma deposition equipment through sensors and actuators. (Photo by Lars Nolle.) (See Color Plate Section, Plate 4.)

that produced classifications from data stored in a file. Many of the more modern AI systems reviewed in this chapter are *situated*—i.e., they interact with their environment through sensors that detect the environment and actuators that operate upon it in real time. This is demonstrated by a recent application of DARBS, which involves monitoring and controlling plasma deposition equipment used in semiconductor device manufacture [96]. As Figure 25 graphically illustrates, the computer system in this application is indeed situated within its environment with a wealth of devices for interacting with the environment.

If AI is to become more widely situated into everyday environments, it needs to become smaller, cheaper, and more reliable. The next key stage in the development of AI is likely to be a move towards *embedded* AI, i.e., AI systems that are embedded in machines, devices, and appliances. The work of Choy et al. [97] is significant in this respect, in that they have demonstrated that the DARBS blackboard system can be ported to a compact platform of parallel low-cost processors.

Increasingly, we are likely to see intelligent agents that serve as personal consultants to advise and inform us, and others that function silently and anonymously while performing tasks such as data interpretation, monitoring, and control. We are therefore relatively close to the possibility of intelligent behavior, based on the techniques reviewed in this chapter, in everyday domestic, workplace, and public environments. Examples that have already appeared include washing machines that incorporate knowledge-based control systems, elevators that use fuzzy logic to decide at which floor to wait for the next passenger, robotic vacuum cleaners that incorporate the BDI model of intelligent agents, and personal organizers that use neural networks to learn the characteristics of their owner's handwriting. It is surely only a matter of time before AI becomes truly ubiquitous.

8.4 Has AI Delivered?

The introduction to this chapter proposed a simple definition of artificial intelligence, i.e., the science of mimicking human mental faculties in a computer. The subsequent sections have reviewed a wide range of techniques. Some approaches are pre-specified and structured, while others specify only low-level behavior, leaving the intelligence to emerge through complex interactions. Some approaches are based on the use of knowledge expressed in words and symbols, whereas others use only mathematical and numerical constructions.

Overall, the tools and techniques of AI are ingenious, practical, and useful. If these were the criteria by which the success of AI were measured, it would be heralded as one of the most accomplished technological fields. However, human mental faculties are incredibly complex and have proved to be extremely difficult to mimic.

Nevertheless, the techniques presented here have undoubtedly advanced humankind's progress towards the construction of an intelligent machine. AI research has made significant advances from both ends of the intelligence spectrum shown in Figure 1, but a gap still exists in the middle. Building a system that can make sensible decisions about unfamiliar situations in everyday, non-specialist domains remains difficult. This development requires progress in simulating behaviors that humans take for granted—specifically perception, language, common sense, and adaptability.

Acknowledgements

This chapter is based in part on my book *Intelligent Systems for Engineers and Scientists* [11], from which several of the figures are taken. Some parts are also derived from an article in *IEEE Computer* [3]. The demonstration of a neural network for interpreting satellite images is based on a project originally devised in collaboration with Tony Hirst and Hywel Evans, using data supplied by Paul Mather. I am indebted to my research students, past and present, for several of the developments presented here.

REFERENCES

[1] Penrose R., *The Emperor's New Mind*, Oxford University Press, London, 1989.
[2] Turing A.M., "Computing machinery and intelligence", *Mind* **59** (1950) 433–460.
[3] Hopgood A.A., "Artificial intelligence: hype or reality?" *IEEE Computer* **6** (2003) 24–28.
[4] Holmes N., "Artificial intelligence: arrogance or ignorance?" *IEEE Computer* **36** (2003) 118–120.
[5] http://asimo.honda.com.
[6] http://www.sony.net/SonyInfo/QRIO.
[7] Buchanan B.G., Sutherland G.L., Feigenbaum E.A., "Heuristic DENDRAL: a program for generating explanatory hypotheses in organic chemistry", *Machine Intelligence* **4** (1969) 209–254.
[8] Shortliffe E.H., *Computer-Based Medical Consultations: MYCIN*, Elsevier, Amsterdam, 1976.
[9] http://www.lpa.co.uk.
[10] Hopgood A.A., "Rule-based control of a telecommunications network using the blackboard model", *Artificial Intelligence in Engineering* **9** (1994) 29–38.
[11] Hopgood A.A., *Intelligent Systems for Engineers and Scientists*, second ed., CRC Press, Boca Raton, FL, 2001.
[12] Forgy C.L., "Rete: a fast algorithm for the many-pattern/many-object-pattern match problem", *Artificial Intelligence* **19** (1982) 17–37.
[13] Fulton S.L., Pepe C.O., "An introduction to model-based reasoning", *AI Expert* (January 1990) 48–55.
[14] Fenton W.G., Mcginnity T.M., Maguire L.P., "Fault diagnosis of electronic systems using intelligent techniques: a review", *IEEE Transactions on Systems Man and Cybernetics Part C—Applications and Reviews* **31** (2001) 269–281.
[15] Wotawa F., "Debugging VHDL designs using model-based reasoning", *Artificial Intelligence in Engineering* **14** (2000) 331–351.
[16] Mateis C., Stumptner M., Wotawa F., "Locating bugs in Java programs—first results of the Java diagnosis experiments project", in: *Lecture Notes in Artificial Intelligence*, vol. 1821, Springer-Verlag, Berlin/New York, 2000, pp. 174–183.
[17] Montani S., Magni P., Bellazzi R., Larizza C., Roudsari A.V., Carson E.R., "Integrating model-based decision support in a multi-modal reasoning system for managing type 1 diabetic patients", *Artificial Intelligence in Medicine* **29** (2003) 131–151.
[18] Bruninghaus S., Ashley K.D., *Combining case-based and model-based reasoning for predicting the outcome of legal cases*, in: *Lecture Notes in Artificial Intelligence*, vol. 2689, Springer-Verlag, Berlin/New York, 2003, pp. 65–79.
[19] De Koning K., Bredeweg B., Breuker J., Wielinga B., "Model-based reasoning about learner behaviour", *Artificial Intelligence* **117** (2000) 173–229.
[20] Fink P.K., Lusth J.C., "Expert systems and diagnostic expertise in the mechanical and electrical domains", *IEEE Transactions on Systems, Man, and Cybernetics* **17** (1987) 340–349.

[21] Xing H., Huang S.H., Shi J., "Rapid development of knowledge-based systems via integrated knowledge acquisition", *Artificial Intelligence for Engineering Design Analysis and Manufacturing* **17** (2003) 221–234.

[22] Minton S., Carbonell J.G., Knoblock C.A., Kuokka D.R., Etzioni O., Gil Y., "Explanation-based learning: a problem-solving perspective", *Artificial Intelligence* **40** (1989) 63–118.

[23] Riesbeck C.K., Schank R.C., *Inside Case-based Reasoning*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[24] Aamodt A., Plaza E., "Case-based reasoning—foundational issues, methodological variations, and system approaches", *AI Communications* **7** (1994) 39–59.

[25] Shortliffe E.H., Buchanan B.G., "A model of inexact reasoning in medicine", *Mathematical Biosciences* **23** (1975) 351–379.

[26] Buchanan B.G., Shortliffe E.H. (Eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison–Wesley, Reading, MA, 1984.

[27] Barnett J.A., "Computational methods for a mathematical theory of evidence", in: *7th International Joint Conference on Artificial Intelligence (IJCAI'81), Vancouver*, 1981, pp. 868–875.

[28] Schafer G., *A Mathematical Theory of Evidence*, Princeton University Press, Princeton, NJ, 1976.

[29] Quinlan J.R., "Inferno: a cautious approach to uncertain inference", *The Computer Journal* **26** (1983) 255–269.

[30] Hopgood A.A., Woodcock N., Hallam N.J., Picton P.D., "Interpreting ultrasonic images using rules, algorithms and neural networks", *European Journal of Nondestructive Testing* **2** (1993) 135–149.

[31] Li H.X., Yen V.C., *Fuzzy Sets and Fuzzy Decision-Making*, CRC Press, Boca Raton, FL, 1995.

[32] Wooldridge M.J., "Agent-based software engineering", *IEE Proc. Software Engineering* **144** (1997) 26–37.

[33] Ulrich I., Mondada F., Nicoud J.D., "Autonomous vacuum cleaner", *Robotics and Autonomous Systems* **19** (1997) 233–245.

[34] Russell S., Norvig P., *Artificial Intelligence: A Modern Approach*, Prentice Hall, New York, 1995.

[35] Wooldridge M.J., Jennings N.R., "Intelligent agents: theory and practice", *Knowledge Engineering Review* **10** (1995) 115–152.

[36] Brooks R.A., "Intelligence without reason", in: *12th International Joint Conference on Artificial Intelligence (IJCAI'91), Sydney*, 1991, pp. 569–595.

[37] Newell A., "The knowledge level", *Artificial Intelligence* **18** (1982) 87–127.

[38] Bratman M.E., Israel D.J., Pollack M.E., "Plans and resource-bounded practical reasoning", *Computational Intelligence* **4** (1988) 349–355.

[39] Wooldridge M.J., "Intelligent agents", in: Weiss G. (Ed.), *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, MIT Press, Cambridge, MA, 1999, pp. 27–77.

[40] Kinny D., Georgeff M. "Commitment and effectiveness in situated agents", in: *12th International Joint Conference on Artificial Intelligence (IJCAI'91), Sydney*, 1991, pp. 82–88.

[41] Ferguson I.A., "Integrated control and coordinated behaviour: a case for agent models", in: Wooldridge M., Jennings N.R. (Eds.), *Intelligent Agents: Theories, Architectures and Languages*, in: *Lecture Notes in Artificial Intelligence*, vol. 890, Springer-Verlag, Berlin/New York, 1995, pp. 203–218.

[42] Jennings N.R., "On agent-based software engineering", *Artificial Intelligence* **117** (2000) 277–296.

[43] Smith R.G., Davis R., "Frameworks for cooperation in distributed problem solving", *IEEE Transactions on Systems, Man, and Cybernetics* **11** (1981) 61–70.

[44] Wooldridge M.J., Jennings N.R., "Formalising the cooperative problem solving process", in: *13th International Workshop on Distributed Artificial Intelligence (IWDAI'94), Lake Quinalt, WA*, 1994, pp. 403–417.

[45] Wooldridge M.J., Jennings N.R., "Towards a theory of cooperative problem solving", in: *6th European Conference on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'94), Odense, Denmark*, 1994, pp. 15–26.

[46] Li G., Hopgood A.A., Weller M.J., "Shifting matrix management: a model for multi-agent cooperation", *Engineering Applications of Artificial Intelligence* **16** (2003) 191–201.

[47] Bratman M.E., *Intentions, Plans, and Practical Reason*, Harvard University Press, Cambridge, MA, 1987.

[48] Mintzberg H., *The Structuring of Organizations*, Prentice Hall, New York, 1979.

[49] Finin T., Labrou Y., Mayfield J., "KQML as an agent communication language", in: Bradshaw J.M. (Ed.), *Software Agents*, MIT Press, Cambridge, MA, 1997, pp. 291–316.

[50] Bonabeau E., Dorigo M., Théraulaz G., *Swarm Intelligence: From Natural to Artificial Systems*, Oxford University Press, London, 1999.

[51] Dorigo M., Stützle T., *Ant Colony Optimization*, MIT Press, Cambridge, MA, 2004.

[52] Dorigo M., "Ant algorithms solve difficult optimization problems", in: *Lecture Notes in Artificial Intelligence*, vol. 2159, Springer-Verlag, Berlin/New York, 2001, pp. 11–22.

[53] Lumer E.D., Faieta B., "Diversity and adaptation in populations of clustering ants", in: Cliff D., Husbands P., Meyer J., Wilson S. (Eds.), *Proc. 3rd Internat. Conf. on Simulation of Adaptive Behavior: From Animal to Animats*, The MIT Press/Bradford Books, Cambridge, MA, 1994.

[54] Ramos V., Almeida F., "Artificial ant colonies in digital image habitats—a mass behaviour effect study on pattern recognition", in: Dorigo M., Middendorf M., Stüzle T. (Eds.), in: *Proc. 2nd Internat. Workshop on Ant Algorithms: From Ant Colonies to Artificial Ants, Brussels, Belgium*, 2000, pp. 113–116.

[55] Holland J.H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Harbor, 1975.

[56] Syswerda G., "Uniform crossover in genetic algorithms", in: Schaffer J.D. (Ed.), *4th International Conference on Genetic Algorithms*, Morgan Kaufmann, San Francisco, CA, 1989.

[57] Mitchell M., *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, MA, 1996.

[58] Hopfield J.J., “Neural networks and physical systems with emergent collective computational abilities”, in: *Proc. National Academy of Science, USA*, vol. 79, 1982, pp. 2554–2558.

[59] Muezzinoglu M.K., Guzelis C., Zurada J.M., “A new design method for the complex-valued multistate Hopfield associative memory”, *IEEE Transactions on Neural Networks* **14** (2003) 891–899.

[60] Tong S., Koller D., “Support vector machine active learning with applications to text classification”, *Journal of Machine Learning Research* **2** (2001) 45–66.

[61] Broomhead D.S., Lowe D., “Multivariable functional interpolation and adaptive networks”, *Complex Systems* **2** (1988) 321–355.

[62] Peng H., Ozaki T., Haggan-Ozaki V., Toyoda Y., “A parameter optimization method for radial basis function type models”, *IEEE Transactions on Neural Networks* **14** (2003).

[63] Kohonen T., “Adaptive, associative, and self-organising functions in neural computing”, *Applied Optics* **26** (1987) 4910–4918.

[64] Kohonen T., *Self-Organization and Associative Memory*, second ed., Springer-Verlag, Berlin/New York, 1988.

[65] Germen E., “Increasing the topological quality of Kohonen’s self organising map by using a hit term”, in: *ICONIP ’02: 9th International Conference on Neural Information Processing*, vol. 2, 2002, pp. 930–934.

[66] Yin H., Allinson N.M., “Interpolating self-organising map (iSOM)”, *Electronics Letters* **35** (1999) 1649–1650.

[67] Carpenter G.A., Grossberg S., “ART2: Self-organization of stable category recognition codes for analog input patterns”, *Applied Optics* **26** (1987) 4919–4930.

[68] Davenport M.P., Titus A.H., “Multilevel category structure in the ART-2 network”, *IEEE Transactions on Neural Networks* **15** (2004) 145–158.

[69] Baretto G., Arajo A., “Time in self-organizing map: an overview of models”, *International Journal of Computer Research* **10** (2001) 139–179.

[70] Voegtlin T., “Recursive self-organizing maps”, *Neural Networks* **15** (2002) 979–991.

[71] Hallam N.J., Hopgood A.A., Woodcock N., “Defect classification in welds using a feed-forward network within a blackboard system”, in: *International Neural Network Conference (INNC’90), Paris*, vol. 1, 1990, pp. 353–356.

[72] Hornick K., Stinchcombe M., White H., “Multilayer feedforward networks are universal approximators”, *Neural Networks* **2** (1989) 359–366.

[73] Rumelhart D.E., Hinton G.E., Williams R.J., “Learning representations by back-propagating errors”, *Nature* **323** (1986) 533–536.

[74] Rumelhart D.E., Hinton G.E., Williams R.J., “Learning internal representations by error propagation”, in: Rumelhart D.E., McClelland J.L. (Eds.), *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, vol. 1, MIT Press, Cambridge, MA, 1986.

[75] Knight K., “Connectionist ideas and algorithms”, *Communications of the ACM* **33** (11) (November 1990) 59–74.

[76] Lesser V.R., Fennell R.D., Erman L.D., Reddy D.R., “Organization of the Hearsay-II speech understanding system”, *IEEE Transactions on Acoustics, Speech, and Signal Processing* **23** (1975) 11–23.

[77] Erman L.D., Hayes-Roth F., Lesser V.R., Reddy D.R., “The Hearsay-II speech understanding system: integrating knowledge to resolve uncertainty”, *ACM Computing Surveys* **12** (1980) 213–253.
[78] Nii H.P., “Blackboard systems, part one: the blackboard model of problem solving and the evolution of blackboard architectures”, *AI Magazine* **7** (1986) 38–53.
[79] Brzykcy G., Martinek J., Meissner A., Skrzypczynski P., “Multi-agent blackboard architecture for a mobile robot”, in: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 4, IEEE Press, New York, 2001, pp. 2369–2374.
[80] Hopgood A.A., Phillips H.J., Picton P.D., Braithwaite N.S.J., “Fuzzy logic in a blackboard system for controlling plasma deposition processes”, *Artificial Intelligence in Engineering* **12** (1998) 253–260.
[81] Nolle L., Wong K.C.P., Hopgood A.A., “DARBS: a distributed blackboard system”, in: Bramer M., Coenen F., Preece A. (Eds.), *Research and Development in Intelligent Systems, XVIII*, Springer-Verlag, Berlin/New York, 2001, pp. 161–170.
[82] Feigenbaum E.A., in: Englemore R.S., Morgan A.J. (Eds.), *Blackboard Systems*, Addison–Wesley, Reading, MA, 1988.
[83] Yao X., “Evolving artificial neural networks”, *Proceedings of the IEEE* **87** (1999) 1423–1447.
[84] Karr C.L., “Genetic algorithms for fuzzy controllers”, *AI Expert* (February 1991) 26–33.
[85] Karr C.L., “Applying genetics to fuzzy logic”, *AI Expert* (March 1991) 38–43.
[86] Altug S., Chow M.Y., Trussell H.J., “Heuristic constraints enforcement for training of and rule extraction from a fuzzy/neural architecture—part II: Implementation and application”, *IEEE Transactions on Fuzzy Systems* **7** (1999) 151–159.
[87] Chow M.Y., Altug S., Trussell H.J., “Heuristic constraints enforcement for training of and knowledge extraction from a fuzzy/neural architecture—part I: Foundation”, *IEEE Transactions on Fuzzy Systems* **7** (1999) 143–150.
[88] El Mihoub T., Hopgood A.A., Nolle L., Battersby A., “Performance of hybrid genetic algorithms incorporating local search”, in: Horton G. (Ed.), *Proceedings of 18th European Simulation Multiconference*, 2004, pp. 154–160.
[89] Schwefel H.P., *Numerical Optimization of Computer Models*, Wiley, New York, 1981.
[90] Holland J.H., Reitman J.S., “Cognitive systems based on adaptive algorithms”, in: Waterman D.A., Hayes-Roth F. (Eds.), *Pattern-Directed Inference Systems*, Academic Press, San Diego, CA, 1978, pp. 313–329.
[91] Goldberg D.E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison–Wesley, Reading, MA, 1989.
[92] Tsukimoto H., “Extracting rules from trained neural networks”, *IEEE Transactions on Neural Networks* **11** (2000) 377–389.
[93] Picton P.D., Johnson J.H., Hallam N.J., “Neural networks in safety-critical systems”, in: *3rd International Congress on Condition Monitoring and Diagnostic Engineering Management, Southampton, UK*, 1991.
[94] Johnson J.H., Hallam N.J., Picton P.D., “Safety critical neurocomputing: explanation and verification in knowledge augmented neural networks”, in: *Colloquium on Human–Computer Interaction*, IEE, London, 1990.

[95] Watson I., Gardingen D., “A distributed case-based reasoning application for engineering sales support”, in: *16th International Joint Conference on Artificial Intelligence (IJCAI’99), Stockholm, Sweden,* vol. 1, Morgan Kaufmann, San Mateo, CA, 1999, pp. 600–605.

[96] Al-Kuzee J., Matsuura T., Goodyear A., Nolle L., Hopgood A.A., Picton P.D., Braithwaite N.S.J., “Intelligent control of low pressure plasma processing”, in: *IEEE Industrial Electronics Society Conference (ICON03)*, 2003.

[97] Choy K.W., Hopgood A.A., Nolle L., O’Neill B.C., “Design and implementation of an inter-process communication model for an embedded distributed processing network”, in: *International Conference on Software Engineering Research and Practice (SERP’03), Las Vegas*, 2003, pp. 239–245.

Software Model Checking with SPIN

GERARD J. HOLZMANN

Laboratory for Reliable Software
NASA/JPL, Pasadena, CA 91109
USA
gholzmann@acm.org

Abstract
The aim of this chapter is to give an overview of the theoretical foundation and the practical application of logic model checking techniques for the verification of multi-threaded *software* (rather than hardware) systems. The treatment is focused on the logic model checker SPIN, which was designed for this specific domain of application. SPIN implements an automata-theoretic method of verification. Although the tool has been available for over 15 years, it continues to evolve, adopting new optimization strategies from time to time to help it tackle larger verification problems. This chapter explains how the tool works, and which types of software verification problems it is designed to handle.

ISSN: 0065-2458/DOI 10.1016/S0065-2458(05)65002-4

1. Introduction

This chapter is concerned with the development of automated procedures for the verification of software systems, with particular emphasis on the verification of process interactions in either logically or physically distributed software systems. Several verification tools are available today that can prove interesting facts for a significant class of such systems. An up to date overview can be found on the web.[1] In this description we will focus on SPIN [36] as one of the leading tools in this class. SPIN is distributed freely in source form.[2]

Two notable trends have contributed to the recent successes of logic model checkers in the verification of distributed software systems. The first trend is the continuing improvement in algorithms and tool design in this area, which make it possible to handle ever larger and more complex verification problems. We will review the main improvements of this type in this chapter. A second significant trend is the steady increase in basic compute power, which continues to follow the curve that was tentatively suggested by Gordon Moore nearly forty years ago [42].

The trends that have turned software verification from a theoretical curiosity into a practical reality are paralleled by similar trends in hardware verification. The difference in the nature of hardware and software, though, makes that there is surprisingly little overlap in the algorithms, data structures, and specific logics that are used in these two fields. We will discuss some of the main reasons for these differences towards the end of this chapter.

The most commonly used method to validate software systems today remains testing. In a unit test, a single process or module of the system is placed in isolation and probed on its functional correctness. Once successful, a series of unit tests is followed by a system integration test. In an integration test multiple units are linked together to form part or all of the envisioned system. The limitations of this method of system validation are as well known as its benefits. For sequential software systems, where one is primarily interested in verifying the computational aspects of a system, the classical testing techniques still have few competitors, even though much

[1] http://archive.comlab.ox.ac.uk/comp/formal-methods.html.
[2] http://spinroot.com/whatispin.html.

could be done to improve precision and coverage by a more aggressive use of formal methods based approaches. In distributed software systems, the verification task is larger, since now we do not just need to worry about computational correctness but also about a range of concurrency related problems that can prevent proper execution. Concurrency does not just increase the obligations of the tester or verifier, it also significantly complicates the already existing obligations for demonstrating the correctness of sequential computations. Concurrency can introduce race conditions, data corruption, delay, process or thread starvation, or even system-wide deadlock.

The unpredictable nature of the interleaving of process executions in distributed systems makes that test executions are not always reproducible. Each single execution is typically only one of a virtually unimaginably large set of possible interleaved executions. What is needed to address these problems is an effective method for probing the system for conveniently defined *classes of behavior*, rather than isolated *instances of behavior*. Logic model checkers promise to provide such a technique, but they too come with some limitations. The current limitations of model checking are of two kinds: computational complexity and user friendliness. In this chapter we provide a synopsis of the model checking procedure as it applies to the verification of distributed software systems, and summarize the progress that has been made in diminishing the effects of these last two limitations.

We will begin by sketching the development of automated verification systems since the late seventies. We then introduce the main building blocks of the software model checking procedure. We define what a formal model is, and how we can formally state the logic properties of a model, using a standard definition of automata on infinite words. Next we discuss the automata theoretic verification procedure, as defined by Vardi and Wolper [52], and show how it can be implemented efficiently, following the methodology that was used in the design of the SPIN model checker.

We then move to some more practical considerations: the problem of semi-automatically extracting models from software applications, to enable us to apply the model checking procedure. Systematic abstraction techniques play a central role in this area. We discuss abstraction techniques and give examples of significant applications of software model checking in practice. We conclude the chapter with a brief comparison of software model checking with techniques used in hardware verification and a perspective of likely developments in this field in the near future.

2. Background

The first attempts to built automated verification systems targeted formal definitions of communication protocols. In 1979, Jan Hajek used a graph exploration tool, called Approver [21], to formally verify basic correctness properties of the protocols in Tanenbaum's primer on computer networks [47]. The details of Hajek's system were documented only fairly recently,[3] but remained unknown at the time. Unfortunately, this limited the influence that the Approver system could have on later developments. Independently, Colin West developed a different protocol validation procedure [53], triggered by a collaboration with Pitro Zafiropulo on the validation of a CCITT recommendation [54]. Initially unaware of this earlier work, I developed and implemented yet another protocol verification procedure in 1980 at Bell Labs, initially based on the definition of a process algebra [22,23].

Communications protocols were a convenient target for the early work since their behavior could fairly easily be formalized in terms of automata. A well-known example of this type, that served as a litmus test for early validation systems throughout the eighties, is the definition of the alternating bit protocol as two communicating finite state machines. The protocol is illustrated in Figure 1 as it was first defined in 1969 [3].

Two state machines are defined in Figure 1, formalizing a sender process and a receiver process. The edge labels specify message exchanges. Each label consists of two characters: the first specifies the origin of the message being sent or received and the second specifies a sequence number for the message. This sequence number, termed the *alternation bit* in [3], is either zero or one, and is toggled between the two values on each successful transmission. Underlined names represent send actions; the other names represent receives. The double arrows, finally, indicate the states where new data is to be fetched for transmission by the sender, or received data is to

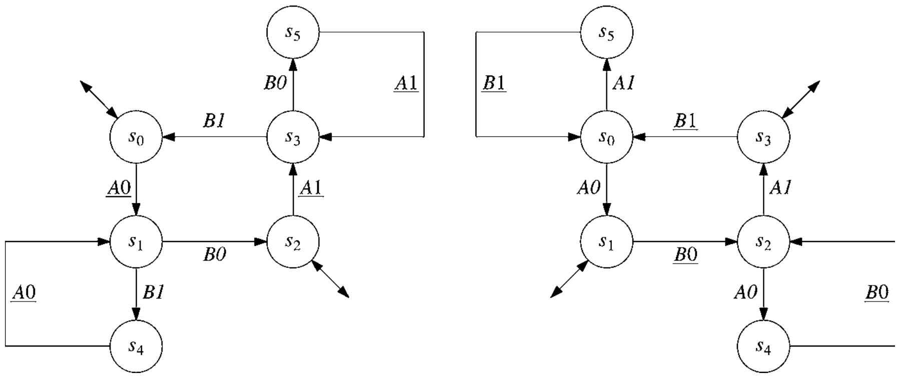


FIG. 1. Alternating Bit Protocol.

[3] http://www.matheory.info/hajekit/approver.txt.

be stored by the receiver. The protocol starts with sender and receiver in the states labeled s_0. The sender will then transmit a message with sequence number zero ($\underline{A0}$) to the receiver. If all is well, the receiver will receive the message ($A0$) and both processes will move to the states labeled s_1. The receiver will now acknowledge receipt by transmitting $\underline{B0}$, and both processes move to the states labeled s_2. The same sequence now repeats with the sequence number toggled from zero to one. If for some reason, e.g., the loss or duplication of a message, the receiver process sees a data message with the wrong sequence number, it will reply with the matching acknowledgment but not proceed. If the sender receives an acknowledgement with a wrong sequence number it will repeat the last transmission and hope for the best.

If we abstract from the data being transmitted, we can see that each process in this system can be in no more than six distinct states. The combination of sender and receiver, therefore, can be in no more than 6×6 or 36 distinct system states. In this simple case, a brute force exhaustive enumeration of all reachable states of the system will suffice to establish most of its logical properties. The combined behavior of the system defines a new finite state automaton, and can similarly to Figure 1 be formalized as a graph. This graph can be constructed and analyzed with a standard depth-first search procedure [48] at a cost that is linear in the size of the graph.

Manual techniques for the analysis of state machine models of protocols had been pioneered in the early seventies, e.g., [5], but not surprisingly these methods quickly proved too cumbersome and too errorprone, as demonstrated in [21]. Although the first automated systems had greater potential, they were mostly restricted to proving only a small set of mostly predefined properties, and would quickly run into seemingly unsurmountable barriers of computational complexity. The types of properties that could be demonstrated for small protocol models included absence of deadlock (i.e., the absence of reachable states in the global execution graph without successors) and the preservation of system invariants on system states (i.e., the absence of reachable states in which one or more of the required invariants would evaluate to false). In the eighties a more general framework for proving logic properties of finite state models took shape and found general acceptance.

The development of automated verification systems has taken a somewhat different path for hardware and for software applications, leading to two different sets of verification tools that are based on different logics and that exploit different types of search and optimization algorithms. The dominant techniques in formal hardware verification, e.g., [13,41], are founded on the early work of Clarke and Emerson in the U.S. [11], and of work by Queille and Sifakis in France [45]. In software model checking, the development can be traced through the early work of Pnueli [44] on temporal logic, to the development of the *automata theoretic verification method* by Vardi and Wolper in the mid eighties [52,55]. It should be noted, though, that the new theories were not immediately of practical use. It took a while for algorithms to be developed that could be implemented efficiently, and for desktop machines to provide the required compute power to execute them. It appears now generally agreed that this turning point was reached in the mid to late nineties.

With improved algorithms and ever increasing compute power, the attention in recent years has shifted from the development of the basic capability to perform logic model checking on hand-built system models towards the automated extraction of verification models from implementation level source code. Before discussing these methods, though, we will first cover the basic theoretical framework that underlies specifically the SPIN model checking system.

3. Finite Automata

In this section we introduce the notion of an automaton and of ω-acceptance, which we use to develop the automata theoretic verification method in subsequent sections. We begin with the definition of an extended finite state automaton.

Definition 3.1. An *extended finite state automaton* A is a tuple $\{S, s_0, D, L, T, F\}$, with S a finite set of 'states,' $s_0 \in S$, called the 'initial state,' D a finite set of named 'data objects', L a finite set of named 'actions' on objects in D, $T \subseteq S \times L \times S$, called the 'transition relation,' and $F \subseteq S$, called the set of 'final' states.

We will be brief here about the definition of 'data objects' and 'actions.' Model checking languages such as PROMELA [27,30,36] give precise semantics to these notions, which guides the operation of the model checker. For our purposes here, it will suffice to assume that each data object has a unique name and finitely many possible 'values' of arbitrary type. One value in the domain of each object is always tagged as the initial value of an object of that type. Each data object also has a 'current value' that can only be changed through the application (or 'execution') of 'actions' from set L.

Definition 3.2. An *action* on set of data objects D consists of two parts: a guard and an effect. The 'guard' is a boolean expression on the values of elements in D. The 'effect' can change the values of elements in D as a function of the current values of all elements.

The intuition is that (the effect part of) an action can only be applied when the guard condition is true. Every transition in the automaton is labeled with an action, which blocks the transition until the guard condition is satisfied and applies the effect when the guard condition is true and the transition is executed.

Definition 3.3. A transition is said to be *executable* if and only if the guard expression from the corresponding action evaluates to *true*, otherwise it is said to be 'unexecutable' or 'blocking.'

We will use this notion of 'executable' and 'unexecutable' actions below in the definition of the 'runs' of a system.

As examples of useful data objects, consider the following PROMELA message channel structures.

```
    chan s2r = [1] of { mtype, bit };
    chan r2s = [1] of { mtype, bit };
```

According to PROMELA semantics, these channels are initially empty and can each store one message consisting of two typed fields [36]. Some actions from PROMELA on the channel `s2r` are:

```
    full(s2r)
    s2r!A,0
    r2s?B,1
```

The first action has a guard that returns *true* only when the channel currently stores one message and is thereby filled to capacity. The effect part of the first action is *skip*, a null-operation that has no effect. That is: the actions acts as a condition without side-effects. The second action has a guard that returns *true* only when the channel is non-full. Its effect changes the value of the data object `s2r` by appending the message `A`, `0`, with `A` of type `mtype`, and `0` of type bit. The third action has a guard that returns *true* only when the channel holds a message with the fields `B, 1`; its effect part deletes that message from the channel.

An extended finite state automaton, as defined, can be represented conveniently by a directed graph with the nodes representing states and the edges representing the transition relation T. The edges are labeled with actions from L. A system like this is therefore also known as a 'labeled transition system.' Our aim is to use extended finite state automata to represent process behavior in a distributed system. Set F can be used to mark the normal termination points of a process, or they can be used to mark special acceptance nodes in the graph that can serve to define and check the liveness properties or a system, as we shall describe shortly.

The two state machines in Figure 1 can also be defined as extended finite state automata. The automaton for the sender (on the left side in Figure 1), for instance, can be defined as follows, using the two data objects that we introduced in the example above and PROMELA syntax for the actions

$$S = \{s_0, s_1, s_2, s_3, s_4, s_5\},$$

$$s = s_0,$$

$$D = \{\text{s2r}, \text{r2s}\},$$

$$L = \{\text{r2s?B}, 0, \text{r2s?B}, 1, \text{s2r!A}, 0, \text{s2r!A}, 1\},$$
$$T = \big\{(s_0, \text{s2r!A}, 0, s_1), (s_1, \text{r2s?B}, 1, s_4), (s_1; \text{r2s?B}, 0, s_2), (s_4, \text{s2r!A}, 0, s_t), (s_2, \text{s2r!A}, 1, s_3), (s_3, \text{r2s?B}, 1, s_0), (s_3, \text{r2s?B}, 0, s_5), (s_5, \text{s2r!A}, 1, s_3)\big\},$$
$$F = \{s_0, s_2\}.$$

In general, the finite state automata that we will consider can be non-deterministic, e.g., we allow transitions such that: $(v, a, w) \in T$, $(v, a, w') \in T$, with $w \neq w'$. Non-determinism is an important mechanism for building an abstract model of a distributed system. It can be used to generalize a model and to remove implementation level detail [32,36].

3.1 Automaton Runs

A *run* $\sigma = t_0, t_1, t_2, \ldots, t_k$ of automaton A is a sequence of transitions that satisfies the following conditions:

- the source state for t_0, the first transition in σ, is always s_0, i.e., the initial state of the automaton,
- $\forall i,\ 0 \leqslant i \leqslant k$: $t_i \in T$,
- $\forall i,\ 0 \leqslant i < k$: $t_i \equiv \{a, b, c\}$, and $t_{i+1} \equiv \{d, e, f\} \rightarrow c \equiv d$,

that is, the run defines a path in the graph of A. Note that a 'run' only defines uninterpreted potential executions of a system, it does not take the manipulation of data objects through actions into account just yet. We will distinguish between 'valid' and 'invalid' runs in an expanded finite state automaton shortly.

According to the classic definition of acceptance a *finite* run is said to be *accepted* by A if its final state is in set F, i.e., for run σ with final transition $t_k \equiv \{a, b, c\}$ if $c \in F$. If set F is used to mark the normal termination points of a process then a run will not be accepted by the automaton unless it terminates at such a marked state.

3.2 Omega Acceptance

The classic notion of acceptance given above applies only to finite runs, i.e., to terminating executions. Looking at the automata in Figure 1, though, it is unclear if termination should be considered proper behavior or an error. As long as data is available from the unspecified source, the sender process should continue to transmit it to the receiver. If the protocol terminates, we would like it to terminate in either state s_0 or s_2, with the last data message properly acknowledged, but it need not terminate at all. We will therefore define a notion of ω-acceptance that can be applied to both the *infinite* and the *finite* runs of an automaton. An infinite run of an automaton is called an ω-run.

Definition 3.4. An ω-run σ is *accepted* by extended finite state automaton A if it contains at least one state from set F infinitely often.

The above notion of acceptance is known as Büchi acceptance [8,50]. For the automata definition we gave for the processes in Figure 1 it would suffice to limit the set of accepting states to one of the two states s_0 and s_2, since clearly neither can be visited infinitely often unless the other is too. It is also clear that any ω-run for a finite state automaton will have to repeat states, i.e., it will necessarily be cyclic.

We now define the ‘stutter-extension’ of a finite run to make sure that the rules of ω-acceptance can be applied equally to infinite and finite runs of an automaton.

Definition 3.5. The *stutter-extension* of a finite run σ of finite state automaton A is the ω-run that is derived from σ by appending an infinite number of nil-actions $\{s_k, nil, s_k\}$ to it, where s_k is the final state that is reached in σ, and *nil* is an action with guard *true* and effect *skip*.

3.3 Asynchronous Product

The combined behavior of asynchronously executing processes in a distributed system can be formalized as a simple product of automata.

Definition 3.6. The *asynchronous product* of the extended finite automata $A = \{S, s, D, L, T, F\}$ and $B = \{S', s', D, L', T', F'\}$ is another extended finite automaton $\{S'', s'', D'', L'', T'', F''\}$ such that $S'' = S \times S'$, $s'' = (s, s')$, $D'' = D \cup D'$, $L'' = L \cup L'$, $T'' \subseteq S'' \times L'' \times S''$, $F'' = F \times F'$, and

$$\forall\big((n, n'), l, (m, m')\big) \in T'' \colon \big(l \in L \wedge (n, l, m)\big) \vee \big(l \in L' \wedge (n', l, m')\big).$$

That is, the states of the asynchronous product define combinations of states in the individual automata, but the edges correspond to the individual transitions of the two automata: the transitions are interleaved.

As an example, the asynchronous product of the two automata from Figure 1 has 6×6 or 36 states. The initial state of that automaton is (s_0, s_0). Set F has four states. Set D contains two data objects {s2r, r2s}, and set L contains the eight actions {r2s!B, 0, r2s!B, 1, r2s?B, 0, r2s?B, 1, s2r!A, 0, s2r!A, 1, s2r?A, 0, s2r?A, 1}.

3.4 Automata Expansion

Clearly the data objects in an extended finite state automaton also carry state information. We can map an extended finite state automaton to a *pure* finite state automaton by moving the state information from set D into set S. In effect, this expansion multiplies set S with the set of values of all data objects. To construct a pure automaton we can replicate each state in S, except the initial state, as many times as there are distinct combinations of values for all data objects in D. For the initial state, the initial value for each data object is used. Each copy of $s \in S$ has a copy of all incoming and outgoing transitions of s in the original automaton.

Next, we can mark the transitions in this new automaton as either *valid* or *invalid*, depending on whether the corresponding action from L is executable in that state. Since data values are now explicit, the validity of each transition can be determined unambiguously.

Let $\mu(n)$ be the valuation of all data objects in state n, i.e., a finite and ordered set of values, and let $\gamma(l, n)$ be the valuation of all data objects in state n *after* the effect part of action l is applied.

Definition 3.7. A transition $\{n, l, m\}$ from the set of transitions of an expanded finite state automaton is *valid* if the guard of action l is *true* for $\mu(n)$, and $\mu(m) \equiv \gamma(l, n)$.

The expansion process of an automaton is completed by first omitting all transitions that are not valid, and next omitting all states that are no longer reachable from the initial state.

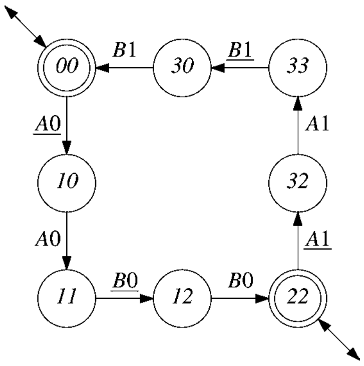


FIG. 2. Expanded asynchronous product of automata in Figure 1. Accepting states have double circles.

For the automata in Figure 1 the full set of actions is, assuming *ideal* full-duplex communication between sender and receiver:

$\underline{A0}$:	`s2r!A,0`	$A0$:	`s2r?A,0`
$\underline{A1}$:	`s2r!A,1`	$A1$:	`s2r?A,1`
$\underline{B0}$:	`r2s!B,0`	$B0$:	`r2s?B,0`
$\underline{B1}$:	`r2s!B,1`	$B1$:	`r2s?B,1`

The complete expansion of the asynchronous product of the two automata, after deleting invalid transitions and unreachable states, has eight states, and permits just one ω-run, as illustrated in Figure 2. In the product automaton in Figure 2 we can now easily annotate each state n with its valuation $\mu(n)$, giving the explicit value of each data object.

4. Temporal Logic

The correctness properties of a distributed system can be formalized in linear temporal logic (LTL), as first proposed by Pnueli in [44]. Any boolean expression over the state of a system and its associated data values will be called a *state formula*. Every guard from a action in an automaton definition, for instance, is defined by a state formula. In the following, the lower case symbols p, q, r represent state formulae and f, q, h represent temporal formulae, which are defined as follows.

Definition 4.1. Every state formula p is also a *temporal formula*.
If f is a temporal formula, then so are $\neg f$, (f), and $\mathsf{X} f$.
If f and g are temporal formulae, then so are $f \wedge g$, $f \vee g$, and $f \; \mathsf{U} \; g$.

The temporal operator X is pronounced 'next,' and the temporal operator U is pronounced 'until.'

We write $v(f, s_i) \equiv$ *true* to express that temporal formula f holds at state s_i. We can then define the standard Boolean operators as follows:

$$v(f \vee g, s_i) \Leftrightarrow v(f, s_i) \vee v(g, s_i),$$

$$v(f \wedge g, s_i) \Leftrightarrow v(f, s_i) \wedge v(g, s_i),$$

$$v(\neg f, s_i) \Leftrightarrow \neg v(f, s_i).$$

The semantics of X and U are defined over an ω-run σ. Let $s_0, s_1, s_2, \ldots, s_i, s_{i+1}, \ldots$, be the set of states that is traversed in σ. We can then define:

$$v(\mathsf{X} f, s_i) \Leftrightarrow v(f, s_{i+1}),$$

$$v(f \,\mathsf{U}\, g, s_i) \quad\Leftrightarrow\quad v(g, s_i) \vee \big(v(f, s_i) \wedge v(f \,\mathsf{U}\, g, s_{i+1})\big).$$

The definition of U requires that either g is *true* now or that f remain *true* until g becomes *true*. If, however, f remains *true* invariantly then g is not required to become *true*. The operator U is therefore called a 'weak until' operator. There is also a 'strong until' operator $\boldsymbol{U}$, which can be defined as follows.

$$v(f \boldsymbol{U} g, s_i) \quad\Leftrightarrow\quad v(f \,\mathsf{U}\, g, s_i) \ \wedge\ \exists j, j \geqslant i\colon\ v(g, s_j).$$

Two other frequently used temporal operators can be defined in terms of the operators we have defined so far. They are the $\Box$, or 'always' operator, and the $\Diamond$, or 'eventually' operator:

$$v(\Box f, s_i) \quad\Leftrightarrow\quad (f \,\mathsf{U}\, \mathit{false}),$$
$$v(\Diamond f, s_i) \quad\Leftrightarrow\quad (\mathit{true}\ \boldsymbol{U}\ f).$$

4.1 Standard Formulae

Many standard types of correctness requirements can be expressed with the temporal operators we have defined here. We give two examples of commonly used patterns.

A *progress* property is a temporal formula that can be written in the form $\Box\Diamond p$. This formula states that at any point in an execution the state formula p is either true or it will become true at some point in the future.

A *guarantee* property is a temporal formula that can be written in the form $\Diamond\Box p$. This formula states that the state formula p is guaranteed to become invariantly true at some point in the future.

Progress and guarantee are in many ways dual properties. It is, for instance, not hard to show that $\neg\Box\Diamond f \Leftrightarrow \Diamond\Box\neg f$.

Some other equivalences [40] are:

$$\neg\Box f \quad\Leftrightarrow\quad \Diamond\neg f,$$
$$\neg\Diamond f \quad\Leftrightarrow\quad \Box\neg f,$$
$$\Box(f \wedge g) \quad\Leftrightarrow\quad \Box f \wedge \Box g,$$
$$\Diamond(f \vee g) \quad\Leftrightarrow\quad \Diamond f \vee \Diamond g,$$
$$\Box\Diamond(f \vee g) \quad\Leftrightarrow\quad \Box\Diamond f \vee \Box\Diamond,$$
$$\Diamond\Box(f \wedge g) \quad\Leftrightarrow\quad \Diamond\Box f \wedge \Diamond\Box g,$$
$$f \,\mathsf{U}\, (g \vee h) \quad\Leftrightarrow\quad (f \,\mathsf{U}\, g) \vee (f \,\mathsf{U}\, h),$$
$$(f \wedge g) \,\mathsf{U}\, h \quad\Leftrightarrow\quad (f \,\mathsf{U}\, h) \wedge (g \,\mathsf{U}\, h),$$

$$
\begin{aligned}
f\ \boldsymbol{U}\ (g \vee h) &\quad\Leftrightarrow\quad (f\ \boldsymbol{U}\ g) \vee (f\ \boldsymbol{U}\ h),\\
(f \wedge g)\ \boldsymbol{U}\ h &\quad\Leftrightarrow\quad (f\ \boldsymbol{U}\ h) \wedge (g\ \boldsymbol{U}\ h),\\
\neg(f\ \boldsymbol{U}\ g) &\quad\Leftrightarrow\quad (\neg g)\ \mathsf{U}\ (\neg f \wedge \neg g),\\
\neg(f\ \mathsf{U}\ g) &\quad\Leftrightarrow\quad (\neg g)\ \boldsymbol{U}\ (\neg f \wedge \neg g).
\end{aligned}
$$

So far we have defined the evaluation of temporal formulae for specific ω-runs. We will be interested in proving properties of a system for *all* possible executions starting from its initial system state. When we say that f holds for finite state automaton A we mean that it holds for *all* ω-runs that start from A's initial state. We may, for instance, want to prove that $\Box(p \to \Diamond q)$ for the automaton in Figure 2, with p and q defined as state properties:

$$p \equiv \texttt{empty}(\texttt{s2r}), \qquad q \equiv \texttt{empty}(\texttt{r2s}).$$

Equivalently, we may want to prove that the negation of this formula is *not* satisfied. Using the equivalences and the definition of logical implication ($p \to q \Leftrightarrow \neg p \vee q$) we can write the negation as:

$$
\begin{aligned}
\neg\Box(p \to \Diamond q) &\quad\Leftrightarrow\quad \Diamond\neg(\neg p \vee \Diamond q) \quad\Leftrightarrow\quad \Diamond(p \wedge \neg\Diamond q)\\
&\quad\Leftrightarrow\quad \Diamond(p \wedge \Box\neg q)
\end{aligned}
$$

This formula is satisfied if at some point in an execution the state formula p becomes *true* while q is *false* and remains *false* forever thereafter. Note that this indeed captures the violation of the formula $\Box(p \to \Diamond q)$ that we started with. Now consider the automaton in Figure 3.

This automaton has two states s_0 and s_1, with s_0 the initial state. Set D is identical to those of the automata in Figures 1 and 2, $D = \{\texttt{s2r}, \texttt{r2s}\}$. Set L has three elements:

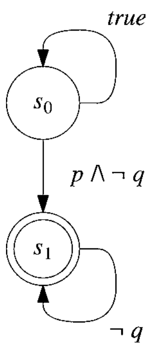


FIG. 3. Non-deterministic automaton for $\Diamond(p \wedge \Box\neg q)$, with initial state s_0 and accepting state s_1.

- *true* which is an action with guard (*true*) and effect *skip*,
- $p \wedge \neg q$ which is an action with guard $(p \wedge \neg q)$ and effect *skip*, and
- $\neg q$ which is an action with guard $(\neg q)$ and effect *skip*.

Set F, finally, has one element: s_1.

The accepting runs of this automaton have the following form, written as a sequence of transitions:

$$(true)^+ \; ; \; (p \wedge \neg q) \; ; \; (\neg q)^\omega$$

where ; indicates concatenation, + indicates finitely many repetitions, and ω indicates infinitely many repetitions. Note that this matches the semantics of $\Diamond(p \wedge \Box\neg q)$, and could be useful in automating the verification process. The automaton in Figure 3 need not be discovered by trial and error: there are efficient algorithms for constructing it mechanically from the LTL formula [19,17,18,52].

4.2 Synchronous Product

How can we use the automaton from Figure 3 to prove our sample property for the alternating bit protocol, i.e., for the automaton from Figure 2? Somehow we must 'match' the runs of the automaton in Figure 3 with the runs of the automaton in Figure 2. We can do precisely this by computing the synchronous product of these two automata.

Definition 4.2. The *synchronous product* of the extended finite automata $A = \{S, s, D, L, T, F\}$ and $B = \{S', s', D', L', T', F'\}$ is an extended finite automaton $\{S'', s'', D'', L'', T'', F''\}$ identical to the asynchronous product except for the definitions of L'' and T''. L'' is a set of *ordered* pairs $L \times L'$, and $T'' \subseteq S'' \times L'' \times S''$ with $\forall((n, n'), (l, l'), (m, m')) \in T''$: $(l \in L \wedge (n, l, m)) \wedge (l \in L' \wedge (n', l', m'))$.

That is, the edges of the synchronous product of the automata correspond to *joint* transitions of the automata. Since every transition now carries two actions, the guards of *both* actions must evaluate to true for the transition to be valid. For a property automaton that is derived from a temporal formula (like the one in Figure 3) the effect part on each action will always be *skip*, so the order in which the effects are executed is unimportant. (In general this order will matter, so that the synchronous product $A \times B$ may be different from $B \times A$.)

The synchronous product of the automata in Figures 2 and 3 is shown in Figure 4. Valid transitions are drawn solid and invalid transitions are dashed. The only valid ω-run in the automaton from Figure 4 contains no accepting states. We can conclude that formula $\Diamond(p \wedge \Box\neg q)$ cannot be satisfied in the automaton from Figure 2, and therefore that formula $\Box(p \to \Diamond q)$ cannot be violated.

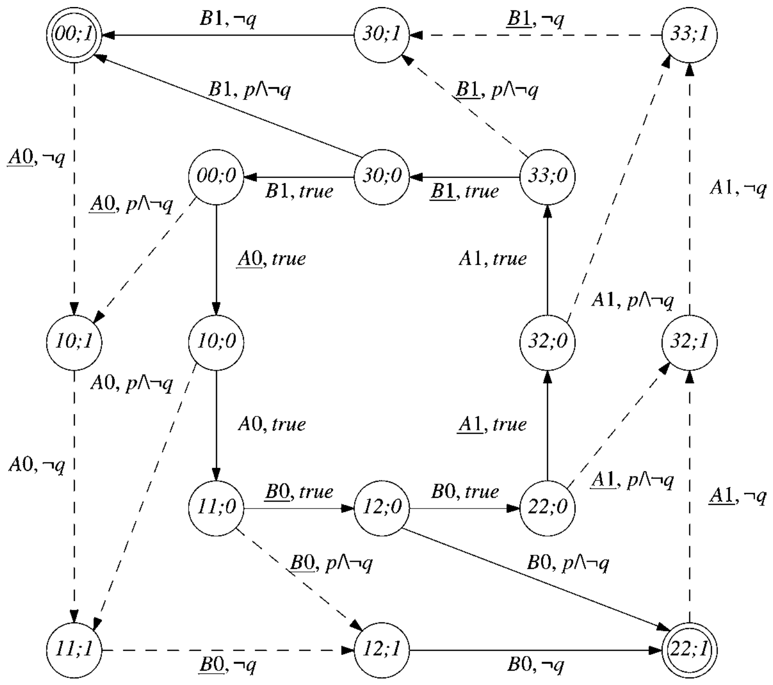


FIG. 4. Synchronous product of the automata in Figures 2 and 3.

5. LTL Model Checking

The set of all *ω-runs* that an automaton accepts is often referred to as the *language* that is recognized by the automaton. Let $L(M)$ be the language recognized by the automaton that represents the system behavior we are studying, and let f be a temporal logic formula that is required to be satisfied by the system. The verification proceeds in four steps:

(1) Mark all states in M as accepting states, to make sure that all ω-runs of M are considered.
(2) Compute a Büchi automaton B for $\neg f$ (the negation of f, capturing all possible ways in which f might be violated).
(3) Compute the language intersection of $L(B)$ and $L(M)$, by computing the synchronous product of B and M.
(4) If the intersection is empty, i.e., if the product automaton accepts no ω-runs at all, M cannot violate f and therefore property f is satisfied.

> If the intersection is non-empty, there is at least one ω-run that is accepted by both M and B. Because it is accepted by B it constitutes a violation of property f. The run can be used as concrete evidence that f is not satisfied by M.

In this section we will consider how this automata theoretic method for the verification of LTL formulae can be implemented efficiently.

5.1 Depth-First Search

For an accepting ω-run to exist, there must be at least one execution of the product automaton defined above that traverses an accepting state infinitely often. This means that there must exist at least one accepting state in the product automaton that is both reachable from the initial state of that automaton *and* that is reachable from itself. For this to be true the reachability graph for the product automaton must have at least one strongly connected component with an accepting state. The strongly connected components in a graph can be computed in linear time with Tarjan's algorithm [48]. The product of M and B also depends linearly on the numbers of states in the automata M and B.

More problematic is, though, that the sizes of M and B can depend exponentially on the problem size. M is generally given as an asynchronous interleaving product of automata. This means that the size of M can increase exponentially with the number of automata (asynchronous processes) that we consider. B is extracted from an LTL automata, and in the worst case the size of B can also be exponentially larger than the size of the formula, measured by the number of state subformulae in the formula [52].

Fortunately, in practice things are not quite this bad. LTL formulae of practical interest rarely contain more than two or three temporal operators, and the automata generated from them have rarely more than five or six states [18]. The reason is simple: the precise meaning of formulae with more than three temporal operators can be hard to determine. The chains of reasoning required to interpret such a requirement quickly becomes too long to be meaningful in systems verification. In almost all cases of interest, a more complex system requirement can be broken down into smaller steps of just a few basic types: invariance (expressed as $\Box p$), inevitability ($\Diamond p$), progress ($\Box\Diamond p$), and conditional response ($\Box(p \to (q \mathbin{\mathsf{U}} r))$) [39,43].

The size of the synchronous product of M and B is almost completely determined by the size of M, which can indeed be large. Contributing factors to the size of M can be the number of asynchronous processes, and the number and the value ranges of data objects used. A number of techniques have been developed to reduce the size of M, and the cost (in time and memory) of analyzing it. We will review the most important of these here. The most frequently used techniques include model reduction and abstraction, partial order reduction, symmetry reduction, on-the-fly verification, state compression, machine minimization, and proof approximation.

5.2 Nested Depth-First Search

First let us briefly revisit the central problem in LTL model checking: detecting the existence of at least one cycle through an accepting state, in a finite graph. In the worst case the algorithm must visit every node in the graph, therefore the complexity cannot be less than linear in the size of the graph. But if the construction of the strongly connected components can be avoided, this problem may be solved with lower overhead than Tarjan's algorithm.

Tarjan's algorithm stores the nodes of a graph in a single depth-first traversal. Each node is typically annotated with two integer numbers, a *lowlink* and a *depth-first* number, e.g., [2]. This requires storing with each node $2\log(R)$ additional bits of information, to represent the lowlink and the depth-first number of a node, if R is the number of nodes in the graph. In practice, with R unknown, one typically uses two 32-bit integers to store this information. We will explore an alternative method that allows us to solve the cycle detection problem while adding just two bits of information to each node.

We begin by discussing a simple algorithm for a restricted class of ω-properties, i.e., proving the absence or existence of non-progress cycles in a finite graph [26,27]. The algorithm works by splitting the depth-first search into two phases with the help of a two-state demon automaton. We then continue with a discussion of a similar but stronger two-phase search algorithm that can be used to prove the absence or existence of acceptance cycles (accepted ω-runs), so that it can be used to perform LTL model checking [14,29].

This is done by the addition of a two-state demon D, as illustrated in Figure 5. The demon can non-deterministically decide to move from its initial state s_0 into an alternate state s_1, where it will then stay forever. The label on this transition has guard *true* and effect *skip*. We assume that some of the states in the automaton M to be analyzed are marked 'progress' states. We will be interested in finding any ω-run that contains only finitely many such progress states. This corresponds to solving the model checking problem for LTL properties of the type $\Diamond\Box np$, with np a predefined state property that is *true* if and only if the system is not in a progress state.

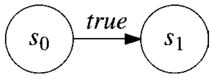


FIG. 5. Two-state non-deterministic demon automaton for detecting non-progress cycles.

We compute the asynchronous product of M and D, and perform a slightly modified depth-first search in the reachability graph for that product. The product machine will be at most twice the size of the original M, containing one copy of each state with the demon in state s_0, and possibly one more copy with the demon in state s_1.

Recall that each state s is a tuple consisting of a state of the demon and a state of M. Let $\mathrm{dm}(s)$ be *true* if the demon machine is in state s_0, and let $\mathrm{np}(s)$ be *true* if s is not marked as a progress state. The non-progress cycle detection algorithm is then as follows. The search starts from the initial state of the product of M and D, with the demon in state s_0.

```
dfs_A(s)
{       add s to visited
        if dm(s) or np(s)
        {       push s onto stack
                for each successor s′ of s
                {       if s′ not in visited
                        {       dfs_A(s′)
                        } else if s′ in stack and
                          ¬dm(s′)
                        {       report non-progress
                                cycle
                                stop
                }       }
                pop s from stack
}       }
```

Note that we do not consider any successors of progress states when the demon is in state s_1. Every cycle in the second state space (with the demon in state s_1) is therefore necessarily a non-progress cycle.

Property 5.1. *If non-progress cycles exist,* `dfs_A()` *will report at least one of these.*

Proof. Suppose there exists a reachable state that is part of a non-progress cycle, i.e., it can be reached from itself without passing through progress states. Consider the *first* such state that is entered into the second state space (upon the transition of the demon automaton into its alternate state), and call it r.

State r is reachable from itself in the second state space and must find itself in the depth-first search below r unless that search truncates at a previously visited state outside the current search stack. Call that state v. We know that r is reachable from v (or else it would not block r from reaching itself) and that v is reachable from r. This means that v is reachable from itself in the second statespace via r. This, however, contradicts the assumption that r was the first state such state entered into the second state space. This means that r either revisits itself or a successor of r revisits itself before that happens. In both cases the existence of a non-progress cycle is reported. □

Whenever a cycle is detected, the corresponding ω-run can be reproduced exactly from the contents of the stack: it will contain a finite prefix of non-repeated states, and a finite suffix, starting at the state within the stack that was revisited, with only non-progress states.

To implement the algorithm it is not necessary to store two full copies of each reachable state. It suffices to store the states once with the addition of two bits [20]. The first of the two bits records if the state was encountered in the first statespace, and the second bit records if the state was encountered in the second statespace. Initially both bits are off. We can encounter only the bit combinations 01, 10, and 11, but not 00. (Note that the state is neither present in the first nor the second statespace when the bit combination is 00.) Note that states may be either encountered first in the second statespace, and later in the first statespace, or vice versa. One bit, e.g., to record only the state of the demon automaton, therefore would not suffice. The second of the two bits is always equal to the state of the demon automaton, which therefore need not be stored separately.

This non-progress cycle detection algorithm was first implemented in 1988 in the tool `sdlvalid`, the immediate predecessor of the SPIN model checker [25], and later incorporated also in SPIN [26,27]. A stronger version of this type of two-phase search algorithm was introduced in [14], and can be used to solve the general LTL model checking problem. This algorithm is known as the *nested depth-first search*.

This time the transitions of the demon automaton D are placed under the control of the search algorithm. The call `dfs_B`(s, d) performs a depth-first search from state s in M and state d in D. Let acc(s) be *true* if and only if state s is accepting. The search starts with the call `dfs_B`(s_0, s_0)

```
dfs_B(s, d)
{       add s to visited
        push s onto stack
        for each successor s′ of s
        {       if s′ not in visited
                {       dfs_B(s′, d)
                } else if s′ $\equiv$ seed and $d \equiv s_t$
                {       report acceptance cycle
                        stop
        }       }
        if $d \equiv s_0$ and acc(s)
```

```
                {       // remember the root of the
                         second search
                         seed = s
                         // perform second search in
                         postorder
                         // with demon moved to
                         state $s_1$
                         dfs_B ($s, s_1$)
                }
                pop s from stack
        }
```

The search tries to locate at least one accepting state that is reachable from itself. The demon machine moves only from accepting states and the move is explored only after all successors of the accepting state have been explored (i.e., in postorder). It is now no longer sufficient for the second search to find any state within the depth-first search stack, we must require that the seed state from which the second search was initiated itself is revisited. The proof of correctness for this version of the algorithm is as follows [14].

Property 5.2. *If acceptance cycles exist,* `dfs_B()` *will report at least one of these.*

Proof. Let r be the first accepting state reachable from itself for which the second search is initiated. State r cannot be reachable from any state that was previously entered into the second state space.

Suppose there was such a state w. To be in the second state space w either is an accepting state, or it is reachable from an accepting state. Call that accepting state v. If r is reachable from w in the second state space it is also reachable from v. But, if r is reachable from v in the second state space, it is also reachable from v is the first state space. There are now two cases to consider. Either (a) r is reachable from v in the first state space without visiting states on the depth first search stack, or (b) it is reachable only by traversing at least one state x (cf. Figure 6) that is on the depth first search stack. In case (a), r would have been entered into the second state space before v, due to the postorder discipline, contradicting the assumption that v is entered before r. In case (b), v is necessarily an accepting state that is reachable from itself, which contradicts the assumption that r is the first such state entered into the second state space.

State r is reachable from all states on the path from r back to itself, and therefore none of those states can already be in the second statespace when this search begins. The path therefore cannot be truncated and r is guaranteed to find itself in the successor tree. □

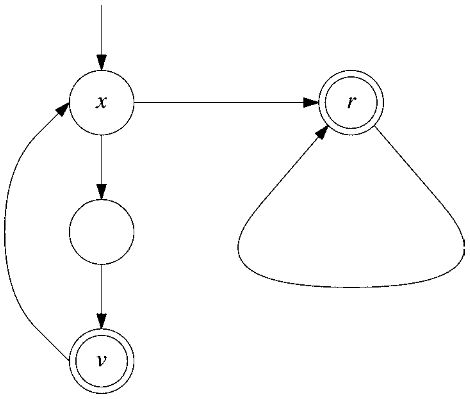


FIG. 6. States v, w and r.

Like `dfs_A`, this algorithm requires no more than two bits to be added to every reachable state in M, so the overhead remains minimal. A significant advantage of this method of model checking is also that the entire verification procedure can be performed *on-the-fly*: errors are detected during the exploration of the search space, and the search process can be cut short as soon as the first error is found. It is not necessary to first construct an annotated search space before the analysis itself can begin.

We can check non-progress properties with algorithm `dfs_B` by defining the temporal logic formula $\Diamond\Box np$, with np equal to *true* if and only if the system is in a non-progress state. The automaton that corresponds to this formula is a two-state automaton shown in Figure 7.

To perform model checking we can now take the synchronous product of the automaton in Figure 7 with a system M, and use algorithm `dfs_B` to detect the accepting ω-runs. We thus potentially incur two doublings of the search space: one due to the nested search inherent in `dfs_B` and one due to the product with the property automaton from Figure 7. The earlier algorithm `dfs_A` solves this specific

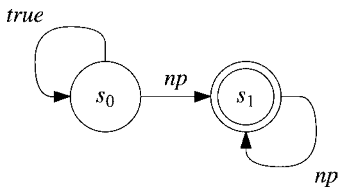


FIG. 7. Two-state automaton for $\Diamond\Box np$.

problem more efficiently by incurring only the doubling from the demon automaton. The advantage of `dfs_B` is that it can handle any type of LTL property, not just non-progress properties.

5.3 Adding Fairness

LTL is rich enough to express many fairness constraints directly, e.g., in properties of the form $(\Box \mathit{trigger}) \rightarrow (\Diamond \mathit{response})$ or $(\Box \text{guard}(t) \equiv \mathit{true}) \rightarrow (\Diamond\, \text{effect}(t))$, where t is a transition. More specific types of process fairness can also be predefined and incorporated into a model checking algorithm. Recall that the asynchronous product of finite automata that is the ultimate subject of LTL model checking is built as an interleaving of transitions from smaller automata, $M = M_1 \times M_2 \cdots M_k$. Each of the automata $M_1 \times M_2 \cdots M_k$ contributes transitions to the runs of M. Component automaton M_i, is said to be 'enabled' at state s of the global automaton M if s has at least one valid outgoing transition from M_i. We can now define two fairly standard notions of fairness.

Definition 5.1. *Weak fairness:* An ω-run σ is weakly fair if every component automaton that is enabled infinitely often contributes at least one transition infinitely often to σ.

Definition 5.2. *Strong fairness:* An ω-run σ is strongly fair if every component automaton that is enabled infinitely long contributes at least one transition infinitely often to σ.

We can include weak fairness into the nested depth-first search algorithm by using Choueka's flag construction method [10]. The following informally describes the method that is implemented in the SPIN system [30,36]. We multiply the state space $k + 1$ times, with k the number of component automata in the asynchronous product. Only the first copy of the state space retains its acceptance states; the corresponding states in the k additional copies are made non-accepting. Next we change every transition contributed by component i in the original product into a transition from the source state of that transition in copy i of the new product to the destination state of the transition in copy $i + 1$. For the last copy, $k + 1$ all transitions move back to the first copy of the state space. Any accepted ω-run in the new unfolded state space now necessarily includes transitions from each of the k component automata. We have to make one further adjustment to this procedure to account for the fact that a component automaton that is permanently not enabled after some point in the run (strong fairness) or a component automaton that is repeatedly not enabled (weak fairness) need not participate in the run. To implement weak fairness, for instance, we can add a null transition from every state s in copy i to state s in copy $i+1$ if component i is not enabled at s.

Unfolding the state space k times can be costly, but we can reduce the memory cost to a minimum by storing each copy of a state just once, and annotating it with $k+1$ bits to record in which copy of the state space the state has been encountered. If we use the cycle detection method from algorithm `dfs_A` or `dfs_B` the memory overhead per reachable state remains limited to $2(k+1)$ bits.

5.4 SPIN's On-the-Fly Implementation

The model checker SPIN performs the LTL model checking procedure on-the-fly, applying the nested depth first search algorithm `dfs_B` during the construction in a single pass of the product $B\ x\ (M_1 \times M_2 \cdots M_k)$ where B is the property automaton for the negation of an LTL formula that should be satisfied, and where x indicates synchronous product, and $\times$ asynchronous product. The construction is optionally modified for Choueka's flag construction to enforce weak fairness. SPIN derives the automaton B from an LTL formula using the algorithm from [19] with some optimizations from [18]. Optionally, the user can also specify B manually, and thus gain an increase in expressive power to the full range of ω regular properties. Alternatively, SPIN also allows the use of the conversion procedure from [18], which adds existential quantification to LTL and thereby also extends the expressive power to the ω regular properties.

The advantage of the on-the-fly procedure is that the construction of the product automaton can stop as soon as an accepting ω-run is found, having delivered proof that the system can violate the requirement. If a system contains an error it usually suffices to construct only a small portion of the product automaton. If the system satisfies the requirement the complete product must be computed. In many cases, though, the property automaton B acts as a constraint on the system, limiting the synchronous product to the executions that are relevant to the property being proven. Therefore, in those cases computing the product $B\ x\ (M_1 \times M_2 \cdots M_k)$ will be cheaper than computing $(M_1 \times M_2 \cdots M_k)$.

5.5 Partial Order Reduction

The validity of an LTL formula is insensitive to the precise order in which independent transitions from different component automata are interleaved in any given ω-run of the global automaton. SPIN uses partial order reduction to exploit this fact and to reduce the cost of a typical verification. Instead of generating a full asynchronous product that captures all possible interleavings of transitions, the model checker generates a reduced product, with only a few representatives from each class of ω runs that are indistinguishable for a given LTL formula [28,29]. This reduction can, in the best case, reduce the cost of a verification by a factor that grows exponentially with the number of component automata that are used to construct the asynchronous product. In effect, by applying partial order reduction rules one can achieve that every ω-run that is inspected by the model checker represents a large class of equivalent runs. If at least one run from each equivalence class is considered, all other runs can be ignored. The correctness of the reduction algorithm used in SPIN was verified independently with a theorem prover [9].

Partial order reduction can also be combined with other types of reduction to increase the benefits in some cases, for instance by exploiting possible symmetries in a model, e.g., [16].

5.6 Compression Techniques

Memory and time are bounded resources. The challenge in the construction of practical model checking tools is to economize the memory requirements *without* incurring unrealistic increases in runtime requirements. The model checker must be able to determine at each newly generated state from the global product automaton whether or not the state already appears in the state space (named the set `visited` in algorithms `dfs_A` and `dfs_B` above). To do so one typically stores the states in a hash-table and compares the memory image of the new state against that of previously visited states with the same hash-value. We can reduce memory use by storing all states in compressed form. The comparisons can similarly be done on the compressed memory images of the state. Better still, the compression need not be reversible. The model checker SPIN includes a number of optional lossless compression techniques, allowing for user defined trade-offs between reducing running time and increasing memory savings.

The most effective compression method SPIN supports avoids storing the set `visited` completely, and instead computes a minimized finite automaton that can recognize (optionally compressed) memory images of states as finite words over a predefined alphabet. To add a state, the automaton is updated in a way that secures its continued minimality [35]. The technique is comparable to techniques based on the use of binary decision diagrams that have proven effective in applications of model checking in hardware circuit verification, e.g., [7,13].

5.7 Bitstate Hashing

Model checking can be computationally expensive, even with aggressive use of compression and reduction techniques. Large problem sizes can easily defeat the available bounds on memory use and compute time. In cases like these it can be of great value to be able to approximate the answer to a verification problem with an accuracy that depends on the ratio by which the problem size exceeds the available resources. We can use lossy compression methods to address this problem. A good example of such a method is the bitstate hashing, or *supertrace* algorithm [24,31,36, 56]. This algorithm uses a fixed number of bits of memory per reachable state. The addresses for each of these bits are computed as hash values from the full memory image of a state, with statistically independent hash functions. In the current versions of SPIN the number of bits used can be chosen arbitrarily by the user, but it defaults to two. An elegant theoretical explanation of the working of bitstate hashing can be based on the theory of Bloom filters [4]. A short description will suffice for the purposes of this chapter. A more detailed account can be found in [15,36].

Suppose M bytes of main memory is available to store the set of `visited` states. On average workstations M is typically 2^{30} bytes, and likely to increase in coming years. We now compute N independent hash values of $\log_2 8M$ bits for each state (assuming 8 bits per byte). Instead of storing $N \log_2 8M$ bits, though, we interpret the N hash-values as a bit-address in M, and store only one single bit at each address (by changing that bit from 0 to 1). If the (compressed) memory image of a state is longer than $N \log_2 8M$ bits, this method will lose information. It is now possible that two different states generate the same N bit addresses. The model checker will then assume that a newly generated state matches a previously visited state and fail to generate the successors of the new state. Because only *visited* states are stored in this way, and not state information from the depth-first search stack (sometimes called *open* states or *stack* states), the omissions due to hash-collisions can cause the model checker to miss error states, but it cannot cause it to generate false error reports.

By virtue of the accuracy of all information saved on the depth-first search stack, the depth-first search is guaranteed to proceed correctly, generating only accurate complete execution sequences, though perhaps not all of them in the presence of bitstate hash collisions. The coverage of the search is truncated *randomly*, by a factor that depends on the amount of information that is lost in the hashing.

Because it is impossible to predict systematically which execution sequences of a model might lead to error, an unbiased random truncation of the search space turns out to be a desirable feature of this search process.

Trivially, if the minimum amount of memory to store one reachable system state without loss of information requires S bytes of memory, and if our machine has M bytes of memory available, the model checker exhausts memory after generating M/S states. If the true number of reachable states R exceeds M/S, then the *problem coverage* of that verification run is $M/(R \times S)$. If, for example, M is 10^8 bytes, S is 10^3 bytes, and R is 10^6 states, then the problem coverage can be no more than 0.1, meaning that no more than 10% of the reachable states are visited. Under the same system constraints, the bitstate hashing algorithm, storing 2 bits per state, can record up to 4 states per byte and could still achieve close to 100% coverage, given that $M/R \gg 4$. In general, when $M < R \times S$, a bitstate hashing technique almost always realizes greater problem coverage than a standard model checking run [31]. Since its first introduction in 1987 the bitstate hashing method has become a trusted technique that was adopted in almost all academic and commercial verification tools to deal with problems that exceed the normal bounds for exhaustive verification.

The bitstate hashing technique allows the user to set the range of bit addresses that can be used, typically matching the maximum amount of memory that is available for a verification run. Clearly, the larger this hash-array, the more states can be stored in it, the larger the coverage will be, and the longer it may take to complete the verification. This gives the user additional control over the search process: by artificially limiting the available hash-array, the user can obtain a very fast and very coarse approximation. By slowly increasing the size of the hash-array, the coverage and the runtime expense can be increased in a controlled manner. Each increase in coverage that fails to locate errors also increases our confidence in the likely correctness of the system. When the system contains errors, it usually takes only a small number of approximations to locate representative samples. Only when the system is correct, in the last phase of a design, more significant resources need to be invested to prove it. This *iterative search refinement* technique was used in the verification of the call processing software for a telephone switch [34].

6. Model Extraction and Abstraction

The construction of models of real-world applications for the purposes of verification is hardly novel and assuredly not restricted to the field of distributed software. There is a long tradition of the use of physical and mathematical models in civil engineering, and scientific disciplines like physics or chemistry would be almost unthinkable without theoretical models that attempt to capture aspects of nature. A model is always an abstraction: by abstracting from detail deemed immaterial to properties of interest we lose scope but gain analytical power.

It is well known that even simple properties of arbitrary software are undecidable or only semi-decidable [51]. This means that model construction for the verification of distributed software is not just an option: it is a necessary step. By defining an abstraction we can reduce a given software application to a finite model, consisting of finite state automata, that can be analyzed with the procedures outlined in this chapter. This reduction will bring a loss of information, so it has to be chosen in such a way that relevant information is preserved and irrelevant detail removed. What is *relevant* and what is not depends on the properties that we are interested in proving.

We can remove the detail in such a way that the soundness of the model checking procedure itself is not endangered [1,6,12,38]. This means that if the model checker indicates that a model satisfies a property, the original software necessarily also satisfies the property. Conversely, if the model checker generates an error, the error sequence can be checked against the original software to determine its validity. If it is valid, an error in the application has been exposed. If it is not valid, we have obtained proof that the abstraction was chosen incorrectly, and the error sequence itself can be used to determine unambiguously how the abstraction should be revised.

A simple abstraction method of this type was used in the application of the SPIN model checker to complex call processing software for a commercial telephone switch, e.g., [33,34]. With this method, a parser automatically *extracts* an annotated control flow skeleton from the source code of the application. A lookup table defines precisely which statements from the program should be omitted from the model (i.e., replaced with *skip* statements), which should be abstracted with user-defined functions, and which should be preserved within the model. Within the model, every statement from the original source code of the application is mapped into a finite domain and represented as a transition in an extended finite state automaton, expressed in guards and effects that operate on finite data objects, often with a reduced range of possible values.

Function calls have to be treated with some care, in the interest of controlling the complexity of a model. The very presence of a function call, however, can be taken as a hint from the programmer that an abstraction can be made. In the call processing application function calls were treated like statements: they were either omitted if the functionality provided was outside the scope of the verification, or they were abstracted, either with a small inline routine or with a non-deterministic choice of the possible return values. As an example, a routine that determines the availability of a resource, like a tone circuit, is best abstracted with a non-deterministic choice between the two possible result values: available or not-available. No useful gain is made if we were to include more detail than this. As another example, the call of a routine that issues billing records can be omitted from the model if billing is not the focus of the verification.

In the call processing application the focus was on the verification of correct feature behavior for the telephone switch. Requirements for over twenty different feature packages, such as call waiting, call forwarding, call screening, conference calling, etc., are specified in Telcordia standards for call processing [49]. Each relevant property was formalized in linear temporal logic. Aspects of the system that were outside the scope of our verification effort (e.g., billing, process management, memory management, device driver code) was mechanically omitted from the model, and helped to reduce the cost of the verification of the remaining aspects of the code.

The advantage of this method is that it can be almost completely automated. When a new version of the source code is prepared, the model extraction program can prompt the user to provide missing and redundant entries in the lookup table. Once the lookup table has been updated, the model checking process can be repeated with a new accurate model being extracted from the source code of the application mechanically, typically in a fraction of a second.

The verification method allowed us to track the evolution of the call processing code for this application over a period of 18 months. The source code for the application grew fivefold in size in this period, and went through approximately 300 different versions, often changing daily. Approximately 75 critical errors were intercepted with the model checking technique we have outlined, at an early stage of the design, giving a clear indication of the considerable power and value of software model checking techniques. Many of the errors found involved subtle race conditions in the code that could disturb required functionality. Such errors are virtually impossible to find with conventional testing techniques.

6.1 Other Uses of Abstraction

The model extraction method sketched above was greatly facilitated by a relatively recent extension of the SPIN model checker that allows for the inclusion of *embedded C code* inside higher level verification models [36]. This capability to use verify embedded C code fragments can be used in a number of other ways to increase the power of the model checking approach. In [37] a method is described that allow SPIN to verify a code module at implementation level, by compiling it with, and linking it to the model checker. The model checker now generates the non-deterministic input sequences for the code module and keeps track of the code's state. To achieve this, the user identifies the concrete data objects inside the code module that contain state information. The user can at this point also define an abstraction function, in C code, that takes the concrete representation of the state information and abstracts it for use by the model checker. In this way we can use SPIN to combine model abstraction without model extraction, which may prove to be a very effective technique for handling large verification problems in years to come.

7. Perspective

The software model checking techniques that we have reviewed in this chapter are based on finite automata, linear temporal logic, depth-first search, partial order reduction, and explicit state representation combined with powerful memory management techniques. It has been successfully applied in many domains, but typically to verification problems that involve asynchronous threads of computation from software systems e.g., [46]. An overview of applications can also be found in [30].

It is interesting to compare the general framework used in SPIN with the one that has been developed for hardware circuit verification. The most commonly used logic in hardware verification is the branching time logic CTL [11], the search strategy is often breadth-first, instead of partial order reduction techniques one uses BDD based algorithms [7], and instead of explicit state representation one uses symbolic model checking [41]. These differences in approach to the verification problem can be understood better if we look at some of the differences between the two domains of application. Hardware is typically clock-driven, operating in a synchronous fashion, while the processes in a distributed system are necessarily asynchronous. At the hardware level information travels as signals, in software applications the information is represented, manipulated, and moved in composite data structures. A bit level representation is clearly not helpful for these types of objects. The structure of a hardware system, finally, can often be defined statically, while in a software system one must deal with dynamically growing and shrinking numbers of asynchronous processes and data objects. These differences mean that few of the reduction techniques that work well in software model checkers show benefit when used in hardware model checkers, and vice versa.

ACKNOWLEDGEMENTS

The research described in this chapter was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[1] Abadi M., Lamport L., “The existence of refinement mappings”, *Theoretical Computer Science* **82** (2) (May 1991) 253–284.

[2] Aho A.V., Hopcroft J.E., Ullman J.D., *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.

[3] Bartlett K.A., Scantlebury R.A., Wilkinson P.T., “A note on reliable full-duplex transmission over half-duplex lines”, *Comm. of the ACM* **12** (5) (1969) 260–265.

[4] Bloom B.H., “Spacetime trade-offs in hash coding with allowable errors”, *Comm. of the ACM* **13** (7) (2004) 422–426.

[5] Bochmann G.V., “Finite state description of communications protocols”, Publication No. 236, Département d’informatique, Université de Montreal, July 1976.

[6] Bozga D.M., "Verification symbolique pour les protocoles de communication", PhD Thesis (in French), University of Grenoble, France, December 1999, Chapter 4.

[7] Bryant E., "Graph-based algorithms for Boolean function manipulation", *IEEE Trans. on Computers* **C-35** (8) (August 1986) 677–691.

[8] Büchi J.R., "On a decision method in restricted second order arithmetic", in: *Proc. Internat. Congr. on Logic, Methodology and Philosophy of Science*, Stanford Univ. Press, Stanford, CA, 1960, pp. 1–11.

[9] Chou C.-T., Peled D., "Verifying a model-checking algorithm", in: *Proc. Tools and Algorithms for the Construction and Analysis of Systems, TACAS, March 1996, Passau, Germany*, in: *Lecture Notes in Comput. Sci.*, vol. 1055, Springer-Verlag, Berlin/New York, 1996, pp. 241–257.

[10] Choueka Y., "Theories of automata on ω-tapes: a simplified approach", *Journal of Computer and System Science* **8** (1974) 117–141.

[11] Clarke E.M., Emerson E.A., "Synthesis of synchronization skeletons for branching time temporal logic", in: *Workshop on Logic of Programs, Yorktown Heights, NY, May 1981*, in: *Lecture Notes in Comput. Sci.*, vol. 131, Springer-Verlag, Berlin/New York, 1982.

[12] Clarke E.M., Grumberg O., Long D.E., "Model checking and abstraction", *ACM-TOPLAS* **16** (5) (September 1994) 1512–1542.

[13] Clarke E.M., Grumberg O., Peled D., *Model Checking*, MIT Press, Cambridge, MA, 1999.

[14] Courcoubetis C., Vardi M.Y., Wolper P., Yannakakis M., "Memory efficient algorithms for the verification of temporal properties", in: *Formal Methods in Systems Design*, vol. I, 1992, pp. 275–288. First published in: *Proc. 2nd Conference on Computer Aided Verification*, Rutgers University, New Jersey, June 1990.

[15] Dillinger P.C., Manolios P., "Fast and accurate bitstate verification for SPIN", in: *Proc. 11th* SPIN *Workshop, Barcelona, Spain*, in: *Lecture Notes in Comput. Sci.*, vol. 2989, Springer-Verlag, Berlin/New York, April 2004.

[16] Emerson E.A., Jha S., Peled D., "Combining partial order reduction and symmetry reduction", in: *Proc. Tools and Algorithms for the Construction and Analysis of Systems, TACAS, Enschede, The Netherlands, 1997*, in: *Lecture Notes in Comput. Sci.*, vol. 1217, Springer-Verlag, Berlin/New York, 1997, pp. 19–34.

[17] Etessami K., "Stutter-invariant languages, ω-automata, and temporal logic", in: *Proc. Conf. on Computer Aided Verification, CAV*, 1999, pp. 236–248.

[18] Etessami K., Holzmann G.J., "Optimizing Büchi automata", in: *Proc. CONCUR2000*, in: *Lecture Notes in Comput. Sci.*, vol. 1877, Springer-Verlag, Berlin/New York, August 2000, pp. 153–167.

[19] Gerth R., Peled D., Vardi M., Wolper P., "Simple on-the-fly automatic verification of linear temporal logic", in: *Proc. Symp. on Protocol Specification, Testing, and Verification, Warsaw, Poland, 1995*, Chapman and Hall, London, 1995, pp. 3–18.

[20] Godefroid P., Holzmann G.J., "On the verification of temporal properties", in: *Proc. Internat. Conf. on Protocol Specification, Testing, and Verification, Liege, Belgium*, May 1993, pp. 109–124.

[21] Hajek J., "Automatically verified data transfer protocols", in: *Proc. 4th ICCC, Kyoto*, 1978, pp. 749–756.

[22] Holzmann G.J., "PAN: a protocol specification analyzer", Technical Report TM81-11271-5, AT&T Bell Laboratories, March 1981.
[23] Holzmann G.J., "A theory for protocol validation", *IEEE Trans. on Computers* **C-31** (8) (1982) 730–738.
[24] Holzmann G.J., "An improved protocol reachability analysis technique", *Software, Practice and Experience* **18** (2) (February 1988) 137–161.
[25] Holzmann G.J., Patti J., "Validating SDL specifications: an experiment", in: *Proc. Internat Conf. on Protocol Specification, Testing, and Verification, Twente, Netherlands*, June 1989, pp. 317–326.
[26] Holzmann G.J., "SPIN—A protocol analyzer", in: *Unix Research System, vol. II, Papers*, tenth ed., Saunders College Publ., January 1990, pp. 423–429.
[27] Holzmann G.J., *Design and Validation of Computer Protocols*, Prentice Hall, Englewood Cliffs, NJ, 1991.
[28] Holzmann G.J., Peled D., "An improvement in formal verification", in: *Proc. Conf. on Formal Description Techniques, FORTE, Bern, Switzerland*, October 1994, pp. 177–194.
[29] Holzmann G.J., Peled D., Yannakakis M., "On nested depth-first search", in: *Proc. 2nd Spin Workshop, Rutgers Univ., New Brunswick, NJ, August 1996*, in: *DIMACS*, vol. 32, American Mathematical Society, Providence, RI, 1996.
[30] Holzmann G.J., "The model checker SPIN", *IEEE Trans. on Software Engineering* **23** (5) (May 1997) 279–295.
[31] Holzmann G.J., "An analysis of bitstate hashing", *Formal Methods in System Design* **13** (3) (November 1998) 287–305.
[32] Holzmann G.J., "Designing executable abstractions", in: *Proc. Formal Methods in Software Practice, Clearwater Beach, FL*, ACM Press, March 1998.
[33] Holzmann G.J., Smith M.H., "A practical method for the verification of event driven systems", in: *Proc. Internat. Conf. on Software Engineering, ICSE99, Los Angeles*, May 1999, pp. 597–608.
[34] Holzmann G.J., Smith M.H., "Software model checking—extracting verification models from source code", in: *Formal Methods for Protocol Engineering and Distributed Systems*, Kluwer Academic, Dordrecht/Norwell, MA, 1999, pp. 481–497.
[35] Holzmann G.J., Puri A., "A minimized automaton representation of reachable states", *Software Tools for Technology Transfer* **2** (3) (November 1999) 270–278.
[36] Holzmann G.J., *The* SPIN *Model Checker—Primer and Reference Manual*, Addison–Wesley, Boston, MA, 2004.
[37] Holzmann G.J., Joshi R., "Model-driven software verification", in: *Proc. 11th* SPIN *Workshop, Barcelona, Spain, April 2004*, in: *Lecture Notes in Comput. Sci.*, vol. 2989, Springer-Verlag, Berlin/New York, 2004, pp. 77–92.
[38] Kurshan R.P., "Homomorphic reduction of coordination analysis", in: *Mathematics and Applications*, in: *IMA Series*, vol. 73, Springer-Verlag, Berlin/New York, 1995, pp. 105–147.
[39] Manna Z., Pnueli A., "Tools and rules for the practicing verifier", Stanford University, Report STAN-CS-90-1321, July 1990, 34 p.
[40] Manna Z., Pnueli A., *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, Berlin/New York, 1991.

[41] McMillan K.L., *Symbolic Model Checking*, Kluwer Academic, Boston, 1993.

[42] Moore G.E., “Cramming more components onto integrated circuits”, *Electronics* **19** (April 1965).

[43] Peled D., “On projective and separable properties”, in: *Colloquium on Trees in Algebra and Programming, Edinburgh, Scotland, 1994*, in: *Lecture Notes in Comput. Sci.*, vol. 787, Springer-Verlag, Berlin/New York, 1994, pp. 291–307.

[44] Pnueli A., “The temporal logic of programs”, in: *Proc. 18th IEEE Symposium on Foundations of Computer Science, 1977*, Providence, RI, pp. 46–57.

[45] Queille J.P., Sifakis J., “Specification and verification of concurrent systems in Cesar”, in: *Proc. of Fifth Internat. Symp. on Programming*, 1981, pp. 337–350.

[46] Schneider F., Easterbrook S.M., Callahan J.R., Holzmann G.J., “Validating requirements for fault tolerant systems using model checking”, in: *Proc. Internat. Conf. on Requirements Engineering, ICRE, Colorado Springs, CO*, IEEE, April 1998, pp. 4–14.

[47] Tanenbaum A.S., *Computer Networks*, first ed., Prentice Hall, Englewood Cliffs, NJ, 1981 (second edition 1988).

[48] Tarjan R.E., “Depth first search and linear graph algorithms”, *SIAM J. Computing* **1** (2) (1972) 146–160.

[49] *LATA Switching Systems Generic Requirements (LSSGR)*, FR-NWT-000064, 1992 edition. Feature requirements, including: *SPCS Capabilities and Features*, SR-504, Issue 1, March 1996. Telcordia/Bellcore.

[50] Thomas W., “Automata on infinite objects”, in: Van Leeuwen J. (Ed.), *Handbook of Theoretical Computer Science*, vol. B, Elsevier, Amsterdam, 1990, pp. 133–187.

[51] Turing A.M., “On computable numbers, with an application to the Entscheidungsproblem”, *Proc. London Math. Soc. Ser. 2* **42** (1936) 230–265, see p. 247.

[52] Vardi M.Y., Wolper P., “An automata-theoretic approach to automatic program verification”, in: *Proc. Symp. on Logic in Comput. Sci., Cambridge*, June 1986, pp. 322–331.

[53] West C.H., “General technique for communications protocol validation”, *IBM J. Res. Develop.* **22** (3) (1978) 393–404.

[54] West C.H., Zafiropulo P., “Automated validation of a communications protocol: the CCITT X. 21 recommendation”, *IBM J. Res. Develop.* **22** (1) (1978) 60–71.

[55] Wolper P., Vardi M.Y., Sistla A.P., “Reasoning about infinite computation paths”, in: *Proc. 24th IEEE Symp. on Foundations of Comput. Sci., Tucson*, 1983, pp. 185–194.

[56] Wolper P., Leroy D., “Reliable hashing without collision detection”, in: *Proc. Conf. on Computer Aided Verification, Crete, June 1993*, in: *Lecture Notes in Comput. Sci.*, vol. 697, Springer-Verlag, Berlin/New York, 1993, pp. 59–70.

Early Cognitive Computer Vision[1]

JAN-MARK GEUSEBROEK

Intelligent Systems Lab Amsterdam, Informatics Institute
Faculty of Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam
The Netherlands
mark@science.uva.nl

Abstract
This chapter outlines computational models for the first stages of visual cognition. For both biological and technical systems, we are examining which architectural components are necessary in such systems, and how experience can be acquired and used to steer perceptual interpretation. Since human perception has evolved to interpret the structure of the world around us, a necessary boundary condition of the vision system must be the common statistics of natural images. Searching for generality, it is observed that a limited set of physical laws of image formation will impress common statistics on the images offered to the eye as sensory input. The physical laws are largely scene and domain independent, as they cover the universally applicable laws of light reflectance from materials.

The chapter focuses on the physical and statistical constrains in the sensory input, and how this can be exploited to construct cognitive vision systems. Visual cognition may be based on a weak description of the important features in the scene, as long as mutual correspondence between observation and objects in the world is maintained. For such a computational theory, the first few steps will be outlined: visual measurement, invariant representation, and focal attention.

[1]This work is sponsored by the Netherlands Organization for Scientific Research (NWO).

1. Introduction

Cognitive vision is the processing of visual sensory information in order to act and react in a dynamic environment. The human visual system is an example of a very well-adapted cognitive system, shaped by millions of years of evolution. Vision dominates our senses for personal observation, societal interaction, and cognitive skill acquisition. Understanding visual perception to such a level of detail that a machine could be designed to mimic it is a long-term goal, and one which is unlikely to be achieved within the next few decades. Since vision requires 30% of our brain capacity, and what is known about it points to it being a highly distributed task interwoven with many other modules, it is clear that modelling human vision—let alone understanding it—is still a long way off. However, as computers are expected in the next twenty years to reach the capacity of the human brain, now is the time to start thinking about methods of constructing modules for cognitive vision systems.

This chapter outlines computational models for visual cognition. For both biological and technical systems, we are examining which architectural components are necessary in such systems, and how experience can be acquired and used to steer perceptual interpretation. Since human perception has evolved to interpret the structure of the world around us, a necessary boundary condition of the vision system must be the common statistics of natural images. As such, a cognitive sensory system will adapt itself to the outside world, specifically to the stochastics of the input signals [1]. When that point of departure is accepted, we note that the statistics of the sensory input are dominated by physical laws of image formation [17,26,21].

Searching for generality, it is observed that a limited set of physical laws of image formation will impress common statistics on the images offered to the eye as sensory input. The physical laws are largely scene and domain independent, as they cover the universally applicable laws of light reflectance from materials. The chapter focuses on the physical and statistical constraints in the sensory input, and how this can be exploited to construct cognitive vision systems. Visual cognition may be based on a weak description of the important features in the scene, as long as mutual correspondence between observation and objects in the world is maintained. For such a computational theory, the first few steps will be outlined: visual measurement, invariant representation, and focal attention.

1.1 Visual Measurements

Equipped with a hypothetical infinitely precise sensor we would see the world around us up to the microscopic level of detail. This view in all its full complexity is useless to the observer as it would swamp the perceptual processing system. To escape the influx of so much information, a large reduction in information is implemented at the retina where the outside world is integrated over discrete sensory receptive fields. The observation by receptors of finite size imposes spatial coherence to the picture while reducing the complexity of the observed scene.

There is only a limited set of measurements that are sensibly calculated from visual input. The set must be derived from the signs to be probed in visual data by integration over receptive fields, be it a spatial area, or a spectral bandwidth, or an interval in time. The integration over the visual input is weighted by a sensitivity curve, effectuated by a receptive field, thereby emphasizing various aspects of the stimulus. The integration sensitivity determines which physical parameter is probed.

Neurobiological studies have found a dozen or so different types of receptive fields in the visual system of primates. As the receptive fields have evolved to capture the world around us, they are likely to be dual to our physical surrounding. These fields must be derived from the statistical structures that are probed in visual data. In Section 2, we have initially derived several receptive field assemblies, each characterizing a physical quantity from the visual stimulus.

1.2 Physical Constraints: Invariance

As the visual stimulus involves a very reductive projection of the physical world onto a limited set of visual measurements, only correlates to relevant entities can be measured directly. Invariants transform visual measurements to true physical quantities, thereby removing those degrees of freedom not relevant for the observer. Hence, a first source of knowledge involved in visual interpretation is the incorporation of physical laws. In Section 3, we have used color invariance as a well-founded principle to separate color into its correlates of material reflection, being illuminant color, highlights, shadows, shading components and the true object reflectance. Such invariants allow a system to be sensitive to obstacles, while at the same time being insensitive to shadows. The representation of the visual input into a plurality of invariant representations is a necessary information-reduction stage in any cognitive vision system.

1.3 Statistical Constraints: Natural Image Statistics and Focal Attention

Natural images are highly structured in their spatial configuration. Where one would expect a different spatial distribution for every image, as each image has a different spatial layout, Section 4 shows that the spatial statistics of recorded images can be explained by a single process of sequential fragmentation.

The statistical structure of the world introduces subjective perception of our environment. For example, the a-priori occurrence of objects at certain distances determines our subjective estimation of distances in the world [71]. Furthermore, parts of an image which deviate from the common statistics around us are likely to contain perceptually salient details, leading to focal attention mechanisms. Hence, our motivation to study the statistical regularities in natural images as it implies a better understanding of cognitive vision systems.

The spatial statistics of large ensembles of natural images are known to be scale invariant [11,55]. That is, when examining the marginal distribution of derivative filters or gradient magnitude, an inverse power-law distribution is found in the Fourier domain. However, the statistics of individual images may vary across scale. Consequently the statistical properties for individual images may be affected by the observation at finite resolution.

To limit the enormous computational burden arising from the complex task of interpretation and learning, any efficient general vision system will ignore the common statistics in its input signals. Hence, the apparent occurrence of invariant representations decides what is salient and therefore requires attention. Such focal attention is a necessary selection mechanism in any cognitive vision system, critically reducing both the processing requirements and the complexity of the visual learning space, and effectively limiting the interpretation task.

Expectation about the scene is then inevitably used to steer attention selection. Hence, focal attention is not only triggered by visual stimuli, but is affected by knowledge about the scene, initiating conscious behavior. In this principled way, knowledge and expectation may be included at an early stage in cognitive vision, see Figure 1. In the following sections, we discuss the main building blocks of this scheme in more detail.

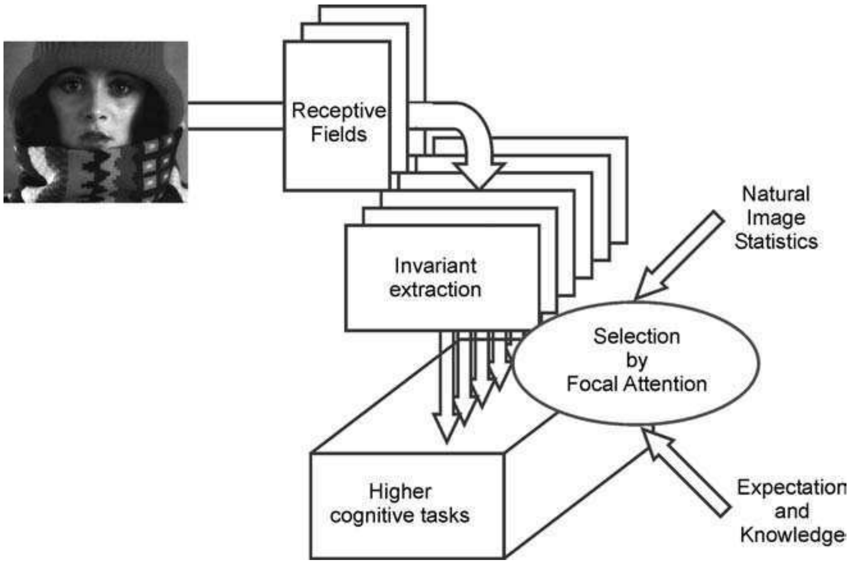


FIG. 1. Early cognitive vision system overview. (See Color Plate Section, Plate 5.)

2. Visual Measurements

Images are only defined in terms of observations. An image is observed by integrating for a certain time over some spatial extent and over a spectral bandwidth. Hence, physical realizable measurements inherently imply integration over spectral and spatial dimensions. Before observation, a color image may be regarded as a three dimensional energy density function $E(x, y, \lambda)$, where (x, y) denotes the spatial coordinate and λ denotes the wavelength. Observation of the energy density $E(x, y, \lambda)$ boils down to correlation of the incoming signal with a measurement probe $p(x, y, \lambda)$ (see Figure 2),

$$\widehat{E}(x, y, \lambda) = \iiint E(x, y, \lambda)p(x, y, \lambda)\,dx\,dy\,d\lambda. \tag{1}$$

Note that for a linear spatially shift invariant imaging system, correlation boils down to convolution. Furthermore, for a system probing the spatial, temporal, and spectral dimensions simultaneously, the measurement function should be separable per dimension, $p(x, y, \lambda) = p_{xy}(x, y)p_{\lambda}(\lambda)$. The yet unknown measurement function $p(x, y, \lambda)$ estimates quantities of the energy density $E(x, y, \lambda)$. Such a measurement function is referred to as “receptive field.”

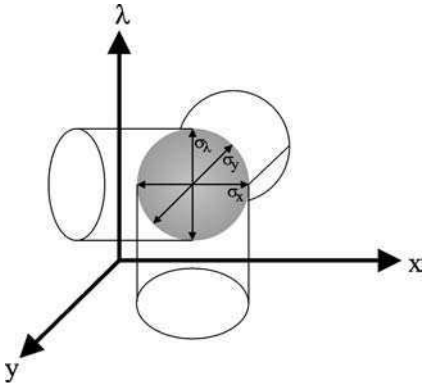


FIG. 2. The probes for spatial color consists of probing the product of the spatial and the spectral energy space with a given aperture. (See Color Plate Section, Plate 6.)

The theory of scale-space [39,69,64,47] adheres to the fact that observation and scale are intervened; a measurement is performed at a certain resolution. Scale-space theory formalizes the fact that the probe needs to have a finite resolution. Furthermore, common image processing sense tells us that the grey-value of a particular pixel is not a meaningful entity. The value 42 by itself tells us little about the meaning of the pixel in its environment. It is the local spatial structure of an image that has a close geometrical interpretation [39]. The difference between neighboring pixels exhibit this structure. Differentiation is one of the fundamental operations in image processing, and one which is nicely defined [16] in the context of scale-space.

Modern analysis of color observation has started in colorimetry where the spectral content of tri-chromatic stimuli are matched by a human, resulting in the well-known XYZ color matching functions [70]. However, from the pioneering work of Land [45] we know that a perceived color does not directly correspond to the spectral content of the stimulus; there is no one-to-one mapping of spectral content to perceived color. For example, a colorimetry purist will not consider brown to be a color, but as computer vision practisers would like to be able to define brown in an image when searching on colors. Hence, it is not only the spectral energy distribution coding color information, but also the spatial configuration of colors.

In this section, we outline several receptive field measurements to extract the spatial structure of color and grey value images. Scale space theory suggests that receptive fields should have a Gaussian shape in order to prevent the probe from adding extra details to the function when observed at a coarser scale [39], in one dimension given by

$$G(x; x_0, \sigma_x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{1}{2}\frac{(x - x_0)^2}{\sigma_x^2}\right\}. \tag{2}$$

We consider the Gaussian as a general probe for the measurement of spatio-spectral differential quotients.

2.1 Opponent Color Receptive Fields

Measurement of the spectral energy distribution with a Gaussian aperture yields a weighted integration over the spectrum. The observed energy in the Gaussian color model, at infinitely small spatial resolution, approaches in second order to

$$\widehat{E}^{\sigma_\lambda}(\lambda) = \widehat{E}^{\lambda_0,\sigma_\lambda} + \lambda \widehat{E}_\lambda^{\lambda_0,\sigma_\lambda} + \frac{1}{2}\lambda^2 \widehat{E}_{\lambda\lambda}^{\lambda_0,\sigma_\lambda} + \cdots \tag{3}$$

where

$$\widehat{E}^{\lambda_0,\sigma_\lambda} = \int E(\lambda) G(\lambda; \lambda_0, \sigma_\lambda)\, d\lambda \tag{4}$$

measures the spectral intensity,

$$\widehat{E}_\lambda^{\lambda_0,\sigma_\lambda} = \int E(\lambda) G_\lambda(\lambda; \lambda_0, \sigma_\lambda)\, d\lambda \tag{5}$$

measures the first order spectral derivative, and

$$\widehat{E}_{\lambda\lambda}^{\lambda_0,\sigma_\lambda} = \int E(\lambda) G_{\lambda\lambda}(\lambda; \lambda_0, \sigma_\lambda)\, d\lambda \tag{6}$$

measures the second order spectral derivative. Further, G_λ and $G_{\lambda\lambda}$ denote derivatives of the Gaussian with respect to λ. Note that, throughout the thesis, we assume scale normalized Gaussian derivatives to probe the spectral energy distribution (see Figure 3).

Definition 1 (*Gaussian color model*). The Gaussian color model measures the coefficients $\widehat{E}^{\lambda_0,\sigma_\lambda}$, $\widehat{E}_\lambda^{\lambda_0,\sigma_\lambda}$, $\widehat{E}_{\lambda\lambda}^{\lambda_0,\sigma_\lambda}$, ... of the Taylor expansion of the Gaussian weighted spectral energy distribution at λ_0 and scale σ_λ.

One might be tempted to consider a higher, larger than two, order structure of the smoothed spectrum. However, the subspace spanned by the human visual system is of dimension 3, and hence higher order spectral structure cannot be observed by the human visual system.

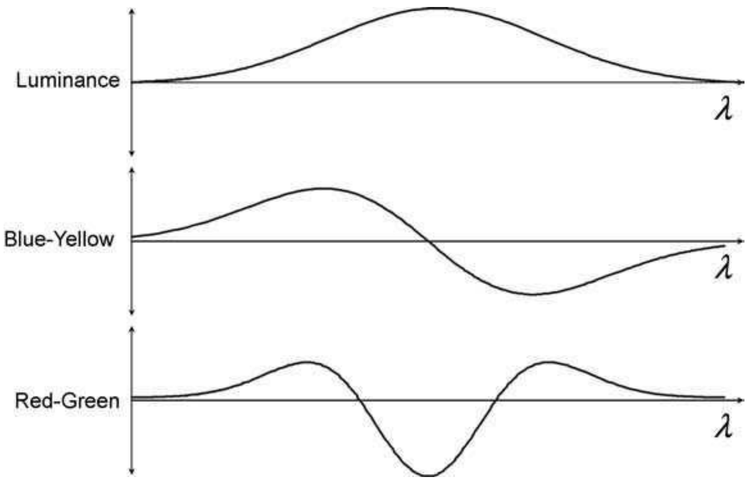


FIG. 3. The Gaussian color sensitivities as function of wavelength. Three-stimulus values are obtained by correlating the sensitivities with the incoming spectral energy distribution. (See Color Plate Section, Plate 7.)

Introduction of spatial extent in the Gaussian color model yields a local Taylor expansion at wavelength λ_0 and position $\vec{x}_0$. Each measurement of a spatio-spectral energy distribution has a spatial as well as spectral resolution. The measurement is obtained by probing an energy density volume in a three-dimensional spatio-spectral space, where the size of the probe is determined by the observation scale σ_λ and σ_x (recall Figure 2). It is directly clear that we do not separately consider spatial scale and spectral scale, but actually probe an energy density volume in the three-dimensional spectral-spatial space where the "size" of the volume is specified by the observation scales.

We can describe the observed spatial-spectral energy density $\widehat{E}(\lambda, \vec{x})$ of light as a Taylor series for which the coefficients are given by the energy convolved with Gaussian derivatives:

$$\widehat{E}(\lambda, \vec{x}) = \widehat{E} + \begin{pmatrix} \vec{x} \\ \lambda \end{pmatrix}^{\mathrm{T}} \begin{bmatrix} \widehat{E}_{\vec{x}} \\ \widehat{E}_{\lambda} \end{bmatrix} + \frac{1}{2} \begin{pmatrix} \vec{x} \\ \lambda \end{pmatrix}^{\mathrm{T}} \begin{bmatrix} \widehat{E}_{\vec{x}\vec{x}} & \widehat{E}_{\vec{x}\lambda} \\ \widehat{E}_{\lambda\vec{x}} & \widehat{E}_{\lambda\lambda} \end{bmatrix} \begin{pmatrix} \vec{x} \\ \lambda \end{pmatrix} + \cdots \tag{7}$$

where

$$\widehat{E}_{\vec{x}^i \lambda^j}(\lambda, \vec{x}) = E(\lambda, \vec{x}) * G_{\vec{x}^i \lambda^j}(\lambda, \vec{x}; \sigma_\lambda, \sigma_x). \tag{8}$$

Here, $G_{\vec{x}^i \lambda^j}(\lambda, \vec{x}; \sigma_\lambda, \sigma_x)$ are the spatio-spectral probes, or color receptive fields. The coefficients of the Taylor expansion of $\widehat{E}(\lambda, \vec{x})$ represent the local image structure completely. Truncation of the Taylor expansion results in an approximate representation, optimal in least squares sense.

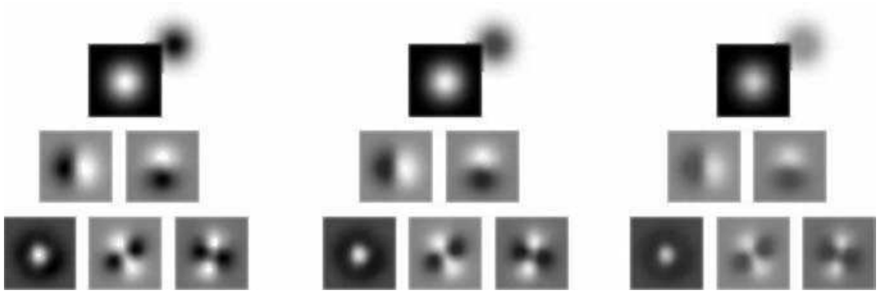


FIG. 4. The color receptive fields up to second spatial derivative order (polar axes system). Note that the luminance receptive fields are compatible with the well-known Gaussian intensity scale-space operators. The yellow–blue and red–green receptive fields extent this scale-space and represent the Hering opponent color receptive fields. (See Color Plate Section, Plate 8.)

For human vision, it is known that the Taylor expansion is spectrally truncated at second order. Hence, higher order derivatives do not affect color as observed by the human visual system. Therefore, three receptive field families should be considered; the luminance receptive fields as known from intensity scale-space [40] extended with a yellow–blue receptive field family measuring the first order spectral derivative, and a red–green receptive field family probing the second order spectral derivative. When centering the spectral Gaussians at $\lambda_0 \simeq 520$ nm and choosing a bandwidth of $\sigma_\lambda \simeq 55$ nm, the Gaussian color model closely resembles [25,27] the Hering opponent color model [33]. For human vision, the Taylor expansion for luminance is spatially truncated somewhere around the fourth order [72]. The combined color and spatial receptive fields are illustrated in Figure 4.

In practise, spectral differential quotients are obtained by a linear combination of given (RGB) sensitivities, whereas spatial differential quotients are obtained by convolution with Gaussian derivative filters. When camera response is linearized, a RGB-camera approximates the CIE 1964 XYZ basis for colorimetry by the linear transform [34]

$$\begin{bmatrix} \widehat{X} \\ \widehat{Y} \\ \widehat{Z} \end{bmatrix} = \begin{pmatrix} 0.62 & 0.11 & 0.19 \\ 0.3 & 0.56 & 0.05 \\ -0.01 & 0.03 & 1.11 \end{pmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{9}$$

The best linear transform from XYZ values to the Gaussian color model is given by Geusebroek et al. [25]

$$\begin{bmatrix} \widehat{E} \\ \widehat{E}_\lambda \\ \widehat{E}_{\lambda\lambda} \end{bmatrix} = \begin{pmatrix} -0.48 & 1.2 & 0.28 \\ 0.48 & 0 & -0.4 \\ 1.18 & -1.3 & 0 \end{pmatrix} \begin{bmatrix} \widehat{X} \\ \widehat{Y} \\ \widehat{Z} \end{bmatrix}. \tag{10}$$

The product of Equations (9) and (10) gives the desired implementation of the Gaussian color model in RGB terms,

$$\begin{bmatrix} \widehat{E} \\ \widehat{E}_\lambda \\ \widehat{E}_{\lambda\lambda} \end{bmatrix} = \begin{pmatrix} 0.06 & 0.63 & 0.27 \\ 0.3 & 0.04 & -0.35 \\ 0.34 & -0.6 & 0.17 \end{pmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{11}$$

Note that we try to achieve derivative filters in the spectral domain by transforming the spectral responses as given by the RGB-filters. The transformed filters may be imperfect, but are likely to offer accurate estimates of differential measurements. When the spectral responses of the RGB-filters are known, a better transform can be obtained.

2.2 Anisotropic Receptive Fields: Edge and Line Detection

One of the most important tasks in computer vision is edge and line detection. The detection and orientation analysis of such oriented structures is considered non trivial when disturbing influences have to be ignored, like gaps in dashed lines, weak (local) image contrast or heavy corruption by noise, or clutter in the background. In these cases, one would like to have a detection method which ignores the distorting data next to the edge or line, while accumulating evidence of the line data along its orientation. Hence, taking advantage of the anisotropic nature of lines.

Orientation analysis is often approached by steerable filters. Freeman and Adelson [18] put forward the conditions under which a filter can be tuned to a specific orientation by making a linear combination of basis filters. Their analysis included orientation tuning of the xy-separable first order isotropic Gaussian derivative filter.

Line detection is particularly difficult when lines run close together or cross each other, as is the case in engineering drawings. Under these circumstances, isotropic filtering strategies as used in, e.g., [3,38,62] are not sufficient. Isotropic smoothing causes parallel lines to be blurred into one single line. Crossing lines are not well detected due to the marginal orientation selectivity of the Gaussian filter [52]. The use of anisotropic Gaussian filters yields superior orientation selectivity, and superior discrimination between thin and thick lines.

For historical reasons, linear scale-space is often put forward as dealing with isotropic Gaussian's, imposed by simplicity of derivation and efficiency in computation [39]. The assumption of isotropy for front-end vision [15,39,47,64] does not imply the scale-space operator to be isotropic, rather imposes the complete sampling of all possible orientations of the scene. The notion of orientation sampling suggests a combined scale and orientation space [40,38,65]. The problem of line detection is clearly anisotropic in nature. The use of anisotropic Gaussian filters yields superior orientation selectivity, and superior discrimination between thin and thick lines. The huge computational burden involved in exhaustive filtering until now has prohibited application of orientation scale-space analysis. Hence, it is important to consider the computational efficiency of anisotropic Gauss filtering.

An orientation scale-space $\mathcal{F}(x, y, \sigma_u, \sigma_v, \theta)$ of an image f is obtained by applying an anisotropic Gaussian filter bank to the image f containing all orientations θ and scales σ_u, σ_v of interest,

$$\mathcal{F} = f(x, y) * G(x, y; \sigma_u, \sigma_v, \theta). \tag{12}$$

The Gaussian anisotropic filter is given by

$$G(x, y; \sigma_u, \sigma_v, \theta) = \frac{1}{2\pi\sigma_u\sigma_v} \exp\left\{-\frac{1}{2}\left(\frac{(x\cos\theta + y\sin\theta)^2}{\sigma_u^2} + \frac{(-x\sin\theta + y\cos\theta)^2}{\sigma_v^2}\right)\right\} \tag{13}$$

where σ_u represents the scale in the direction of θ, and σ_v orthogonal to θ (see Figure 5).

Theoretically, in two-dimensions, line points are detected by considering the second order directional derivative in the gradient direction [62]. For a line point, the first order directional derivative perpendicular to the line vanishes, where the second order directional derivative exhibits an extremum. Hence, the second order directional derivative perpendicular to the line, normalized by the line brightness is a measure of line contrast [23],

$$\lambda(x, y, \sigma_u, \sigma_v, \theta) = f(x, y) * G_{vv}(x, y; \sigma_u, \sigma_v, \theta)\frac{\sigma_v^2}{b} \tag{14}$$

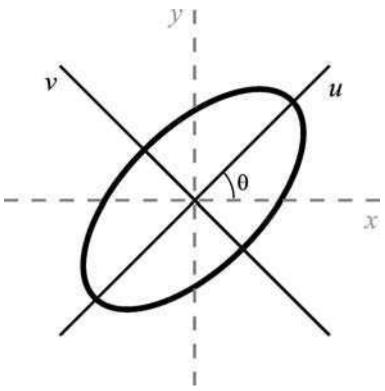


FIG. 5. Coordinate system for directional filtering for line detection.

where line brightness b is given by

$$b = f(x, y) * G(x, y; \sigma_u, \sigma_v, \theta). \tag{15}$$

Bright lines are observed when $\lambda(.) < 0$ and dark lines when $\lambda(.) > 0$. For both types of lines, the magnitude $|\lambda(.)|$ indicates line contrast. Note that this formulation is free of parameters. The differential measurement is of dimension [intensity/pixel^2]. Multiplication by σ_v^2, which is of dimension [pixel^2], normalizes line contrast Equation (14) for the differential scale. Normalization by line brightness b results in a dimensionless quantity. As a consequence, the value of $\lambda(.)$ is within $[0 \ldots 1]$.

The differential equation (14) is a point measure, indicating whether a given pixel belongs to a line structure or not. The result is not the line structure itself, but a set of points in a 5-dimensional parameter space $(x, y, \sigma_u, \sigma_v, \theta)$ accumulating evidence for a line. Maxima in the parameter space indicate line points. Hence, the parameter space can be projected onto a two-dimensional angular parameter image $\Theta(x, y)$, a line width image $\Sigma_v(x, y)$, and the maximum magnitude $\Lambda(x, y)$,

$$\begin{aligned} \Theta(x, y), \Sigma_v(x, y) &= \operatorname*{argmax}_{\sigma_v, \theta} \left|\lambda(x, y, \sigma_u, \sigma_v, \theta)\right|, \\ \Lambda(x, y) &= \max_{\sigma_u, \sigma_v, \theta} \left|\lambda(x, y, \sigma_u, \sigma_v, \theta)\right|. \end{aligned} \tag{16}$$

Note that the value of σ_u at maximum response is less useful as it is affected by both local line curvature and intensity variations over the line, e.g., gaps.

Non-maxima suppression can be applied by evaluating the intensity profile for $\Lambda(x, y)$ in the direction of $\Theta(x, y)$ [7]. If the value of $\Lambda(x, y)$ is larger than its two neighboring pixels in the $\Theta(x, y)$ and $\Theta(x, y) + \pi$ direction, (x, y) is a local maxima. Otherwise, the pixel is suppressed in the $\Lambda(x, y)$, $\Theta(x, y)$, $\Sigma_v(x, y)$ images.

Grouping of line points is now achieved by labelling points with similar orientation and scale parameters within an eight-connected neighborhood. Note that a more robust algorithm is obtained by considering a larger neighborhood.

The algorithm is illustrated in Figure 6 by an example of a line drawing. Orientation scale-space is calculated according to Equation (14). The resulting images Θ, Λ, and the non-maxima suppressed image are shown in Figure 6(c), (d), (e), respectively. Most of the lines are correctly detected, including dashed lines. The fan-shapes at line-endpoints are caused by various thin filters at slightly different angle which fit the line-width.

Although it is trivial that the anisotropic Gaussian filter is separable in its coordinate system u, v, doing so is computationally expensive due to the misalignment between the image in memory and the direction of filtering. An efficient algorithm, by separating the filter along the x-axis and an axis t, where the direction t depends on the filter parameters $\theta, \sigma_u, \sigma_v$, is given in Geusebroek et al. [24]. Inspired by the

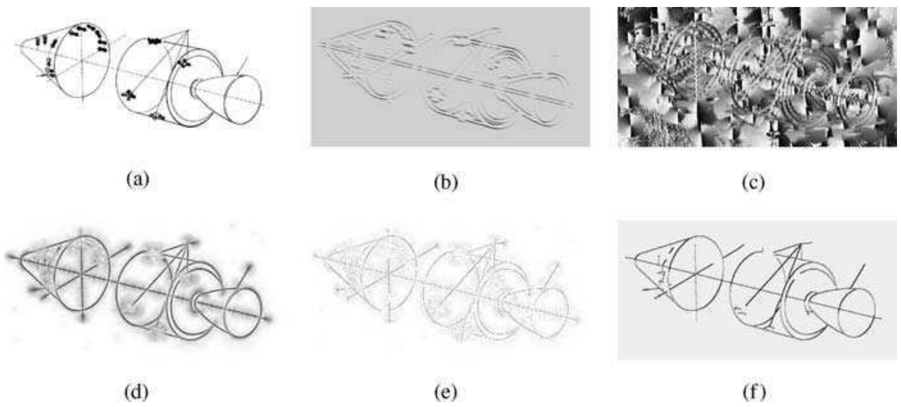


FIG. 6. Example orientation scale-space filtering on a line drawing (a). The response for $\lambda(.)$ Equation (14) for $\theta = -15°$ (b). The orientation Θ (c) at the maximum response Λ (d), combined in the non-maxima suppressed result (e). Thresholding and line orientation evaluation (f) results in clustering. Image courtesy of NASA History Office.

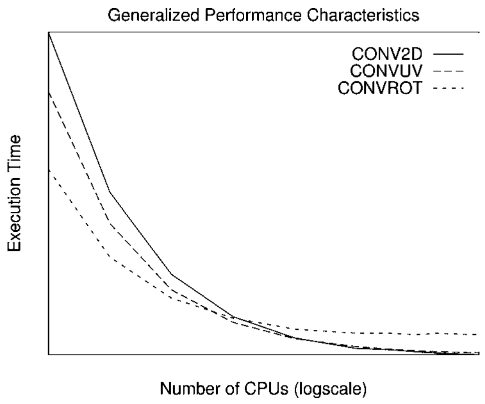


FIG. 7. Execution time as function of the number of processors. Exact timings depend on image size, parameter space, and processor speed. For details, see Seinstra et al. [57], Seinstra and Koelma [58].

human visual front-end, one could use a parallel computer system to further increase performance [57,58]. Figure 7 indicates the gain in computation time as function of the number of processors that can be achieved in this case.

2.3 Texture Analysis

In the case of texture, we are interested in the local spatial frequency characteristics of $E(x, y, \lambda)$. These properties are best investigated in the domain of spatial frequency. This leads to the use of Gabor filter [4] on the opponent color representation, as empirically derived by [35]. Thus, it is appropriate to represent the joint color-texture properties in a combined *wavelength-Fourier* domain $\mathcal{E}(u, v, \lambda)$, where λ remains the wavelength of the light energy, and (u, v) denotes the spatial frequency,

$$\widehat{\mathcal{E}}(u, v, \lambda) = \int \mathcal{E}(u, v, \lambda)\mathcal{P}(u, v, \lambda)\, d\lambda. \tag{17}$$

The measurement of the signal $\mathcal{E}(u, v, \lambda)$ at a given spatial frequency (u_0, v_0) and wavelength λ_0 is obtained by a 3D Gaussian probe centered at (u_0, v_0, λ_0) at a frequency scale σ_f and wavelength scale σ_λ,

$$\widehat{\mathcal{M}}(u, v, \lambda) = \int \mathcal{E}(u, v, \lambda)G(u - u_0, v - v_0, \lambda - \lambda_0; \sigma_f, \sigma_\lambda)\, d\lambda. \tag{18}$$

Note that the Gaussian probe is a separable function, we can rewrite Equation (18) as,

$$\widehat{\mathcal{M}}(u, v, \lambda) = \int \mathcal{E}(u, v, \lambda)G(u - u_0, v - v_0; \sigma_f)G(\lambda - \lambda_0; \sigma_\lambda)\, d\lambda. \tag{19}$$

Frequency selection is achieved by tuning the parameters u_0, v_0, and σ_f, and color information is captured by the Gaussian specified by λ_0 and σ_λ.

We now transform Equation (19) back to the wavelength-spatial domain (x, y, λ), taking the Gaussian derivative probes into account. The multiplication with the shifted Gaussian $G(u - u_0, v - v_0; \sigma_f)$ in Equation (19) is equivalent to the convolution with a Gabor filter in the spatial domain [4]. Therefore, the combined color-texture measurement in the (x, y, λ) domain at wavelength λ_0 is:

$$\widehat{M}(x, y, \lambda) = h(x, y) * \int E(x, y, \lambda)G_n(\lambda - \lambda_0; \sigma_\lambda)\, d\lambda, \tag{20}$$

or for short

$$\widehat{M}(x, y, \lambda) = h(x, y) * \widehat{E}_{\lambda^{(n)}}(x, y), \tag{21}$$

where:

$$h(x, y) = \frac{1}{2\pi\sigma_s^2}\, \mathrm{e}^{-\frac{x^2+y^2}{2\sigma_s^2}}\, \mathrm{e}^{2\pi j(Ux+Vy)}, \tag{22}$$

is the 2D Gabor function at the radial center frequency $F = \sqrt{U^2 + V^2}$ (cycles/pixel) and the filter orientation $\tan(\theta) = V/U$, and $j^2 = -1$. Furthermore, the color probes are given by the Gaussian derivatives,

$$G_n(\lambda - \lambda_0; \sigma_\lambda) = \frac{\sigma_\lambda^n}{\sqrt{2\pi\sigma_\lambda}} \frac{\partial^n}{\partial\lambda^n}\, \mathrm{e}^{-\frac{(\lambda-\lambda_0)^2}{2\sigma_\lambda^2}}. \tag{23}$$

Application of the filter is illustrated in Figure 8.

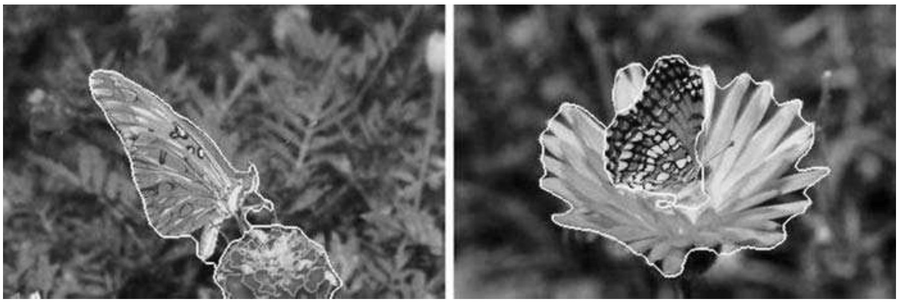


FIG. 8. Example of texture segmentation by the combined color-spatial Gabor filter set. Segmentation is achieved by applying a color Gabor filter set, followed by clustering. Note that the algorithm is capable of distinguishing both differences in color and in texture. (See Color Plate Section, Plate 9.)

3. Invariance

From a computer vision perspective, a fundamental question is *how to integrate the physical laws of light reflection into receptive field measurements*? The question boils down to deriving the invariant properties of color vision. With *invariance* we mean a property f of object t which receives value $f(t)$ regardless unwanted conditions W in the appearance of t. For human color vision, the group of disturbing conditions W' are categorized by *shadow*, *highlights*, *light source*, and *scene geometry*. Scene geometry is determined by the number of light sources, light source directions, viewing direction, and object shape. The invariant class W' is referred to as *photometric invariance*. For observation of images, *geometric invariance* is of importance [14,31,39,47,64]. The group of spatial disturbing conditions is given by *translation*, *rotation*, and *observation scale*. Since the human eye projects the three-dimensional world onto a two-dimensional image, the group may be extended with *projection* invariance.

Rotational invariance aims at keeping values constant when rotating the image under the camera. For spatial observations, derivatives are taken in the (x, y)-coordinates of the camera. Hence, when rotating the object viewed by the camera, the values of the derivatives in the image change, while the local image structure remains constant. In order to deal with this variation, we rotate the derivatives of each pixel to the gradient direction, a well known technique to obtain rotational invariance [16]. Rotating derivatives to the direction of the gradient yields the *gradient gauge*, given for the nth derivative in the x-direction and the mth derivative in the y-direction by

$$\partial_{v^n}\partial_{w^m} = \left(\sin\theta\frac{\partial}{\partial x} - \cos\theta\frac{\partial}{\partial y}\right)^m \left(\cos\theta\frac{\partial}{\partial x} + \sin\theta\frac{\partial}{\partial y}\right)^n, \quad n, m \geqslant 0, \quad (24)$$

where θ is the angle of the gradient, $\tan\theta = E_y/E_x$. Hence, the first order gradient gauge is given by

$$Ew = \sqrt{{E_x}^2 + {E_y}^2}, \tag{25}$$

$$Ev = 0 \tag{26}$$

yielding the gradient magnitude. The second order gradient gauge,

$$E_{vv} = \frac{E_x^2 E_{yy} - 2E_x E_y E_{xy} + E_y^2 E_{xx}}{E_w^2}, \tag{27}$$

$$E_{vw} = \frac{E_x E_y (E_{xx} - E_{yy}) - (E_x^2 - E_y^2) E_{xy}}{E_w^2}, \tag{28}$$

$$E_{ww} = \frac{E_x^2 E_{xx} + 2E_x E_y E_{xy} + E_y^2 E_{yy}}{E_w^2}. \tag{29}$$

Note that the Laplacian is obtained by $E_{vv} + E_{ww} = E_{xx} + E_{yy}$.

Both photometric and geometric invariance are required for a cognitive vision system to reduce the complexity intrinsic to images [26]. In this chapter we emphasize on photometric invariance, and put this into geometrical context. The effect of lighting conditions on the value of a pixel is investigated [26], resulting in local color invariance. From an analytical point of view this effectively separates scene accidental conditions from the true object characteristics. When seen from a reverse point of view, this constraints the probability of a pixel color by the light reflectance characteristics and the local coherence with neighboring pixels.

3.1 Color Invariance

Modelling the physical process of spectral image formation provides insight into the effect of different parameters on object reflectance [13,17,19,20,27,28,56,10]. The problem, known as color constancy is easily explained by considering the physical process of image formation (see Figure 9).

Object reflectance is well modelled by Shafer [59], based on the older Kubelka–Munk theory [43,44]. The Kubelka–Munk theory models the reflected and transmitted spectrum of a colored layer, based on a material dependent scattering and absorption function, under the assumption that light is isotropically scattered within the material. The theory has proven to be successful for a wide variety of materials and applications [37,70]. The theory unites spectral color formation for both reflecting materials as well as transparent materials into one photometric model. Therefore, the Kubelka–Munk theory is well suited for determining material properties from color measurements. In an earlier paper [26], the author demonstrates the use of the

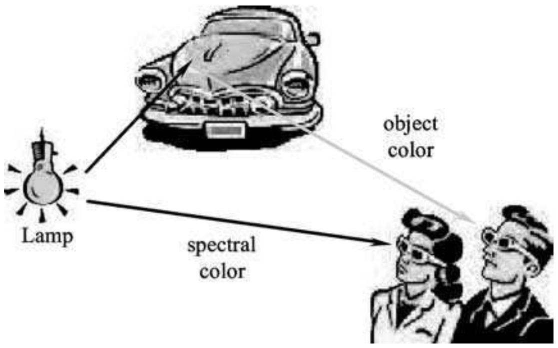


FIG. 9. The problem of color constancy. The light emitted by the lamp is reflected by the (yellow) car, causing a color sensation in the brain of the observers. The physical composition of the reflected light depends on the color of the light source. However, this effect is compensated for by the visual system. Hence, regardless the color of the light source, we will see a yellow car. This light source compensation is not trivial to obtain with a color camera in an unconstrained scene. (See Color Plate Section, Plate 10.)

Kubelka–Munk model to measure object reflectance from color images, under various general assumptions regarding imaging conditions. This section concentrates on color constant measurement of object color under both reflectance of light as well as light transmission.

When considering the estimation of material properties on the basis of local measurements, differential equations constitute a natural framework to describe the physical process of image formation. The color receptive fields as outlined earlier in this chapter provide a physical basis, which is compatible with colorimetry, for the measurement of color constant object properties.

3.1.1 Color Formation for Reflection of Light

Consider a homogeneously colored material patch of uniform thickness d and infinitesimal area, characterized by its absorption coefficient $k(\lambda)$ and scatter coefficient $s(\lambda)$. When illuminated by incident light with spectral distribution $e(\lambda)$, light scattering within the material causes diffuse body reflection (Figure 10), while Fresnel interface reflectance occurs at the surface boundaries.

When the thickness of the layer is such that further increase in thickness does not affect the reflected color, Fresnel reflectance at the back surface may be neglected. The incident light is partly reflected at the front surface, and partly enters the material, is isotropically scattered, and a part again passes the front-surface boundary. The reflected spectrum in the viewing direction $\vec{v}$, ignoring secondary scattering after

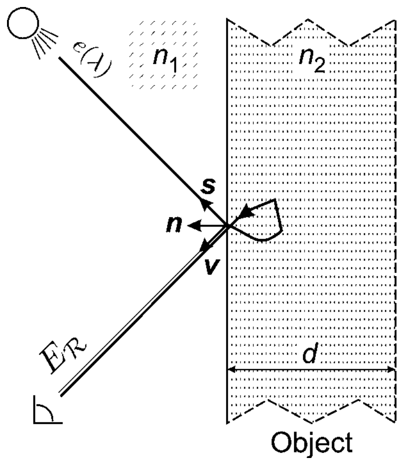


FIG. 10. Illustration of the photometric model. The object, refractive index n_2, is illuminated by $e(\lambda)$ (medium refractive index n_1), and light is reflected and scattered in the viewing direction.

internal boundary reflection, is given by [37,70]:

$$E_{\mathcal{R}}(\lambda) = e(\lambda)\bigl(1 - \rho_f(\lambda, \vec{n}, \vec{s}, \vec{v})\bigr)^2 R_\infty(\lambda) + e(\lambda)\rho_f(\lambda, \vec{n}, \vec{s}, \vec{v}) \tag{30}$$

where $\vec{n}$ is the surface patch normal and $\vec{s}$ the direction of the illumination source, and ρ_f the Fresnel front surface reflectance coefficient in the viewing direction. The body reflectance

$$R_\infty(\lambda) = a(\lambda) - b(\lambda) \tag{31}$$

depends on the absorption and scattering coefficient by

$$a(\lambda) = 1 + \frac{k(\lambda)}{s(\lambda)}, \qquad b(\lambda) = \sqrt{a(\lambda)^2 - 1}. \tag{32}$$

Simplification is obtained by considering neutral interface reflection, assuming that the Fresnel reflectance coefficient has a constant value over the spectrum. For commonly used materials, interface reflection is constant with respect to wavelength within a few percent across the visible spectrum [37,53]. Equation (30) reduces to

$$E_{\mathcal{R}}(\lambda) = e(\lambda)\bigl(1 - \rho_f(\vec{n}, \vec{s}, \vec{v})\bigr)^2 R_\infty(\lambda) + e(\lambda)\rho_f(\vec{n}, \vec{s}, \vec{v}). \tag{33}$$

The influence of the Fresnel reflectance varies from perfectly diffuse body reflectance $\rho_f = 0$, or Lambertian reflection, to total mirroring of the illuminating source ($\rho_f = 1$). Hence, the spectral color of $E_{\mathcal{R}}$ is an additive mixture of the color of the light source and the perfectly diffuse body reflectance color.

Because of projection of the energy distribution on the image plane, vectors $\vec{n}$, $\vec{s}$ and $\vec{v}$ will depend on the position at the imaging plane. The energy of the incoming spectrum at a point $\vec{x}$ on the image plane is then related to

$$E_{\mathcal{R}}(\lambda, \vec{x}) = e(\lambda, \vec{x})\big(1 - \rho_{\mathrm{f}}(\vec{x})\big)^2 R_\infty(\lambda, \vec{x}) + e(\lambda, \vec{x})\rho_{\mathrm{f}}(\vec{x}) \tag{34}$$

where the spectral distribution at each point x is generated off a specific material patch.

The major assumption made for the model of Equation (34) is that locally planar surface patches are examined, for which the material is homogeneously colored. These constraints are imposed by the Kubelka–Munk theory, resulting in isotropic scattering of light within the material. The assumption is valid when the resolution is fine enough to consider locally uniform colored patches, whereas individual staining particles are not resolved. Further, the thickness of the layer is assumed to be such that no light reaches the other side of the material. For every day scenes, these assumptions seems to be justified. Concerning the Fresnel reflectance, the photometric model assumes a neutral interface at the surface patch. As discussed in [53,59], deviations of ρ_{f} over the visible spectrum are small for commonly used materials, therefore the Fresnel reflectance coefficient may be considered constant. The internally Fresnel reflected light contributes little in many cases [70], and is ignored in the model.

3.1.2 Color Formation for Transmission of Light

Consider a homogeneously colored material patch of uniform thickness d and infinitesimal area, characterized by its absorption coefficient $k(\lambda)$ and scatter coefficient $s(\lambda)$. When illuminated by incident light with spectral distribution $e(\lambda)$, absorption and scattering by the material determines its transmission color (Figure 11), while Fresnel interface reflectance occurs at both the front and back surface boundaries.

When the layer is thin, such that the material is transparent, the transmitted spectrum through the layer in the viewing direction $\vec{v}$, ignoring the effect of interreflections between the material surfaces, is given by [37,70]:

$$E_{\mathcal{T}}(\lambda) = \frac{e(\lambda)(1 - \rho_{\mathrm{f}}(\lambda, \vec{n}, \vec{s}, \vec{v}))(1 - \rho_{\mathrm{b}}(\lambda, \vec{n}, \vec{s}, \vec{v}))b(\lambda)}{a(\lambda)\sinh[b(\lambda)s(\lambda)l(\vec{n}, \vec{s}, \vec{v})c] + b(\lambda)\cosh[b(\lambda)s(\lambda)l(\vec{n}, \vec{s}, \vec{v})c]} \tag{35}$$

where again $\vec{n}$ is the material patch normal and $\vec{s}$ is the direction of the illumination source. Further, c is the staining concentration and l the distance traveled by the light through the material. The terms ρ_{f} and ρ_{b} denote the Fresnel front and back surface reflectance coefficient, respectively. The factors a and b depend on the absorption and scattering coefficients as given by Equation (32).

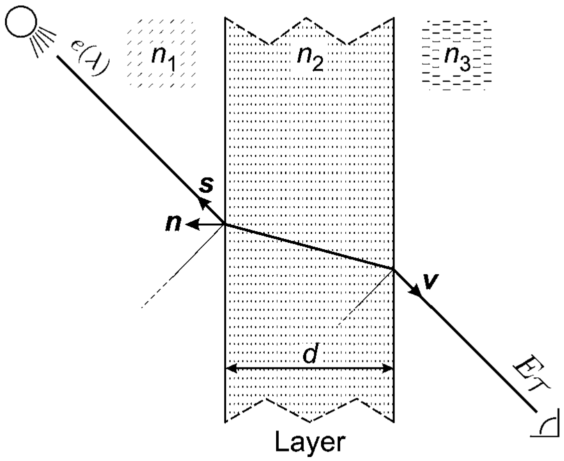


FIG. 11. Illustration of the photometric model. The object, refractive index n_2, is illuminated by $e(\lambda)$ (medium refractive index n_1). When the material is transparent, light is transmitted through the material, enters medium n_3, and is observed.

Simplification is obtained by considering neutral interface reflection, assuming that the Fresnel reflectance coefficients have a constant value over the spectrum. In that case, the Fresnel reflectance affects the intensity of the transmitted light only. Further, by considering a small angle of incidence at the transparent layer, the path length $l(\vec{n}, \vec{s}, \vec{v}) = d$. Equation (35) reduces to

$$E_{\mathcal{T}}(\lambda) = \frac{e(\lambda)(1 - \rho_{\mathrm{f}}(\vec{n}, \vec{s}, \vec{v}))(1 - \rho_{\mathrm{b}}(\vec{n}, \vec{s}, \vec{v}))b(\lambda)}{a(\lambda)\sinh[b(\lambda)s(\lambda)dc] + b(\lambda)\cosh[b(\lambda)s(\lambda)dc]}. \tag{36}$$

Because of projection of the energy distribution on the image plane, vectors $\vec{n}$, $\vec{s}$ and $\vec{v}$ will depend on the position $\vec{x}$ at the imaging plane,

$$\begin{aligned} E_{\mathcal{T}}(\lambda, \vec{x}) = {} & e(\lambda, \vec{x})\big(1 - \rho_{\mathrm{f}}(\vec{x})\big)\big(1 - \rho_{\mathrm{b}}(\vec{x})\big)b(\lambda, \vec{x}) \\ & /\big(a(\lambda, \vec{x})\sinh\big[b(\lambda, \vec{x})s(\lambda, \vec{x})d(\vec{x})c(\vec{x})\big] \\ & + b(\lambda, \vec{x})\cosh\big[b(\lambda, \vec{x})s(\lambda, \vec{x})d(\vec{x})c(\vec{x})\big]\big) \end{aligned} \tag{37}$$

where the spectral distribution at each point x is generated off a specific transparent patch.

One of the assumptions made for the model of Equation (37) is that locally planar material patches are examined, with parallel sides, for which the material is homogeneously colored. The assumption is valid when the material is non-fluorescent nor in any sense optically active, and the resolution is fine enough to consider locally uniform colored patches, while individual stain particles are not resolved. Again, these constraints are imposed by the Kubelka–Munk theory. Further, normal incidence of light at the layer is assumed, so that the optical path length through the layer approximates its thickness. In transmission light microscopy, the preparation and observation conditions fairly justify these assumptions. Concerning the Fresnel reflectance, the photometric model assumes a neutral interface at the transparent patch. As discussed in Pluta [53], deviations of ρ_f, ρ_b over the visible spectrum are small for commonly used materials. For example, the refractive index of immersion oil often used in microscopy only varies 3.3% over the visible spectrum. Therefore, the Fresnel reflectance coefficients ρ_f and ρ_b may be considered constant over the spectrum. The contribution of internally Fresnel reflected light is small in many cases [70], and is therefore ignored in the model.

3.1.3 Special Cases

Thus far, we have achieved a photometric model for spectral color formation, which is applicable for both reflecting and transmitting materials, and valid under a wide variety of circumstances and materials. The following special cases can be derived.

For matte, dull surfaces, the Fresnel coefficient can be considered neglectable, $\rho_\mathrm{f}(\vec{x}) \approx 0$, for which $E_\mathcal{R}$ Equation (34) reduces to the Lambertian model for diffuse body reflection,

$$E_\mathcal{R}(\lambda, \vec{x}) = e(\lambda, \vec{x}) R_\infty(\lambda, \vec{x}) \tag{38}$$

as expected.

By introducing $c_b(\lambda) = e(\lambda) R_\infty(\lambda)$, $c_i(\lambda) = e(\lambda)$, $m_b(\vec{n}, \vec{s}, \vec{v}) = (1 - \rho_\mathrm{f}(\vec{n}, \vec{s}, \vec{v}))^2$ and $m_i(\vec{n}, \vec{s}, \vec{v}) = \rho_\mathrm{f}(\vec{n}, \vec{s}, \vec{v})$, Equation (33) may be reformulated as

$$E_\mathcal{R}(\lambda) = m_b(\vec{n}, \vec{s}, \vec{v}) c_b(\lambda) + m_i(\vec{n}, \vec{s}, \vec{v}) c_i(\lambda) \tag{39}$$

which corresponds to the dichromatic reflection model proposed by Shafer [59].

For light transmission, when the scattering coefficient is low compared to the absorption coefficient, $s(\lambda) \ll k(\lambda)$, $E_\mathcal{T}$ Equation (37) reduces to Bouguer's or Lambert–Beer's law for absorption [70],

$$E_\mathcal{T}(\lambda, \vec{x}) = e(\lambda, \vec{x}) \big(1 - \rho_\mathrm{f}(\vec{x})\big) \big(1 - \rho_\mathrm{b}(\vec{x})\big) \exp\big(-k(\lambda, \vec{x}) d(\vec{x}) c(\vec{x})\big) \tag{40}$$

as expected.

Further, a unified model for both reflection and transmission of light is obtained when considering Lambertian reflection and a uniform illumination for both cases.

For matte, dull surfaces, and a uniform illumination affected by shading, $E_{\mathcal{R}}$ Equation (34) reduces to a multiplicative (Lambertian) model for body reflection,

$$E_{\mathcal{R}}(\lambda, \vec{x}) = e(\lambda) i(\vec{x}) R_{\infty}(\lambda, \vec{x}) \tag{41}$$

where $e(\lambda)$ is the colored but spatially uniform illumination and $i(\vec{x})$ denotes the intensity distribution due to the surface geometry. Similar, for a uniform illuminated transparent material, intensity affected by shading and Fresnel reflectance, $E_{\mathcal{T}}$ Equation (37) may be rewritten as

$$E_{\mathcal{T}}(\lambda, \vec{x}) = e(\lambda) i(\vec{x}) C(\lambda, \vec{x}) \tag{42}$$

where $e(\lambda)$ is the uniform illumination, $i(\vec{x})$ denotes the intensity distribution, including Fresnel reflectance at front and back surface, and $C(\lambda, \vec{x})$ represents the total extinction coefficient, that is the total absorption—and scattering coefficient, within the transparent layer. A general model for spectral image formation useful in both reflectance and transmission of light may now be written as a multiplicative model,

$$E(\lambda, \vec{x}) = e(\lambda) i(\vec{x}) m(\lambda, \vec{x}) \tag{43}$$

where $m(\lambda, \vec{x})$ denotes the material transmittance or reflectance function. Again, $e(\lambda)$ is the colored but spatially uniform illumination and $i(\vec{x})$ denotes the intensity distribution. The validness of the model may be derived from models Equations (34) and (37). For reflectance of light, the model is valid for matte, dull surfaces, for which the Fresnel reflectance is neglectable, and for isotropic light scattering within the material. For light transmission, the model is valid for neutral interface reflection, small angle of incidence to the surface normal, and isotropic light scattering within the material. The model as such is used in the next sections to derive color invariant material properties.

3.1.4 Invariants for Object Reflectance or Transmittance

Any method for finding invariant color properties relies on a photometric model and on assumptions about the physical variables involved. For example, hue is known to be insensitive to surface orientation, illumination direction, intensity and highlights, under a white illumination [28]. Normalized *rgb* is an object property for matte, dull surfaces illuminated by white light. When the illumination color varies or is not white, other object properties which are related to constant physical parameters should be measured. In this section, expressions for determining material changes in images will be derived, robust to a change in illumination color over time. Therefore, the photometric model derived above are taken into account.

Consider the photometric reflection model Equation (43) and an illumination with locally constant color,

$$E(\lambda, \vec{x}) = e(\lambda) i(\vec{x}) m(\lambda, \vec{x}) \tag{44}$$

where $e(\lambda)$ represents the illumination spectrum. The assumption allows for the extraction of expressions describing material changes independent of the illumination. Without loss of generality, we restrict ourselves to the one dimensional case; two dimensional expressions may be derived according to [26]. Differentiation of Equation (44) with respect to λ results in

$$\frac{\partial E}{\partial \lambda} = i(x) m(\lambda, x) \frac{\partial e}{\partial \lambda} + i(x) e(\lambda) \frac{\partial m}{\partial \lambda}. \tag{45}$$

Dividing Equation (45) by Equation (44) gives the relative differential,

$$\frac{1}{E(\lambda, x)} \frac{\partial E}{\partial \lambda} = \frac{1}{e(\lambda)} \frac{\partial e}{\partial \lambda} + \frac{1}{m(\lambda, x)} \frac{\partial m}{\partial \lambda}. \tag{46}$$

The result consists of two terms, the former depending on the illumination color and the latter depending on material properties. Since the illumination color is constant with respect to x, differentiation to x yields a material property only,

$$\frac{\partial}{\partial x} \left\{ \frac{1}{E(\lambda, x)} \frac{\partial E}{\partial \lambda} \right\} = \frac{\partial}{\partial x} \left\{ \frac{1}{m(\lambda, x)} \frac{\partial m}{\partial \lambda} \right\}. \tag{47}$$

Within the Kubelka–Munk model, assuming matte, dull surfaces or transparent layers, and assuming a single light source, $N_{\lambda x}$ determines changes in object reflectance or transmittance,

$$N_{\lambda x} = \frac{1}{E(\lambda, x)} \frac{\partial^2 E}{\partial \lambda \partial x} - \frac{1}{E(\lambda, x)^2} \frac{\partial E}{\partial \lambda} \frac{\partial E}{\partial x} \tag{48}$$

which determines material changes independent of the viewpoint, surface orientation, illumination direction, illumination intensity and illumination color. The expression results from differentiation of Equation (47).

The expression given by Equation (48) is the fundamental lowest order illumination invariant. Any spatio-spectral derivative of Equation (48) inherently depends on the body reflectance or object transmittance only. According to Olver et al. [49], a complete and irreducible set of differential invariants is obtained by taking all higher order derivatives of the fundamental invariant,

$$N_{\lambda x \lambda^m x^n} = \frac{\partial^{m+n}}{\partial \lambda^m \partial x^n} \left\{ \frac{1}{E(\lambda, x)} \frac{\partial^2 E}{\partial \lambda \partial x} - \frac{1}{E(\lambda, x)^2} \frac{\partial E}{\partial \lambda} \frac{\partial E}{\partial x} \right\} \tag{49}$$

for $m \geqslant 0$, $n \geqslant 0$.

Application of the chain rule for differentiation yields the higher order expressions in terms of the spatio-spectral energy distribution. For instance, the spectral derivative of $N_{\lambda x}$ is given by

$$N_{\lambda\lambda x} = \frac{E_{\lambda\lambda x}E^2 - E_{\lambda\lambda}E_x E - 2E_{\lambda x}E_\lambda E + 2E_\lambda^2 E_x}{E^3} \tag{50}$$

where $E(\lambda, x)$ is written as E for simplicity and indices denote differentiation. Note that these expressions are valid everywhere $E(\lambda, x) > 0$. These invariants may be interpreted as the spatial derivative of the normalized spectral slope N_λ and curvature $N_{\lambda\lambda}$ of the reflectance function R_∞. Expressions for higher order derivatives are straightforward.

A special case of Equation (49) is for Lambert–Beer absorption Equation (40) and slices of locally constant thickness. Under these circumstances, ratios of invariants from set N,

$$N' = \frac{N^{m,n}}{N^{p,q}} \tag{51}$$

for $m, p \geqslant 1$ and $n, q \geqslant 0$, are independent of the slice thickness. The property is proven by considering differentiation with respect to λ of Equation (40), and division by Equation (40), which results in

$$\frac{1}{E_{\mathcal{T}}(\lambda, x)} \frac{\partial E_{\mathcal{T}}}{\partial \lambda} = \frac{1}{e(\lambda)} \frac{\partial e}{\partial \lambda} - dc(x) \frac{\partial k}{\partial \lambda}. \tag{52}$$

Differentiation of the expression with respect to x yields

$$\frac{\partial}{\partial x} \left\{ \frac{1}{E_{\mathcal{T}}(\lambda, x)} \frac{\partial E_{\mathcal{T}}}{\partial \lambda} \right\} = -dc(x) \frac{\partial^2 k}{\partial \lambda \partial x} - d \frac{\partial k}{\partial \lambda} \frac{\partial c}{\partial x}. \tag{53}$$

By taking ratios of higher order derivatives, the constant thickness d is eliminated.

Summarizing, we have derived a complete set of color constant expressions determining object reflectance or transmittance. The expressions are invariant for a change of illumination over time. The major assumption underlying the proposed invariants is a single colored illumination, effectuating a spatially constant illumination spectrum. For an illumination color varying slowly over the scene with respect to the spatial variation of the object reflectance or transmittance, simultaneous color constancy is achieved by the proposed invariant.

We have proven that spatial differentiation is necessary to achieve color constancy when pre-knowledge about the illuminant is not available. Hence, any color constant system should perform both spectral as well as spatial comparison in order to be invariant against illumination changes, which confirms the theory of relational color constancy as proposed by Foster and Nascimento [17].

3.1.5 Invariants for Specular Reflection

Consider the photometric reflection model Equation (34). For white illumination, the spectral components of the source are approximately constant over the wavelengths. Hence, a spatial component $i(x)$ denotes intensity variations, resulting in

$$E(\lambda, x) = i(x)\bigl\{\rho_{\mathrm{f}}(x) + \bigl(1 - \rho_{\mathrm{f}}(x)\bigr)^2 R_\infty(\lambda, x)\bigr\}. \tag{54}$$

The assumption allows the extraction of expressions describing object reflectance independent of the Fresnel reflectance. Differentiating Equation (54) with respect to λ results in

$$\frac{\partial^n E}{\partial \lambda^n} = i(x)\bigl(1 - \rho_{\mathrm{f}}(x)\bigr)^2 \frac{\partial^n R_\infty}{\partial \lambda^n}.$$

Hence, ratio of derivatives depend on derivatives of the object reflectance functions R_∞ only,

$$H = \frac{\frac{\partial E}{\partial \lambda}}{\frac{\partial^2 E}{\partial \lambda^2}} \tag{55}$$

independent of viewpoint, surface orientation, illumination direction, illumination intensity and Fresnel reflectance coefficient.

The property H describes the hue $= \arctan(\lambda_{\max})$ of the material. The expression given by Equation (55) is a fundamental lowest order invariant. Any spatio-spectral derivative of the fundamental invariant is an invariant under the same imaging conditions according to Olver et al. [49]. As a result, differentiation of the expression for H with respect to x results in object reflectance properties under a white illumination. Note that H is ill-defined when the second order spectral derivative vanishes. We prefer to compute the $\arctan(H)$, for which the spatial derivatives yield better numerical stability.

In conclusion, within the Kubelka–Munk model, a complete and irreducible set of invariants for dichromatic reflection and a white illumination is given by

$$H_{x^n} = \frac{\partial^n}{\partial x^n}\left\{\arctan\left(\frac{\frac{\partial E}{\partial \lambda}}{\frac{\partial^2 E}{\partial \lambda^2}}\right)\right\} \tag{56}$$

for $n \geqslant 0$.

Application of the chain rule for differentiation yields the higher order expressions in terms of the spatio-spectral energy distribution. For illustration, the hue spatial derivative is given by

$$H_x = \frac{E_{\lambda\lambda} E_{\lambda x} - E_\lambda E_{\lambda\lambda x}}{E_\lambda^2 + E_{\lambda\lambda}^2} \tag{57}$$

where $E(\lambda, x)$ is written as E for simplicity, admissible for $E_\lambda^2 + E_{\lambda\lambda}^2 > 0$.

3.2 Invariant Image Interpretation

An interesting problem in the segmentation of man made objects is the segmentation of edges into the “real” object edges, or “artificial” edges caused by shadow boundaries or highlights [29]. Consider an image captured under white illumination. A common model for the reflection of light by an object to the camera is given by the Kubelka–Munk theory Equation (34). Edges may occur under three circumstances:

- shadow boundaries due to edges in $i(\vec{x})$;
- highlight boundaries due to edges in $\rho_{\mathrm{f}}(\vec{x})$;
- material boundaries due to edges in $R_\infty(\lambda, \vec{x})$.

For the model given by Equation (34), material edges are detected by considering the ratio between the first and second order derivative with respect to λ, or

$$\frac{\partial}{\partial x}\left\{\frac{E_\lambda}{E_{\lambda\lambda}}\right\}$$

where E represents $E(\lambda, \vec{x})$ and indices denote differentiation. Further, the ratio between $E(\lambda, \vec{x})$ and its spectral derivative are independent of the spatial intensity distribution. Hence, the spatial derivatives

$$\frac{\partial}{\partial x}\left\{\frac{E_\lambda}{E}\right\}, \qquad \frac{\partial}{\partial x}\left\{\frac{E_{\lambda\lambda}}{E}\right\} \tag{58}$$

depend on Fresnel and material edges. Finally, the spatial derivatives of E, E_λ, and $E_{\lambda\lambda}$ depend on intensity, Fresnel, and material edges.

In order to measure the invariant properties, the partial derivative operator is replaced by Gaussian derivative convolutions. Hence, the incoming spatio-spectral energy density function is probed with Gaussian and Gaussian derivative functions. These measurements approximate the differential expressions derived above. Hence, measurement of the above expressions is obtained by substitution of E, E_λ, and $E_{\lambda\lambda}$ for the measured values $\widehat{E}$, $\widehat{E}_\lambda$, and $\widehat{E}_{\lambda\lambda}$ Equation (10) smoothed at scale σ_x. For the spatial derivatives, convolution with a Gaussian derivative function at scale σ_x yields the correct approximation. Combining these expressions in gradient magnitudes yields Figure 12.

4. Natural Image Statistics

An image typically consists of a million of pixels, each pixel being one value out of millions. Despite this overwhelming amount of choices to generate an image, there is a limited amount of configurations that represent a natural scene. Investigation

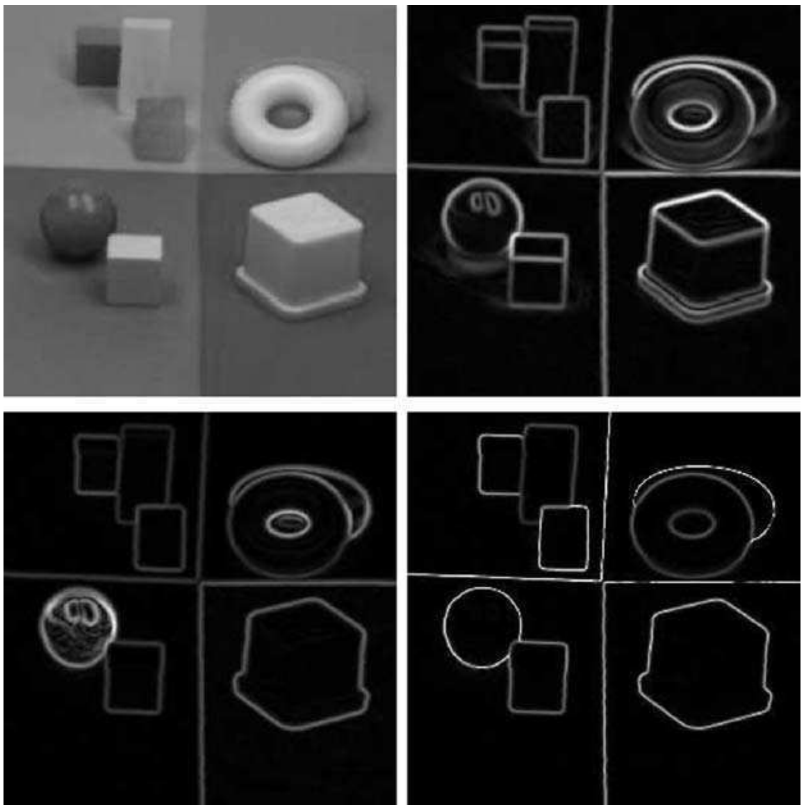


FIG. 12. Photometric interpretation of edges in a toy example. Top right shows all edges by combining the spatial derivatives of E, E_λ, $E_{\lambda\lambda}$. Bottom left shows edges due to object reflectance and Fresnel reflectance (specularities). Note the suppression of shading edges in the blocks. Further, notice that highlights are still detected in the red ball. Bottom right shows the case of shadow, shading and highlight invariant edge detection. Note that the highlights on the ball's surface are suppressed, together with shadow and shading edges. Only edges caused by true changes in object surface color are detected. Comparison between these invariants allows the interpretation of photometric edges. (See Color Plate Section, Plate 11.)

of the statistics of natural images is an important topic for texture synthesis and recognition. Empirical studies [36,60,67] concentrate on the fitting of distributions to the response of linear operators for (large) sets of images. Empirical methods lack a physical basis, hence are difficult to interpret. The distributions determined are not easily proven to be the correct ones. Especially when the marginal statistics are considered important, as is often the case for reasoning in knowledge, finding the correct distribution is crucial.

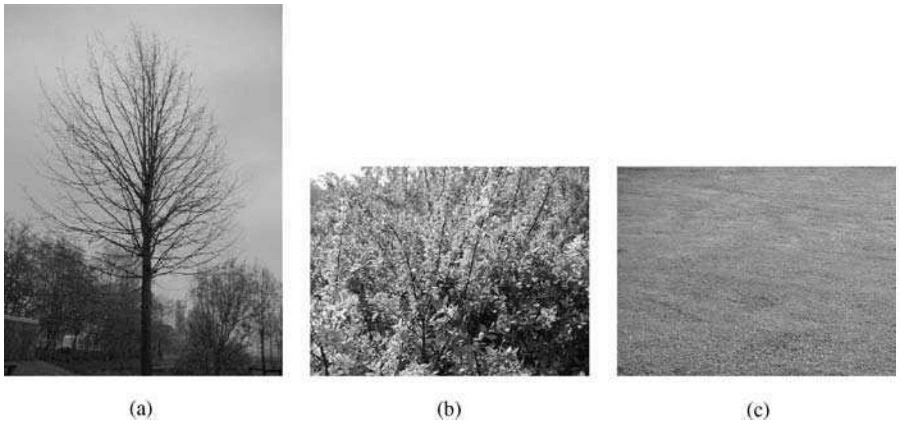


FIG. 13. An opaque object fragments the scene into stochastic patches of foreground and background (a). When more and more details are added (b), the background is sequentially fragmented, until detail size is beyond the visual resolution (c).

Theoretical studies based on the statistics of surface reflectance properties include [8,41,42,66]. These methods consider the physics of reflection to derive image statistics. Knowledge about the characteristics of the reflecting surface is explicitly assumed, for instance the slope distribution at the surface is Gaussian. Hence, there is a physical ground and a sufficient explanation of the parameters of the model. However, the model is only valid for a limited amount of natural images.

In this section, a physical explanation is given for the statistics of local image structure of natural images. The analysis is started by introducing the Weibull distribution from the field of sequential fragmentation. The distribution originates from the study of particle distributions after milling. The Weibull distribution describes the number of particles as function of particle size or mass, hence the result of sieving processes. A picture is composed of many details of larger and smaller size, which in turn are composed of even smaller details (Figure 13). In the projection of the details onto the receptive fields in the retina, some of the detail is larger than the scale of resolution, whereas other details are smaller and effectively integrated in the response of one receptive field. The size distribution of the details may be inferred from the contours of the details and their shadows. The projection of the contours is a linear transform of the three-dimensional detail shape. Hence, the intensity differences in a view are indicative for the size distribution of the details in the scene [50].

4.1 Sequential Fragmentation Theory

Local image structure is completely determined by the Taylor expansion of the image at a given point (x, y),

$$\widehat{E}(x, y) = \widehat{E} + \begin{pmatrix} x \\ y \end{pmatrix}^{\mathrm{T}} \begin{bmatrix} \widehat{E}_x \\ \widehat{E}_y \end{bmatrix} + \frac{1}{2} \begin{pmatrix} x \\ y \end{pmatrix}^{\mathrm{T}} \begin{bmatrix} \widehat{E}_{xx} & \widehat{E}_{xy} \\ \widehat{E}_{xy} & \widehat{E}_{yy} \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \cdots . \tag{59}$$

The measurement is obtained by integrating over a certain spatial extent, the observation scale σ. Differentiation may be transported using Gaussian derivative filters,

$$\widehat{E}_{x^n y^m}(x, y) = E(x, y) * G_{x^n y^m}(x, y; \sigma) \tag{60}$$

which results in the well-known N-Jet [15]. The coefficients of the Taylor expansion of $\widehat{E}(x, y)$ together form a complete representation of the local image structure. Truncation of the Taylor expansion results in an approximate representation, the local equivalence class.

The respective statistics of each of the N-Jet components reflect the dependence between neighboring pixels. For the zeroth order measurement, the statistics are represented by the histogram of the Gaussian smoothed image intensity. The histogram is, neglecting smoothing effects, invariant under permutations of the pixels. Hence, the zeroth order statistics do not include joined statistics, and are irrelevant when considering pixel dependence.

Of considerable importance is the statistics of the first order derivatives,

$$\widehat{E}_x \equiv \frac{\partial E}{\partial x} \left[\frac{\text{intensity}}{\text{length}} \right]. \tag{61}$$

For construction of the response probability density function one takes a given response interval and sums over its spatial occurrences. When considering a unity step along the response $\partial \widehat{E}_x$, one considers a certain variable distance ∂x in the spatial domain. This distance depends on the local slope in the image function $\widehat{E}$.

As a result from scale-space theory, we consider that small details are occurring more often than large structures [39]. This is a direct implication of causality. Diffusion of numerous small structures will result in fewer large structures. Inversely, increasing magnification at large structures will resolve many smaller structures One may rephrase the statement in that, when resolving power increases, large structures will break-up into new structures, of which some of them are relatively large, but most of them will be small details.

Consider the response of an edge detector to an image containing one dominant edge. The edge detector will respond strongly to the edge, and yield smaller values elsewhere. The histogram of responses typically shows a power-law distribution,

$$f(x) = \left(\frac{x}{\beta} \right)^{\gamma - 1} . \tag{62}$$

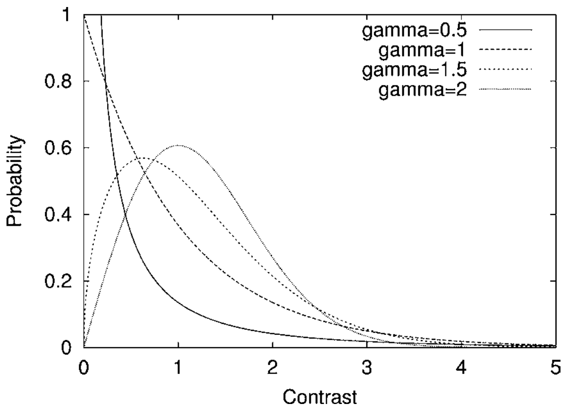


FIG. 14. Illustration of the Weibull distribution $p(r)$ for various values of γ.

When more objects are added to the scene, the image will be fragmented into various patches, each giving rise to an edge of varying contrast. The histogram over the various edges is the results of integrating over the various power-laws caused by every edge,

$$n(x) = c \int_{x}^{\infty} n(x')f(x)\,dx' \tag{63}$$

where $n(x)$ indicates the number of pixels with response magnitude between x and $x + dx$, contributed by all edges with contrast $x' > x$. The integration over a sufficient number of power-laws yields a Weibull distribution (Figure 14),

$$n(x) = \frac{1}{\beta}\left(\frac{x}{\beta}\right)^{\gamma-1} \mathrm{e}^{-\frac{1}{\gamma}(\frac{x}{\beta})^{\gamma}}. \tag{64}$$

In integral form, the Weibull distribution given by

$$N(> x) = \int_{x}^{\infty} n(x)\,dx = \mathrm{e}^{-\frac{1}{\gamma}|\frac{x}{\beta}|^{\gamma}} \tag{65}$$

indicates the relative amount of edges of (positive or negative) contrast larger than x.

A similar reasoning is known in the sequential fragmentation of particles by milling [5,6], which shows much resemblance with the present theory. Brown and Wohletz [6] theoretically derived the power-law process to describe the particle size distribution for the crushing of particles in a mill, providing a solid physical basis for the distributions in Equations (64) and (65) with the shape parameter γ related to fractal dimension.

Simoncelli [60] empirically found the generalized Laplacian,

$$P(c) = z\,\mathrm{e}^{-|\frac{c}{s}|^p} \tag{66}$$

to fit to the marginal statistics of wavelet coefficients. Here, c indicates the wavelet coefficient, and s indicates the variance. The exponent p is related to γ in Equation (65). The generalized Laplacian is the integral form Equation (65) of the Weibull distribution. We have given a physical explanation for the empirical results as obtained by Simoncelli [60].

The Weibull shape parameter γ is related to the fractal dimension of the image [21]. Note that fractal dimension is a strictly spatial property of an image.

The parameters of the Weibull distribution completely characterize the spatial layout of stochastically ergodic textures. Furthermore, the Weibull parameters generate a complete orthogonal basis for stochastic textures. The parameters of the Weibull distribution indicate the contrast in the image (β), the grain size (γ) relative to resolving power, and the global shape of the object (μ) as can be derived from shape from shading [51].

4.2 Consequences for Natural Image Statistics

As a consequence of the sequential fragmentation theory, spatial image statistics are limited to conform to one out of five options:

Power-law: When resolution is extremely fine compared to detail size, spatial layout follows a power-law distribution, being the result of a single fragmentation event, or foreground-background. This is the case when we examine a single object against a highly contrasting background. When contrast is reduced by local normalization, a power-law is no longer observed.

Normal: When resolution is too coarse to resolve the details, spatial layout becomes normal distributed. This is the case when we look at sand, or at hairs. On closer inspection with higher resolution, we may resolve the details and spatial statistics convert to Weibull again.

Weibull: In general, with the fine but limited resolution used for the vast majority of scenes we encounter, views we observe are fragmented and their details therefore Weibull distributed. Spatial detail statistics deform from power-law to normal through the Weibull type distribution as resolving power decreases, while the field of view enlarges.

Composition: When the scene is composed of a few fragmentations, the Weibull distribution will not appear. In this case, individual parts of the scene may conform to the Weibull distribution with varying parameters. However, the composition of the scene results in the addition of Weibull distributions of different parameters, resulting in a non-Weibull distribution. This is often the case for a scene composed of two or three objects, or scenes with sharply distinct distances. As the Weibull distribution describes sequential fragmentation, a requirement for the Weibull distribution is that the composition of the scene is sufficiently complex.

Regular: For repetitive patterns, the visual responses interfere with the repetition in the sensing field, in which case any distribution may describe spatial statistics. Hence, the proposed fragmentation theory breaks for regular textures.

4.3 Experiment on Corel Image Collection

We experimentally investigated the statistics of spatial detail on the Corel[1] photo collection [22]. The collection consists of 46,695 images covering a broad class of general pictures. As edge information is of crucial importance in the coding of images, we expect the sequential fragmentation process so dominantly present in image statistics that it will be preserved by any compression method. The large and diverse collection is used to provide evidence for the sequential fragmentation theory to be present in a broad imaging domain, much broader than only dead-leaves occlusion processes, as in Lee et al. [46], or only transparent (additive) image formation, as in Grenander and Srivastava [32]; Srivastava et al. [61]. The collection is originally compressed by a wavelet compression technique. For processing purposes, we converted the whole collection to JPEG compressed images, compression factor 0.7.

A detailed study was performed on views of material textures in the Curet collection [9]. The image collection is calibrated and uncompressed, image size 768×576. The collection consists of 61 materials, each taken under various illumination and viewing directions. The sample contains a diverse collection of materials, including plaster, styrofoam, straw, corduroy, paper, brick, fur, and so on, effectively covering a range of Lambertian reflection, polarized reflection with highlights, to the mirror reflection of Aluminum foil. We will use this database mainly for illustration purposes.

4.3.1 Experimental Setup

Edge strength is accessed by Gaussian derivative filters measured in 72 directions by steering the x, y-derivative filters. The effective resolution of the system is given by the spatial width of the filter, here set to correspond to a standard deviation of 3

[1] Corel Gallery, see http://www.corel.com.

pixels in all experiments. Note that changing the width to another constant will not change the major results of this paper (data not shown). Further note that applying any alternative zero-average filter will not affect the major results (data not shown). Responses per image were accumulated into histograms, and three distributions were fitted to each histogram. They are a power-law distribution, a Weibull distribution, and a normal distribution. As the histogram contains both positive and negative edge responses, we used the symmetrical versions of the Weibull and power-law distribution.

For filter responses, high values indicate strong correlation between image content and filter shape. Hence the tails of the distribution are much more important in terms of image content [2] than low values, representing uniform areas and noise. Where most hypothesis tests, including the Kullback–Leibler divergence, assign more weight to the often occurring values, we put our emphasis in the tails of the distribution. As the Anderson–Darling test is sensitive to the tails, goodness-of-fit was evaluated by this hypothesis test [12]. Note that this test is commonly used in statistics when accessing goodness-of-fit for both power-law and Weibull distributions. A second important characteristic of the Anderson–Darling test is that critical values are tuned to the distribution at hand, including the free parameters. Models of different complexity –with different degrees of freedom—may be compared at similar confidence levels. Hence, the Anderson–Darling test allows fair comparison between power—law, Gaussian, and Weibull distributed portion of a database, despite the different number of parameters of these models.

The Weibull distribution symmetric integral form is given by

$$f(x) = C \exp\left(-\frac{1}{\gamma}\left|\frac{(x-\mu)}{\beta}\right|^{\gamma}\right), \tag{67}$$

the parameters μ, β, and γ representing the center, width, and shape of the distribution, and C being a normalization constant. The shape parameter, γ, ranges from 0 to 2 [30]. For $\gamma = 2$ the Weibull distribution is equivalent to the normal distribution, and for $\gamma = 1$ it is a double exponential. The distribution is also known as the generalized Laplacian [48,60]. Brown [5] showed the close connection of this integral form to the original distribution proposed by Weibull [68]. For our experiments, the values of the Weibull parameters were estimated using the maximum likelihood method. Goodness-of-fit was evaluated at a significance level $\alpha = 0.05$ ($A^2 < 0.757$) [12] for all cases. Furthermore, we rejected the Weibull distribution for an estimation of $\gamma > 2.2$, resulting in a value of γ too far out of range to yield a stable statistical process [30].

We consider the symmetric form of the power-law,

$$f(x) = \frac{1}{2}\delta|x|^{-\delta-1}. \tag{68}$$

The parameter δ was estimated by the maximum likelihood method, and goodness-of-fit was evaluated by the Anderson–Darling statistic [54] at a significance level of $\alpha = 0.05$ ($A^2 < 1.341$) [63].

The parameters for the normal distribution were obtained by maximum likelihood estimation, and goodness-of-fit was evaluated by the Anderson–Darling statistic at a significance level of $\alpha = 0.05$ ($A^2 < 0.787$) [63].

4.3.2 Results

For the Corel general photo stock, the Anderson–Darling test indicated 48% of the pictures to be Weibull distributed. This is a remarkable result given the variety of generating processes for the pictures in the collection, and the compression of the images. The Weibull distribution apparently describes the spatial statistics for outdoor scenes, indoor scenes, close-ups, and materials of various kinds. Approximately 1% of the Corel collection is normally distributed. Note that these pictures are included in the Weibull distributed part of the collection, the normal distribution being an extreme case with $\gamma = 2$. An additional 9% of the Corel collection is distributed according to a power-law, while being rejected as a Weibull distribution. A portion of 4% is accepted as both power-law distributed and Weibull distributed, the fraction being included in the reported 48%.

For the uncompressed images in the van Hateren collection, similar results are obtained. For this collection, a Weibull distribution is present in 54% of the images. A neglectable amount of images is normally distributed, which can be explained by the high resolution at which these outdoor images are taken. Furthermore, 28% of the images is power-law distributed, indicating that much of the images consist of a high contrasting object against a more or less uniform background. A portion of 5% is accepted as both power-law distributed and Weibull distributed, the fraction being included in the 54% Weibull distributed.

To understand this widespread presence of the Weibull distribution, a detailed study was performed on views of material textures in the Curet collection [9]. It appeared that 54 materials out of 61 (88%) consistently render a Weibull distribution. The estimated Weibull parameter values varied with illumination and viewing direction, consistently rendering a Weibull distribution over all imaging conditions. Note that in the continuing report of our result, examples of the Curet database do not provide anecdotal evidence for our theory, rather yields detailed illustrations of the different stages of the fragmentation process.

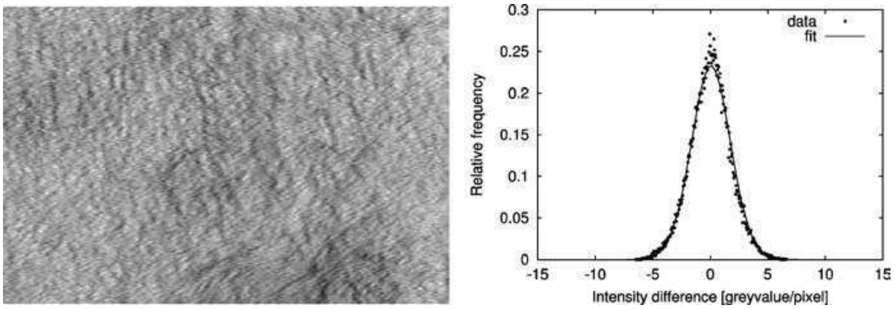
As an explanation for a normal distribution of filter responses consider an infinitely precise sensor. We would see the details around us at infinitely many scales. Increasing the size of the sensor to a finite extent imposes spatial coherence and a limited local scale of detail. When the sensor resolution is much larger than the common size of the random details in the field of view, each sensor response is an average over many impulses. From the central limit theorem, local intensity differences will be normally distributed as is observed in the rabbit fur of the Curet collection (Figure 15(a)), with individual hairs as the random details, each much smaller than the resolution of observation.

For the power-law distributed images, visual inspection showed much of these pictures to contain an exhibited item, or to display land-sky, thereby fragmenting the scene into a foreground and a background region, while details of intermediate size are missing. When the shape of the foreground figure is sufficiently fractal, the distribution of intensity differences will follow a power-law. See the orange peel in the Curet collection (Figure 15(b)) as an example.

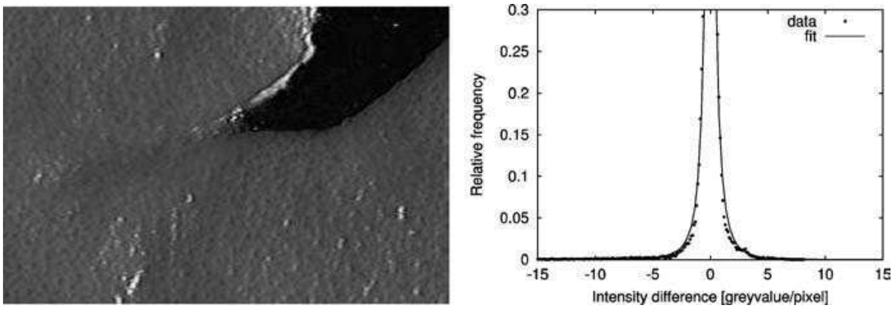
Between these two extremes, the Weibull distribution occurs. It arises when the scene is progressively fragmented by the addition of objects or detail. Such a process of sequential fragmentation results in a Weibull size distribution [5,6], where the power-law describes a single fragmentation event. Material textures, with small details extending over a limited depth range, as well as everyday scenes and even mountain views extending over considerable depth, all follow a Weibull sequential fragmentation process. When the size of the details underlying the texture is such that a receptive field typically covers part of a structure, the observed distribution follows Weibull, illustrated by the aluminum foil in the Curet collection (Figure 15(c)). The power-law process gives the extreme case for a single fragmentation, segmenting the field of view into foreground and background. On the other end of the spectra of size distributions, we have the normal distribution, representing the extreme of fragmentation beyond the visual resolution.

When the spatial detail is not randomly distributed but regular, such that there is repetition between the responses in an image, a Weibull distribution will no longer be found. For the Curet collection, ribbed paper, straw, corduroy, and corn husk break the distribution in the direction of orientation (Figure 15(d)). In these examples, however, a Weibull distribution reappears when measuring the receptive field response in the perpendicular direction. For the rug and the painted spheres in the Curet collection, the specular reflectance of the material causes a regular pattern of highlights, breaking the standard distribution in any direction.

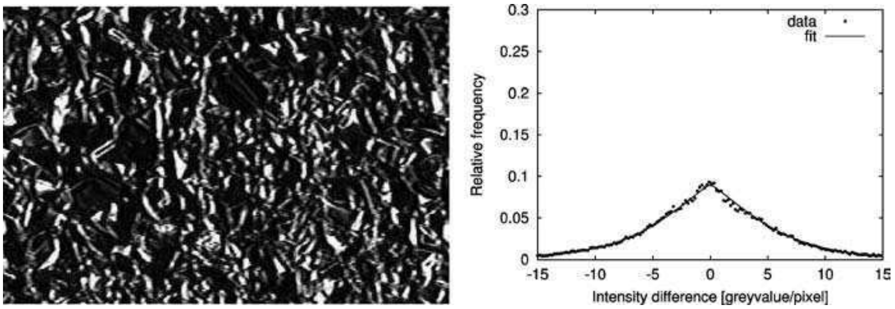
The distribution of the fragmentation exponent γ as given in Figure 16 indicates the relative importance of the power-law process and the resolution limited case of the normal distribution. Note that the γ values inherently depend on the observation resolution, hence on the pixel resolution and filter size. For high resolution images observed at small scale, power-law will be dominant. For a low resolution collection or a large observation scale, the distribution of γ-values will shift toward the normal extreme. This scaling behavior is illustrated in Figure 16. For a general vision system, observing at a variety of scales, spatial statistics will cover the complete spectrum of γ-values.



(a)

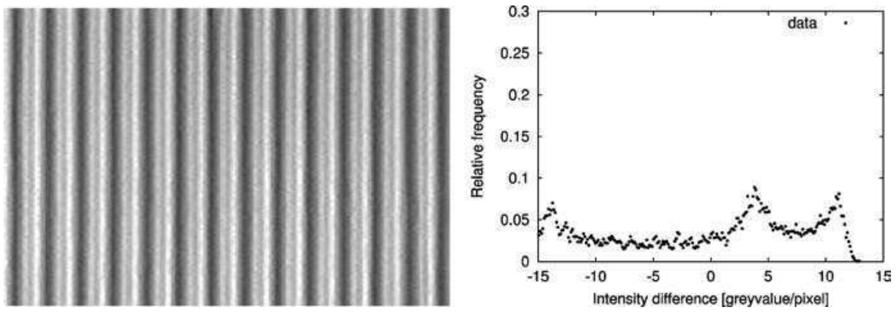


(b)



(c)

FIG. 15. Histograms of intensity differences in the x-direction for images from [9]. The rabbit fur (a) demonstrates the resolution limiting case, for which the Weibull distribution with $\gamma_x = 1.94$ approaches the normal distribution. Orange peel (b) is an example of a single object fragmentation, for which the histogram follows a power-law distribution ($\delta_x = 2.54$). The aluminum foil (c) shows a Weibull distribution with $\gamma_x = 1.27$. Ribbed paper (d) with its regular structure has a non-Weibull distribution in the x-direction, but shows a Weibull distribution for the y-direction. (*Continued on next page*.)



(d)

FIG. 15. Continued.

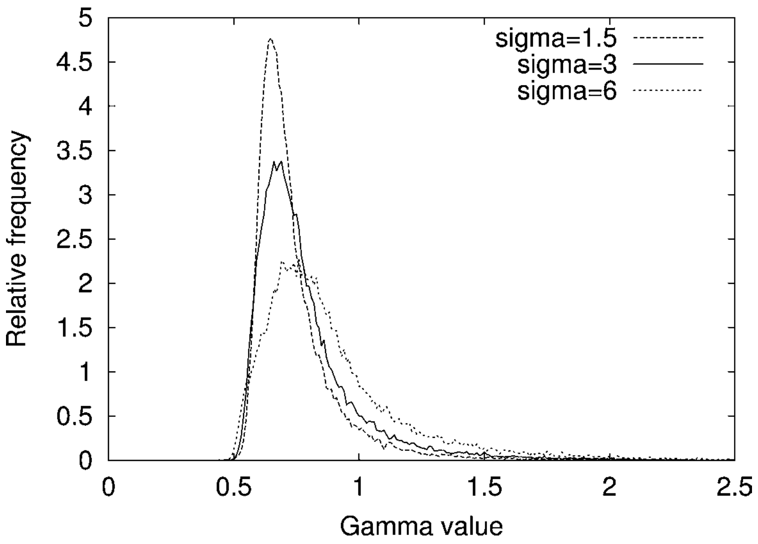


FIG. 16. Distribution of the values of γ for the Weibull distribution as estimated for the Corel collection at spatial filter sizes $\sigma = \{1.5, 3, 6\}$.

For the gradient magnitude, $f_w = \sqrt{{f_x}^2 + {f_y}^2}$, which represents a rotationally invariant filter, 32% of the Corel collection conforms to the Weibull distribution. In this case, we tested for the true Weibull distribution, as magnitude is a strictly positive entity. Furthermore, a photometric invariant $f_n = f_x/f$ is tested. In that case, 45% of the collection is Weibull distributed, whereas the power-law is no longer present. Hence, non-linear sensory combinations still result in a Weibull distribution of observed spatial detail.

The Weibull or power-law distribution is not observed when the image exhibits large uniform regions. Visual inspection of some of the Corel images not conforming to the three distributions revealed these images to show a composition of a few objects. Typical examples are objects exposed against a uniform background, and landscape images under a uniform sky. In such cases, the histogram of the filter responses consists of an addition of two Weibull distributions, with large differences between the parameters. Note that such compositions are not observed in the Curet database, which contains homogeneous textured materials.

4.3.3 Summary of Experimental Results

In conclusion, in a world full of fractal objects, with infinite observation resolution, the dominant distribution of spatial detail size would be the power-law, seen in 9% in the Corel photo stock. When the resolution is coarse, that distribution is normal, seen in 1% of the stock. The extreme cases explain just a small part of all scenes. In general, with the fine but limited resolution used for the vast majority of scenes we encounter, views we observe are fragmented and their details therefore Weibull distributed, seen in 47% of the stock and in 97% of at least one-ninth of the image. The fragmentation process breaks for repetitive patterns, causing interference between the receptive field responses. Regions where statistics break with the common statistics of natural images are likely to attract attention, see Figure 17.

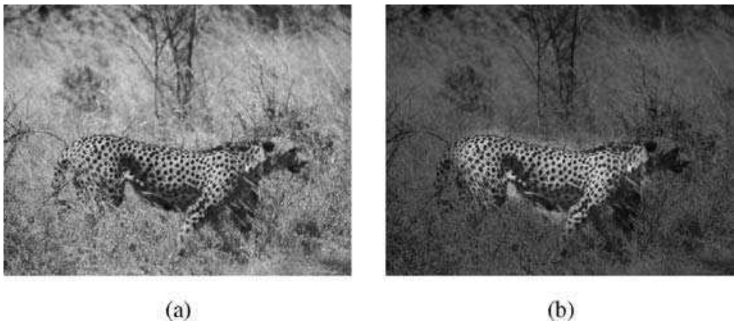


FIG. 17. Focal attention at regions where spatial statistics break with the generally observed Weibull statistics in natural scenes, that is, at regular patterns. (See Color Plate Section, Plate 12.)

5. Conclusions

It is an important research question is in which way knowledge and expectation steer focal attention to yield an efficient vision system:

- Knowledge and expectation—given a recognized phenomenon in the field of view, adaptation of the remaining statistical structure implies the tuning to its a-priori occurrences imposing new constrains on the visual input. Expectation about the scene is then inevitably used to steer attention selection. Hence, focal attention is not only triggered by visual stimuli, but is affected by knowledge about the scene, initiating conscious behavior. In this principled way, knowledge and expectation may be included at an early stage in cognitive vision.
- Learning of semantics—semantic information starts playing a role as soon as we pay attention to the visual stimulus. The selection of statistical descriptive properties by focus of attention mechanisms limits the dimensionality of the learning space drastically. Visual learning may be considered as the connection with the lingual system of denomination, language taken the carrier of semantics of information.

A long tradition in philosophy predicts or prescribes one specific module for the translation of visual impressions to linguistically formulated thoughts. However, when visual information enters the brain it is redirected to dozens of cognitive modules in the brain, interconnected by various visual pathways. That fact will complicate the model considerably, as dozens of modules are expected to be involved in the generation of linguistic information, each module contributing to a more complex description of the observed phenomena. Learning the fact that all balls are round is a task easily derived from visual evidence, which may be accomplished at a rudimentary form of cognitive vision. However, gathering visual evidence and extracting general rules to recognize memorial meetings is likely to be a much more complex task, which may be solved at completely different levels of consciousness.

REFERENCES

[1] Barlow H.B., "Possible principles underlying the transformation of sensory messages", in: W. Rosenblith (Ed.), *Sensory Communication*, MIT Press, Cambridge, MA, 1961, p. 217.

[2] Bell A.J., Sejnowski T.J., "The independent components of natural scenes are edge filters", *Vision Res.* **37** (1997) 3327–3338.

[3] Bigün J., Granlund G.H., Wiklund J., "Multidimensional orientation estimation with applications to texture analysis and optic flow", *IEEE Trans. Pattern Anal. Machine Intelligence* **13** (1991) 775–790.

[4] Bovik A., Clark M., Geisler W., "Multichannel texture analysis using localized spatial filters", *IEEE Trans. Pattern Anal. Machine Intelligence* **12** (1) (1990) 55–73.

[5] Brown W.K., "A theory of sequential fragmentation and its astronomical applications", *J. Astrophys. Astr.* **10** (1989) 89–112.

[6] Brown W.K., Wohletz K.H., "Derivation of the Weibull distribution based on physical principles and its connection to the Rosin–Rammler and lognormal distributions", *J. Appl. Phys.* **78** (1995) 2758–2763.

[7] Canny F.J., "A computational approach to edge detection", *IEEE Trans. Pattern Anal. Machine Intelligence* **8** (1986) 679–698.

[8] Dana K.J., Nayar S.K., "Histogram model for 3d textures", in: *Proc. CVPR.*, IEEE Computer Society, Los Alamitos, CA, 1998.

[9] Dana K.J., van Ginneken B., Nayar S.K., Koenderink J.J., "Reflectance and texture of real world surfaces", *ACM Trans. Graphics* **18** (1999) 1–34.

[10] D'Zmura M., Lennie P., "Mechanisms of color constancy", *J. Opt. Soc. Amer. A* **3** (10) (1986) 1662–1672.

[11] Field D.J., "Relations between the statistics of natural images and the response properties of cortical cells", *J. Opt. Soc. Amer. A* **4** (1987) 2370–2393.

[12] Filliben J.J., et al., *NIST/SEMTECH Engineering Statistics Handbook*, NIST, Gaithersburg, 2002, http://www.itl.nist.gov/div898/handbook.

[13] Finlayson G.D., "Color in perspective", *IEEE Trans. Pattern Anal. Machine Intelligence* **18** (10) (1996) 1034–1038.

[14] Florack L., *Image Structure*, Kluwer Academic Publishers, Dordrecht, 1997.

[15] Florack L.M.J., Romeny B.M.t.H., Koenderink J.J., Viergever M.A., "Scale and the differential structure of images", *Image Vision Comput.* **10** (6) (1992) 376–388.

[16] Florack L.M.J., Romeny B.M.t.H., Koenderink J.J., Viergever M.A., "Cartesian differential invariants in scale-space", *J. Math. Imaging Vision* **3** (4) (1993) 327–348.

[17] Foster D.H., Nascimento S.M.C., "Relational colour constancy from invariant cone-excitation ratios", *Proc. R. Soc. London B* **257** (1994) 115–121.

[18] Freeman W.T., Adelson E.H., "The design and use of steerable filters", *IEEE Trans. Pattern Anal. Machine Intelligence* **13** (1991) 891–906.

[19] Funt B.V., Finlayson G.D., "Color constant color indexing", *IEEE Trans. Pattern Anal. Machine Intelligence* **17** (5) (1995) 522–529.

[20] Geusebroek J.M., Dev A., van den Boomgaard R., Smeulders A.W.M., Cornelissen F., Geerts H., "Color invariant edge detection", in: *Scale-Space Theories in Computer Vision*, in: *Lecture Notes in Comput. Sci.*, vol. 1682, Springer-Verlag, Berlin/New York, 1999, pp. 459–464.

[21] Geusebroek J.M., Smeulders A.W.M., "A physical explanation for natural image statistics", in: Chantler M. (Ed.), *Proceedings of the 2nd International Workshop on Texture Analysis and Synthesis (Texture 2002)*, Heriot–Watt University, Berlin/New York, 2002, pp. 47–52.

[22] Geusebroek J.M., Smeulders A.W.M., "Fragmentation in the vision of scenes", in: *Proc. 9th Internat. Conf. Comput. Vision, vol. 1*, IEEE Computer Society, Los Alamitos, CA, 2003, pp. 130–135.

[23] Geusebroek J.M., Smeulders A.W.M., Geerts H., "A minimum cost approach for segmenting networks of lines", *Internat. J. Comput. Vision* **43** (2) (2001) 99–111.

[24] Geusebroek J.M., Smeulders A.W.M., van de Weijer J., "Fast anisotropic Gauss filtering", *IEEE Trans. Image Processing* **12** (8) (2003) 938–943.

[25] Geusebroek J.M., van den Boomgaard R., Smeulders A.W.M., Dev A., "Color and scale: The spatial structure of color images", in: Vernon D. (Ed.), *Sixth European Conference on Computer Vision (ECCV), vol. 1*, in: *Lecture Notes in Comput. Sci.*, vol. 1842, Springer-Verlag, Berlin/New York, 2000, pp. 331–341.

[26] Geusebroek J.M., van den Boomgaard R., Smeulders A.W.M., Geerts H., "Color invariance", *IEEE Trans. Pattern Anal. Machine Intelligence* **23** (12) (2001) 1338–1350.

[27] Geusebroek J.M., van den Boomgaard R., Smeulders A.W.M., Gevers T., "Color constancy from physical principles", *Pattern Recognition Lett.* **24** (11) (2003) 1653–1662.

[28] Gevers T., Smeulders A.W.M., "Color based object recognition", *Pattern Recognition* **32** (1999) 453–464.

[29] Gevers T., Stokman H.M.G., "Classification of color edges in video into shadow-geometry, highlight, or material transitions", *IEEE Trans. Multimedia* **5** (2003) 237–244.

[30] Gnedenko B.V., Kolmogorov A.N., *Limit Distributions for Sums of Independent Random Variables*, Addison–Wesley, Boston, 1968.

[31] Gool L.J.V., Moons T., Pauwels E.J., Oosterlinck A., "Vision and Lie's approach to invariance", *Image Vision Comput.* **13** (4) (1995) 259–277.

[32] Grenander U., Srivastava A., "Probability models for clutter in natural images", *IEEE Trans. Pattern Anal. Machine Intelligence* **23** (4) (2001) 424–429.

[33] Hering E., *Outlines of a Theory of the Light Sense*, Harvard University Press, Cambridge, MA, 1964.

[34] "ITU-R Recommendation BT. 709, Basic parameter values for the HDTV standard for the studio and for international programme exchange", Tech. Rep. BT. 709 (formerly CCIR Rec. 709), ITU, 1211 Geneva 20, Switzerland, 1990.

[35] Jain A., Healey G., "A multiscale representation including opponent color features for texture recognition", *IEEE Trans. Image Processing* **7** (1) (January 1998) 124–128.

[36] Jolion J.M., "Image and the Benford's law", *J. Math. Imaging Vision* **14** (2001) 73–81.

[37] Judd D.B., Wyszecki G., *Color in Business, Science, and Industry*, Wiley, New York, 1975.

[38] Kalitzin S., ter Haar Romeny B., Viergever M., "Invertible orientation bundles on 2d scalar images", in: *Scale-Space Theories in Computer Vision*, Springer-Verlag, Berlin/New York, 1997, pp. 77–88.

[39] Koenderink J.J., "The structure of images", *Biol. Cybern.* **50** (1984) 363–370.

[40] Koenderink J.J., van Doorn A.J., "Receptive field families", *Biol. Cybern.* **63** (1990) 291–297.

[41] Koenderink J.J., van Doorn A.J., "Illuminance texture due to surface mesostructure", *J. Opt. Soc. Amer. A* **13** (3) (1996) 452–463.

[42] Koenderink J.J., van Doorn A.J., Dana K.J., Nayar S., "Bidirectional reflection distribution function of thoroughly pitted surfaces", *Internat. J. Comput. Vision* **31** (1999) 129–144.

[43] Kubelka P., “New contribution to the optics of intensely light-scattering materials, Part I”, *J. Opt. Soc. Amer.* **38** (5) (1948) 448–457.
[44] Kubelka P., Munk F., “Ein beitrag zur optik der farbanstriche”, *Z. Techn. Physik* **12** (1931) 593.
[45] Land E.H., “The retinex theory of color vision”, *Sci. Amer.* **237** (1977) 108–128.
[46] Lee A.B., Mumford D., Huang J., “Occlusion models for natural images: A statistical study of a scale-invariant dead leaves model”, *Internat. J. Comput. Vision* **41** (2001) 35–59.
[47] Lindeberg T., *Scale-Space Theory in Computer Vision*, Kluwer Academic Publishers, Boston, 1994.
[48] Mallat S.G., “A theory for multiresolution signal decomposition: The wavelet representation”, *IEEE Trans. Pattern Anal. Machine Intelligence* **11** (1989) 674–693.
[49] Olver P., Sapiro G., Tannenbaum A., “Differential invariant signatures and flows in computer vision: A symmetry group approach”, in: ter Haar Romeny B.M. (Ed.), *Geometry-Driven Diffusion in Computer Vision*, Kluwer Academic Publishers, Boston, 1994.
[50] Pentland A.P., “Fractal-based description of natural scenes”, *IEEE Trans. Pattern Anal. Machine Intelligence* **6** (1984) 661–674.
[51] Pentland A.P., “Linear shape from shading”, *Internat. J. Comput. Vision* **4** (1990) 153–163.
[52] Perona P., “Steerable-scalable kernels for edge detection and junction analysis”, *Image Vision Comput.* **10** (1992) 663–672.
[53] Pluta M., *Advanced Light Microscopy, vol. 1*, Elsevier, Amsterdam, 1988.
[54] Rigdon S.E., “Testing goodness-of-fit for the power law process”, *Commun. Statist. Theory Methods* **18** (1989) 4665–4676.
[55] Ruderman D.L., Bialek W., “Statistics of natural images: Scaling in the woods”, *Phys. Rev. Lett.* **73** (1994) 814–817.
[56] Sapiro G., “Color and illuminant voting”, *IEEE Trans. Pattern Anal. Machine Intelligence* **21** (11) (1999) 1210–1215.
[57] Seinstra F., Koelma D., Geusebroek J., “A software architecture for user transparent parallel image processing”, *Parallel Comput.* **28** (7–8) (2002) 967–993.
[58] Seinstra F.J., Koelma D., “User transparency: A fully sequential programming model for efficient data parallel image processing”, *Concurrency Comput.* **16** (6) (2004) 611–644.
[59] Shafer S.A., “Using color to separate reflection components”, *Color Res. Appl.* **10** (4) (1985) 210–218.
[60] Simoncelli E.P., “Modeling the joint statistics of images in the wavelet domain”, in: *Proc. SPIE, vol. 3813*, SPIE, Bellingham, WA, 1999, pp. 188–195.
[61] Srivastava A., Liu X., Grenander U., “Universal analytical forms for modeling image probabilities”, *IEEE Trans. Pattern Anal. Machine Intelligence* **24** (9) (2002) 1200–1214.
[62] Steger C., “An unbiased detector of curvilinear structures”, *IEEE Trans. Pattern Anal. Machine Intelligence* **20** (1998) 113–125.
[63] Stephens M.A., “EDF statistics for goodness of fit and some comparisons”, *J. Amer. Statist. Assoc.* **69** (1974) 730–737.
[64] ter Haar Romeny B.M. (Ed.), *Geometry-Driven Diffusion in Computer Vision*, Kluwer Academic Publishers, Boston, 1994.

[65] van Ginkel M., Verbeek P.W., van Vliet L.J., “Improved orientation selectivity for orientation estimation”, in: Frydrych M., Parkkinen J., Visa A. (Eds.), *Proceedings of the 10th Scandinavian Conference on Image Analysis*, 1997, pp. 533–537.
[66] van Ginneken B., Koenderink J.J., Dana K.J., “Texture histograms as a function of irradiation and viewing direction”, *Internat. J. Comput. Vision* **31** (1999) 169–184.
[67] van Hateren J.H., van der Schaaf A., “Independent component filters of natural images compared with simple cells in primary visual cortex”, *Proc. R. Soc. London B* **265** (1998) 359–366.
[68] Weibull W., “A statistical distribution function of wide applicability”, *J. Appl. Mech.* **18** (1951) 293–297.
[69] Witkin A.P., “Scale-space filtering”, in: *Proc. Image Understanding*, 1984, pp. 79–95.
[70] Wyszecki G., Stiles W.S., *Color Science: Concepts and Methods, Quantitative Data and Formulae*, Wiley, New York, 1982.
[71] Yang Z., Purves D., “A statistical explanation of visual space”, *Nature Neurosci.* **6** (2003) 632–640.
[72] Young R.A., “The Gaussian derivative theory of spatial vision: Analysis of cortical cell receptive field line-weighting profiles”, Tech. Rep. GMR-4920, General Motors Research Center, Warren, MI, 1985.

This page intentionally left blank

Verification and Validation and Artificial Intelligence

TIM MENZIES

Computer Science
Portland State University
Oregon
USA
tim@menzies.us

CHARLES PECHEUR

Université Catholique de Louvain
Dept. of Computer & Software Engineering
2 Place Sainte-Barbe
B-1348 Louvain-la-Neuve
Belgium
pecheur@info.ucl.ac.be

Abstract
Artificial Intelligence (AI) is useful. AI can deliver more functionality for reduced cost. AI should be used more widely but won't be unless developers can trust adaptive, nondeterministic, or complex AI systems.

Verification and validation is one method used by software analysts to gain that trust. AI systems have features that make them hard to check using conventional V&V methods. Nevertheless, as we show in this chapter, there are enough alternative readily-available methods that enable the V&V of AI software.

1. Introduction

Artificial Intelligence (AI) is no longer some bleeding technology that is hyped by its proponents and mistrusted by the mainstream. In the 21st century, AI is not necessarily amazing. Rather, it is often *routine*. Evidence for the routine and dependable nature of AI technology is everywhere (see the list of applications in [1]).

The AI approach has always been at the forefront of computer science research. Many hard tasks were first tackled and solved by AI researchers before they transitioned to standard practice. Those examples include time-sharing operating systems, automatic garbage collection, distributed processing, automatic programming, agent systems, reflective programming and object-oriented programming.

This tradition of AI leading the charge and solving the hard problems continues to this day. AI offers improved capabilities at a reduced operational cost. For example, Figure 1 describes the AI used in NASA's Remote Agent Experiment (RAX) [2]. For over a day, this system ran a deep-space probe without any help from mission control. Such AI-based autonomy is essential to future deep space missions. NASA needs such autonomous software so that deep space probes can handle unexpected or important events billions of miles away from earth when they are hours away from assistance by mission control.

AI software can be complex and the *benefits* of complexity are clear. Some applications such as the RAX described in Figure 1 are inherently complex and require an extension to existing technology.

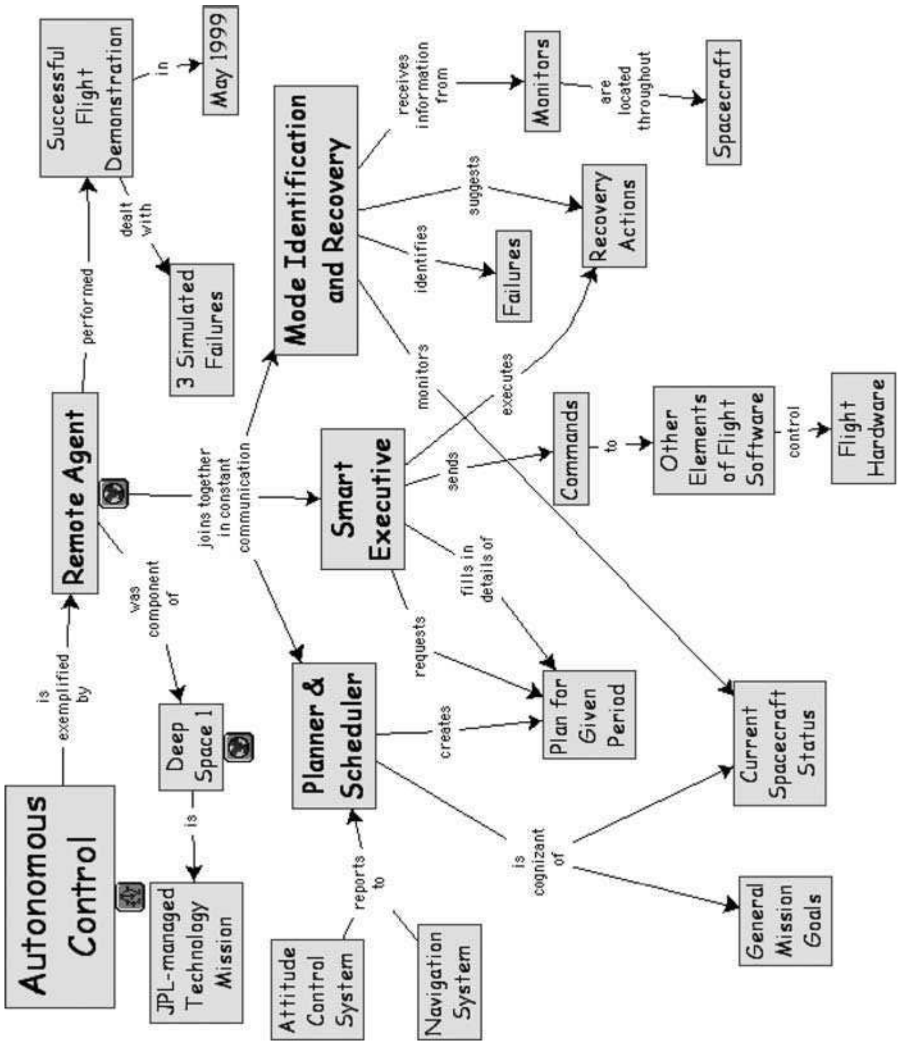

Autonomous Control
is exemplified by
Remote Agent
performed
Successful Flight Demonstration
in
May 1999
dealt with
3 Simulated Failures
was component of
Deep Space 1
is
JPL-managed Technology Mission
joins together in constant communication
Mode Identification and Recovery
receives information from
Monitors
are located throughout
Spacecraft
suggests
Recovery Actions
identifies
Failures
monitors
Smart Executive
executes
sends
Commands
to
Other Elements of Flight Software
control
Flight Hardware
fills in details of
requests
Plan for Given Period
Planner & Scheduler
creates
is cognizant of
Current Spacecraft Status
General Mission Goals
reports to
Attitude Control System
Navigation System

FIG. 1. The Remote Agent Experiment, from http://cmex-www.arc.nasa.gov/CMEX/RemoteAgent.html. NASA's deep space missions require *autonomous spacecrafts* that don't rely on ground control. The Remote Agent Experiment (RAX) was an experiment in autonomy technology: for two days, RAX provided on-board AI control for the Deep Space One probe, while it was 60,000,000 miles from Earth. RAX had three main components. The *Planner & Scheduler* (*PS*) took general goals and determined detailed activities needed to achieve the goals. If a hardware problem developed that prevents execution of the plan, the planner made a new plan, taking into account degraded capabilities. The *Smart Executive* (*EXEC*) interpreted the plans and added more detail to them, then issued commands to rest of the satellite. Lastly, the *Mode Identification and Recovery* (*MIR*) component (based on the Livingstone diagnosis system) acted like a doctor, monitoring the spacecraft's health. If something went wrong, MIR would detect it and report it to EXEC. Exec could then consult the "doctor" for simple procedures that may quickly remedy the problem. If those simple procedures could not resolve the problem, EXEC asked PS for a new plan that still achieved the mission goals while accounting for the degraded capabilities. (See Color Plate Section, Plate 13.)

However, the *cost* of complexity is that complex systems are harder to understand and hence harder to test. Complex systems like can hide intricate interactions which, if they happen during flight, could compromise the mission. For example, despite a year of extensive testing, when Remote Agent was first put in control of NASA's Deep Space One mission, it froze because of a software deadlock problem (RAX was re-activated two days later and successfully completed all its mission objectives). After analysis, it turns out that the deadlock was caused by a highly unlikely race condition between two concurrent threads inside Remote Agent's executive. The scheduling conditions that caused the problem to manifest never happened during testing but indeed showed up in flight [2].

Hardware engineers solve such problems with hardware redundancy (when one component fails, its back-up wakes up and takes over). However, redundancy may not solve software reliability problems. Hardware components fail statistically because of wear or external damage. Software components programs fail almost exclusively due to latent design errors. Failure of an active system is thus highly correlated with failure of a duplicate back-up system (unless the systems use different software designs, as in the Space Shuttle's on-board computers). The classic example of how redundancy does not solve reliability is the infamous Ariane 5 rocket explosion where the backup Internal Reference system failed on the same software bug several milliseconds before the primary system failed.[1]

If redundancy may not increase the reliability of AI software, what else should we do to check our AI software? How should standard verification and validation (V&V) be modified to handle AI systems? What are the traps of V&V of AI software? What leverage for V&V can be gained from the nature of AI software? This chapter offers

[1] See http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

an overview of the six features of AI systems that a V&V analysts must understand. An AI system may be:

(1) *complex* software;
(2) *declarative* and *model-based*, and sometimes *knowledge-level* software;
(3) *nondeterministic* or even *adaptive* software.

Fortunately, not all AI systems have all the above features since each can come with a *significant cost*. However, each of these features grants *significant benefits* that can make the costs acceptable.

The rest of this chapter is structured around this list of features of an AI system. Each feature will be defined and their associated cost and benefits will be discussed. To help the reader who is a not an AI specialist, many of our sections start with short tutorials.

Note that the approach of this chapter is different to the traditional reviews of AI verification (e.g., [3–15]) or AI (e.g., [16–24]). To some extent, this difference is due to the domain within which we work. The first generation of expert system validation tools worked mostly on the rule-based or frame-based representations that were common in the 1980s and 1990s. Since then, much has changed. We work with complex model-based autonomous vehicles of the kind flown by NASA. Hence, while other articles offer success stories with rule-based or frame-based representations (e.g., [25–28] and Chapters 8, 30, 31, 34 in [29]), this chapter focuses more on the features of modern AI that distinguishes it from conventional procedural software; e.g., nondeterministic adaptive knowledge-level systems.

If the reader is interested in that traditional view, then they might care to read the references in this paragraph (in particular [24,4,28,29]) or one of the many excellent on-line bibliographies on V&V of AI systems.[2]

2. AI Software Can Be Complex

The rest of this chapter stresses what is *different* about AI systems and how those differences affect V&V for AI. Before moving on to that material, this section observes that AI software is still software, albeit sometimes quite complex software. Hence, methods developed for assessing normal software systems still apply to AI systems. V&V analysts should view this chapter as techniques that *augment*, not *replace* their standard V&V methods such as peer reviews, automated test suites, etc. [30–32].

[2] E.g., http://www.csd.abdn.ac.uk/~apreece/Research/vvbiblio.html.

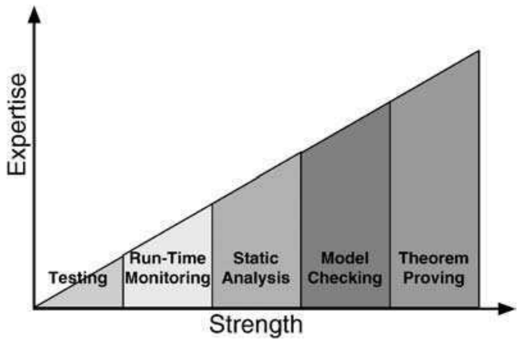


FIG. 2. The verification methods spectrum. (See Color Plate Section, Plate 14.)

In this section, we review the state-of-the-art in verifying complex software. Many of these techniques are routinely applied at NASA when verifying complex AI systems such as RAX.

We start with conventional *testing* as a baseline, then introduce more advanced formal methods: *run-time monitoring*, *static analysis*, *model checking* and *theorem proving*. Those methods vary in the strength of the verdicts they provide, as well as the level of expertise they require—generally speaking, more thorough approaches require more expertise. This can be laid out as the "verification methods spectrum" shown in Figure 2.[3]

2.1 Testing

Traditionally, software verification is done using scenario-based testing. The system to be verified is embedded into a test harness that connects to the inputs and outputs of that component, and drives it through a suite of test runs. Each test run is an alternated sequence of provided inputs and expected outputs, corresponding to one scenario of execution of the tested component. An error is signaled when the received output does not meet the expected one.

Even for simple systems, the design and maintenance of test suites is a difficult and expensive process. It requires a good understanding of the system to be tested, to ensure that a maximum number of different situations are covered using a minimum number of test cases. Running the tests is also a time-consuming task, because the

[3] Adapted from John Rushby, in slides of [33]. This spectrum is notional and expresses trends rather than absolute truths. In particular, a comparative experimental evaluation conducted at NASA [34] concluded that static analysis may require a deep understanding of its underlying algorithms to be used effectively.

whole program code has to be executed and everything must be re-initialized before each test run. In the development of complex systems, it is quite common that testing the software actually takes more resources than developing it.

While traditional testing may be enough for more conventional software, it falls short for complex AI software, mainly because the range of situations to be tested is incomparably larger. A typical intelligent program implicitly incorporates responses to a very large space of possible inputs (for example, consider the possible interaction sequences that an autonomous agent like RAX may face). The internal state of the program is typically huge, dynamically allocated (heap memory, garbage collection) and may involve complex data structures (knowledge, rules, theories). It depends on the past history of the system in intricate ways. AI systems may involve several concurrent components, or other sources of nondeterminism.

Because of all these factors, a test suite can only exercise a very limited portion of the possible configurations, and it is very hard to assess how much has been covered and therefore measure the residual risk of errors.

2.2 Run-Time Monitoring

Run-time monitoring, or run-time verification, refers to advanced techniques for scrutinizing artifacts from an executing program (variables, events, typically made available through program instrumentation), in order to detect effective or potential misbehaviors or other useful information.

Simple runtime monitoring methods have been used for decades. For example, it is standard practice for programmers to add conditionals to their code that print warning messages if some error condition is satisfied, or to generate and scan additional logging messages for debugging purposes. In essence, run-time verification automates the otherwise strenuous and error-prone task of reviewing these logs manually.

This analysis can be conducted after-the-fact on stored execution traces, but also on-the-fly while the program is executing. In this latter case, the need to store the execution trace is alleviated, and the monitor can also trigger recovery mechanisms as part of a software fault protection scheme.

Recently, the sophistication of runtime monitoring methods has dramatically increased. Commercial tools such as Temporal Rover [35] can instrument a program to execute inserted code fragments based on complex conditions expressed as *temporal logic formulae* (see Figure 3). New algorithms can detect suspicious concurrent programming patterns (data races [36], deadlocks [37]) that are likely to cause an error, even if no error occurs on the observed trace.

Runtime monitoring typically requires little computing resources and therefore scales up well to very large systems. On the other hand, it will only observe a limited

A temporal logic is a classical logic augmented with operators to reason about the evolution of the model over time. Temporal logic allows to express conditions over time, such as "any request is always eventually fulfilled." For example, (propositional) *linear temporal logic*, or LTL uses the following operators:

Property	Reads	Means p holds ...
$\circ p$	"next p"	... in the next state
$\Box p$	"henceforth p"	... in all future states
$\Diamond p$	"eventually p"	... in some future state
$p \cup q$	"p until q"	... in all states until q holds

FIG. 3. About temporal logic.

number of executions and thus gives only uncertain results. In the case of error predictions, it can also give false negatives, i.e., flag potential errors that cannot actually occur.

One example of a runtime monitoring system is Java Path Explorer (JPaX) [37]. Given a specification of properties to be monitored, JPaX instruments a Java program to generate a trace of relevant events. JPaX also produces an observer program that reads that trace and verifies the properties. The trace can be streamed through a socket, to allow local or remote on-the-fly monitoring. Both user-provided temporal logic conditions and generic deadlock and data race conditions can be monitored.

In [38], run-time monitoring is applied (in conjunction with automated test-case generation) to verify the controller of the K9 planetary rover. The controller is a large multi-threaded program (35,000 lines of C++) that controls the rover according to a flexible plan generated by a planning program. Each test case amounts to a plan, for which a set of temporal properties are derived, according to the semantics of the plan. The EAGLE system [39] is then used to monitor this properties. This system is fully automated and unveiled a flaw in plan interpretation, that indeed occurred during field tests before it was fixed in the controller. A potential deadlock and a data race were also uncovered.

2.3 Static Analysis

Static Analysis consists in exploring the structure of the source code of a program without executing it. It is an important aspect of compiler technology, supporting type analysis and compiler optimizations. A good reference textbook on the topic can be found in [40]. Static analysis includes such aspects as *control- and data-flow analysis* (tracking the flows of execution and the propagation of data in the code), *abstract interpretation* (computing abstract approximations of the allowed ranges of program variables) and *program slicing* (capturing the code portions that are relevant to a particular set of variables or a function).

In principle, static analysis can be applied to source code early in the development and is totally automatic. There is, however, a trade-off between the cost and the precision of the analysis, as the most precise algorithms have a prohibitive complexity. More efficient algorithms make approximations that can result in a large number of false positives, i.e., spurious error or warning messages.

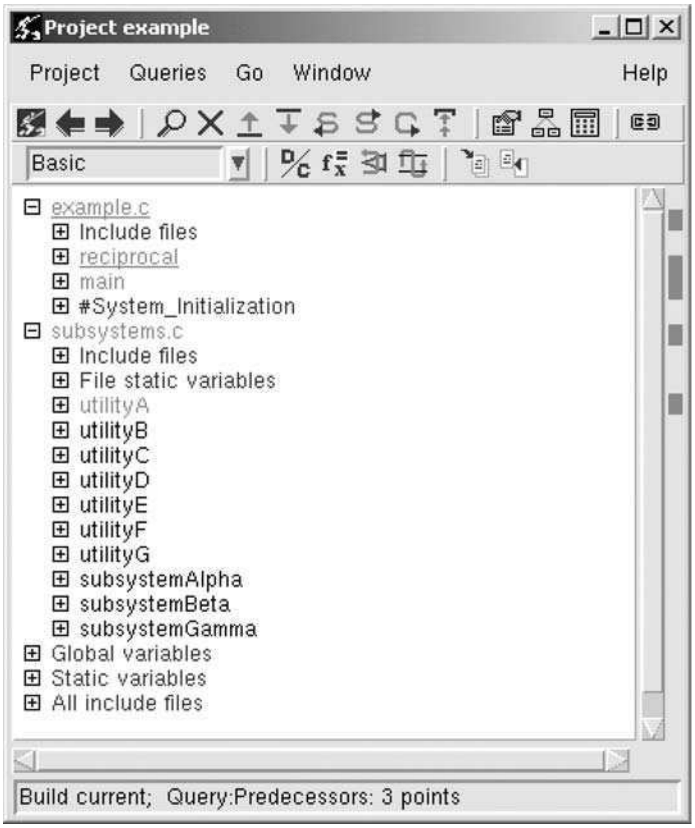
Two commercial static analysis tools are Grammatech's CodeSurfer and the PolySpace Verifier [41]. Figure 4 shows CodeSurfer using a static control flow analysis to find the code *not* reachable from the `main` function. Such code is dead code and represents either over-specification (i.e., analysts exploring too many special cases) or code defects (e.g., the wrong items are present in a conditional). In Figure 4 the code sections reached from `main` are shown with dark colored marks in the right-hand-side of the display. Note that in this case, most of the code is *not* reachable from the `main` program.

PolySpace uses abstract interpretation to find potential run-time errors in (C or Ada) programs such as:

- access to non-initialized variables,
- unprotected shared variables in concurrent programs,
- invalid pointer references,
- array bound errors,
- illegal type conversions,
- arithmetic errors (overflow, underflow, division by zero, ...),
- unreachable code.

The output of the tool consists of a color-coded version of the program source code, as shown on Figure 5. Green code is guaranteed free from errors, red code is sure to cause errors, orange code may cause errors (depending on execution paths or because the analysis was inconclusive), and grey code is unreachable.

Once the code to be analyzed has been identified, static analysis tools such as CodeSurfer and PolySpace are fully automatic. However, the experiment reported in [34] concludes that their use can be a labor-intensive, highly iterative process, in order to: (i) isolate a proper self-standing code set to be analyzed, (ii) understand and fix the reported red errors, (iii) adjust analysis parameters to reduce the number of false alarms. The last point can be very detrimental, as the analysis tends to return a very large number of mostly spurious "orange" warnings, making it very hard to identify real errors. The C Global Surveyor tool (CGS), currently under development at NASA, drastically reduces this problem by specializing the analysis algorithms to the coding practices of a specific class of applications [42].



```
⊟ example.c
    ⊞ Include files
    ⊞ reciprocal
    ⊞ main
    ⊞ #System_Initialization
⊟ subsystems.c
    ⊞ Include files
    ⊞ File static variables
    ⊞ utilityA
    ⊞ utilityB
    ⊞ utilityC
    ⊞ utilityD
    ⊞ utilityE
    ⊞ utilityF
    ⊞ utilityG
    ⊞ subsystemAlpha
    ⊞ subsystemBeta
    ⊞ subsystemGamma
⊞ Global variables
⊞ Static variables
⊞ All include files
```

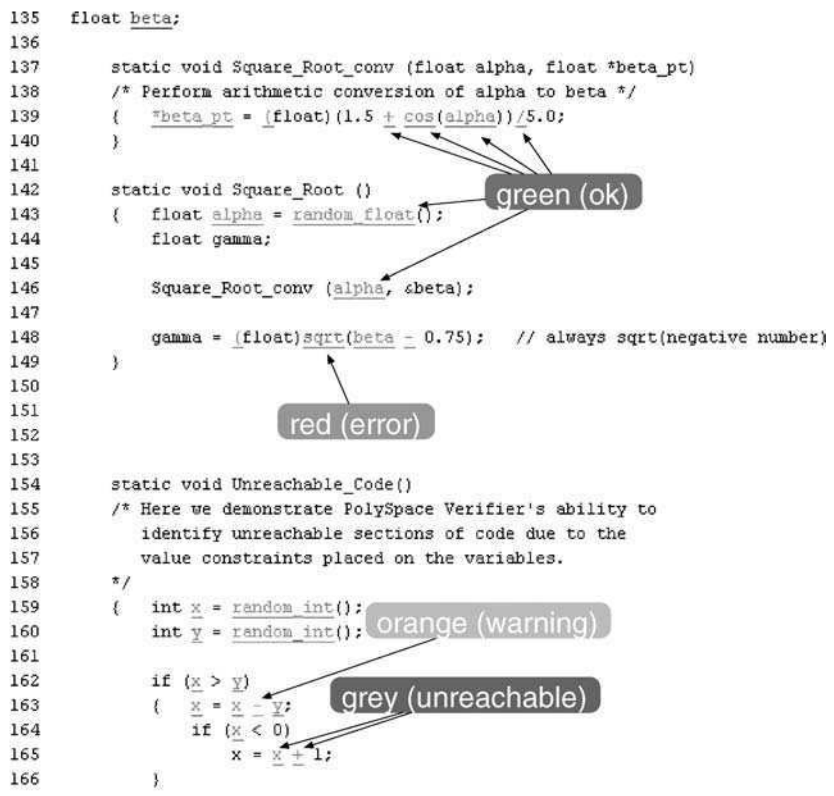
Build current; Query:Predecessors: 3 points

FIG. 4. Slicing in Grammatech's CodeSurfer tool (see http://www.grammatech.com/products/codesurfer/example.html). (See Color Plate Section, Plate 15.)

2.4 Model Checking

Model Checking consists in verifying that a system (or a model thereof) satisfies a property by exhaustively exploring all its reachable states. Model checking was invented in the 1980s to analyze communication protocols [43,44], and is now routinely employed in verification of digital hardware. Several mature and powerful model checkers are available and widely used in the research community; SPIN [45, 46] and SMV [47,48] are probably the best known. See [49] for a comprehensive theoretical presentation, and [50] for a more practical introduction.

A model checker searches all pathways of the system looking for ways to violate the property. This requires that this state space be finite and tractable: model checking



```
float beta;

    static void Square_Root_conv (float alpha, float *beta_pt)
    /* Perform arithmetic conversion of alpha to beta */
    {   *beta_pt = (float)(1.5 + cos(alpha))/5.0;
    }

    static void Square_Root ()
    {   float alpha = random_float();
        float gamma;

        Square_Root_conv (alpha, &beta);

        gamma = (float)sqrt(beta - 0.75);   // always sqrt(negative number)
    }




    static void Unreachable_Code()
    /* Here we demonstrate PolySpace Verifier's ability to
       identify unreachable sections of code due to the
       value constraints placed on the variables.
    */
    {   int x = random_int();
        int y = random_int();

        if (x > y)
        {   x = x - y;
            if (x < 0)
                x = x + 1;
        }
```

FIG. 5. An example of color-coded source code produced by the PolySpace Verifier. From http://www.polyspace.com/datasheets/c_psde.htm (callouts added). (See Color Plate Section, Plate 16.)

is limited by the state space explosion problem, where the number of states can grow exponentially with the size of the system.[4] Tractability is generally achieved by abstracting away from irrelevant details of the system. If a violation is found, the model checker returns the counter examples showing exactly how the property is violated. Such counter examples are useful in localizing and repairing the source of the violation.

Model checking requires the construction of two models:

- The *systems model* is an abstract description of the dynamic, generally concurrent behavior of a program or system. For tractability reasons, it is usually not

[4] When the state space is too large or even infinite, model checking can still be applied: it will not be able to prove that a property is satisfied, but can still be a very powerful error-finding tool.

the complete code from the implementation but rather some abstract verification model of the application, capturing only the essential features that are relevant to the properties to be checked.

- The *properties model* is a specification of the requirements that should hold across the systems model. The properties model is often expressed as a temporal logic constraint.[5]

Model checkers often impose their own modeling language for both the systems and the properties, though more and more tools now apply directly to common design and programming languages (UML, Java), either natively or through translation.

Although model checkers are automatic tools, the benefits of model checking come at a cost that is often very significant. That cost can be divided into three components:

- *writing cost*: the initial cost of developing the systems model and the properties model, in a form accepted by the model checker;
- *running cost*: the cost of actually performing the model checking, as many times as necessary; and
- *re-writing cost*: the cost of iteratively modifying the models until model checking can complete successfully and provide acceptable results.

With traditional model checking tools, both the systems model and the properties model have to be written in their own tool-specific language, which often results in high *writing cost*. In particular, scarce and expensive PhD-level mathematical expertise may be required to properly encode properties in formal temporal logic.

Once models have been completed, the model checker explores all the interactions within the program. In the worst case, the number of such interactions is exponential on the number of different assignments to variables in the system. Hence, the *running cost* of this query can be excessive. This large running cost generally forces analysts to simplify the formal models, for example by removing parts or functions, abstracting away some details, or restricting the scope of the analysis. Such a rewrite incurs the *rewrite cost*. On the other hand, simplification may take away elements that are relevant to the properties being verified, so a balance must be found between tractability and accuracy of the analysis. This typically requires several iterations and a significant amount of expertise in both the tools and the application being verified.

Much research has tried to reduce these costs. The writing of systems models can be avoided by applying model checkers directly to readily available representations,

[5] Depending on the model checker being used, properties are sometimes expressed in other forms, such as invariants that must hold in every state, regular expressions that execution traces must match, or other dynamic system models whose executions must agree (in some precisely defined sense) with those of the verified system (see, e.g., [51]).

such as design models or program code. For example, the BANDERA system extracts systems models from JAVA source code and feeds them into SMV or SPIN machine [52]. Java PathFinder applies model checking directly to Java bytecode, using its own custom Java virtual machine [53]. Other approaches such as SCR [54–56] or simple influence diagrams [57] propose simplified modeling environments where users can express their models in a simple and intuitive framework, and then map them into model checkers such as SPIN.

Interestingly enough, AI software can offer good opportunities for model checking, as a consequence of using model-based or knowledge-based approaches. This aspect is discussed in detail in Section 3; for now let us point out that the models that are interpreted in AI systems tend to be abstract enough to be readily amenable to model checking, with only minor adaptation writing costs. Pecheur and Simmons' verification of models used for autonomous diagnosis using the SMV symbolic model checker is a good example [58].

On the properties side, Dwyer, Avrunin and Corbett [59,60] have developed a taxonomy of *temporal logic patterns* that covers most of the properties observed in real-world applications. For each pattern, they have defined an expansion from the intuitive pseudo-English form of the pattern to a formal temporal logic formula.[6] These patterns are simpler and more intuitive than their logical counterpart, and shield the analysts away from the complexity of formal logics. For example, consider the following property on an elevator:

> *Always, the elevator door never opens more than twice between the source floor and the destination floor.*

If P is the elevator doors opening and Q is the arrival at the source floor and R arrival at the destination floor, then the temporal logic formula for this property is:

$$\Box\big((Q \wedge \Diamond R) \rightarrow \big((\neg P \wedge \neg R) \mathbin{\mathsf{U}} \big(R \vee \big((P \wedge \neg R) \mathbin{\mathsf{U}} \big(R \vee \big((\neg P \wedge \neg R) \\ \mathbin{\mathsf{U}} \big(R \vee \big((P \wedge \neg R) \mathbin{\mathsf{U}} \big(R \vee (\neg P \vee R)\big)\big)\big)\big)\big)\big)\big)\big)\big).$$

In contrast, this property can be expressed using the "bounded existence" temporal logic pattern as "*transitions to P-states occur at most* 2 *times between Q and R*," and then be automatically translated to the form above. The translator in [58] offers a similar facility, though focused on more specific property classes.

These tools reduce the *writing cost* but don't necessary reduce the *running cost* or the *rewriting cost*. The *rewriting cost* is incurred only when the *running cost* is too high and the models or constraints must be abbreviated. There is no guarantee that model checking is tractable over the constraints and models built quickly using

[6] Actually, to several logic variants, including Linear Temporal Logic (LTL), Computation Tree Logic (CTL), Graphical Interval Logic (GIL), Quantified Regular Expressions (QRE) and INCA Queries.

temporal logic patterns and tools like SCR. Restricted modeling languages *may* generate models simple enough to be explored with model checking-like approaches, but the restrictions on the language can be excessive. For example, checking temporal properties within simple influence diagrams can take merely linear time [57], but such a language can't model common constructs such sequences of actions or recursion. Hence, analysts may be forced back to using more general model checking languages.

Despite decades of work and ever-increasing computing power, the high *running cost* of such general model checking remains a major challenge. Different techniques have proven to be useful in reducing the running cost of model checking:

- *Symbolic Methods* represent and process parts of the state symbolically to avoid enumerating each individual value that they can take. This can be done at the level of individual model variables [61] or over the model as a whole, using boolean encodings such as binary decision diagrams [47] or propositional satisfiability solvers [62].
- *Abstraction* can be applied in a principled way to map large or infinite concrete spaces into small abstract domains. *Data abstraction* applies to individual variables (e.g., abstracting an integer to its sign) [63], while *predicate abstraction* uses the values of predicates obtained from the model [64]. In either case, the abstraction is usually not exact and produces spurious traces (i.e., for which no concrete trace exists in the original model).
- *Partial Order Reduction* analyzes dependency between concurrent operations to avoid exploring multiple equivalent permutations of independent operations. This is further discussed in Section 5.3.
- *Symmetry Reduction* uses symmetry in the system (e.g., identical agents) to avoid exploring multiple symmetrical states or paths [65,66].
- *Compositional Reasoning* divides the systems model into separate components, which can be reasoned about separately [67–69]. This generally involves some form of assume/guarantee reasoning, to capture the interdependency between the different components.
- *Model Reduction* replaces the systems model by a reduced, simpler model that is equivalent with respect to the property being verified. For example, the BANDERA system [52] can automatically extract (slice) the minimum portions of a JAVA program's byte codes which are relevant to particular properties models.

While the above techniques have all been useful in their test domains, not all of them are universally applicable. Certain optimizations require expensive pre-processing, or rely on certain features of the system being studied. Not all systems exhibit reducible symmetry or concurrency that is amenable to partial order reduction. Compositional analysis is hard for tightly connected models. In the general case, model checking techniques are still limited to models of modest size, and obtaining these models from real applications requires significant work and expertise.

Despite these limitations, model checking is a widely used tool for AI systems at institutions like NASA. For example, in 1997, a team at NASA Ames used the Spin model checker to verify parts of the RAX Executive and found five concurrency bugs [70]. Although it took less than a week to carry out the verification activities, it took about 1.5 work-months to manually construct a model that could be run by Spin in a reasonable length of time, starting from the Lisp code of the executive. The errors found were acknowledged and fixed by the developers of the executive. As it turns out, the deadlock that occurred during the in-flight experiment in 1999 was caused by an improper synchronization in another part of the Executive, but was exactly of the same nature as one of the five bugs detected with Spin two years before. This demonstrates how this kind of concurrency bug can indeed pass through heavy test screens and compromise a mission, but can be found using more advanced techniques such as model checking. In the same vein, compositional verification has been applied to the K9 Rover executive [71]. Further examples can be found in the Proceedings of the recent AAAI 2001 Spring Symposium on Model-Based Validation of Intelligence.[7]

2.5 Theorem Proving

Theorem provers build a computer-supported proof of the requirement by logical induction over the structure of the program. In principle, theorem provers can use the full power of mathematical logic to analyze and prove properties of any design in its full generality. For example, the PVS system [72] has been applied to many NASA applications (e.g., [73]). However, these provers require a lot of efforts and skills from their users to drive the proof, making them suitable for analysis of small-scale designs by verification experts only. In contrast, the other methods discussed in the previous sections are largely automated, and thus more convenient for verification as part of a software development process. For this reason, theorem proving is still mostly limited to a few academic studies and regarded as inapplicable in an industrial setting.

This view may change, however, as proof systems feature increasingly powerful proof strategies, that can automatically reduce most of the simpler proof obligations. Also note that recent developments in formal verification are blurring this distinction between proof-based and state-based approaches: on one side, proof systems are

[7] http://ase.arc.nasa.gov/mvi/.

extended with model checkers that can be used as decision procedures inside larger proofs [74]; on the other side, novel symbolic model checking approaches use proof-based solvers to prune out impossible paths in the symbolic state space [75,61].

3. Model-Based AI Systems

Having discussed how V&V AI can leverage techniques developed for software engineering problems, we now turn to the special features of AI that change the V&V task. This section discusses *model-based*. Subsequent sections will discuss *knowledge-level* systems, nondeterminism, adaptation, and their implications for V&V.

Every V&V analyst knows that reading and understanding code is much harder than reading and understanding high-level descriptions of a system. For example, before reading the “C” code, an analyst might first study some high-level design documents. The problem with conventional software is that there is no guarantee that the high-level description actually corresponds to the low-level details of the system. For example, *after* the high-level block diagram is designed, a programmer might add a call between blocks and forget to update the high-level block diagram.

This disconnect between the specification and the implementation raises significant issues for formal verification. Often, formal verification techniques based on model checking (as opposed to those based on proof systems[8]) are able to efficiently check all possible execution traces of a software system in a fully automatic way. However, the system typically has to be manually converted beforehand into the syntax accepted by the model checker. This is a tedious and complex process, that requires a good knowledge of the model checker, and is therefore usually carried externally by a formal methods expert, rather than by the system designer themselves.

A distinct advantage of model-based AI systems is that the high-level description *is* the system. A common technique used in AI is to define a specialized, succinct, high-level modeling language for some domain. This high-level language is then used to model the domain. If another automatic tool is used to directly execute that notation, then we can *guarantee* that the high-level model has a correspondence to the low-level execution details.[9]

These models are often *declarative* and V&V analysts can exploit such declarative knowledge for their analysis.

[8] Which can provide even more general results but require an even more involved and skilled guidance.

[9] This chapter defines the term “model” in its most common usage; i.e., the thing that is generated by analysts when they record information about their domain. Logic programming theorists prefer the term “theory.” In logic, a “model” is some *instance* of a “theory” and is generated automatically at runtime.

3.1 About Declarative Knowledge

Declarative representations can best be understood by comparing them to *procedural representations* used in standard procedural languages such as “C.” Procedural representations list the ordering of activities require to complete some task. Procedural knowledge often manifests itself in the *doing of something* and may be hard to share with others except in the specific context where the procedural knowledge was developed.

V&V analysts know how hard it can be to un-tangle procedural knowledge such as a “C” program. Declarative knowledge describes facts and relationships within a domain. Declarative knowledge can be easier to understand than procedural knowledge; it can be easier to modify, easier to communicate to others, and easier to reuse for different purposes.

While procedural knowledge is about *how*, declarative knowledge is often statements about *what* is true in a domain. For example, consider the following piece of procedural knowledge. This implementation reports that you have “X” if it finds any evidence for any of the sub-types of “X.” Note that this is knowledge about *how* to navigate a hierarchy of `disease`s, and their `symptom`s.

```
if ((record.disease(X)==found) &&
   (diseases = record.disease(X).subtypes))
      for(disease in diseases)
        for(symptom in disease.symptoms)
          for (observation in observations)
            if symptom == observation
              printf("You have %s which is a type of %s!", disease, X);

              return 1
```

Suppose we wanted to report the `disease` that we have the *most* evidence for; i.e., the `disease` that has the most `symptoms` amongst the available `observations`. In this procedural representation, this change would imply extensive modification to the code. An alternate approach would be to use declarative representations that queried the following facts:

```
subtype(bacterial,   measles).      symptom(measles, temperature).
subtype(injury,      carAccident).  symptom(measles, spots).
subtype(bacterial,   gastro).       symptom(gastro,  temperature).
symptom(gastro,      dehydration).
symptom(carAccident, wounds).
```

Declarative representations free the analyst from specifying tedious procedural details. For example, the core logic of the above procedural code is that we have evidence for a `disease` is we have any `observation`s consistent with `subtyes` of that `disease`. This can be expressed directly and declaratively as follows:

```
% comments start with a percent sign
evidence(Disease,SubType,Evidence):-  % we have evidence if..
   subtype(Disease,SubType),       % we can find a subtype AND
   symptom(SubType,Evidence),      % AND that subtype has symptoms
   observation(Evidence).          % AND we can observe those symptoms
```

This declarative representation is useless without some interpreter that can exercise it. Our example here uses the syntax of the Prolog logic programming language [76]. In that language, upper case words are variables and lower case words are constants.

The major disadvantage of procedural knowledge is that it can be opaque and inflexible. Declarative knowledge, on the other hand, is far more flexible since the knowledge of *what* is separated from the *how*, This means that the *what* can be used in many ways.

For example, suppose we want to drive the diagnosis *backwards* and find what might cause spots. To do this, we first must fool Prolog into believing that all observations are possible. Since we are using declarative representations, this is simple to do: we just make an “anything goes” assertion:

```
observation(_).
```

Here, the “_” is an *anonymous variable* that matches anything at all. In the language of Prolog, this means that we will assume any observation at all. With this “anything goes” in place, we can now drive the `evidence` rule backwards to find that `spots` can be explained via a `bacterial` infection.

```
?- evidence(Disease,SubType,spots).

Disease = bacterial
SubType = measles
```

A more complicated query might be to find evidence that *disproves* some current hypothesis. For example, suppose we believe the last query; i.e., the observed `spots` can be explained via a `bacterial` infection. Before we commence treatment, however, it might be wise to first check for evidence of other diseases that share some of the `evidence` for `measles`. Since our knowledge is declarative, we need not change any of the above `evidence` rule. Instead, we just reuse it in a special way:

```
differentialDiagnosis(Disease,Old,Since,New,If) :-
   evidence(Disease,Old,Since),
   evidence(Disease,New,Since),  % Old and New share some evidence
   evidence(Disease,New,If),
   not evidence(Disease,Old,If). % New has some evidence not seen
                                 % in Old
```

With this in place, we can run the following query to learn that `measles` can be distinguished from `gastro` if `dehydration` can be detected:

```
?- differentialDiagnosis(bacterial,Old,Since,New,If).

Old   = measles
Since = temperature
New   = gastro
If    = dehydration
```

While the declarative version of knowledge might be faster to write and change, it may be slower to run. Procedural knowledge can be highly optimized to take advantage of low-level details of a system. In the 1970s, this was taken to be a major disadvantage of declarative knowledge. In the 21st century, this is much less of an issue. Not only are computers much faster now, but so are the interpreters for our declarative systems (e.g., [77,78]), particularly when the representation can be restricted to some subset of full logic (e.g., [79,80]).

3.2 Declarative Models and V&V

The ability to build simple queries for a declarative model greatly reduces the effort required for V&V. For example, one method for V&V of model-based systems is to build a profile of an *average model*. The TEIREISIAS [81] rule editor applied a clustering analysis to its models to determine what parameters were *related*; i.e., are often mentioned together. If proposed rules referred to a parameter, but not its related parameters, then TEIREISIAS would point out a possible error.

Declarative modeling tend to only use a small number of modeling constructs. This simplifies the construction of translators from one modeling language to another, and in particular from AI modeling languages to verification modeling languages such as those used by model checkers (as discussed in Section 2.4). Moreover, the reasoning algorithms used in AI typically suffer from similarly high complexity metrics as model checking, and therefore the size and complexity of AI models is already limited by the scalability of their intended usage. Indeed, AI fields such as planning and scheduling have much in common with model checking, in the way they both explore a state space described by a model. This similarity has lead to some cross-fertilization between the two fields in the recent years, with verification adopting search heuristics from planning on one hand [82] and planners based on verification technology such as BDDs on the other hand [83].

For example, Pecheur and Simmons have developed a translator to convert Livingstone models to SMV [58]. Livingstone is a model-based health monitoring system developed at NASA [84]. It tracks the commands issued to the device and monitors device sensors to detect and diagnose failures. To achieve this, Livingstone relies on a model of the device that describes, for each component, the normal and abnormal functioning modes, how these modes are affected by commands and how they affect sensors. The translator enables exhaustive analysis of those models using the powerful SMV model checker. The essence of the translation is fairly straightforward, thanks to the similar semantics framework used in both Livingstone and SMV. The translator also supports user-oriented specification patterns and variables to express common classes of properties such as consistency or existence of a broken component. These declarations are captured and converted into the core temporal logic syntax accepted by SMV. Finally, the execution traces returned by SMV can be converted back into Livingstone syntax. Together, these three translation capabilities (model, properties, trace) isolate the Livingstone application developer from the peculiarities of SMV and provide the functional equivalent of a symbolic model checker for Livingstone, as depicted if Figure 6.

The translator has been used at NASA Kennedy Space Center by the developers of a Livingstone model for the In-Situ Propellant Production (ISPP), a system that will produce spacecraft propellant using the atmosphere of Mars [85]. The latest version of the ISPP model, with 10^{50} states, could still be processed in less than a minute using SMV optimizations (re-ordering of variables). The Livingstone model of ISPP features a huge state space but little depth (all states can be reached within at most three transitions), for which the symbolic processing of SMV is very appropriate.

This tool can be used to check general sanity properties (e.g., consistency, absence of ambiguity, no conflicting transitions) or specific expected properties of the modeled device (e.g., flow conservation, functional dependency between variables). More recently, the technique has been extended to verify *diagnosability properties*, i.e., the possibility for an ideal diagnosis system to infer accurate and sufficient information on the state of the device from its observed behavior [86]. Diagnosability amounts to the absence of certain pair of traces with identical observations, which

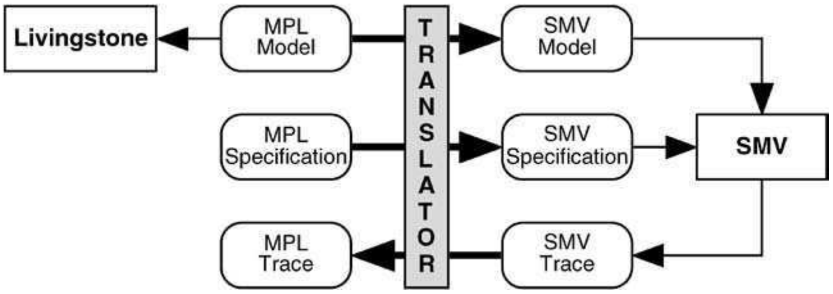


FIG. 6. Translator from Livingstone to SMV (MPL is Livingstone's modeling language).

can be turned into a simple model checking problem over a duplicated version of the original model.

Model checking is powerful, but can be complicated. Feather and Smith report that a much simpler model-based technique can still be very insightful [87]. When asked to check the planner module of NASA's Remote Agent Experiment (RAX; see Figure 1 and [88]), they developed the architecture of Figure 7. RAX's planner automatically generated plans that responded to environmental conditions while maintaining the constraints and type rules specified by human analysts. An important feature of the planner was the declarative nature of the constraints being fed into the planner *and* the plans being generated. Feather and Smith found these plans could be easily and automatically converted into the rows of a database. Further, the constraints could also be easily and automatically converted to queries over the database. As a result, given the same input as the planner, they could build a simple test oracle that could check if the planner was building faulty plans. Note that, formally, the Feather and Smith method can be considered as an example of (after-the-fact) run-time monitoring (which was discussed above in Section 2.2).

The Feather and Smith method can be very cost-effective and applied quite widely:

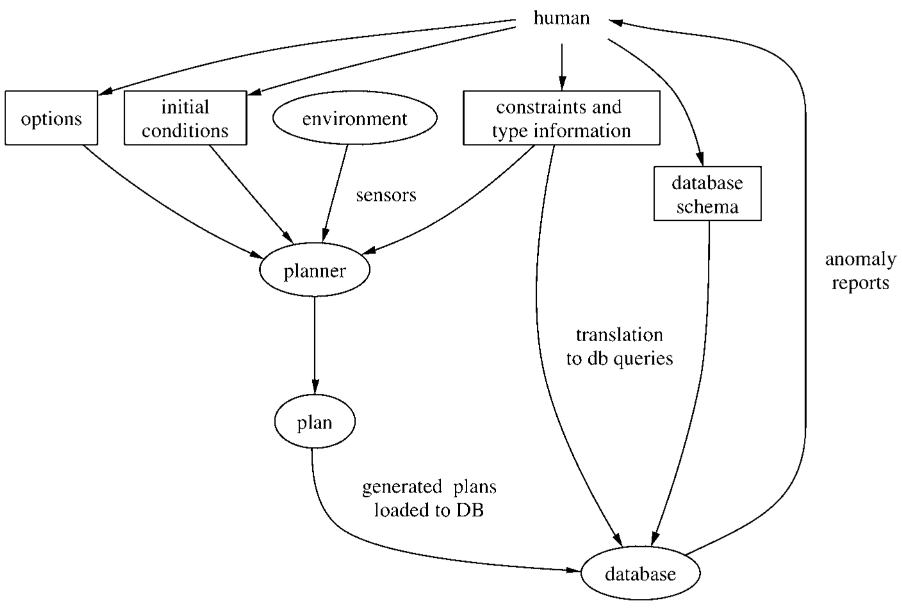


FIG. 7. A framework for model-based V&V.

- The rectangles in Figure 7 denoting the sections that must be built manually. Once these sections are built, they can be reused to check any number of plans.
- The architecture of Figure 7 could be generalized to any device that accepts declarative constraints as inputs and generates declarative structures as output.

Preece reports other simple but effective V&V tools that utilize the model-based nature of AI systems [4]. Preece studied rule-based models which are lists of statements of the following form:

$$\text{if} \quad \overbrace{L_a \wedge L_b \wedge L_c \wedge \cdots}^{\text{premise}} \quad \text{then} \quad \overbrace{L_x \wedge L_y \wedge L_z \wedge \cdots}^{\text{conclusion}}.$$

The Preece analysis defined a taxonomy of verification issues for rule-based models (see Figure 8) and argued that a variety of AI model-based verification tools target different subsets of these issues (perhaps using different terminology).

The Preece taxonomy require meta-knowledge about the terms found within a knowledge base (which Preece et al. call *literals*):

- A literal L_i is *askable* if it represents a datum that the rule-base can request from the outside world.
- A literal L_i is a *final hypothesis* if it is declared to be so by the rule-base's author and only appears in a rule conclusion.
- A rule is *redundant* if the same final hypotheses are reachable if that rule was removed. An *unusable* redundant rule has some impossible premise. A rule-base is *deficient* if a consistent subset of *askables* leads to no final hypotheses. A *duplicate redundant rule* has a premise that is a subset of another rule premise.
- Preece defined *duplicate rules* for the propositional case and *subsumed redundant rules* for the first-order case. In the first-order case, instantiations have to be made to rule premise variables prior to testing for subsets.

$$\text{Anomaly} \leftarrow \left\{ \begin{array}{l} \text{Redundant} \leftarrow \left\{ \begin{array}{l} \text{Unusable} \\ \text{Redundant} \leftarrow \left\{ \begin{array}{l} \text{Duplicate} \\ \text{Subsumed} \end{array} \right. \end{array} \right. \\ \text{Ambivalence} \leftarrow \text{Conflict} \\ \text{Circularity} \\ \text{Deficiency} \leftarrow \left\{ \begin{array}{l} \text{Missing rules} \\ \text{Missing values} \end{array} \right. \end{array} \right.$$

FIG. 8. The Preece hierarchy of verification errors.

TABLE I
RATIO OF ERRORS/ANOMALIES SEEN IN REAL-WORLD EXPERT SYSTEMS

	Application					
	mmu	tapes	neuron	displan	dms1	errors/anomalies
subsumed	0	5/5	0	4/9	5/59	14/73 = 19%
missing rules	0	16/16	0	17/59	0	33/75 = 44%
circularity	0	0	0	20/24	0	20/24 = 83%

From [7]. “Subsumed” reports repeated rule conditions. “Missing rules” denote combination of attribute ranges not seen in any rule. “Circularity” reports loops found in the dependency graph between variables in the system.

- Preece defined *ambivalence* as the case where, given different consistent subset of askables, a rule-base can infer the same final hypotheses.

Preece stresses that the entries in their taxonomy of rule-base anomalies may not be true errors. For example, the dependency network from a rule-base may show a circularity anomaly between literals. However, this may not be a true error. Such *circularities* occur in (e.g.) user input routines that only terminate after the user has supplied valid input.

More generally, Preece argued convincingly that automatic verification tools can never find “errors.” Instead, they can only find “anomalies” which must be checked manually. The percentage of true errors, i.e., errors/anomalies can be quite small. For example, Table I shows the errors/anomalies ratios seen in five KBSs. Note that not all anomalies are errors.

4. The Knowledge Level

After decades of model-based programming, certain common model-based tasks were identified. That is, model-based programming became viewed like assembler code on top of which a range of *knowledge-level problem solving methods* (PSMs) are implemented.

A standard hierarchy of PSMs is shown in Figure 9. In this view, RAX’s MIR (diagnosis) component (described in Figure 1) is one kind of diagnosis PSM. Another diagnosis PSM is shown in Figure 10. In that figure ovals denote functions and rectangles denote data structures. Formally, that figure represents a mapping from data d to an hypothesis h via intermediaries Z and other data R_i:

$$abstract\big(data(d), R_1, obs(Z)\big) \wedge hypothesis\big(obs(Z), R_2, hyp(h)\big)$$

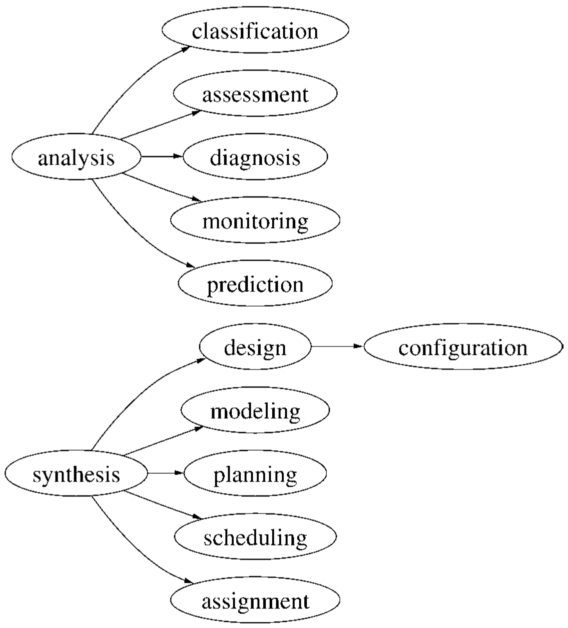


FIG. 9. A hierarchy of problem solving methods.

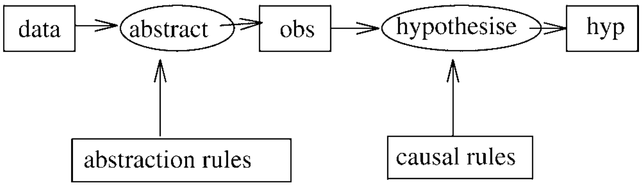


FIG. 10. Explicit problem solving (PSM) meta-knowledge: A simple KADS-style PSM for diagnosis. `Abstract` and `hypothesis` are primitive inferences which may appear in other PSMs. From [89].

Various design and code libraries were built around these knowledge-level PSMs, e.g., cognitive patterns [90]; CommonKADS [91–93]; configurable role-limiting methods [94,95]; MIKE [96]; the Method-To-Task approach [97]; generic tasks [98]; SPARK/BURN/FIREFIGHTER [99]; model construction operators [100]; components of expertise [101]; and the systems described in [102,103,98,104,105,93,90]. The advantage of these PSMs is that they offer an organizational layer on top of model-based methods. V&V analysts can use this layer as an indexing method for their evaluation techniques.

For example, van Harmelen and Aben [89] discuss formal methods for repairing the diagnosis PSMs of Figure 10. V&V analysts can restrict their analysis of this model to the three ways this process can fail:

(1) It can fail to prove $\mathit{abstract}(\mathit{data}(d), R_1, \mathit{obs}(Z))$; i.e., it is missing abstraction rules that map d to observations.
(2) It can fail to prove $\mathit{hypothesize}(\mathit{obs}(Z'), R_2, \mathit{hyp}(h))$; i.e., it is missing causal rules that map Z' to an hypothesis h.
(3) It can prove either subgoal of the above process, but not the entire conjunction; i.e., there is no overlap in the vocabulary of Z and Z' such that $Z = Z'$.

Case #1 and #2 can be fixed by adding rules of the missing type. Case #3 can be fixed by adding rules which contain the overlap of the vocabulary of the possible Z values and the possible Z' values. More generally, given a conjunction of more than one sub-goal representing a PSM, fixes can be proposed for any sub-goal or any variable that is used by > 1 sub-goal.

Another knowledge-level V&V technique is to audit how the PSMs are built. Knowledge not required for the PSM of the application is superfluous and can be rejected. In fact, numerous AI editors are *PSM-aware* and auto-configure their input screens from the PSM such that only PSM-relevant knowledge can be entered by the user. For example:

- RIME's rule editor [106,25] acquired parts of the KB minority-type meta-knowledge for the XCON computer configuration system [28]. RIME assumed that the KB comprised operator selection knowledge which controlled the exploration of a set of problem spaces. After asking a few questions, RIME could auto-generate complex executable rules.
- SALT's rule editor interface only collected information relating directly to its propose-and-revise inference strategy. Most of the SALT rules ($2130/3062 \approx 70\%$) were auto-generated by SALT.
- Users of the SPARK/BURN/FIREFIGHTER (SBF) [99] can enter their knowledge of computer hardware configuration via a click-and-point editor of business process graphs. SBF reflects over this entered knowledge, then reflects over its library of PSMs. When more than one PSM can be selected by the entered knowledge, SBF automatically generates and asks a question that most differentiates competing PSMs.

PSM-aware editors can not only assist in entering knowledge, but also in testing and automatically fixing the entered data. For example, in the case where numerous changes have to be made to a PSM, if the user does not complete all those changes, then the PSM may be broken. Gil and Tallis [107] use a scripting language to control the modification of a multi-PSM to prevent broken knowledge. These *KA scripts*

TABLE II
CHANGE TIMES FOR ETM WITH FOUR SUBJECTS: S1...S4. FROM [107]

	Simple task #1				Harder task #2			
	No ETM		With ETM		No ETM		With ETM	
	S4	S1	S2	S3	S2	S3	S1	S2
Time completing transactions	16	11	9	9	53	32	17	20
Total changes	3	3	3	3	7	8	10	9
Changes made automatically	n/a	n/a	2	2	n/a	n/a	7	8

are controlled by the EXPECT TRANSACTION MANAGER (ETM) which is triggered when EXPECT's partial evaluation strategy detects a fault. Table II shows some speed up in maintenance times for two change tasks for EXPECT KBS, with and without ETM. Note that ETM performed some automatic changes (last row of Table II).

5. AI Software Can Be Nondeterministic

The main challenge in verifying AI software (or, for that matter, any kind of complex system) comes from the number of different possible executions that have to be taken into account. We refer broadly to this uncertainty on a system's future behavior as *nondeterminism*. Nondeterministic choices come from incoming external events, scheduling of concurrent tasks, or even intentional random choices in the program, to name a few. Every choice point in the execution results in as many possible subsequent execution branches. Typically, those choices compound into exponentially many possible executions. This is known as the *state space explosion* phenomenon.

Nondeterminism can be either *external* or *internal*:

- *External nondeterminism* results from input or events coming from the *environment*. Examples include system configuration and initialization, invocation parameters, messages, discrete events, continuous data streams. In the case of model-based systems, the models themselves constitute a huge choice space, as far as the interpreter is concerned.
- *Internal nondeterminism* result from the system itself. A common source is *concurrency*, where scheduling choices are made between concurrent executions (for example, a knowledge system that processes knowledge updates concurrently). Another case is *stochastic*, where the system itself is deliberately making random (or pseudo-random) choices.

The important distinction between these two is that while external nondeterminism is *controllable* (and therefore can, in principle, be tested), internal nondeterminism is not, which poses an additional problem to the verifier: the same test case may produce different results when run several times, and it is hard to measure or control the coverage of internal nondeterministic choices.

Conventional sequential programs are usually (internally) deterministic, in that they contain hard-wired decision paths that result in a functional mapping from inputs to outputs. The same is true of many applicative AI algorithms taken in isolation. However, as these algorithms get implemented and assembled together to build complex intelligent software, additional nondeterminism is introduced in the form of concurrent components executing asynchronously. This is particularly the case for reactive systems such as robotic controllers, that have to react in a timely manner to external stimuli occurring at unpredictable times. These systems can resort to *anytime* algorithms, whose output depends on how long they are allowed to run. In typical concrete cases, nondeterminism is thus an issue that V&V analysts of AI systems have to face.

The rest of this section discusses V&V and these different types of nondeterminism.

5.1 Environmental Nondeterminism

The standard method of managing environmental nondeterminism is to build a *operational profile* modeling the probability that a certain variable setting will appear at runtime [108]. The range of possible inputs can now be sampled via the operational profile. The operational profile can be used to generate representative nominal inputs. Also, by inverting the profile, unlikely off-nominal test cases can also be generated.

In the case of AI agents performing tasks in dynamic environments for deep space missions, building an accurate operational profile is a very difficult task. In turns out that operational profile errors can be compensated by increasing the number of tests. Figure 11 shows Pasquini's study where the original operational profile *OPe* was compared to three profiles containing an increasing number of errors. The mutants were called (in order of increasing errors) *OP*1, *OP*2, *OP*3. The inaccuracies in the operational profiles were very apparent after a small number of tests. However, above 100 tests randomly selected from each profile, the errors of the different profiles converged and after a 1000 tests, the effects of those errors were negligible.

The system studied by Pasquini was not a complex AI system. It is reasonable to assume that more complex systems would require exponentially more tests to compensate for errors in the operational profile. In many real-world situations, it is not practical to run a very large number of tests. Table III shows the pre-launch test regime for the remote agent. The testing team had to share certain test rigs with

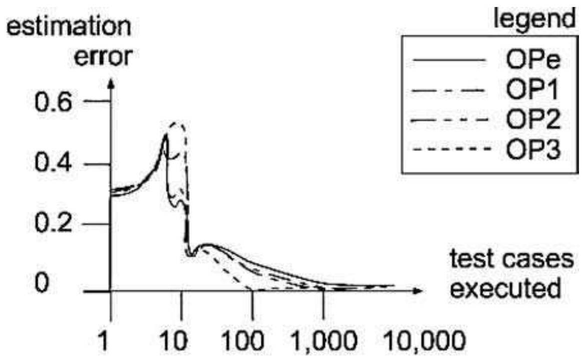


FIG. 11. Sensitivity of reliability-growth models to operational profile errors vs testing accuracy. Hand-translated from [109].

numerous other development teams. As those test rigs got more elaborate (e.g., increasing fidelity to the actual in-flight software/hardware combination), they were slower to run and all the development teams, including the testers, got less and less access. Table III shows that Remote Agent launched after 610 tests. Therefore, mere operational profile sampling may be inadequate for AI systems.

Various researchers have explored intelligent methods for sampling a program's input space. For example, Smith et al. [111] characterized the inputs to the RAX planner as an n-dimensional parameter space and used orthogonal arrays to select a manageable number of cases that exercises all pair-wise combinations of parameter values. In this rig, an intelligent sample of RAX's input space required a number of tests that is logarithmic on the number of RAX's input parameters.

TABLE III
NUMBER OF DOCUMENTED PRE-LAUNCH TESTS FOR THAT RAX NASA SATELLITE AS IT MOVED FROM SOFTWARE SIMULATIONS (IN PHASE 1) TO SOME HARDWARE TEST BENCHES (PHASE 5 AND 6), THE FINALLY TO THE ACTUAL MISSION (PHASE 7). FROM [110]

	Test platform		Speed	
Phase	Hardware	Flight software	Test:real	# tests
1	nil	some	35:1	269
2	nil	some	7:1	≈ 300
3	nil	some	1:1	≈ 20
4	nil	all	1:1	10
5	flight spare	all	1:1	10
6	flight spare	all	1:1	1
7	actual satellite	all	1:1	1
			total:	≈ 611

5.2 Stochastic Nondeterminism

Often, AI algorithms use some internal random choice to break out of dead-ends or to explore their models. Such stochastic methods can be remarkably effective. For example, stochastic choice has been observed to generate plans in AI systems one to two orders of magnitude bigger than ever done before with complete search [79, 112]. Yet another source of stochastic nondeterminism is *adaptation* when a system's behavior changes as a result of new experience. Adaptation is discussed later (see Section 6).

A concern with stochastic nondeterminism is that the variance in the system's output will be so wild that little can be predicted or guaranteed about the system's performance at runtime. Before a V&V analyst can certify such a wildly varying system, they might first need to *constrain* it to the point where definite predictions can be made about the system's behavior. A typical way to find these controlling constraints is to instrument the system so that internal choices become visible and controllable, effectively turning them into external choices.

Some empirical results suggest that if instrumentation is successful, then the subsequent learning of controllers may be quite simple. The *funnel* assumption is that within a nondeterministic program there exists a small number of assumptions that control which nondeterministic option the program will take [113]. This assumption appears in multiple domains, albeit using different terminology: *master-variables* in scheduling [114]; *prime-implicants* in model-based diagnosis [115] or machine learning [116], or fault-tree analysis [117]; *backbones* or back doors in satisfiability [118–120]; *dominance filtering* used in Pareto optimization of designs [121]; and the *minimal environments* in the ATMS [122]. Whatever the name, the intuition is the same: whatever happens in the larger space of the program is controlled by a few variables.

There is much empirical and analytical evidence that narrow funnels are found in many models. Menzies and Singh argue that the whole field of soft computing utilizes that assumption [113]. Later in this chapter, in Section 6.5, we will discuss the TAR3 learner. TAR3 was built as a test of the narrow funnel assumption. If such funnels exist, they would appear as variable bindings with a much higher frequency in preferred situations. TAR3 collects such frequency counts, normalizes and accumulates them. When applied to models with stochastic nondeterminism, TAR3 can find a very small set of treatments that constrain the nondeterminism, while selecting for better overall system behavior (see the examples in [123,124]).

The caveat here is that the nondeterministic system must be exercised sufficiently to give TAR3 enough data to learn the funnels. For more on that issue, see Section 6.2.

5.3 Concurrency Nondeterminism

At the conceptual level, AI algorithms are typically sequential (though possibly nondeterministic). In real applications, though, concurrency is often present. For example:

- A knowledge base receives and processes queries and updates concurrently.
- An intelligent robot controller responds to physical stimuli occurring at unpredictable times.
- A chess playing program is distributed over many processors for improving performance.
- Intelligent web services communicate with each other to negotiate some business contract.

Concurrency errors have been known to cause trouble in AI software. As we mentioned in the introduction, the error that caused a deadlock during the RAX experiment on-board Deep Space One was a typical concurrency error.

Concurrency is a particular form of nondeterminism, and thus a major source of concern for V&V. Concurrency-related flaws typically result in *race conditions*, which manifest themselves only under specific timings of events and are very hard to detect and reproduce. This leads to critical systems being designed to minimize concurrency risks, using strictly sequenced chains or cycles of execution, or with tightly isolated concurrent components.

Model checking is well suited to verifying concurrent systems, and has been originally developed in that domain. As opposed to a conventional test-bed, the model checker has full control over the scheduling of concurrent executions and can therefore explore all nondeterministic alternatives.

Concurrency-related state-space explosion can be addressed using a particular class of optimization called *partial-order reduction* (*POR*). This is based on the "diamond" principle ($\begin{tikzcd}[sep=tiny] & {} \arrow[dl] \arrow[dr] & \\ {} \arrow[dr] & & {} \arrow[dl] \\ & {} & \end{tikzcd}$): if two concurrent operations are independent, then it does not matter in which order they are executed. This can be used to dramatically cut down the search space, provided that (sufficient conditions for) independence between operations can be asserted [125,126]. Implementations exploiting this technique can constrain how the space is traversed [127], or constructed in the first place [128].

6. Adaptive AI Systems

In the language of the previous section, adaptation is another source of nondeterminism. Depending on the learning method, adaptation can either be:

- *Stochastic nondeterminism* when some random choice is used in the learning, e.g., in genetic algorithms;
- *Environmental nondeterminism* when the learning changes according to the data passed into the system from the outside.

Regardless of the source of nondeterminism, the net result is that adaptive systems can adjust their own internal logic at runtime. Adaptive systems have the benefit that the software can fix itself. For example, a planning system might find a new method to generate better plans in less time. The problem with adaptive systems is that the adaption might render obsolete any pre-adaption certification.

There are many different adaptive systems such as decision explanation-based generalization [129,130], chunking [129,130], genetic algorithms [131], simulated annealing [132], tree learners [133], just to name a few. An example of decision tree adaptation is shown in Figure 12. In that figure, the decision tree on the right was generated from the data on the left. We see that we do not play golf on high-wind days when it might rain.

Despite there being many different learning methods, there exist several adaptive V&V criteria that can be applied to all learners: external validity, learning rates, data anomaly detectors, stability, and readability. These are discussed below.

6.1 External Validity

While the *method* of adaptation can vary the *goal* of the different methods is similar. Adaptation builds or tune some theory according to a set of new examples. Therefore, to validate any adaptive system, it is useful to start with validating that enough examples are available to support adequate adaptation.

When checking that enough data was available for the adaptation, good experimental technique is important. If the goal of adaptation is to generate models that have some useful future validity, then the learnt theory should be tested on data not used to build it. Failing to do so can result in a excessive over-estimate of the learnt model. For example, Srinivasan and Fisher report an 0.82 correlation (R^2) between the predictions generated by their learnt decision tree and the actual software development effort seen in their training set [134]. However, when that data was applied to data from another project, that correlation fell to under 0.25. The conclusion from their work is that a learnt model that works fine in one domain may not apply to another.

One standard method for testing how widely we might apply a learnt model is *N-way cross validation*:

- The training set is divided into N buckets. Often, $N = 10$.

Data

```
#outlook, temp, humidity, windy, class
#-------  ----  --------  -----  -----
sunny,    85,   85,       false, dont_play
sunny,    80,   90,       true,  dont_play
overcast, 83,   88,       false, play
rain,     70,   96,       false, play
rain,     68,   80,       false, play
rain,     65,   70,       true,  dont_play
overcast, 64,   65,       true,  play
sunny,    72,   95,       false, dont_play
sunny,    69,   70,       false, play
rain,     75,   80,       false, play
sunny,    75,   70,       true,  play
overcast, 72,   90,       true,  play
overcast, 81,   75,       false, play
rain,     71,   96,       true,  dont_play
```

Learnt theory

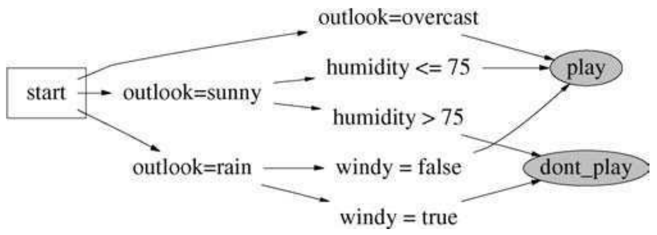


FIG. 12. Decision-tree learning. Classified examples (left) generate the decision tree (right).

- For each bucket in turn, a treatment is learned on the other $N - 1$ buckets then tested on the bucket put aside.
- The prediction for the error rate of the learnt model is the *average* of the classification accuracy seen during the N-way study.

In essence, N-way cross validation is orchestrating experiments in which the learnt model is tested ten times against data not seen during training.

When assessing *different* adaptation mechanisms, the N-ways are repeated M times. In a 10-by-10 cross-validation study, the ordering of examples in a data set is randomized 10 times and a separate 10-way study is conducted for each of the ten random orderings. Such 10-by-10 study generates 100 training and 100 test sets and each of these should be passed to the different learners being studied. This will generate a mean and standard deviation on the classification accuracy for the learners being studied and these should be compared with a t-test with 10 degrees of freedom (and not 99, see [135]).

6.2 Learning Rates

Another important criteria for any learning is that of *learning rates*; i.e., how does the learnt theory change over time as more data is processed.

One way to study learning rates is via a *sequence study*. This is a variant on a 10-way study but this time training occurs on an increasing percentage of the available data. More precisely:

- The data is divided into N buckets.
- $\frac{X}{N}$th of the data for $X \in \{1, 2, \ldots N - 1\}$ is used for training;
- The remaining $\frac{N-X}{N}$ of the data is used for testing.

Note that the sequence stops at $N - 1$ since training on $\frac{N}{N}$ of the data would leave nothing for the test suite $(1 - \frac{N}{N} = 0)$. In a *M-by-N sequence* study, the above process is repeated M times with the ordering of the example data randomized before each N-sequence study. The mean and standard deviation of the accuracy at each N value is reported. M-by-N sequence studies let us check how early learning stabilizes as more data is used in the training.

Figure 13 show results of a M-by-N sequence study for six data sets using $\langle M = N = 20\rangle$. In this study, the same decision tree learner was used as seen in Figure 12. The vertical-axis ranges from zero to 100% accuracy. The horizontal-axis shows the training set growing in size. The whiskers in Figure 13 show ± 1 standard deviation of the 20 experiments conducted at a particular N value.

Figure 13 is divided into three groups. On the left-hand column are data sets where the adaptation needs more examples than what is currently available:

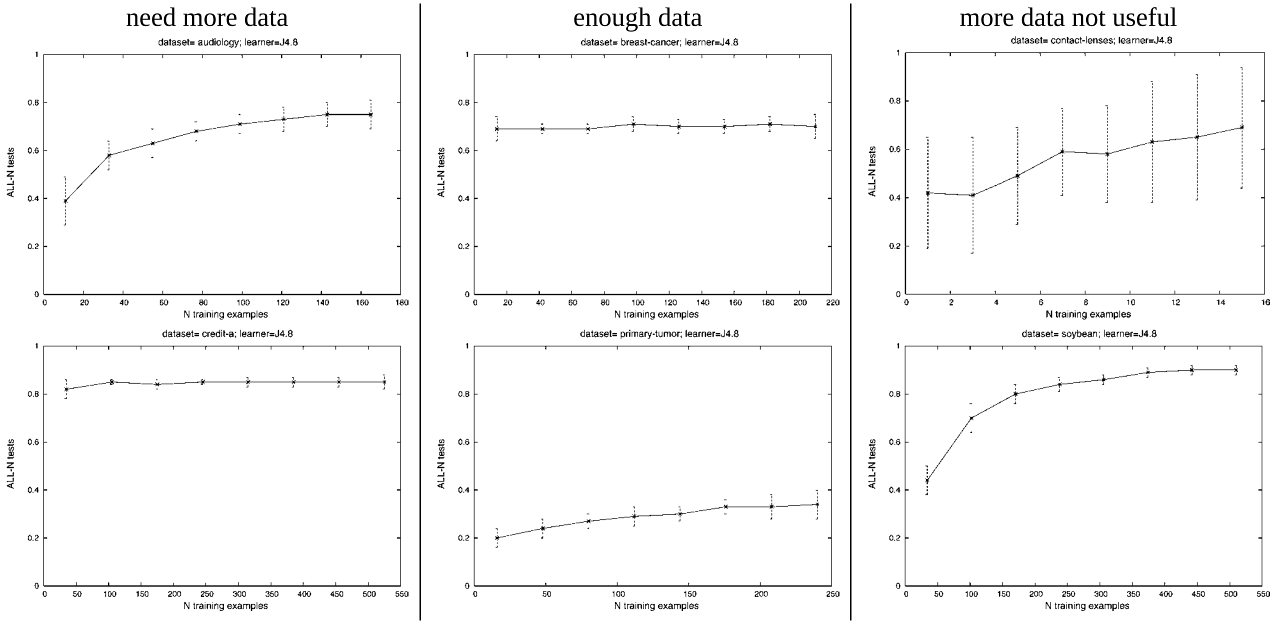


FIG. 13. *M*-by-*N* sequences studies on six data sets.

- The standard deviation in the classification accuracies of the top-left are very large. Clearly, in that data set, training is insufficient for stable future predictions.
- The mean classification of the bottom-left plot is very low (less than 40%), even though all the available training data has been passed to the learner. The accuracy improves as the training set grows but much more data would be required before a V&V analyst could check if adaptation is performing adequately in that domain.

The middle column of Figure 13 shows examples were the learning improves dramatically during the sequence, rises to medium or high level accuracies, then plateaus *before* we exhaust all the data in this domain. The conclusion from these two plots would be that we are collecting adequate amounts of data in this domain and that the benefits of further data collection might be quite low.

The right-hand column of Figure 13 shows examples where, very early in the sequence, the adaption reaches medium to high levels and does not improve as more data is supplied. For these domains, a V&V analyst would conclude that *too much* data was collected in this domain.

Note that the learning never achieves 100% accuracy. Making some errors is fundamental to the learning task. If something adapts perfectly to past example, then it can *over-fit* to the data. Such over-fitted adaption can obsess on minor details in the training data and can perform poorly on future examples if those future examples contain trivial differences to the training example.

Studying the shape of these sequence learning curves is an important V&V technique for adaptive real-time systems. Consider a real-time controller that must adapt to sudden changes to an aircraft; e.g., the flaps on the left wing have suddenly frozen. The V&V task here would be to predict the shape of these curves in the space of possible future input examples. Such predictions would inform questions such as “would the controller adapt fast enough to save the plane?”

There are many ways to explore the space of possible future input examples. One way is the sequence study shown above: the ordering of existing data is randomized many times and, each time, the learner learns for that sequence of data. Another way would be to generate artificial examples from distributions seen in current data or from known distributions in the environment. Yet another way is to define an *anomaly detector* which triggers if newly arriving data is different to data which the learner has previously managed [136,137].

6.3 Data Anomaly Detectors

Figure 14 shows a general framework for wrapping a learner in anomaly detectors. When new data arrives a *pre-filter* could reject the new input if it is too anomalous.

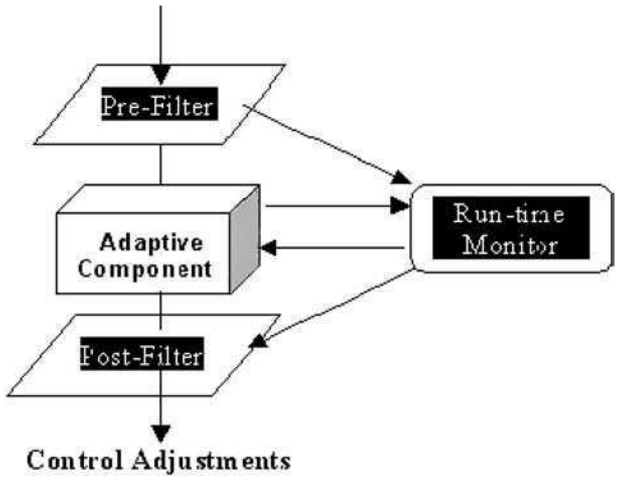


FIG. 14. Applications of anomaly detection. From [138].

Any accepted data is then passed to the adaptive module. This module offers some conclusion; e.g., adds a classification tag. The output of the adaption module therefore contains more structure than the input example so a second *post-filter* anomaly detector might be able to recognize unusual output from the learner. If so, then some repair action might be taken to (e.g.) stop the output from the learner effecting the rest of the system. In effect, Figure 14 is like an automated V&V analyst on permanent assignment, watching over the adaptive device.

Figure 15 shows an example of a pre-filter anomaly detector. That figure is a representation of high dimensional data collected from nominal and five off-nominal modes from a flight simulator passed through a *support vector machine*. *Support vector machines* recognizing the borderline examples that distinguish between different classes of data. Such machines run very quickly and scale very well to data sets with many attributes. The crosses in Figure 15 show training examples and the closed lines around the circles represent the border between “familiar” and “anomalous” data. Our learner should be able to handle failure mode 5 since data from that mode falls mostly in the “familiar” zone. However, Failure Mode 2 worries us the most since much of the data from that mode falls well outside the “familiar” zone.

Figure 15 only shows anomaly *detection*. After detection, some repair action is required. The precise nature of the repair action is domain-specific. For example, in the case of automatic flight guidance systems, the repair action might be either “pass control to the human pilot” or, in critical flight situations, “hit the eject button.”

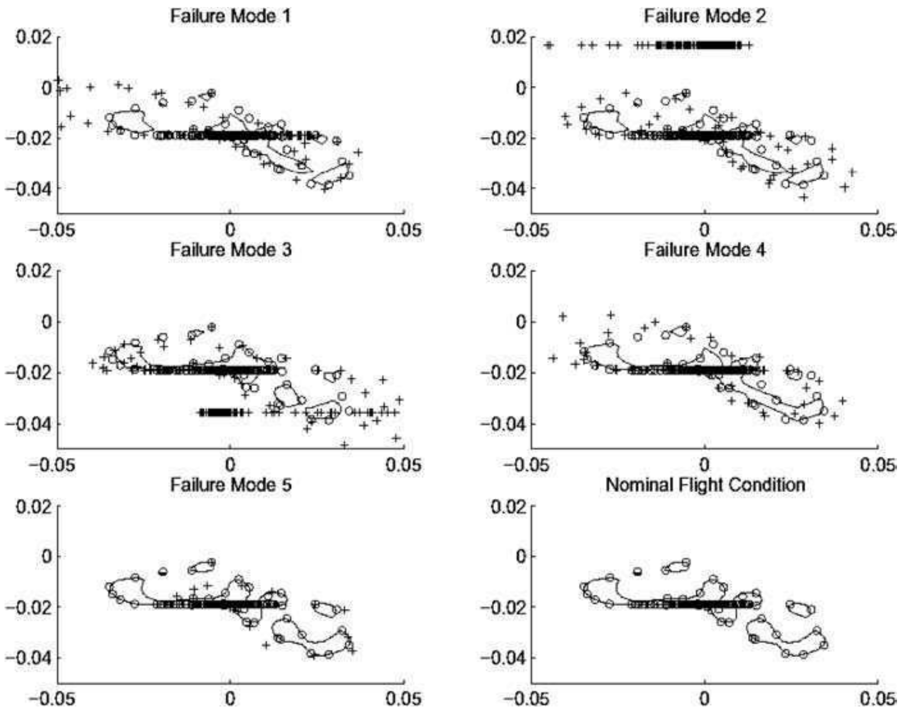


FIG. 15. Identifying anomalous data. From [138].

6.4 Stability

Apart from studying the above, a V&V analyst for an adaptive system might also care to review the *results* of the learning. When a V&V analyst is reading the output of a learner, one important property of the learning is *stability*; i.e., the output theory is the same after different runs of the learner.

Not all learners are stable. For example, decision tree learners like the one used in Figure 12 are *brittle*; i.e., minor changes to the input examples can result in very different trees being generated [139]. Also, learners that use random search can leap around within the learning process. For example, genetic programming methods randomly *mutate* a small portion of each generation of their models. Such random mutations may generate different theories on different runs. Therefore, if an output theory is to be manually inspected, it is wise to select learners that generate *stable conclusions*. For example, Burgess and Lefley report [140] that in ten runs of a neural net and genetic programming system trying to learn software cost estimation models, the former usually converged to the same conclusion while the latter could generate different answers with each run.

6.5 Readability

Lastly, another interesting V&V criteria for a learner is *readability*; i.e., assessing if the output from the learner is clear and succinct.

Not all learners generate readable output. For example, neural net and Bayes classifiers store their knowledge numerical as a set of internal weights or table values that are opaque to the human reader. Research in neural net validation often translate the neural net into some other form to enable visualization and inspection (e.g., [141]). Even learners that use symbolic, not numeric, representations can generate unreadable output. For example, Figure 16 was learnt from 506 examples of low, medium low, medium high, and high quality houses in Boston. While a computer program could apply the learnt knowledge, it is virtually unreadable by a human.

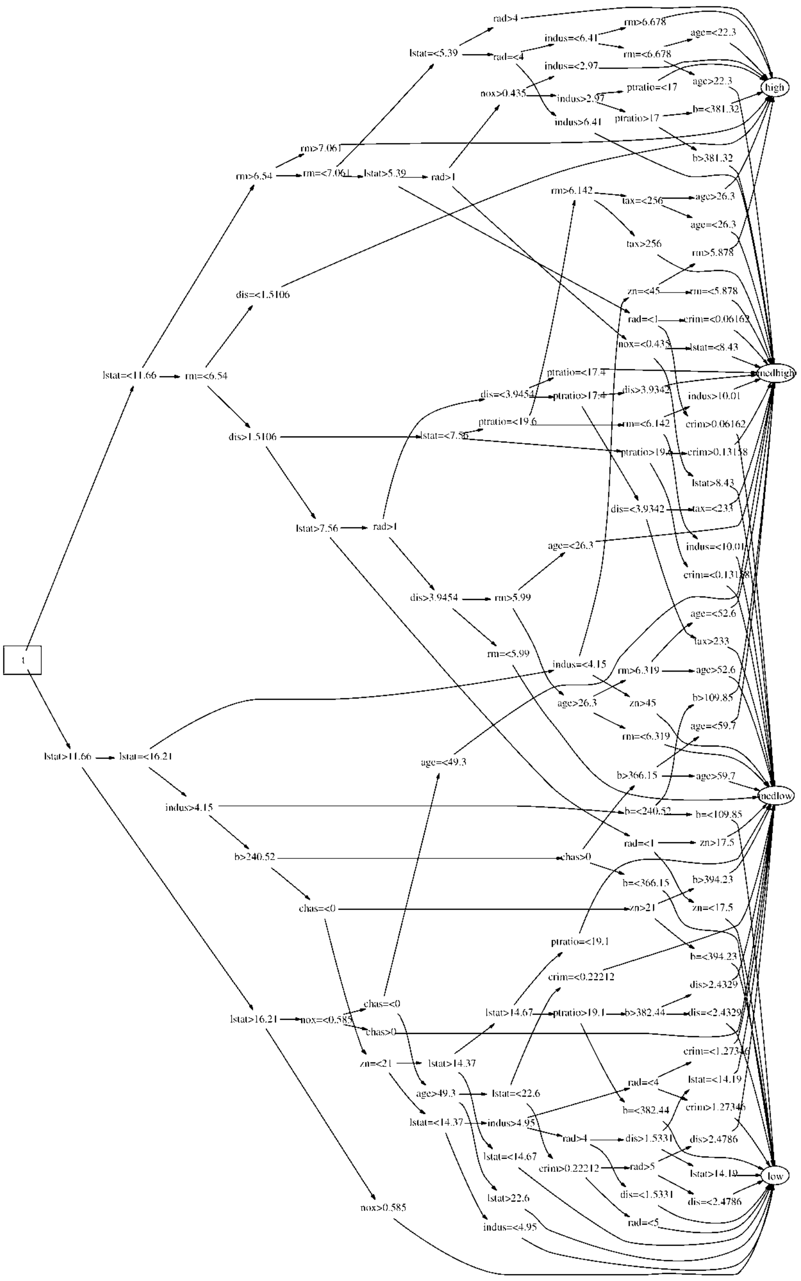
Some learners are specifically designed to generate succinct, simple, readable output. The TAR3 *treatment learner* [142–146,123,124,147] seek the *smallest* number of attribute ranges that *most* select for preferred classes and *least* select for undesired classes. Treatments are like constraints which, if applied to the test set, selects a subset of the training examples. A treatment is *best* if it *most improves* the distribution of classes seen in the selected examples. From the same data as used in Figure 16, TAR3 learns Eq. (1):

$$\mathit{best} = (6.7 \leqslant \mathit{RM} < 9.8) \wedge (12.6 \leqslant \mathit{PTRATION} < 15.9) \tag{1}$$

That is, good houses can be found by favoring houses with 7 to 9 rooms in suburbs with a parent-teacher ratio in the local schools of 12.6 to 15.9; The effects on the distribution of selected houses by Eq. (1) are shown in Figure 17. In the raw data, the quality of houses was evenly distributed. In the treated data, most of the selected houses (97%) are now high quality.

Which is a better representation of the data? The details of Figure 16 or the high-level summary of Eq. (1)? That choice is up to the reader but if they are a busy V&V analyst struggling to understand an adaptive system, they might be attracted to the succinctness of Eq. (1).

FIG. 16. A learnt decision tree. Classes (right-hand side), top-to-bottom, are "high," "med-high," "medlow," and "low" This indicates median value of owner-occupied homes in $1000's. Decision tree learnt from the 506 cases in HOUSING example set from the UC Irvine repository (http://www.ics.uci.edu/~mlearn/).


t
lstat=<11.66
lstat>11.66
rm=<6.54
rm>6.54
dis=<1.5106
dis>1.5106
rm>7.061
rm=<7.061
lstat=<5.39
lstat>5.39
rad>4
rad=<4
rad>1
indus=<6.41
indus>6.41
indus=<2.97
indus>2.97
nox>0.435
rm>6.678
rm=<6.678
age=<22.3
age>22.3
ptratio=<17
ptratio>17
b=<381.32
b>381.32
rm>6.142
tax=<256
tax>256
age>26.3
age=<26.3
rm>5.878
zn=<45
rm=<5.878
rad=<1
crim=<0.06162
nox=<0.435
lstat=<8.43
ptratio=<17.4
ptratio>17.4
dis=<3.9454
dis>3.9342
indus>10.01
ptratio=<19.6
rm=<6.142
crim>0.06162
lstat=<7.56
ptratio>19.6
crim>0.13158
lstat>8.43
lstat>7.56
rad>1
dis=<3.9342
tax=<233
age=<26.3
indus=<10.01
crim=<0.13158
dis>3.9454
rm>5.99
age=<52.6
rm=<5.99
tax>233
indus=<4.15
rm>6.319
age>52.6
age>26.3
zn>45
b>109.85
rm=<6.319
age=<59.7
lstat=<16.21
age=<49.3
b>366.15
age>59.7
indus>4.15
b=<240.52
b=<109.85
rad=<1
zn>17.5
b>240.52
chas>0
b=<366.15
b>394.23
chas=<0
zn>21
zn=<17.5
ptratio=<19.1
b=<394.23
crim=<0.22212
dis>2.4329
chas=<0
lstat>14.67
ptratio>19.1
b>382.44
dis=<2.4329
lstat>16.21
nox=<0.585
chas>0
crim=<1.27346
zn=<21
lstat>14.37
rad=<4
lstat=<14.19
age>49.3
lstat=<22.6
b=<382.44
crim>1.27346
lstat=<14.37
indus>4.95
rad>4
dis>1.5331
dis>2.4786
lstat=<14.67
crim>0.22212
rad>5
lstat>14.19
dis=<1.5331
lstat>22.6
dis=<2.4786
nox>0.585
rad=<5
indus=<4.95
high
medhigh
medlow
low

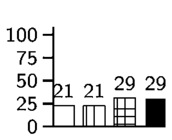
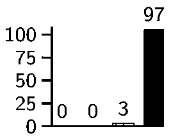
baseline	baseline + Eq. (1)
	
506 examples	38 examples

FIG. 17. Treatments learnt by TAR3 from the data used in Figure 16. That dataset had the class distribution shown left-hand side. Actions that most increase housing values are shown in the right column.

7. Conclusion

This chapter offers a diverse set of technologies to support the V&V of adaptive systems. If there is a single conclusion from such a diverse survey is that developers should not be afraid of AI. AI is useful. AI can deliver more functionality for reduced cost. AI should and will be used more widely.

On the other hand, AI systems have features that make them hard to check using conventional methods. Nevertheless, there are enough alternative readily-available methods that enable the V&V of AI software:

- AI software can be *complex*. Powerful methods like model checkers and static analysis tools has evolved in the software engineering area to simplify the task of checking such complex systems. Many of those methods can be applied to AI systems.
- The *model-based* nature of AI systems makes it easier for V&V analysts to extract features from a system and this can be exploited in several ways.
- Sometimes, the inference associated with those models falls into one of a small set of commonly-used *knowledge-level* problem solving methods and specialized V&V techniques are appropriate for different problem solving methods.
- AI systems can be *nondeterministic*. Different methods apply for the V&V of nondeterministic systems depending on the nature of the nondeterminism (environmental, concurrent, stochastic).
- Adaptive systems are an extreme for of stochastic nondeterministic systems. The V&V of adaptive systems can apply such criteria like external validity, learning rate, stability, etc.

Acknowledgements

On a previous version of this chapter, Dale Pace and Steve Stevenson [148] offered some excellent guidance and Lisa Montgomery provided invaluable editorial advice. The work was partially sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility, and by the NASA Engineering for Complex Systems Program. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not constitute nor imply its endorsement by the Authors or the United States Government.

References

[1] Menzies T., "21st century AI: proud, not smug", Editorial, IEEE Intelligent Systems, Special Issue on AI Pride. Available from http://menzies.us/pdf/03aipride.pdf.

[2] Bernard D., et al., "Spacecraft autonomy flight experience: The DS1 Remote Agent experiment", in: *Proceedings of the AIAA, 1999, Albuquerque, NM*, 1999.

[3] Suwa M., Scott A., Shortliffe E., "Completeness and consistency in rule-based expert systems", *AI Magazine* **3** (4) (1982) 16–21.

[4] Preece A., "Principles and practice in verifying rule-based systems", *The Knowledge Engineering Review* **7** (1992) 115–141.

[5] Rousset M., "On the consistency of knowledge bases: the COVADIS system", in: *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI'88)*, 1988, pp. 79–84.

[6] Preece A., Shinghal R., Batarekh A., "Verifying expert systems: A logical framework and a practical tool", *Expert Systems with Applications* **5** (2) (1992) 421–436.

[7] Preece A., Shinghal R., "Verifying knowledge bases by anomaly detection: An experience report", in: *ECAI '92*, 1992.

[8] Prakash G., Subramanian E., Mahabala H., "A methodology for systematic verification of ops5-based AI applications", in: *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI'91)*, 1991, pp. 3–8.

[9] Nguyen T., Perkins W., Laffey T., Pecora D., "Knowledge base verification", *AI Magazine* **8** (2) (1987) 69–75.

[10] Meseguer P., "Incremental verification of rule-based expert systems", in: *Proceedings of the 10th European Conference on Artificial Intelligence, ECAI-92*, 1992, pp. 840–844.

[11] Meseguer P., "Verification of multi-level rule-based expert systems", in: *Proceedings of the 9th National Conference on Artificial Intelligence*, 1991, pp. 323–328.

[12] Ginsberg A., Weiss S., Politakis P., "Automatic knowledge base refinement for classification systems", *Artificial Intelligence* **35** (1988) 197–226.

[13] Evertsz R., "The automatic analysis of rule-based system based on their procedural semantics", in: *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI'91)*, 1991, pp. 22–27.

[14] Chang C., Combs J., Stachowitz R., "Report on the expert systems validation associate (EVA)", *Expert Systems with Applications* **1** (3) (1990) 217–230.

[15] Ayel M., "Protocols for consistency checking in expert system knowledge bases", in: *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI'88)*, 1988, pp. 220–225.

[16] Grogono P., Batarekh A., Preece A., Shinghal R., Suen C., "Expert system evaluation techniques: A selected bibliography", *Expert Systems* (1992) 227–239.

[17] Hoppe P.M.T., "Vvt terminology: A proposal", *IEEE Expert* **8** (3) (1993) 48–55.

[18] Laurent J., "Proposals for a valid terminology in KBS validation", in: *Proceedings of the 10th European Conference on Artificial Intelligence, ECAI-92, Vienna, Austria*, 1992, pp. 829–834.

[19] Lopez E.P.B., Meseguer P., "Knowledge based systems validation: A state of the art", *Artificial Intelligence Communications* **5** (3) (1990) 119–135.

[20] Meseguer P., "Towards a conceptual framework for expert system validation", *Artificial Intelligence Communications* **5** (3) (1992) 119–135.

[21] O'Keefe R., O'Leary D., "Expert system verification and validation: A survey and tutorial", *Artificial Intelligence Review* **7** (1993) 3–42.

[22] R.O.R.M., Balci O., Smith E., "Validating expert system performance", *IEEE Expert* **87** (1987) 81–89.

[23] Rushby J., "Quality measures and assurance for AI software", sRI-CSL-88-7R, SRI Project 4616 (1988).

[24] Zlatereva N., Preece A., "State of the art in automated validation of knowledge-based systems", *Expert Systems with Applications* **7** (1994) 151–167.

[25] Soloway E., Bachant J., Jensen K., "Assessing the maintainability of XCON-in-rime: Coping with the problems of a very large rule-base", in: *AAAI '87*, 1987, pp. 824–829.

[26] McDermott J., "R1's formative years", *AI Magazine* **2** (2) (1981) 21–29.

[27] Hamilton C.C.D., Kelley K., "State-of-the-practice in knowledge-based system verification and validation", *Expert Systems with Applications* **3** (1991) 403–410.

[28] Bachant J., McDermott J., "R1 Revisited: Four years in the trenches", *AI Magazine* (1984) 21–32.

[29] Buchanan B., Shortliffe E., *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison–Wesley, Reading, MA, 1984.

[30] McConnell S., "The best influences on software engineering", IEEE Software. Available from http://www.computer.org/software/so2000/pdf/s1010.pdf.

[31] Boehm B., "Safe and simple software cost analysis", *IEEE Software* (2000) 14–17. Available from http://www.computer.org/certification/beta/Boehm_Safe.pdf.

[32] Shull F., Basili V., Boehm B., Brown A., Costa P., Lindvall M., Port D., Rus I., Tesoriero R., Zelkowitz M., "What we have learned about fighting defects", in: *Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada*, 2002, pp. 249–258. Available from http://fc-md.umd.edu/fcmd/Papers/shull_defects.ps.

[33] Rushby J., "Disappearing formal methods", in: *High-Assurance Systems Engineering Symposium*, Association for Computing Machinery, Albuquerque, NM, 2000, pp. 95–96, http://www.csl.sri.com/~rushby/hase00.html.

[34] Brat G., Giannakopoulou D., Goldberg A., Havelund K., Lowry M., Pasareanu C., Venet A., Visser W., "Experimental evaluation of verification and validation tools on

Martian rover software", in: *CMU/SEI Software Model Checking Workshop, Pittsburg, USA*, 2003; extended version to appear in *Formal Methods in System Design*.

[35] Drusinsky D., "The temporal rover and the ATG rover", in: *SPIN Model Checking and Software Verification*, in: *Lecture Notes in Computer Science*, vol. 1885, Springer-Verlag, Berlin/New York, 2000, pp. 323–330.

[36] Savage S., Burrows M., Nelson G., Sobalvarro P., Anderson T., "Eraser: A dynamic data race detector for multithreaded programs", *ACM Transactions on Computer Systems* **15** (4) (1997) 391–411.

[37] Havelund K., Roşu G., "Monitoring Java programs with Java PathExplorer", in: 1st Workshop on Runtime Verification (RV'01), Paris, France, *Electronic Notes in Theoretical Computer Science* **55** (2001).

[38] Artho C., Drusinsky D., Goldberg A., Havelund K., Lowry M., Pasareanu C., Roşu G., Visser W., "Experiments with test case generation and runtime analysis", in: Börger E., Gargantini A., Riccobene E. (Eds.), *Abstract State Machines (ASM'03)*, in: *Lecture Notes in Computer Science*, vol. 2589, Springer-Verlag, Berlin/New York, 2003, pp. 87–107, version to appear in the journal of Theoretical Computer Science.

[39] Barringer H., Goldberg A., Havelund K., Sen K., "Rule-based runtime verification", in: *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04), Venice, Italy*, 2004.

[40] Nielson F., Nielson H.R., Hankin C., *Principles of Program Analysis*, Springer-Verlag, Berlin/New York, 1999.

[41] Deutsch A., "Next generation testing tools for embedded applications", white paper, PolySpace Technologies, http://www.polyspace.com/docs/Static_Verification_paper.pdf.

[42] Venet A., Brat G., "Precise and efficient static array bound checking for large embedded C programs", in: *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI04)*, ACM Press, Washington DC, USA, 2004, pp. 231–242.

[43] Queille J.-P., Sifakis J., "Specification and verification of concurrent systems is CESAR", in: *International Symposium on Programming*, in: *Lecture Notes in Computer Science*, vol. 137, Springer-Verlag, Berlin/New York, 1982, pp. 337–351.

[44] Clarke E., Emerson E., Sistla A., "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Transactions on Programming Languages and Systems* **8** (2) (1986) 244–263.

[45] Holzmann G., "Software model checking with SPIN", in: Zelkowitz M.V. (Ed.), *Advances in Computers*, vol. 65, Elsevier, Amsterdam, 2005, pp. 77–108 (this volume).

[46] Holzmann G.J., "The model checker SPIN", *IEEE Transactions on Software Engineering* **23** (5) (1997).

[47] Burch J.R., Clarke E.M., McMillan K.L., Dill D.L., Hwang J., "Symbolic model checking: 10^{20} states and beyond", *Information and Computation* **98** (2) (1992) 142–170.

[48] Cimatti A., Clarke E., Giunchiglia F., Roveri M., "NuSMV: A new symbolic model verifier", in: *Proceedings of International Conference on Computer-Aided Verification*, 1999.

[49] Clarke E.M., Grumberg O., Peled D., *Model Checking*, MIT Press, Cambridge, MA, 1999.

[50] Bérard B., Bidoit M., Finkel A., Laroussinie F., Petit A., Petrucci L., Schnoebelen P., *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer-Verlag, Berlin/New York, 2001.

[51] Garavel H., Jorgensen M., Mateescu R., Pecheur C., Sighireanu M., Vivien B., "Cadp'97—status, applications and perspectives", in: Lovrek I. (Ed.), *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, 1997.

[52] Corbett J., Dwyer M., Hatcliff J., Laubach S., Pasareanu C., "Bandera: Extracting finite-state models from Java source code", in: *Proceedings ICSE2000, Limerick, Ireland*, 2000, pp. 439–448.

[53] Visser W., Havelund K., Brat G., Park S., "Model checking programs", in: *Proceedings of the IEEE International Conference on Automated Software Engineering*, 2000, pp. 3–12.

[54] Heitmeyer C., "Software cost reduction", in: Marciniak J.J. (Ed.), *Encyclopedia of Software Engineering*, 2002. Available from http://chacs.nrl.navy.mil/publications/CHACS/2002/2002heitmeyer-encse.pdf.

[55] Heitmeyer C., Jeffords R., Labaw B., "Automated consistency checking of requirements specifications", *ACM Transactions on Software Engineering and Methodology* **5** (3) (1996) 231–261. Available from http://citeseer.nj.nec.com/heitmeyer96automated.html.

[56] Heitmeyer C., Labaw B., Kiskis D., "Consistency checking of SCR-style requirements specifications", in: *International Symposium on Requirements Engineering, York, England*, March 26–27, 1995.

[57] Menzies T., Powell J., Houle M.E., "Fast formal analysis of requirements via 'topoi diagrams' ", in: *ICSE, 2001*, 2001. Available from http://menzies.us/pdf/00fastre.pdf.

[58] Pecheur C., Simmons R., "From Livingstone to SMV: Formal verification for autonomous spacecrafts", in: Verlag S. (Ed.), *Proceedings of 1st Goddard Workshop on Formal Approaches to Agent-Based Systems*, in: *Lecture Notes in Computer Science*, vol. 1871, NASA Goddard, 2000.

[59] Dwyer M.B., Avrunin G.S., Corbett J., "A system specification of patterns", http://www.cis.ksu.edu/santos/spec-patterns/.

[60] Dwyer M., Avrunin G., Corbett J., "Patterns in property specifications for finite-state verification", in: *ICSE98: Proceedings of the 21st International Conference on Software Engineering*, 1998.

[61] Pasareanu C., Visser W., "Verification of java programs using symbolic execution and invariant generation", in: *Proceedings of SPIN 2004, Barcelona, Spain*, in: *Lecture Notes in Computer Science*, vol. 2989, Springer-Verlag, Berlin/New York, 2004.

[62] Biere A., Cimatti A., Clarke E.M., Zhu Y., "Symbolic model checking without BDDs", in: Cleaveland R. (Ed.), *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, in: *Lecture Notes in Computer Science*, vol. 1579, Springer-Verlag, Berlin/New York, 1999, pp. 193–207.

[63] Clarke E.M., Grumberg O., Long D.E., "Model checking and abstraction", *ACM Transactions on Programming Languages Systems* **16** (5) (1994) 1512–1542.

[64] Graf S., Saidi H., “Verifying invariants using theorem proving”, in: *Conference on Computer Aided Verification CAV’96*, in: *Lecture Notes in Computer Science*, vol. 1102, Springer-Verlag, Berlin/New York, 1996.

[65] Clarke E., Filkorn T., “Exploiting symmetry in temporal logic model checking”, in: *Fifth International Conference on Computer Aided Verification*, Springer-Verlag, Berlin/New York, 1993.

[66] Ip C., Dill D., “Better verification through symmetry”, *Formal Methods in System Design* **9** (1/2) (1996) 41–75.

[67] Clarke E., Long D.E., “Compositional model checking”, in: *4th Annual Symposium on Logic in Computer Science*, 1989.

[68] Clancy D., Kuipers B., “Model decomposition and simulation: A component based qualitative simulation algorithm”, in: *AAAI-97*, 1997.

[69] Giannakopoulou D., Pasareanu C., Barringer H., “Assumption generation for software component verification”, in: *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002), Edinburgh, UK*, 2002.

[70] Havelund K., Lowry M., Park S., Pecheur C., Penix J., Visser W., White J.L., “Formal analysis of the remote agent before and after flight”, in: *Proceedings of 5th NASA Langley Formal Methods Workshop, Williamsburg, Virginia*, 2000.

[71] Giannakopoulou D., Pasareanu C., Barringer H., “Assumption generation for software component verification”, in: *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002), Edinburgh, UK*, 2002.

[72] Owre S., Rajan S., Rushby J.M., Shankar N., Srivas M.K., “PVS: Combining specification, proof checking, and model checking”, in: *Proceedings of the 8th International Conference on Computer Aided Verification*, Springer-Verlag, Berlin/New York, 1996, pp. 411–414.

[73] Crow J., Di Vito B., “Formalizing space shuttle software requirements: four case studies”, *ACM Transactions Software Engineering Methodology* **7** (3) (1998) 296–332.

[74] Owre S., Rushby J., Shankar N., “Integration in PVS: tables, types, and model checking”, in: Brinksma E. (Ed.), *Tools and Algorithms for the Construction and Analysis of Systems TACAS ’97, Enschede, The Netherlands*, in: *Lecture Notes in Computer Science*, vol. 1217, Springer-Verlag, Berlin, 1997, pp. 366–383, http://www.csl.sri.com/papers/tacas97/.

[75] de Moura L., Owre S., Ruess H., Rushby J., Shankar N., “The ICS decision procedures for embedded deduction”, July 2004.

[76] Bratko I., *Prolog Programming for Artificial Intelligence*, third ed., Addison–Wesley, Reading, MA, 2001.

[77] Kalman J.A., *Automated Reasoning with OTTER*, Rinton Press, Reading, MA, 2002.

[78] Stickel M., “A Prolog technology theorem prover: A new exposition and implementation in Prolog”, *Theoretical Computer Science* **104** (1992) 109–128.

[79] Kautz H., Selman B., “Pushing the envelope: Planning, propositional logic and stochastic search”, in: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, AAAI Press/MIT Press, Menlo Park, 1996, pp. 1194–1201. Available from http://www.cc.gatech.edu/~jimmyd/summaries/kautz1996.ps.

[80] Nayak P.P., Williams B.C., "Fast context switching in real-time propositional reasoning", in: *Proceedings of AAAI-97*, 1997. Available from http://ack.arc.nasa.gov:80/ic/projects/mba/papers/aaai97.ps.

[81] Davis R., "Interactive transfer of expertise: Acquisition of new inference rules", *Artificial Intelligence* **12** (2) (1979) 121–157.

[82] Groce A., Visser W., "Heuristic model checking for Java programs", in: *Proceedings of SPIN 2002, Grenoble, France*, 2002.

[83] Cimatti A., Roveri M., "Conformant planning via symbolic model checking", *Journal of Artificial Intelligence Research* **13** (2000) 305–338.

[84] Williams B.C., Nayak P.P., "A model-based approach to reactive self-configuring systems", in: *Proceedings of AAAI-96*, 1996.

[85] Clancy D., Larson W., Pecheur C., Engrand P., Goodrich C., "Autonomous control of an in-situ propellant production plant", in: *Proceedings of Technology 2009 Conference, Miami*, 1999.

[86] Cimatti A., Pecheur C., Cavada R., "Formal verification of diagnosability via symbolic model checking", in: *Proceedings of IJCAI'03, Acapulco, Mexico*, 2003.

[87] Feather M., Smith B., "Automatic generation of test oracles: From pilot studies to applications", in: *Proceedings of the Fourteenth IEEE International Conference on Automated Software Engineering (ASE-99), Cocoa Beach, Florida*, 1999, pp. 63–72. Available from http://www-aig.jpl.nasa.gov/public/planning/papers/oracles-ase.pdf.

[88] Muscettola N., Nayak P.P., Pell B., Williams B., "Remote agent: To boldly go where no AI system has gone before", *Artificial Intelligence* **103** (1–2) (1998) 5–48.

[89] van Harmelen F., Aben M., "Structure-preserving specification languages for knowledge-based systems", *International Journal of Human–Computer Studies* **44** (1996) 187–212.

[90] Gardner K.M., Rush A.R., Crist M., Konitzer R., Odell J.J., Teegarden B., Konitzer R., *Cognitive Patterns: Problem-Solving Frameworks for Object Technology*, Cambridge University Press, Cambridge, UK, 1998.

[91] Wielinga B., Schreiber A., Breuker J., "KADS: a modeling approach to knowledge engineering", *Knowledge Acquisition* **4** (1992) 1–162.

[92] Schreiber A.T., Wielinga B., Akkermans J.M., Velde W.V.D., de Hoog R., "CommonKADS: a comprehensive methodology for KBS development", *IEEE Expert* **9** (6) (1994) 28–37.

[93] Schreiber G. (Ed.), *Knowledge Engineering and Management: The CommonKADS Methodology*, MIT Press, Cambridge, MA, 1999.

[94] Swartout B., Gill Y., "Flexible knowledge acquisition through explicit representation of knowledge roles", in: *1996 AAAI Spring Symposium on Acquisition, Learning, and Demonstration: Automating Tasks for Users*, 1996.

[95] Gil Y., Melz E., "Explicit representations of problem-solving strategies to support knowledge acquisition", in: *Proceedings AAAI' 96*, 1996.

[96] Angele J., Fensel D., Studer R., "Domain and task modelling in MIKE", in: A.G. Sutcliffe, et al. (Eds.), *Domain Knowledge for Interactive System Design*, Chapman & Hall, London/New York, 1996.

[97] Eriksson H., Shahar Y., Tu S.W., Puerta A.R., Musen M.A., “Task modeling with reusable problem-solving methods”, *Artificial Intelligence* **79** (2) (1995) 293–326.

[98] Chandrasekaran B., Johnson T., Smith J.W., “Task structure analysis for knowledge modeling”, *Communications of the ACM* **35** (9) (1992) 124–137.

[99] Marques D., Dallemagne G., Kliner G., McDermott J., Tung D., “Easy programming: Empowering people to build their own applications”, *IEEE Expert* (1992) 16–29.

[100] Clancey W., “Model construction operators”, *Artificial Intelligence* **53** (1992) 1–115.

[101] Steels L., “Components of expertise”, *AI Magazine* **11** (1990) 29–49.

[102] Benjamins R., “Problem-solving methods for diagnosis and their role in knowledge acquisition”, *International Journal of Expert Systems: Research and Applications* **8** (2) (1995) 93–120.

[103] Breuker J., de Velde W.V. (Eds.), *The CommonKADS Library for Expertise Modelling*, IOS Press, Netherlands, 1994.

[104] Motta E., Zdrahal Z., “Parametric design problem solving”, in: *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop*, 1996.

[105] Tansley D., Hayball C., *Knowledge-Based Systems Analysis and Design*, Prentice Hall, New York, 1993.

[106] de Brug A.V., Bachant J., McDermott J., “The taming of R1”, *IEEE Expert* (1986) 33–39.

[107] Gil Y., Tallis M., “A script-based approach to modifying knowledge bases”, in: *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, 1997.

[108] Musa J., Iannino A., Okumoto K., *Software Reliability: Measurement, Prediction, Application*, McGraw–Hill, New York, 1987.

[109] Pasquini A., Crespo A.N., Matrella P., “Sensitivity of reliability-growth models to operational profile errors vs testing accuracy”, *IEEE Transactions on Reliability* **45** (4) (1996) 531–540.

[110] Muscettola N.A.R.C.N., Personal communication (2000).

[111] Smith B., Feather M., Muscettola N., “Challenges and methods in validating the remote agent planner”, in: *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS-2000)*, 2000. Available from http://www-aig.jpl.nasa.gov/public/home/smith/publications.html.

[112] Selman B., Levesque H., Mitchell D., “A new method for solving hard satisfiability problems”, in: *AAAI ’92*, 1992, pp. 440–446.

[113] Menzies T., Singh H., “Many maybes mean (mostly) the same thing”, in: Madravio M. (Ed.), *Soft Computing in Software Engineering*, Springer-Verlag, Berlin/New York, 2003. Available from http://menzies.us/pdf/03maybe.pdf.

[114] Crawford J., Baker A., “Experimental results on the application of satisfiability algorithms to scheduling problems”, in: *AAAI ’94*, 1994.

[115] Rymon R., “An se-tree-based prime implicant generation algorithm”, *Annals of Mathematics and Artificial Intelligence* **11** (1994) (special issue on model-based diagnosis). Available from http://citeseer.nj.nec.com/193704.html.

[116] Rymon R., “An SE-tree based characterization of the induction problem”, in: *International Conference on Machine Learning*, 1993, pp. 268–275.

[117] Lutz R., Woodhouse R., "Bi-directional analysis for certification of safety-critical software", in: *1st International Software Assurance Certification Conference (ISACC'99)*, 1999. Available from http://www.cs.iastate.edu/~rlutz/publications/isacc99.ps.

[118] Parkes A., "Lifted search engines for satisfiability", http://citeseer.nj.nec.com/parkes99lifted.html, 1999.

[119] Singer J., Gent I.P., Smaill A., "Backbone fragility and the local search cost peak", *Journal of Artificial Intelligence Research* **12** (2000) 235–270, http://citeseer.nj.nec.com/singer00backbone.html.

[120] Williams R., Gomes C., Selman B., "Backdoors to typical case complexity", in: *Proceedings of IJCAI, 2003*, 2003, http://www.cs.cornell.edu/gomes/FILES/backdoors.pdf.

[121] Josephson J., Chandrasekaran B., Carroll M., Iyer N., Wasacz B., Rizzoni G., "Exploration of large design spaces: an architecture and preliminary results", in: *AAAI '98*, 1998. Available from http://www.cis.ohio-state.edu/~jj/Explore.ps.

[122] DeKleer J., "An assumption-based TMS", *Artificial Intelligence* **28** (1986) 163–196.

[123] Feather M., Menzies T., "Converging on the optimal attainment of requirements", in: *IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 9–13 September*, University of Essen, Germany, 2002. Available from http://menzies.us/pdf/02re02.pdf.

[124] Menzies T., Chiang E., Feather M., Hu Y., Kiper J., "Condensing uncertainty via incremental treatment learning", in: Khoshgoftaar T.M. (Ed.), *Software Engineering with Computational Intelligence*, Kluwer, Dordrecht/Norwell, MA, 2003. Available from http://menzies.us/pdf/02itar2.pdf.

[125] Valmari A., "A stubborn attack on state explosion", in: Kurshan R.P., Clarke E.M. (Eds.), *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, in: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 3, AMS–ACM, 1990, pp. 25–42.

[126] Godefroid P., *Partial-Order Methods for the Verification of Concurrent Systems—An Approach to the State-Explosion Problem*, *Lecture Notes in Computer Science*, vol. 1032, Springer-Verlag, Berlin/New York, 1996.

[127] Godefroid P., "On the costs and benefits of using partial-order methods for the verification of concurrent systems (invited papers)", in: *The 1996 DIMACS Workshop on Partial Order Methods in Verification, July 24–26, 1996*, 1997, pp. 289–303.

[128] Schneider F., Easterbrook S., Callahan J., Holzmann G., "Validating requirements for fault tolerant systems using model checking", in: *3rd IEEE International Conference On Requirements Engineering*, 1998.

[129] van Harmelen F., Bundy A., "Explanation-based generalisation = partial evaluation", *Artificial Intelligence* (1988) 401–412.

[130] Mitchell T., Keller R., Kedar-Cabelli S.T., "Explanation-based generalization: A unifying view", *Machine Learning* **1** (1986) 47–80.

[131] Goldberg D., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison–Wesley, Reading, MA, 1989.

[132] Kirkpatrick S., Gelatt C.D., Vecchi M.P., "Optimization by simulated annealing", *Science* **4598** (13 May 1983) 220; *Science* **4598** (1983) 671–680; http://citeseer.nj.nec.com/kirkpatrick83optimization.html.

[133] Quinlan R., *C4.5: Programs for Machine Learning*, Morgan Kaufman, San Mateo, CA, ISBN: 1-55860-238-0, 1992.

[134] Srinivasan K., Fisher D., "Machine learning approaches to estimating software development effort", *IEEE Transactions on Software Engineering* (1995) 126–137.

[135] Bouckaert R., "Choosing between two learning algorithms based on calibrated tests". Available from http://www.cs.pdx.edu/~timm/dm/10x10way, 2003.

[136] Liu Y., Menzies T., Cukic B., "Data sniffing—monitoring of machine learning for online adaptive systems", in: *IEEE Tools with AI*, 2002. Available from http://menzies.us/pdf/03datasniffing.pdf.

[137] Liu Y., Menzies T., Cukic B., "Detecting novelties by mining association rules". Available from http://menzies.us/pdf/03novelty.pdf, 2003.

[138] Liu Y., Gururajan S., Cukic B., Menzies T., Napolitano M., "Validating an online adaptive system using SVDD", in: *IEEE Tools with AI*, 2003. Available from http://menzies.us/pdf/03svdd.pdf.

[139] Witten I.H., Frank E., *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, San Mateo, CA, 1999.

[140] Burgess C., Lefley M., "Can genetic programming improve software effort estimation? A comparative evaluation", *Information and Software Technology* **43** (14) (2001) 863–873.

[141] Taylor B., "Development of methodologies for iv&v neural networks: Literature survey of current v&v technology". Available from http://sarpresults.ivv.nasa.gov/ViewResearch/914/11.jsp, 2004.

[142] Hu Y., "Treatment learning: Implementation and application", Masters Thesis, Department of Electrical Engineering, University of British Columbia, 2003.

[143] Menzies T., Hu Y., "Reusing models for requirements engineering", in: *1st International Workshop on Model-based Requirements Engineering*, 2001. Available from http://menzies.us/pdf/01reusere.pdf.

[144] Menzies T., Hu Y., "Constraining discussions in requirements engineering", in: *1st International Workshop on Model-based Requirements Engineering*, 2001. Available from http://menzies.us/pdf/01lesstalk.pdf.

[145] Menzies T., Hu Y., "Agents in a wild world", in: Rouff C. (Ed.), *Formal Approaches to Agent-Based Systems*, 2002. Available from http://menzies.us/pdf/01agents.pdf.

[146] Menzies T., Hu Y., "Just enough learning (of association rules): The TAR2 treatment learner", *Artificial Intelligence Review* (2004). Available from http://menzies.us/pdf/02tar2.pdf.

[147] Menzies T., Hu Y., "Data mining for very busy people", *IEEE Computer* (2003). Available from http://menzies.us/pdf/03tar2.pdf.

[148] Menzies T., Sinsel E., Kurtz T., "Learning to reduce risks with cocomo-ii", in: *Workshop on Intelligent Software Engineering, An ICSE 2000 Workshop, and NASA/WVU Software Research Lab, Fairmont, WV, Tech. report # NASA-IVV-99-027, 1999*, 2000. Available from http://menzies.us/pdf/00wise.pdf.

Indexing, Learning and Content-Based Retrieval for Special Purpose Image Databases

MARK J. HUISKES

Centre for Mathematics and Computer Science
Kruislaan 413, 1098SJ Amsterdam
The Netherlands
Mark.Huiskes@cwi.nl

ERIC J. PAUWELS

Centre for Mathematics and Computer Science
PNA4 (Signals and Images)
Kruislaan 413, 1090GB Amsterdam
The Netherlands
Eric.Pauwels@cwi.nl

Abstract
This chapter deals with content-based image retrieval in special purpose image databases. As image data is amassed ever more effortlessly, building efficient systems for searching and browsing of image databases becomes increasingly urgent. We provide an overview of the current state-of-the art by taking a tour along the entire "image retrieval chain"—from processing raw image data, through various methods of machine learning, to the interactive presentation of query results.

As it often constitutes the key to building successful retrieval systems, we first discuss the issue of content representation and indexing. Here both the computation of global and local characteristics based on image segmentations is reviewed in some detail in the context of interior design images. Also the representation of content by means of MPEG-7 standard metadata is introduced.

In regard to the search system itself, we focus particularly on interfaces and learning algorithms which facilitate relevance feedback, i.e., on systems that allow for natural interaction with the user in refining queries by means of feedback directly in terms of example images. To this end the literature on this subject is reviewed, and an outline is provided of the special structure of the relevance feed-

ADVANCES IN COMPUTERS, VOL. 65
ISSN: 0065-2458/DOI 10.1016/S0065-2458(05)65005-X

back learning problem. Finally we present a probabilistic approach to relevance feedback that addresses this special structure.

1. Introduction

In this chapter we are concerned with *content-based* retrieval in *special purpose* image databases. In this context, “content-based” means that we aim to characterize images primarily by analyzing their intrinsic visual content by machine rather than by relying on “external” descriptions. So we let our systems derive descriptions based on the analysis of the image itself, instead of using manually annotated keywords, or as in the case of for instance Google’s image search, using the image caption or text on the webpage adjacent to the image. With “special purpose” we indicate that we restrict ourselves to domains where queries are limited in the terms by which they are formulated and specialized domain knowledge can help us in modeling these terms. Particular examples of such databases occur in domains such as forensics (e.g., fingerprints or mug shot databases), trademark protection (e.g., [10]), medicine, biomonitoring (e.g., [45]), and interior design (discussed below).

To realize the promise of content-based browsing and searching in image databases, the main obstacle remains the well-known semantic gap. In [55] it is defined as "the lack of coincidence between the information that one can extract from the visual data and the interpretation that the same data have for a user in a given situation." The automatically obtained visual measures, or *features*, are typically low-level and often fall short of the semantics of human subjectivity.

Additionally, we must deal with the fact that people often have different interpretations of the same image, or worse, that one and the same person has different perceptions in different situations.

Relevance feedback has often been suggested as a (partial) solution to these formidable problems, in particular to dealing with the user- and task-dependence of image interpretation. The most natural way to provide such feedback is, arguably, by letting the user select both positive and negative examples to indicate his respective preferences and dislikes. This allows the system to extract which features are important for the query at hand.

Even when using relevance feedback, however, ultimate success will still depend on the richness and accuracy of the *representations* we construct of our images. In this chapter we will discuss both the construction of such representations, as a combination of modeling and learning, and the inference techniques to use the representations in transforming the feedback into image relevance measures. Machine learning will play an important role throughout, both for building a higher level understanding of the designs using lower-level building blocks, and in the task of identifying implied preferences from user feedback.

The methods we discuss here will to a large extent apply to the development of content-based retrieval systems for any type of specialized image database. As a real-life example of a system we describe one for the subdomain of decoration design images.

Decoration design images form a class of images consisting of patterns used in, for instance, various types of textile products (clothes, curtains, carpets) and wallpaper. Figure 1 shows a number of typical examples. Because of the widespread use of images in the textile, fashion and interior decoration industries, the development of retrieval methods for this economically important domain is also valuable in its own right.

As an introduction, we will take a short tour along the main components of the content-based retrieval system developed in the FOUNDIT project [41].

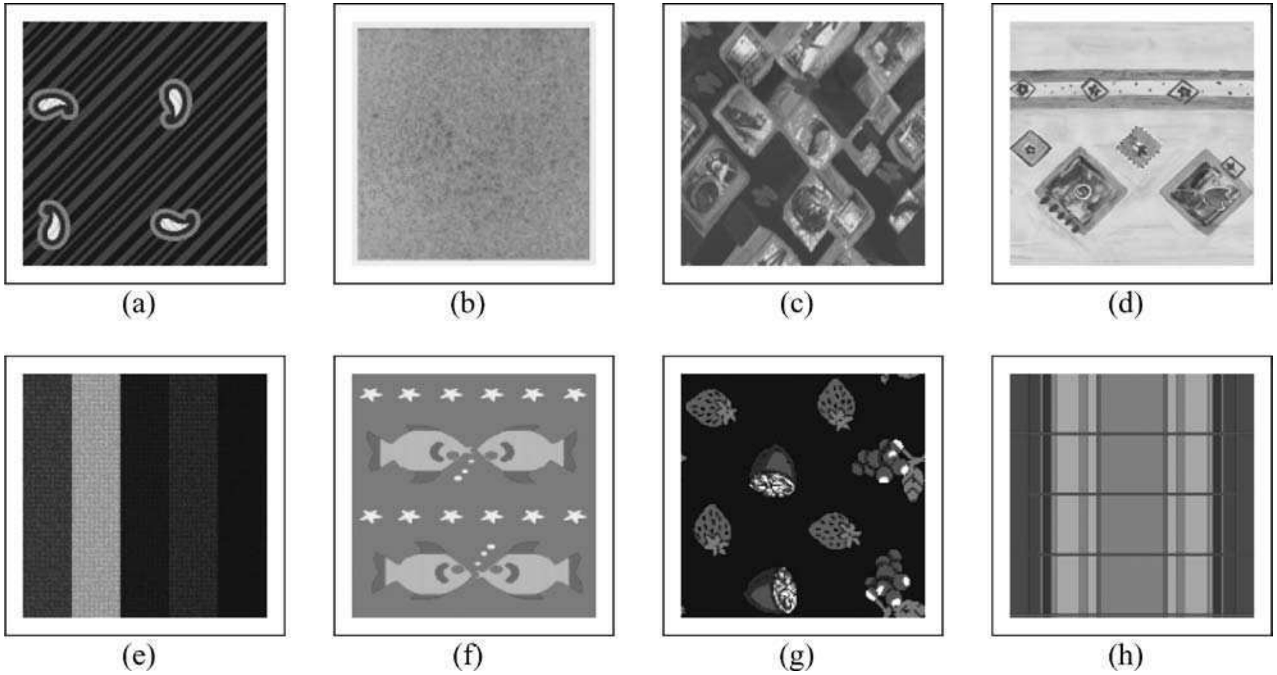


FIG. 1. Examples of design images as used in the textile industry. (See Color Plate Section, Plate 17.)

1.1 The FOUNDIT CBIR System

The European IST project FOUNDIT aims to develop content-based retrieval systems that are particularly geared towards requirements for searching and browsing digital decoration designs. The definition of requirements of the search system has taken place in close collaboration with the textile industry. Scenarios of use include both customer browsing through design collections, and expert search by designers for, among others, re-purposing.

A guiding principle in the design of the search system has been that the only way to elucidate the user's subjective appreciation and preferences, is by continuously soliciting his or her feedback. This feedback is then harnessed to estimate for each image in the database the likelihood of its relevance with respect to the user's goals whereupon the most promising candidates are displayed for further inspection and feedback.

Figure 2 illustrates the main components of the system:

- The *graphical user interface* displays a selection of images from the image database and allows the user to provide the system with relevance feedback by selecting examples and counterexamples.
- The *inference engine* transforms this qualitative feedback into a probabilistic relevance measure for each image by analyzing it in terms of the image features.
- The *feature extraction engine*, or feature factory, generates a feature database of visual characteristics by analyzing the content of the images. Unlike the previous components the feature extraction engine operates off-line.

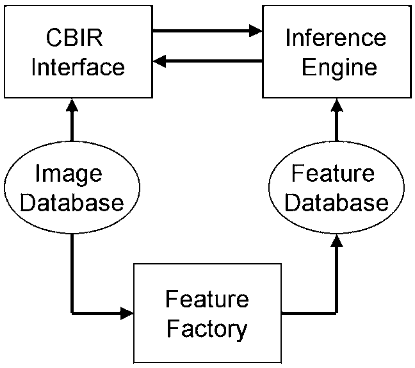


FIG. 2. Main architecture of the FOUNDIT system for content-based image retrieval.

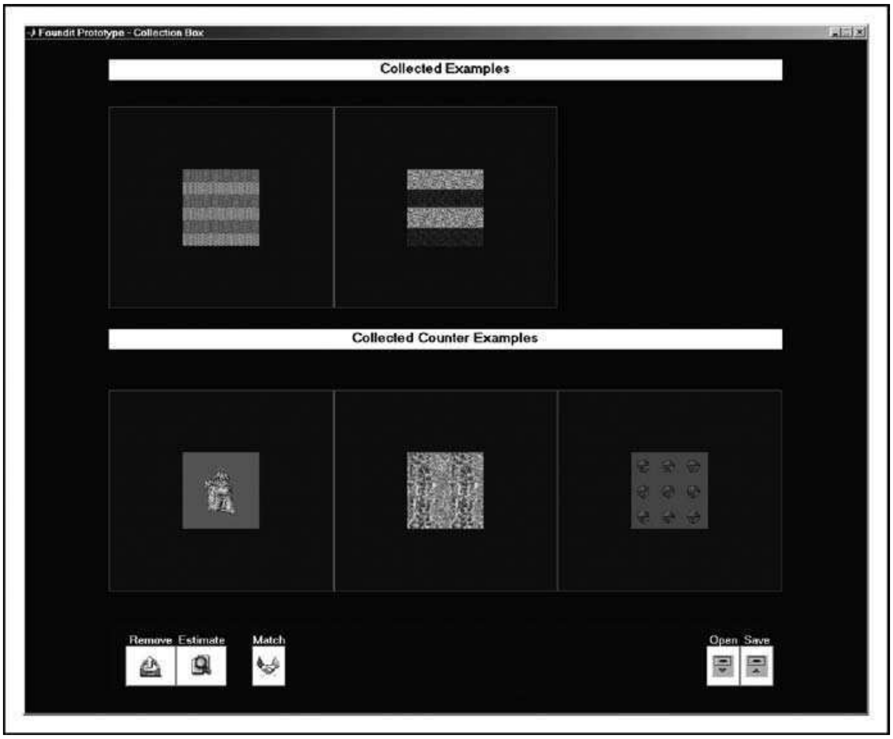

Foundit Prototype - Collection Box
Collected Examples
Collected Counter Examples
Remove
Estimate
Match
Open
Save

FIG. 3. Collection box of positive and negative examples. In this example the user has selected two images with horizontal stripes as positive examples (top row). Additionally 3 images were selected as negative examples (bottom row). (See Color Plate Section, Plate 18.)

A typical query then starts by displaying a random selection of images from the database. The user indicates his preferences by clicking the images, once for images he finds relevant, twice for images that are very dissimilar to what he has in mind. These choices are collected in separate bins of the so-called collection box, as illustrated in Figure 3. Note that for images on which the user has no strong opinion one way or the other, no feedback is provided.

Next, the inference engine is put to work to determine those features or feature combinations that best explain the feedback given. Using a relevance model, this then leads to a ranking of the database images by their predicted relevance which may be used to display the next selection of images (Figure 4).

In the next cycle the user can add further example images to the collection box, or if desired, remove example images if this better represents his wishes. Additionally we have experimented with showing a list of the most promising individual features

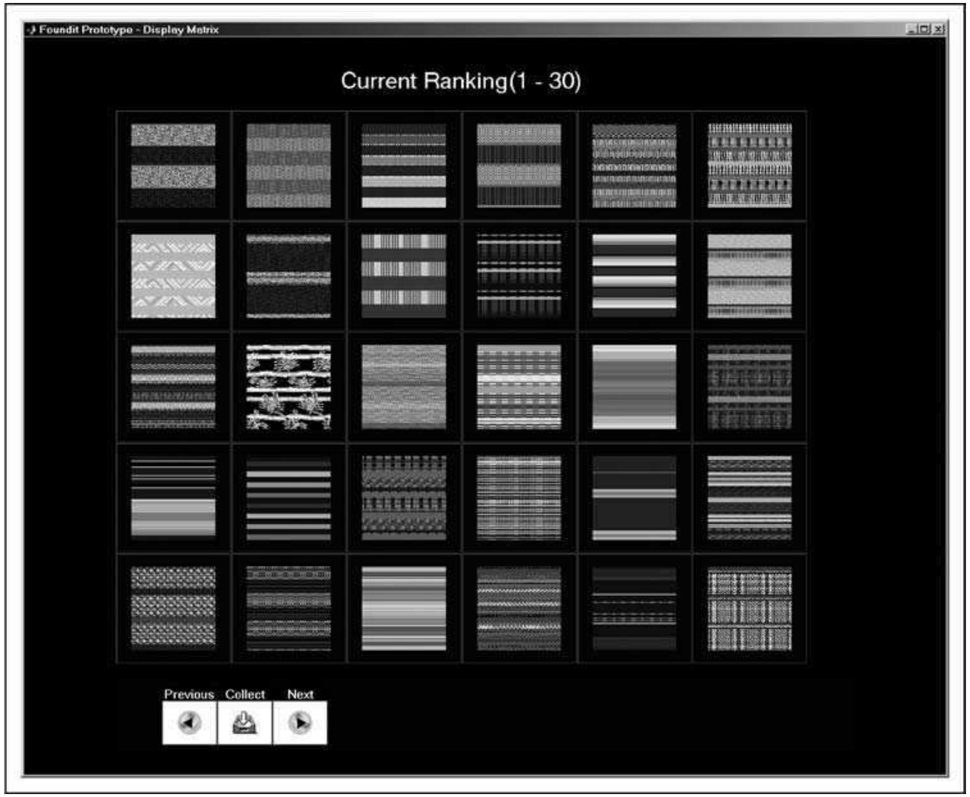
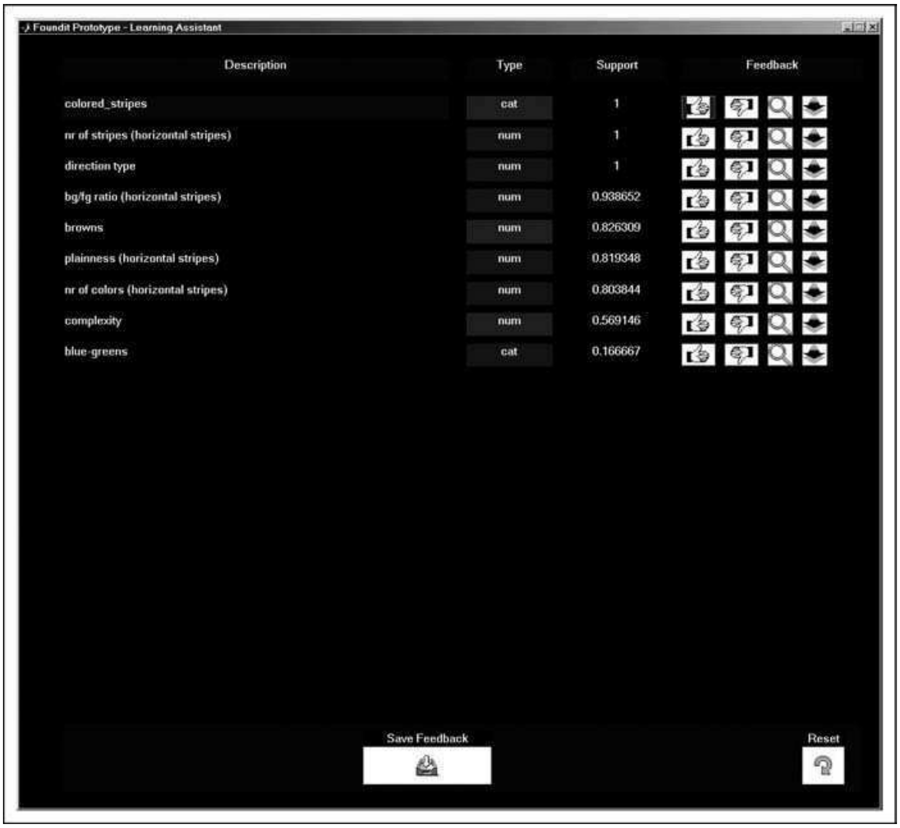


FIG. 4. Relevance ranking of database image based on inference engine analysis. Shown are the 30 top ranked database designs that were found based on the collection box of Figure 3. As expected the selection consists of horizontal stripes; subsequent relevance feedback can now be used for searching on additional properties such as color and number of stripes. (See Color Plate Section, Plate 19.)



Description	Type	Support	Feedback
colored_stripes	cat	1	
nr of stripes (horizontal stripes)	num	1	
direction type	num	1	
bg/fg ratio (horizontal stripes)	num	0.938652	
browns	num	0.826309	
plainness (horizontal stripes)	num	0.819348	
nr of colors (horizontal stripes)	num	0.803844	
complexity	num	0.569146	
blue-greens	cat	0.166667	

FIG. 5. Screenshot of the *learning assistant* interface. On the left the system displays a list of features the inference engine considers particularly important in view of the feedback provided by the user. The user is now free to confirm or reject each of these suggestions. (See Color Plate Section, Plate 20.)

in determining relevance, allowing for a very direct type of feedback in terms of feature names. This type of feedback is expected to be useful mainly for expert users. See Figure 5.

1.2 Outline of the Chapter

The remaining sections of this chapter are largely independent and can be read according the reader’s interest.

Section 2: Presents an overview of methods for feature extraction and the representation of image content.

Section 3: Discusses the detection of salient elements in design images.
Section 4: Provides a short introduction to the MPEG-7 standard for multimedia content description, in particular in the context of design images.
Section 5: Gives an overview of inference and learning techniques for relevance feedback, and focuses on one method specifically aimed at the special structure of feedback data.
Section 6: Conclusion and outlook.

2. Representation of Image Content: Feature Extraction

2.1 Introduction

As indicated above the rich and accurate representation of image content is crucial to the success of any search system. If representation of certain image features is neglected, generally no searching with respect to such feature will be possible other than through accidental correlations with other features.

As generation of features on the fly is generally computationally not feasible with current technology, image representation by features is predominantly a one way street: the full image content is reduced to a single set of fixed, rather inflexible, features. Admittedly, some combination of these features into new features is possible (see, for example, [33]), but this can help us only to a limited extent, i.e., choosing a fixed set of initial features, irrevocably leads to a substantial loss of information. This is in stark contrast to how humans may interpret and re-interpret images depending on different contexts.

Figure 6 provides an illustration of a typical discrepancy between low-level features and high-level perception occurring in the domain of decoration design images.

Another issue in feature extraction is its, often unexpectedly, high level of difficulty. Even in cases where we have sufficient domain knowledge to model all or most features that are expected to be of importance in a search, the abundance of possible variations and special cases we may encounter is often rather stunning. In particular the idiosyncracies of the human attentional system are a great source of problems and ambiguities. To give a simple example: an image could consist almost entirely of small diagonal stripes whereas its predominant perception is horizontal, e.g., the stripes may be grouped in such a way they form a horizontal bar in the foreground. It is, however, of paramount importance that features measure characteristics that are perceptually relevant to the user.

Also note that as an erroneous representation generally leads to correspondingly erroneous behavior in the search system, there is a great need for feature extraction methods that have at least some sort of self- or cross-checking included, such that

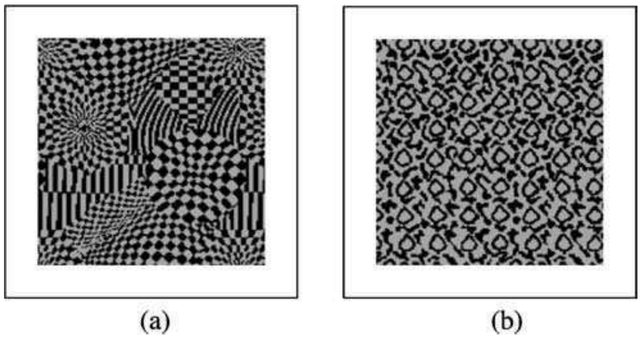


FIG. 6. Illustration of the semantic gap for design images. Both image pairs are similar in terms of low-level features, particularly in their color distributions. However, expert designers do not perceive the designs as similar as they are from different “design classes:” they would classify image (a) as “optical” and (b) as “texture.”

uncertain cases may be treated as such by the search system or, alternatively, be presented for additional supervised evaluation.

We have found it useful to make a distinction between features based on the overall appearance of an image, and features based on the elements occurring in the image. The former will be discussed in Section 2.2, the latter in Section 2.3. The detection of design elements is a topic in itself that will be treated in Section 3. Organization of feature data by means of the MPEG-7 metadata is discussed in Section 4.

2.2 Global Characterization

We start the overview of design image characterization by a discussion of features designed to capture the overall appearance of an image. We expect the features described here to be useful for general images as well. Features are described for overall appearance with respect to color, texture, complexity and periodicity. Most of the features considered may also be used to describe the appearance of a single image region.

2.2.1 Color

Color is a powerful visual attribute in the perception of similarity between images and, as such, has often served as a key source of features in image retrieval systems (e.g., [8,11,58]). Many features have been described in literature for the characterization of color distribution and spatial layout. For the characterization of the global color appearance the challenge lies in the extraction of the perceptually important colors based on their relative occurrence and spatial interactions. See for instance [34] and [40]. Also for decoration designs color may play an important role in determining relevance in a search, both with respect to dominant impression and very specific structural wishes (e.g., a customer wishes the motifs in a certain color). We must note however that in many retrieval applications, it is relatively simple to re-color designs after retrieval (e.g., designs are represented in pseudo-color, and color maps may be modified). For expert users color is thus often of little concern.

For the FOUNDIT system we relied primarily on simple code book histogram approaches. Both data-driven and fixed code books were used. For the data-driven code book pixels are sampled from images in the database and K-means clustering is used to determine a set of representative colors. For the fixed code book we use a list of named colors and associated RGB values covering most of the colors generally used in the decorative designs.

For both code book types a pixel is assigned either fully or partially to a bin weighted by its distance to the color associated with the bin. To determine distances between colors we use the Euclidian metric in CIE Lab space. The features consist of the relative contributions of the image to each bin, using thresholds to prevent the contribution of perceptually irrelevant colors.

Additional features are obtained by adding contributions of color bins that can be associated with a single color name, e.g., all blue-ish colors. These metacolor features show very satisfactory performance with regard to capturing subjective experience of dominant color.

In addition we used a number of the color descriptors defined in the MPEG-7 standard, described in detail in [37]. For the decoration design images we found the following descriptors to be useful:

- The Dominant Color Descriptor (DCD), which provides a very compact description of the dominant colors in an image. Unlike the traditional histogram-based methods, this descriptor computes a small sequence of representative colors in the image with associated area percentages and measures of color variance. Additionally an overall spatial homogeneity coefficient of the dominant colors in the image is computed.
- The Scalable Color Descriptor (SCD), which applies a Haar transform-based encoding scheme across values of a color histogram in HSV color space.
- The Color Structure Descriptor (CSD), which takes into account not only the color distribution of the image, but also the local spatial structure of the colors.

Finally, simple features based on saturation and value from the HSV color space are used as well since we found they correlate well with the experienced degree of “liveliness” and brightness.

2.2.2 Direction and Texture

In design images global and local directions of design patterns are often central to the design, think for instance of striped patterns, or tartans. It is thus important to be able to identify those orientations in an image that are perceived as dominant.

Many methods are available to assess typical line or stripe pattern angles (e.g., [2,12,21]). In the FOUNDIT system we have implemented a method based on the Radon transform of an edge map for detection of horizontal, vertical and diagonal dominant directions. Additionally we use pooled bins of the MPEG-7 Edge Histogram Descriptor. The first method counts edges occurring on lines of certain orientation, the second method is complementary in the sense that it counts edges of certain orientation. Note that these simple features often fail for curves that are not sufficiently straight, e.g., for patterns of horizontal waves. For such cases we must first detect the respective design elements after which we may establish their orientation. Another issue that supports this latter approach is that the features often react to edges that are not relevant, e.g., a design with a vertically oriented texture ground with a horizontal stripe, will often be quantified as predominantly vertical whereas subjective experience would usually favor a horizontal interpretation (remember Figure 7).

The occurrence of stripes in designs is so common that we have developed a template-based method for their detection. This method is based both on grouping of edges and evaluation of homogeneity within the stripe.

A stripe is considered to constitute a region in an image which (i) is bounded by two relatively straight lines spanning the image, and (ii) has a relative homogeneous appearance in between those lines, which differs from the appearance outside these

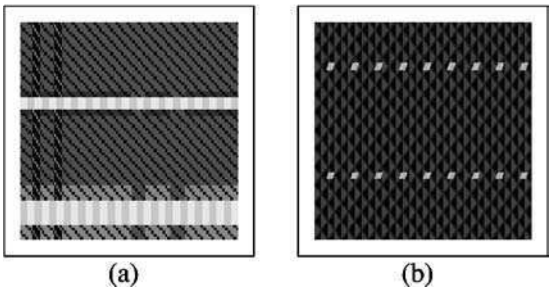


FIG. 7. Examples of design images with conflicting direction features and directional perception. In (a) we see strong perceptual evidence for horizontal (and to lesser extent, vertical) lines, whereas strictly numerically the diagonal lines outnumber these by far. Similarly, in (b) we have many diagonal lines and hardly have any horizontal lines or edges at all; still our perception of this design is predominantly horizontal because of the arrangement of the small motifs. (See Color Plate Section, Plate 21.)

lines. The sense in which the homogeneity is to be understood is not defined in the algorithm. In practice this means we assume that the user provides an indexed image in which each index represents a certain homogeneity type. One can think for instance of an image resulting from a color or texture segmentation (or from a combination of both).

Several aspects with regard to stripes may play a role in perceived similarity of designs. Think of the occurrence and location of stripes, their orientation, the width of the stripes, variation in the stripe widths and color and texture of the stripes. Useful features quantifying such properties are extracted at various stages of the algorithm. For instance after the edge line detection, we count the number of edge lines in an image, or consider the relative number of edge lines. Once stripes have been detected, we compute the density of the stripes in the image; additionally, we compute the average distance between the stripes.

A number of composite features were computed as well, e.g., a feature measuring the occurrence of both horizontal and vertical or two diagonal directions of (thin) stripes simultaneously, for the detection of tartan images; and features assessing the possibility that the background of an image consists of stripes.

Many methods are also available for characterizing image texture (e.g., [5,14,16, 24,25,27,30,31,44,47]). From the MPEG-7 texture descriptors (see [6]), we have used the following descriptors:

- The Homogeneous Texture Descriptor (HTD), which characterizes texture using the mean energy and the energy deviation from a set of 30 frequency channels. The frequency plane partitioning is uniform along the angular direction (equal steps of 30 degrees), but not uniform along the radial direction (which is on an octave scale). There is some evidence that the early visual processing in the human visual cortex can be modelled well using a similar frequency layout.
- The Texture Browsing Descriptor (TBD), which specifies the perceptual characterization of a texture in terms of regularity (4 levels), coarseness (2 quantized scales) and directionality (2 quantized directions). The filtering is performed using a Gabor filter extraction method with a similar frequency layout as in the HTD.
- The Edge Histogram Descriptor (EHD), which describes the spatial distribution of the edges in an image.

2.2.3 Complexity

For designers overall complexity is an influential factor in the perception of design similarity. Of course, complexity is not a well-defined quantity that can be determined objectively. Rather we observe that a number of design features correlate well with subjectively perceived design complexity, e.g., the number of colors occurring in the design, the “amount of discontinuity,” its “crowdedness” or its “level of detail.”

We modeled the level of detail by means of a summarizing feature resulting from a multiresolution analysis, along the lines of for instance [4], where the image is decomposed into terms representing contributions at different levels of scale.

To this end we use a grayscale image of the original (with 64 levels). Its wavelet decomposition using the Daubechies-6 wavelet family type is computed and features are determined by taking the maximum absolute deviation of the coefficients for the approximation and the levels of detail (horizontal, vertical and diagonal) at 4 different scales. The summarizing quantity takes a ratio of energy in the lower scales to energy in the higher scales.

A reasonable correlation with perceived overall complexity was obtained by taking a weighted sum of this quantity and the total number of edges found by the Edge Histogram Descriptor (Section 2.2.2) and the number of colors in the image. The weights were chosen to optimize performance on an annotated test set.

2.2.4 Periodicity

Many images contain patterns which are periodic, i.e., patterns that are invariant to certain translations. In images we may have periodicity in one direction (so-called “friezes”) and periodicity in two independent directions (so-called “wallpapers”). The repeating elements can be extracted by means of autocorrelation analysis or Fourier approaches. See for instance [26,52].

General symmetries (i.e., rigid transformations that keep the pattern invariant) of the plane are compositions of translations, rotations and reflections. In [54] a detailed analysis of plane isometries and plane symmetry groups is presented. It is shown that for two dimensional designs there are 7 frieze groups describing patterns that repeat along one direction and 17 wallpaper groups for patterns that repeat along two linearly independent directions to tile the plane. The groups vary in which additional symmetry types are present. Liu et al. [28] provides a computational model for periodic pattern classification based on these groups. Using this model one can identify the symmetry group of the pattern and extract a representative motif.

For extracting the translation lattice, we have used an autocorrelation approach along the lines of Liu et al. [26]. The maxima of the autocorrelation function give a set of candidate lattice points. The task is then to find the shortest linearly independent translation vectors that generate this lattice. For the frieze patterns this is a single vector, for the wallpaper patterns two vectors are needed. Liu et al. [28] introduce a method based regions of dominance to robustly determine these translation vectors.

An additional method to investigate periodicity is by extracting the motifs by means of figure-ground segregation, followed by an analysis of their arrangement and variation (see Section 2.3.3).

2.3 Representation of Region Properties and Relations

Once design elements have been identified, more detailed characterization of designs becomes possible. Procedures to determine such elements are discussed in Section 3.

2.3.1 Region Properties

For a given connected image region, representing for instance a design motif, we may compute various elementary properties. We mention for example: size (e.g., relative area, extent, equivalent diameter, perimeter, length of major and minor axis); orientation (based on for instance bounding box or fitted ellipse); eccentricity (or elongation, circularity, rectangularity); convexity (or solidity, compactness); color and texture; central and invariant moments; curvature (e.g., total absolute curvature, bending energy); contrast with ground; number of holes; fractal dimension (rate at which the perimeter of an object increases as the measurement scale is reduced). For definitions and formulas of these properties we refer to elementary textbooks on image processing such as [52] and [43]. When a design consists of only one type of motif, we may use these quantities directly as features. In case there are different types of motifs, we detect common properties and properties that show variation.

Of course, similar properties may also be computed for the design ground. Particularly interesting here is ground texture that may correlate with various kind of textile treatments, such as batik, chiné, dots and fine lines. To this end we may use the general texture features discussed in Section 2.2.2.

2.3.2 Shape

Many of the region properties discussed before are related to region shape. The property of shape is very active subject of research and warrants some further discussion. Next to the mentioned simple properties, various shape similarity metrics with associated feature space have been developed. See for instance [63] and [3] for an overview.

We have used the MPEG-7 visual shape descriptors (see [3]):

- The Region Shape Descriptor (RSD), which quantifies the pixel distribution within a 2-D object or region. It is based on both boundary and internal pictures, and it can describe complex objects consisting of multiple disconnected regions as well as simple objects with or without holes. It uses a complex angular radial transform (ART) defined on a unit disk in polar coordinates.
- The Contour Shape Descriptor (CSD), which is based on the curvature scale space (CSS) representation of the contour.

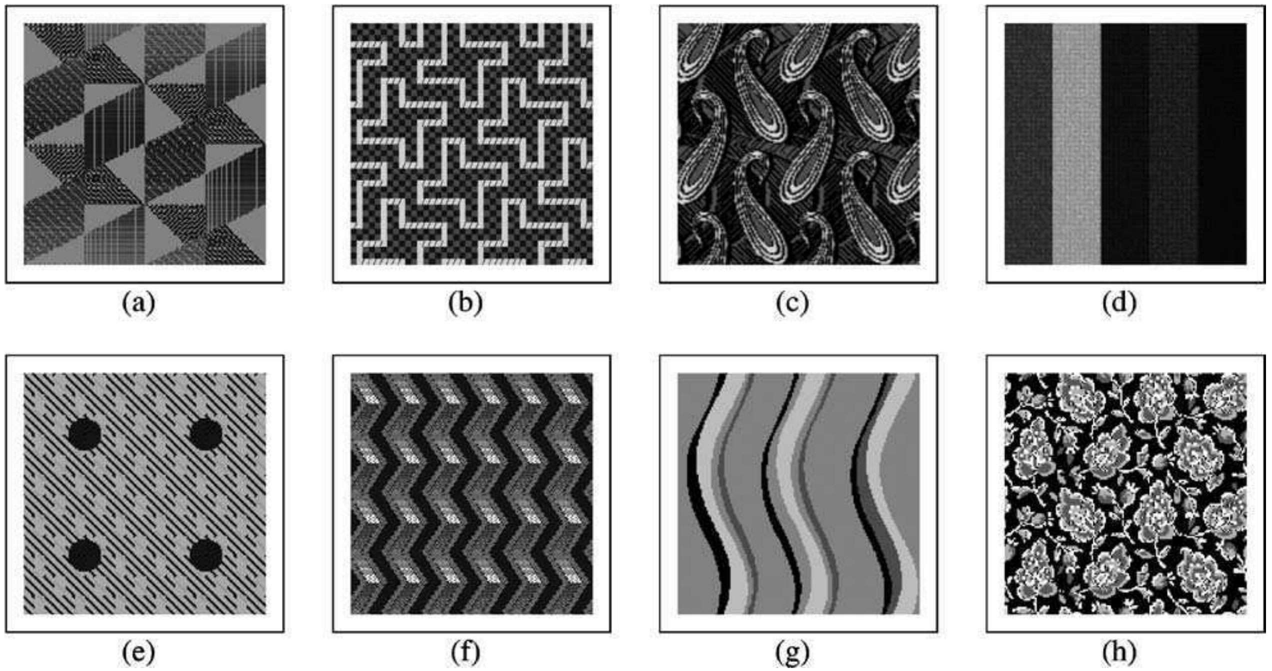


FIG. 8. Examples of shape categories important for designs: (a) diamonds, (b) pied-de-poule, (c) paisley, (d) stripe, (e) pois (circles), (f) chevron, (g) movement (waves), (h) leaves. (See Color Plate Section, Plate 22.)

Based on the shape descriptors we can construct shape features measuring membership to certain shape categories. A few example categories are shown in Figure 8.

For the category modeling we have taken a case-based approach using exemplars, which is a convenient method to develop useful features from high-dimensional feature spaces.

For each shape category we have taken a number of representative cases (e.g., see Figure 9(a)). These are called the exemplars; the shape category feature a given shape is then based on the minimum distance to the exemplars shape. Figure 9(b) shows distances for a few shapes to the paisley exemplar of Figure 9(a) based on the contour shape descriptor.

2.3.3 Object Variation and Spatial Organization

Once the salient elements in an image have been determined it becomes interesting not only to look at their intrinsic properties, but also to consider the relations between such elements. In particular, we are interested in the similarities and differences between the various elements, and in their spatial organization.

Variation between the intrinsic properties can be measured directly, e.g., by using the median absolute deviation (MAD). Using such features we can detect, for instance, if the motifs in a design possess similar shape but differ in orientation.

Further features are based on the distances between the elements (design spacing) and measure if and how they are organized in a grid pattern.

3. Detection of Salient Design Image Elements by Figure-Ground Segregation

As mentioned the performance of the CBIR system relies to an important extent on the quality of the chosen design representations. Additionally we have seen that for meaningful searching and browsing through design collections higher-level characterizations based on the individual elements in the design are essential. Only then, if such elements can be identified, it becomes feasible to quantify visual properties such as shape, spatial pattern and organization, and variation in for instance color, shape and orientation among the elements.

In an important subset of the decoration designs the identification of individual design elements can take place by figure-ground segregation. The definition of the figure-ground segregation problem is, however, by no means trivial. Design elements may be arranged in a large variety of ways: they may be overlaid, may fade over into each other, or may form tilings of the image plane. Furthermore the elements may

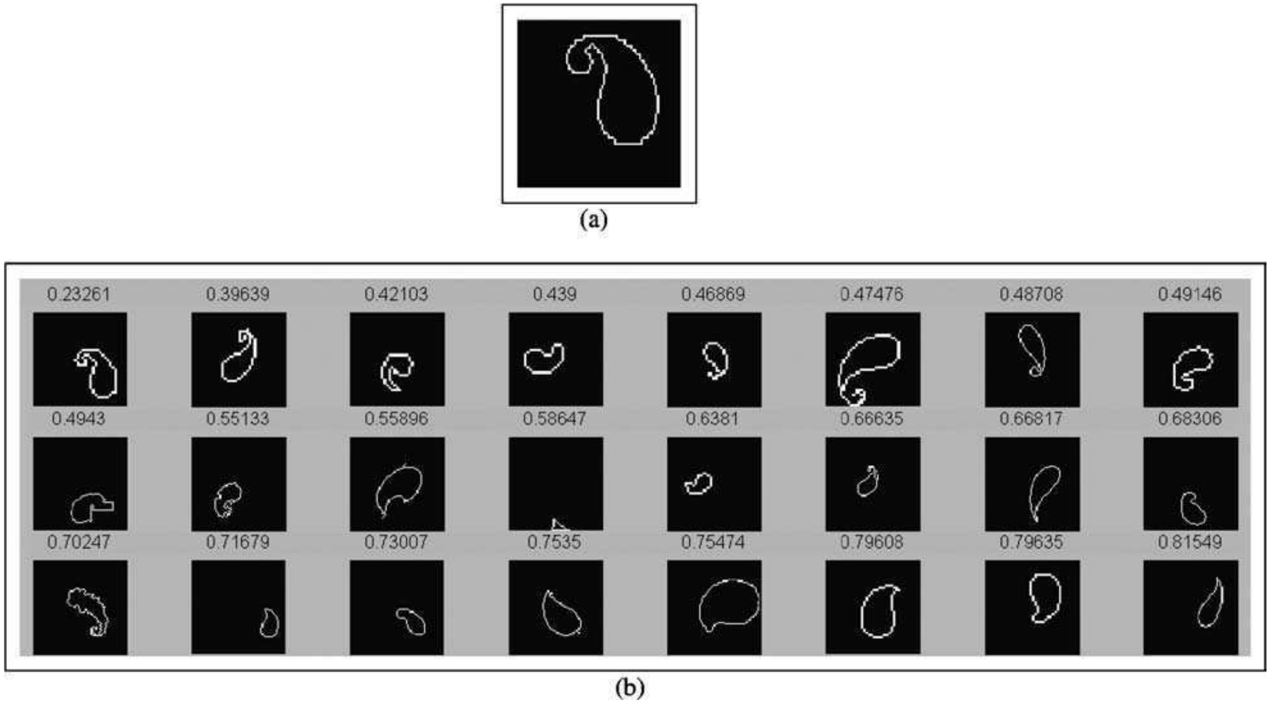


FIG. 9. (a) Exemplar contour for the paisley shape; (b) Contour distance to the paisley exemplar shape as measured by the MPEG-7 Contour Shape Descriptor.

take part in many types of spatial patterns and groupings. Within such arrangements the design elements vary in their level of *salience*, i.e., by the extent to which 'they stand out.' For figure-ground segregation we are interested in those cases where design elements are arranged on a ground, i.e., the case where a number of, usually isolated, salient elements stand out on a non-salient ground. Clearly not all designs possess such ground structure, see for instance Figure 1(a).

Occurrence of ground structure is often not clear due to the situation that the ordering among the design elements in terms of their salience is not clear. In other cases several figure-ground interpretations are possible simultaneously. An example is shown in Figure 10(c). The image can be interpreted as black on white, or as white on black. In some cases the occurrence of ground structure is clear, but it is still hard to determine the ground accurately (Figure 10(d)). The latter two effects are referred to as ground *instability*.

As a final issue we mention the occurrence of nested grounds. An example to this effect is shown in Figure 11(a). The image can be interpreted to consist of three layers: a plain green layer, a layer of heart motifs and four angels. The background can thus be either the plain layer, or this layer together with the hearts. A special case of this problem occurs in relation to designs consisting entirely of texture where the entire image may be taken to consist of background. In such cases it is often still useful to analyze the texture additionally in terms of its figure-ground structure, see Figure 11(b).

In the following we give a concise description of a method for figure-ground segregation based on the identification of salient color-patterns, the color coalitions, which is described in more detail in [19].

Patterns of color and regions of color texture play an important role in the visual structure of decoration designs. Consequently many pixels in design images can be naturally interpreted to take part in various color combinations. As an example consider the design image of Figure 12(a). The background in this image consists of pixels of two colors: red and black. Rather than viewing such pixels as either red or black it is more natural to view both types of pixels as part of a red-and-black region. Moreover, pixels are often part of a nested sequence of such color combinations. This may be seen by repeating the original design to arrive at the image of Figure 12(b). The original background pixels are now part of a larger pattern also including the so-called pied-de-poule motifs, which are yellow. Depending upon the scale at which a design is perceived the red and black pixels may thus also take part in a red–black–yellow combination.

We have set out to explore methods for color texture segmentation by direct analysis of the color combinations occurring in an image, i.e., we intend to find the natural color combinations which we shall then call color coalitions.

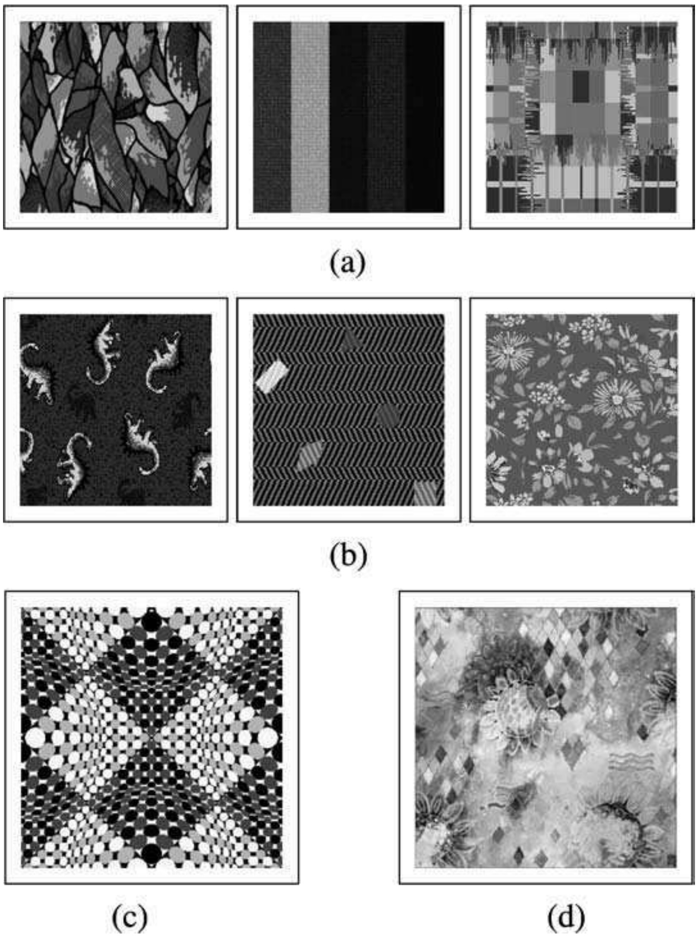


FIG. 10. Examples of design images: (a) images without background; (b) images with background; (c) and (d) images for which the background-foreground structure is unclear. (See Color Plate Section, Plate 23.)

In Section 3.1 we introduce the color coalition labeling and describe its application to the detection of regions of color texture at various scales. Next we discuss its application to figure-ground segregation. We propose a strategy consisting of three main steps:

(1) Obtain initial candidates for the background by multi-scale detection of color texture regions (Section 3.1).

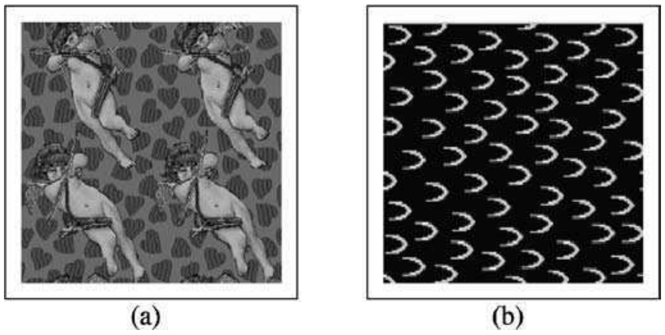


FIG. 11. Examples of design images for which the figure-ground structure consists of multiple levels. (See Color Plate Section, Plate 24.)

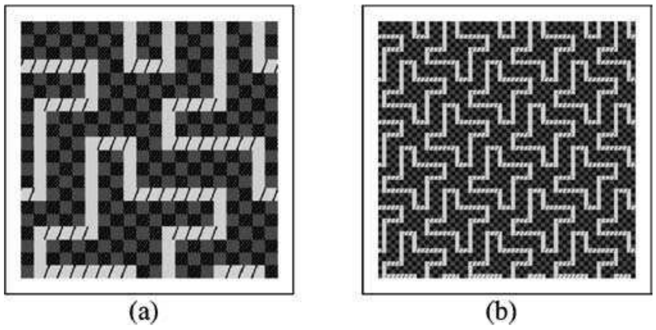


FIG. 12. (a) An example design image. (b) The same design image repeated 9 times and resized to its original size. (See Color Plate Section, Plate 25.)

(2) Assess the appropriateness of the individual candidates by an N-nearest neighbor classification algorithm based on visual cues such as relative size, connectedness and massiveness (Section 3.2).
(3) Integrate the results of the previous steps to produce a hierarchical description of the figure-ground structure of the design (Section 3.3).

The algorithms are tested by application to images from two decoration design databases. One is a database of tie designs from an Italian designer company. The other database has been provided by a manufacturer of CAD/CAM systems for the textile industry and includes a wide range of decoration design types.

In Section 3.4 we list the results obtained for the two test sets and discuss the general performance of the approach.

3.1 Detection of Color Coalitions

The task of finding color texture regions is closely related to general image segmentation. As any segmentation problem it is about the grouping of pixels that, in some sense, belong together. The approach we take here is based on direct analysis of the color combinations occurring in the image. As the level of homogeneity of a region depends on the scale under consideration, we must investigate the occurrence of color combinations at various scales. For each scale we then define a color coalition labeling that provides each pixel with a label uniquely identifying the colors occurring in a structuring element around the pixel.

We restrict ourselves to generate candidate regions for the image background and will not attempt a full bottom-up segmentation here. Moreover unlike in most segmentation methods we will not demand texture regions to be connected, nor will we attempt to assign every pixel to a segment region.

The algorithm for the construction of color texture regions for a fixed scale is divided in the following main stages:

(1) construct color coalition labeling;
(2) erode label image and analyze homogeneity of remaining color combinations;
(3) grow the resulting color coalitions into color texture regions.

These stages are outlined in Figure 13 and will be further detailed below.

3.1.1 Color Coalition Labeling

In the following we consider indexed images where each pixel has an associated integer value that either refers to a color in a colormap or is equal to zero, indicating that the color of the pixel is to be ignored by the algorithm. More formally, we define an image f as a mapping of a subset $\mathcal{D}_f$ of the discrete space $\mathbb{Z}^2$, called the definition domain of the image, into the set of indices:

$$f : \mathcal{D}_f \subset \mathbb{Z}^2 \to \{0\} \cup \mathcal{C}_f = \{0, 1, \dots, N\}, \tag{1}$$

where $\mathcal{C}_f = \{1, \dots, N\}$ is the set of color indices of the image. In practice the definition domain is usually a rectangular frame referred to as the image plane of pixels.

For indexed images we define the *index* or *color set* $\mathrm{cs}_c(f)$ of index c as the set of pixels with index c: $\mathrm{cs}_c(f) = \{x \mid f(x) = c\}$, or as binary image:

$$\left[\mathrm{cs}_c(f)\right](x) = \begin{cases} 1 & \text{if } f(x) = c, \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

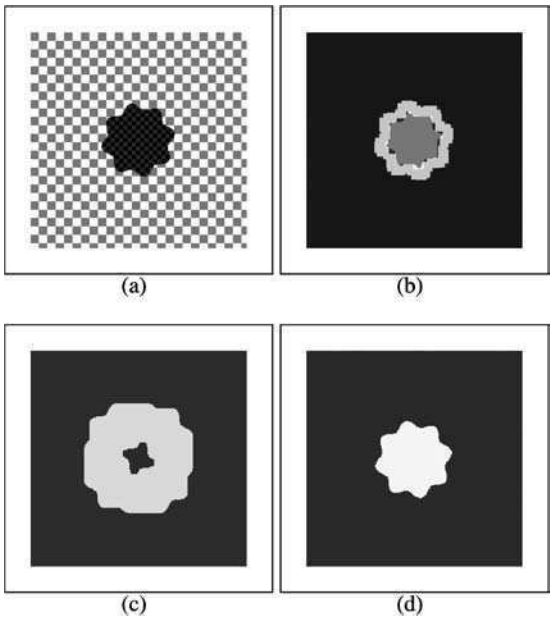


FIG. 13. Main stages in construction of color texture regions: (a) test image of 256 by 256 pixels consisting of two regions of color texture; (b) based on a rectangular window of 13 by 13 pixels structuring element the image has 7 distinct color sets; (c) after erosion and homogeneity checking two color sets remain; white pixels in this image have index 0 and do not correspond to a color combination; (d) growing leads to two regions of color texture. (See Color Plate Section, Plate 26.)

We further define the erosion of an indexed image as the summation of binary erosions performed on the individual color sets while keeping the original indices:

$$\varepsilon_B(f) = \sum_c c\varepsilon_B\big(\mathrm{cs}_c(f)\big), \tag{3}$$

where B is the structuring element and summation and scalar multiplication are pixel-wise.

For each pixel x we consider the set of colors $\omega_{B_s}(x)$ occurring in a structuring element B_s^x of scale s:

$$\omega_{B_s}(x) = \{c \in \mathcal{C}_f \mid \exists y \in B_s^x\colon\ f(y) = c\}. \tag{4}$$

Each such subset of $\mathcal{C}_f$ is referred to as a *color combination*, and $\omega_{B_s}(x)$ is called the color combination associated with pixel x at scale s. For the structuring element we will usually take a rectangular window with the centre pixel as element origin.

We define the color coalition labeling of f as follows. Let Ω_{B_s} be the set of all color combinations occurring in the image at scale s, then we associate with each combination ω in Ω_{B_s} a label $\lambda_{B_s}(\omega)$ in the order of encounter of such combinations in a forward scan of the image. The color coalition labeling $\Lambda_{B_s}(f)$ of f is then defined by

$$\big[\Lambda_{B_s}(f)\big](x) = \lambda_{B_s}\big(\omega_{B_s}(x)\big). \tag{5}$$

An example of a color coalition labeling is shown in Figure 13(b).

3.1.2 Color Coalition Selection

Our aim is to select the principal color combinations of the image, i.e., those color combinations that are most appropriate to extend to full color texture regions for a given scale. To this end we erode each of the index sets of the color coalition labeling under the tentative assumption that the color combinations occurring at the boundaries of regions of color texture are generally thinner than the interiors of such regions. For the erosion we use a structuring element B_t of scale t, i.e., we construct $\varepsilon_{B_t}(\Lambda_{B_s}(f))$. We denote an eroded set associated with ω by $R(\omega)$, i.e., we take

$$R(\omega) = \varepsilon_{B_t}(\mathrm{cs}_{\lambda_{B_s}(\omega)}). \tag{6}$$

As we are interested in finding regions of homogeneous color texture we further investigate homogeneity statistics for color combinations ω for which $R(\omega)$ is non-empty. Note that if statistics are computed based on a structuring element of scale s, taking $t \geqslant s$ ensures that colors surrounding a region of color texture cannot affect the homogeneity of the statistics in an eroded color set.

So let $S_{B_s}(x)$ be the local statistics at pixel x taken over pixels in a structuring element B_s, and consider a surviving color combination ω: $R(\omega) \neq \emptyset$. We accept ω as a color coalition if the following two conditions hold:

(1) $R(\omega)$ still contains all colors of the color combination ω.
(2) The coefficients of variation of S_{B_s} on $R(\omega)$ are smaller than a given threshold.

Both the choice of statistics and of the scale for the erosion structuring element t are subject to a trade-off between the aims of suppression of boundary color combinations and still being able to detect color texture regions that have a relatively large interior[1] scale relative to their exterior scale. We obtained best results by using the erosion as the main mechanism in reducing the number of candidate color combinations (we set $t = 1.5s$), and kept the statistics as simple as possible to allow for maximum detection of color textures. In fact, we take only the relative number of

[1] We define the interior scale of a set as the smallest scale at which a set is homogeneous; the exterior scale as the smallest scale at which the erosion of the set is empty.

pixels of the dominant color in the structuring element as a statistic. The computation of the coalition labeling and the local statistic can both be implemented taking a single forward scan and a moving histograms approach (see [61]).

3.1.3 Region Growing Strategies

Next the color texture regions associated with the principal color combinations are determined by region growing of the eroded color sets. If we denote the final color texture region by $G(\omega) = \mathcal{G}(R(\omega))$ then for a pixel x to be assigned to $G(\omega)$ it should satisfy at least the following conditions:

(1) The pixel must have a color index belonging to the color combination: $f(x) \in \omega$.
(2) the pixel must have the remaining colors of the color combination in its structuring element: $\omega \subset \omega_{B_s}(x)$.

The pixels in $R(\omega)$ satisfy both conditions; also note that the conditions allow pixels at boundaries of texture regions to have additional colors in their structuring element.

This still leaves the important issue of how to assign pixels for which more than one color combination is feasible. Several strategies are possible such as assigning to the closest or the largest feasible eroded set. In our application we have obtained best results so far by assigning to the color combination for which the associated eroded region has an average color histogram that is closest to the color histogram of the structuring element of the pixel.

For each scale we thus get a segmentation of the image in regions corresponding to the principal color combinations and a set of pixels with label zero that are not assigned to any color combination.

3.2 Classification

To determine the background quality of color texture regions, we take a simple yet effective approach based on weighted N-nearest neighbor classification. Based on a number of property variables or features of the region the *ground probability* is estimated that the region is suitable to serve as a background region.

Classification takes place by using a training set of sample regions with features x_i, $i = 1, \ldots, n$, that have been assigned a ground probability $p(x_i)$ by manual annotation. The probability $p(x)$ of a region with features x is determined by taking a weighted average of the probabilities of its N nearest neighbors in feature space, see for instance [9].

The feature variables were chosen by experimentation with the aim of reaching a high level of consistency and a low level of ambiguity:

- Relative area: the region area relative to the total image area.
- Filling coefficient: background regions often possess a complement consisting of components that are not connected to the border and which are removed after filling the background region (see for instance [56] for morphological operations such as hole removal). Let X be the background region, X^c its complement and $\bar{X}$ the background region after hole removal, then the filling coefficient $\mathrm{fc}(X)$ is defined as

$$\mathrm{fc}(X) = \begin{cases} 1 - A([\bar{X}]^c)/A(X^c) & \text{if } X^c \neq \emptyset, \\ 1 & \text{if } X^c = \emptyset, \end{cases} \tag{7}$$

  where $A(X)$ is the area in pixels of region X.
- Spatial reach: measures if the region occurs only in certain parts of the image or all over the image; the image is covered by a grid of boxes and spatial reach is measured by counting the relative number of boxes that are occupied by the region.
- Connectedness: the area of the largest connected component of the region relative to the region area (computed after closing with a small structuring element).
- Massiveness: the median distance of the region pixels to the region boundary.

The N-nearest neighbor approach allows for straightforward evaluation of inconsistencies and ambiguities. Consistency of the samples can be analyzed by comparing the ground probability of a sample region obtained by classification leaving that example out to the probability obtained by manual annotation. Let p_{-i} be the ground probability obtained by classification using all samples except sample i, then we define the consistency for sample i as $\rho_i = |p_{-i} - p(x_i)|$. In our study we took 350 samples of which only 8 had a consistency smaller than 0.75. It is also simple to assess if examples are sufficiently nearby for reliable classification: for instance by comparing distances of examples to new cases to be classified to the average of such distances occurring for the samples in the sample set. If relatively empty regions or problem cases are encountered additional samples may be added.

3.3 Synthesis

Using the color coalition labeling approach of Section 3.1 we obtain color coalitions for a sequence of scales. In this study we took 8 scales that were equally distributed over a range from a smallest window of 3 by 3 pixels to a rectangular window of about 30% of the image size. All resulting regions were classified using the method of Section 3.2. Each region with a ground probability greater than 0.5 is accepted as a potential ground (although ground probabilities are generally found to be either 0.0 or 1.0). If a color combination is found to be feasible for serving as ground at more than one scale, we take two criteria into account to decide on the most appropriate region: (i) the simplicity of the region; (ii) the number of scales at which the particular region, or a region very similar to that region, is found (scale robustness). For the simplicity measure of the region we have taken, rather ad hoc, the sum of the number of connected regions in the background and the foreground (after opening each with a small structuring element; see [56]).

Next we further analyze the determined grounds and their associated color combinations. Every pair of combinations is assigned as either: nested, partially overlapping or disjoint. Large disjoint regions often indicate flipping behavior as in Figure 10(c). Apart from the analysis of such relations that also includes checking the hypothesis that the entire image consists of a single color texture, for every design a highest quality background is determined using the simplicity and robustness criteria. Based on these results each of the images is automatically assigned to one of four distinct categories or *streams*:

- I: no figure-ground structure;
- II: figure-ground structure;
- III: consists entirely of one color texture, which itself possesses figure-ground structure;
- IV: consists entirely of one color texture, and does not possess further figure-ground structure.

Note that such automatic stream assignment allows for data driven feature computations. For example texture features can be computed for the background regions and full texture images, whereas shape features are computed for foreground elements.

3.4 Results

Benchmarking figure-ground segregation algorithms for designs is generally difficult for the reasons sketched in the introduction: even for humans identification of figure-ground structure is often not unambiguous. We thus choose to restrict ourselves to cases where we clearly have a ground or we clearly do not, and check if the algorithm output aligns with human perception for the cases where occurrence of figure-ground structure is clear and stable. This approach recognizes the notion that strict bottom-up processing is generally infeasible, unless some sort of context is assumed: in this case we assume that we are dealing with images where a ground is to be identified.

For testing we take three sets of images:

(i) Collection 1: 500 images from a specialized database of tie designs, used for training the background detection algorithms;
(ii) Collection 2: another 500 images from the same database;
(iii) Collection 3: 500 images from a database containing a wide range of decoration designs from the textile industry.

As such this database provides a representative test set for images the algorithm is likely to encounter in practice. The images of Collections 2 and 3 have not been used in any way to calibrate the algorithms.

We assigned each of the images in the test sets by manual annotation to either one of the four streams discussed in Section 3.3 or to stream

V: occurrence of structure not clear or instable.

Rates of correct performance are reported in Table I.

Example images from stream II with results are shown in Figure 14. Errors can largely be attributed to the following types of designs:

TABLE I
CORRECT PERFORMANCE RATES FOR IMAGES ASSIGNED BY MANUAL ANNOTATION TO STREAM I THROUGH IV

Collection	Stream I	Stream II	Stream III	Stream IV
1	89%	90%	85%	82%
2	92%	91%	77%	90%
3	100%	88%	78%	81%

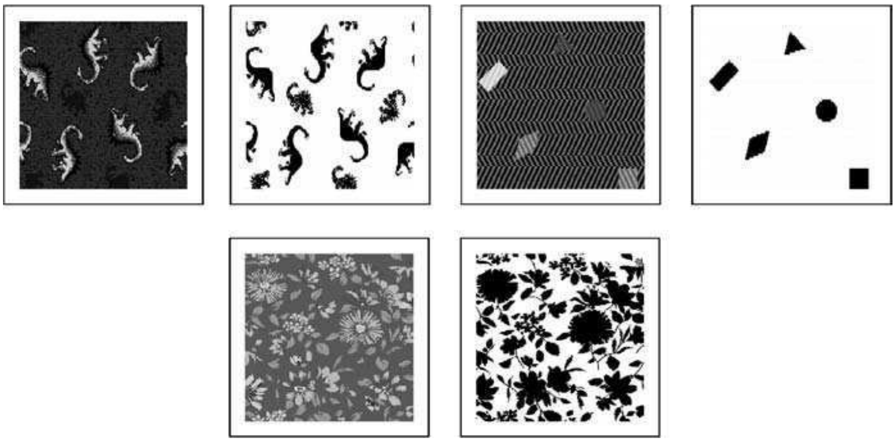


FIG. 14. Results of the figure-ground segregation algorithm for the images of Figure 10(b). (See Color Plate Section, Plate 27.)

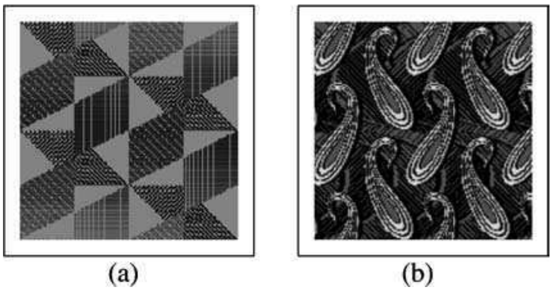


FIG. 15. Examples of design images with (main) elements that cannot be found by figure-ground segregation. (See Color Plate Section, Plate 28.)

- Designs where all colors in the foreground object also occur in the background, and the foreground objects do not disrupt the homogeneity of the background region. An example is shown in Figure 15(a). Other methods must be used to find such additional structure in the ground.
- Cases where the background consists of a composition of regions, see for instance Figure 15(b). Currently no combinations of regions are tested for their suitability to serve as ground.
- Cases for which classification is ambiguous, e.g., in images for which the background consists of small isolated patches, that by their shape and layout would rather be expected to be of foreground type. This type of background is hard to detect automatically and generally requires a higher level of design understanding.
- Cases where the choice of simplicity measure leads inappropriate candidates to be accepted. Occurs very rarely.
- Designs with illumination effects, gradients and special types of noise. Main problem here is the occurrence of noise that is not removed by preprocessing and occurs in only part of a color texture.
- Designs where the interior scale of a background region is large in comparison to its exterior scale. Sometimes the region is not found as a candidate since the color combination region disappears by erosion before it is accepted as homogeneous.

Correct performance is directly related to the occurrence of such types of images in the database. For example the mistakes for Collection 3 are mainly of the first type as the set has a relatively high number of binary images with additional fine structure.

The general conclusion is that the method discussed here works well for a large class of decoration design images. There is, however, a class of designs without figure-ground structure that still contain interesting elements relevant to design retrieval. Images in this class include for example geometric mosaics and designs with overlapping elements. Two few examples are shown in Figure 15.

Detection of such objects is possible to some extent with general segmentation methods [13,39,47]. These methods lead to partitions of the images in homogeneous regions such that the unions of such regions are not homogeneous. However, we have found that individual segmentation methods are often not able to deliver the regions that are of interest from the design interpretation perspective. Among other reasons, this is due to the intricate interplay of the Gestalt grouping principles (see, e.g., similarity, proximity, goodness-of-curve [64]). Currently we are working towards increased robustness by exploiting both redundancy (i.e., using results of complementary segmentation approaches) and non-accidentality (i.e., by detecting an unexpectedly high degree of ordering in terms of the Gestalt principles).

4. MPEG-7 Description of Design Images

As we are working towards “content-based” image retrieval it is natural to investigate the potential of the MPEG-7 metadata system for content description.

MPEG-7, formally named “Multimedia Content Description Interface” [35], is an ISO/IEC standard for describing various types of multimedia information developed by MPEG (Moving Picture Experts Group). Whereas the MPEG-1 through MPEG-4 standards are aimed at representing the content itself, MPEG-7 represents information about the content: “the bits about the bits” so to speak.

There are many types of audiovisual data content that may have associated MPEG-7 descriptions. These include: still pictures, graphics, 3D models, audio, speech, video, and composition information about how these elements are combined in a multimedia presentation (scenarios). The following is based on [20] and will focus on those parts of the standards that are of interest in the context of retrieval of still pictures in general, and of decoration design images in particular.

In these contexts we found that the main purpose of use can be summarized as twofold:

(1) Organization of image description data: as discussed in Section 2 many types of design features are computed; in this process a large variety of data is generated, consisting not only of the numerical values of the features, but also of intermediate results such as image segmentations for which the computation may be time-consuming and which are therefore worth storing. MPEG-7 descriptions can be used to organize all this data in relation to the structure of the image and to provide bookkeeping with respect to, for instance, algorithm version information and chosen parameter settings.

(2) Structural and semantical image content description to facilitate the quick and efficient identification of interesting and relevant information

This will be explained in more detail below; additionally, we shortly discuss the potential of using the MPEG-7 standard in combination with ontology definitions.

4.1 Structure of MPEG-7 Descriptions

MPEG-7 descriptions are defined in terms of *descriptors* and *description schemes*.

A descriptor is a feature representation, i.e., the descriptor defines the syntax and the semantics for the representation of a perceptual feature. Descriptors may be both textual and non-textual and some can be computed automatically whereas others are typically obtained by manual annotation.

Description schemes (DSs) expand on the MPEG-7 descriptors by combining individual descriptors and other description schemes into more complex structures by defining the relationships between its components (which may be both descriptors and description schemes). Description schemes are defined following the MPEG-7 Description Definition Language (DDL), which can also be used to define new description schemes.

In practice one may think of an MPEG-7 description as a (possibly compressed) XML file, and of the description schemes as XML schemas (see for instance [15]). A description always consists of an MPEG-7 top element. It may contain either a partial description consisting of any MPEG-7 description unit desired for a particular application, or a description that is complete in the sense that it follows a pre-defined hierarchy of description units. For an overview of this hierarchy, see [53].

An illustrative snippet from an MPEG-7 description is shown in Figure 16.

4.2 Description Schemes

The following is a concise introduction to the various description schemes provided by the MPEG-7 standard that are most useful for decorative design description.

4.2.1 Content Management

MPEG-7 offers description schemes for creation information, usage information and media description. The *CreationInformation DS* provides functionality to describe the creation and production of the design, e.g., a title of the design, its creator, creation locations and dates. The *UsageInformation DS* describes information related to the usage rights management and protection, usage records and financial information.

```
<CreationInformation>
    <Creation>
        <Creator>
            <Role><Name xml:lang="en">Main designer</Name></Role>
            <Agent xsi:type="PersonType">
                <Name>
                    <GivenName>Mark</GivenName>
                </Name>
             </Agent>
         </Creator>
         <CreationCoordinates>
             <Location>
                <Name xml:lang="en">Amsterdam Design</Name>
                <Region>nl</Region>
             </Location>
             <Date>
                <TimePoint>2003-03-27</TimePoint>
             </Date>
         </CreationCoordinates>
    </Creation>
</CreationInformation>
```

FIG. 16. Snippet of an MPEG-7 description.

The *MediaInformation DS* describes the location, and coding and storage format of various possible instances of a described design, e.g., of the master copy and various compressed versions.

Additionally a *DescriptionMetadata DS* is available to provide metadata about the descriptors themselves. It allows for description of the creation of the description, e.g., which algorithm version is used, where is it located and on which parameter settings is it based.

4.2.2 Content Structure

The content structure description schemes allow for detailed description of the design structure in terms of its constituent regions and moreover provide a framework for organizing the various types of visual descriptors.

Regions or elements of a design are described by means of the *Segment DS*, or more in particular for design images by the *StillRegion DS*. Hierarchical descriptions are possible: segments may be further divided into sub-segments. Decompositions are described by means of the *SegmentDecomposition DS*. An example of the use of these schemes is shown in Figure 17. The original image is denoted by SR0. After preprocessing (e.g., resizing, color quantization) we obtain a simplified image, which we denote by SR1. This region may be decomposed in a striped (background) region

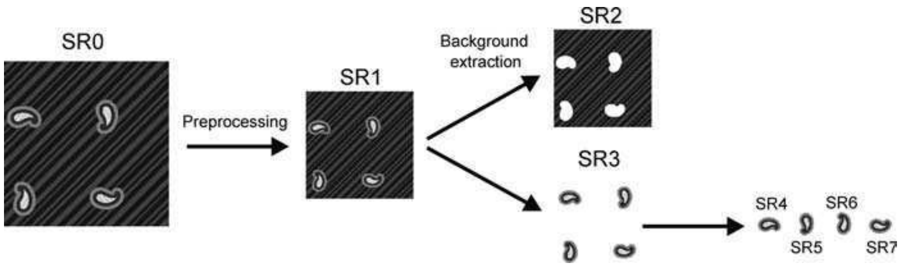


FIG. 17. Example of a design segment decomposition using the *StillRegion DS* and *SegmentDecomposition DS*. SR denotes a *StillRegion*, SD a *SegmentDecomposition*. (See Color Plate Section, Plate 29.)

SR2 and a region consisting of 4 little objects (SR3). If required, SR3 may be further divided into its connected components: SR4 to SR7.

Generally, segments are allowed to overlap, and the union of the segments is not required to cover the full design. For further characterization of the organization of the segments several schemes are available to describe relations between the design regions. As such relations constitute a subset of the semantic relations, they will be discussed in the next section.

Segments may have several types of attributes, most notably the visual descriptors regarding for instance color, shape and texture of the region. The segments and their decompositions thus provide a natural framework for structuring the organization of the various types of perceptual features. In the previous example features regarding the stripes would be associated with SR3, whereas shape information can be stored as attributes of the regions corresponding to the objects. Also color information can be stored for both the entire design and the several regions separately.

Another important type of segment attribute in the context of design structure is element saliency, i.e., the extent to which an elements in a design "stands out." A mechanism to this end is provided by means of the *MatchingHint* descriptors.

4.2.3 Content Semantics

Just as the content structure schemes provide a convenient framework for the low-level description of a design in terms of its regions, the content semantics description schemes provide a rich set of tools for high-level descriptions of a design in terms of its elements. The elements, the properties of the elements and the relationships between the elements are described by semantic entities, semantic attributes and semantic relations, respectively. Generally, close links exist between the regions described in the content structure schemes and the entities used in the semantic description, but the semantic content description allows various other abstract concepts to be defined as well.

Several schemes are available for the description of abstract entities of which the most relevant are the *Object*, *SemanticState* and *Concept* DSs, e.g., the *Object* DS describes perceivable semantic entities.

Semantic attributes are used to describe the semantic entities by means of labels, a textual definition, properties and for instance a link to a region in the actual design (a so-called *MediaOccurrence* attribute). Other attributes allow for the specification of the abstraction level of a given entity. This can be used for searching designs and will be discussed in more detail in the next section.

Semantic relation types are defined by means of *classification schemes* (CSs). These are provided by MPEG-7 for the definition of vocabularies for use inside descriptions. Several such schemes are already available: relations may describe for example how entities relate in a narrative (e.g., agent, patient, instrument, beneficiary), how their definitions relate to each other (e.g., generalizes, componentOf, propertyOf) or how the entities are organized with respect to spatial structure (e.g., above, left).

An example of a semantic content description is shown in Figure 18 for the decorative design of Figure 17. The decorative design is shown to consist of a background and a foreground entity. The background is associated with the abstract concept of 'striped-ness.' Two state entities further specify the quality of the stripes (their orientation, and their size). The foreground consists of a set of motifs. The type of motif set may be further clarified by for instance a variation state (indicating to what extent

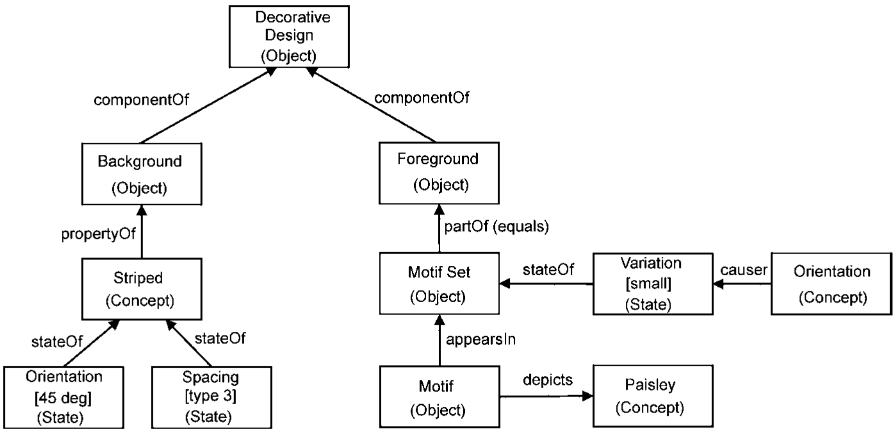


FIG. 18. Example of a semantic content description of the design shown in Figure 17. The boxes represent semantical entities, the arrows semantical relations. Attributes providing links to associated segments are not shown.

the motifs are varying within the set). The motifs that occur in the set, in this case so-called paisley motifs, may be described independently.

Many other schemes are available which have not been mentioned yet, for instance schemes for description of design collections, user interaction or usage histories. See [1] for an overview.

4.3 High-Level Design Description and Retrieval

As shown the MPEG-7 descriptors and content structure description scheme provide a convenient framework for organizing the design metadata obtained by the various design representation methods discussed in Section 2.

Ideally one would also like to be able to automatically generate high-level descriptions of the type discussed in the previous section as this level of content description is ultimately most meaningful for design retrieval. Design retrieval by semantic descriptions can then for instance take the form of listing all images that possess "at least one paisley motif," or of listing those images for which the background texture has a certain quality.

To define an efficient language for design description and query formulation the main challenge is to provide a set of well-founded constructs to describe the semantics of decorative designs. As we have already seen, entities should include

(i) objects such as foreground, motif set;
(ii) concepts, such as texture, variation, pattern types (e.g., halfdrop, horizontal), and style types (e.g., ethnic, tropical, skin);
(iii) states, e.g., variation, spacing, complexity.

Basic capabilities for such description of a vocabulary of terms, the relationships that can exist between terms and the properties that such terms may have, is provided by the Resource Description Framework (RDF) Schema language (RDF, [46]). For more extensive capabilities one may also use an ontology language such as Web Ontology Language (OWL, [38])

The specification of a decorative design ontology provides the opportunity to reason about the concepts defined. Consider for example the case that Geometric, Square and Circle are classes defined in a decorative design taxonomy, and that Square and Circle are both indicated as subclasses of the Geometric class. A search based on a semantic description containing a Geometric object can then be inferred to match descriptions containing Circle or Square objects. An additional use of ontologies is the support for the representation of common knowledge. An example of such knowledge would be that tropical designs typically contain a number of bright colors.

MPEG-7 provides abstraction mechanisms that support reasoning by ontology. The most relevant is formal abstraction which describes patterns that are common to a set of examples. Formal abstractions are created by taking a specific semantic description and by replacing one or more of the entities in the description by variables using the *AbstractionLevel* attribute. In the example above we could for instance replace the paisley concept by a Curved Shape entity of *AbstractionLevel* one (level zero denotes concrete instance; levels higher than one provide abstractions of abstractions). Such description would then match any design with a certain type of striped background for which the motifs exemplify curved shapes (and for which the motifs in the set are varying by orientation).

As a final opportunity we mention the definition of a classification scheme in combination with the *Affective DS*. This provides a means to describe the esthetic appreciation of designs and allows description of measurements of affective response to variables such as balance, ratio, juxtaposition, repetition, variation, pattern, rhythm/tempo, emphasis, contrast, harmony and unity.

We conclude that MPEG-7 semantic descriptions show great promise in organizing high-level design interpretations for meaningful searching, but must note that given the current state-of-the-art fully automatic description is possible only to a limited extent, primarily as reliable recognition of meaningful objects remains a task yet to be solved.

5. Inference and Learning for Relevance Feedback by Examples

5.1 Introduction

Content-based image retrieval revolves to an important extent around the task of *interactively* and *adaptively* reaching an understanding of what the user is looking for. As discussed in the introduction, using relevance feedback may help us deal in particular with the fact that interpretation of image content is user- and task-dependent. An overview of approaches is presented in [68]; in many cases substantial gains in retrieval performances through the use of relevance feedback have been reported (e.g., [7,8,32,51,67]).

In the following we focus on feedback in terms of example images. With this type of feedback the user is presented with a selection of images from the database; he indicates which images he considers relevant examples (positives) and which he considers to be counterexamples (negatives); next, a new selection of images based on the estimated relevance ranking is presented and the cycle may be repeated. This type of interaction is particularly natural for images: unlike for text documents, relevance of images can really be determined “at a glance.”

Many approaches to relevance feedback are based on adaptively re-weighting of feature dimensions, both for query point movement (e.g., [48,51]) and in similarity and relevance measures. In both cases feature variables or feature classes are assigned weights based on the feedback data. The weights should be chosen in accordance with the importance of the associated features to the user. For example, Rui et al. [51] update weights of different feature classes by using the inverse variance of the positive examples, thereby giving higher weights to features for which the positives are close to each other. Many variants of this idea have been developed (e.g., [7,42]) but generally are heuristic in nature: feature weights are assigned such that positives cluster, while negatives stay separated.

In [8] a framework for Bayesian interpretation of relevance feedback data is described. At the heart of this approach lies the probabilistic modeling of the feedback data given that the user has a target image in mind and a certain selection of images is available to the user to choose from. It is assumed there that the user will pick images based on their similarity to the target, but no explicit effort is taken to find out which features are most important in the similarity measure.

Many recent approaches treat the estimation of image relevance based on the relevance feedback by examples as a machine learning or classification problem. The feedback images are taken as training samples and used to train a classifier or other learner that can be used to predict the relevance of the database images. As we have seen, typically two classes or levels of relevance are assumed. Extensions to more levels are usually straightforward (e.g., [51]), but may incur the cost of a less natural interaction with the user. Examples of learning approaches to relevance feedback are: MacArthur et al. [29] (decision trees), Laaksonen et al. [23] (neural networks and self-organizing maps), Vasconcelos and Lippman [62] (Bayesian), Tong and Chang [60] (support vector machines), Wu and Manjunath [65] (nearest neighbors), Wu et al. [66] (linear discriminants) and Tieu and Viola [59] (boosting).

In the following we discuss the special structure of the relevance feedback learning problem that leads to difficulties for many of the methods mentioned earlier; we also describe a new approach which deals naturally with the fact that feedback images are typically selected based of a small set of salient properties. To be able to discuss this more clearly we first introduce the notion of *aspects*.

5.2 Aspect-Based Image Search

As we have seen in Section 2 features measure image quantities; some of these quantities will matter to image relevance and some will not (neutral features). When a feature matters we should find out which feature *values* influence relevance positively, and which negatively. Note that only for neutral features, any feature value has (approximately) the same effect on image relevance, i.e., no effect. For "relevant features," not only will there be feature values that lead to higher perceived relevance, but there must always also be feature values that make images *less* relevant.

In our approach we will not analyze the relevance of features as a whole, but rather the relevance of an image having feature values satisfying certain conditions or belonging to a certain set. We consider for instance the influence of "high complexity," where "high" is defined as a range of complexity feature values. We will refer to such derived binary features which model a specific perceptual quality, and which an image either has or has not, as aspects. To be more precise, we will understand an *aspect* as:

> a proposition with predicate in terms of a feature or set of features variables (which for a given image is either true false), for which we intend to resolve its effect on image relevance as a unit.

When the image features satisfy the aspect predicate we say the image *possesses*, or simply *has*, the aspect.

As mentioned, even though any constraint on feature values can be used, in practice the proposition will usually state that the image has a feature value in a certain natural range or interval. We will refer to such range or interval as the aspect *cell*. Such cells can be fixed beforehand, allowing for very fast feedback processing as we shall see later on, or be determined adaptively. The construction of aspects for different feature types is discussed in Section 5.3.

We believe that when a user is searching in an image database, he does so, consciously or unconsciously, in terms of aspects. The aspects he wishes the images to possess, and which make an image at least partially relevant, we call *relevance enhancing*, or simply *relevant*, aspects. Similarly we have *neutral* and *relevance inhibiting* aspects.

As an illustrative example, suppose a user is interested in finding designs that:

(i) have a blue background;
(ii) have simple round motifs that are relatively far apart; and
(iii) have a high contrast between motifs and ground.

Depending on the available features, we can translate this to requirements in terms of aspects. Some aspects are clearly relevant, e.g., the blue-ness of the ground should be high, dominant motif shape should be round, and relative amount of background should be high. Aspects that are in opposition to the relevant aspects are relevance inhibiting, e.g., the user does not want closely spaced motifs, a ground that is red or a dominant motif shape that is square. Additional inhibiting aspects may be discovered during the feedback process, e.g., a user may decide that he is in fact not interested in yellow motifs. Other aspects are neutral as the user does not care whether images possess these. For example we may not care about the pattern in the ground: it may be plain or have some texture.

In the following we discuss the implications of feedback examples being chosen based on *partial* relevance, i.e., based solely on one or a few *salient* aspects. To this end it will be useful to quantify the saliency or importance of an aspect by measuring how often, or rather how rarely, the aspect occurs in the database. In practice we will mainly need the fraction $p_{\text{db}}(a)$ of images in the database that possess a given aspect a, which in analogy to the information retrieval term of "document frequency" could also be referred to as the "image frequency" of the aspect. A natural definition of aspect saliency, in this narrow sense, can then be based directly on the definition of inverse document frequency (see, for example, [57]), giving the *inverse image frequency* $\text{iif}(a) = \log(1/p_{\text{db}}(a))$.

5.3 Aspects and Feature Types

More and more methods for image feature extraction become available and there is a correspondingly large heterogeneity in feature types. Several divisions of feature types can be made. Feature values may be continuous or discrete. Within the discrete, or categorical, features we have ordered (ordinal) and unordered features. An example of an ordered feature is a complexity feature divided into five levels, e.g., very simple, simple, medium, complex and very complex. An unordered feature is, for instance, "direction type" (horizontal, vertical, "diagonal grid" etc.). We may also have to deal with feature spaces. As mentioned the MPEG-7 standard defines a number of visual descriptors for color, texture and local shape, where each of such descriptors consists of a feature space and a predefined similarity metric. In some cases the similarity metric consists of a complicated algorithm, for instance in the case of the MPEG-7 contour shape descriptor based on a curvature scale space approach.

In many databases manually annotated labels are available, typically of the discrete type, that have high semantic value and can be used with high reliability. Such labels may be used in hybrid systems that allow for direct selection of image subsets, but it may also be useful to let the system determine their relevance based on feedback by examples. At the other end of the spectrum we may have features that hardly have any semantic interpretation at all; such features may for instance be obtained through dimension reduction techniques such as principal component analysis or the like. Features may also be learned through interaction with the user (see, for instance, [33]).

We finally observe that several features or feature spaces may be available for the characterization of more or less the same higher-order aspects, e.g., we may have several different implementations of color similarity. Which of these is most appropriate may vary and depend on the given situation.

For discrete feature types associated aspects naturally follow from the available levels. Also for single dimensional continuous features it is usually straightforward to quantize the feature, either automatically or by inspection, into a number of meaningful classes. High-dimensional feature spaces are the most interesting in this respect. Our preferred solution is to use derived features obtained by an exemplar or case-based approach. For instance, we may select a number of red example images, determine a prototype or set of prototypes (e.g., by means of LVQ, see [22]), and define a derived red-ness feature based on the distances to one or more prototypes. Another approach constructs data-driven aspects by mining for clusters of images in the given space. Aspects then follow from cluster membership.

Using the evidential aspect-based approach detailed below, we can treat all feature types in a unified way, and use the feedback data to establish which aspects are most promising in determining image relevance. More importantly however, it directly confronts a number of issues concerning the structure of the relevance feedback learning problem.

5.4 Special Structure of the Relevance Feedback Learning Problem

The number of training samples obtained as a result of the relevance feedback process is usually small, whereas the dimension of feature space is large (typically more than 100). This makes many of the standard learning methods unsuitable for the task of relevance prediction.

Apart from this difficulty many of the learning methods do not take the special structure of the relevance feedback problem into account. We mention three issues:

(i) feature value distributions are often highly skewed;
(ii) the selection of examples is usually based on partial relevance;
(iii) there is a lack of symmetry between positive and negative examples.

Features often have value distributions that are highly skewed. This is particularly the case for features, common for special purpose databases, measuring detailed salient properties. As examples one may think of binary features such as “contains-a-paisley,” “has-colored-stripes” or “is-a-tartan.” For many such features, the great majority of images will not possess the aspect thus leading to a highly skewed value distribution. Also, if we take a feature measuring yellow-ness, say divided into three classes: “no yellow,” “some yellow” and “very yellow,” then by far most of the database images will be in the first class, and very few will be in the last. In combination with the next issue, this skewness in the population distributions leads to a tendency for feedback data to be misleading.

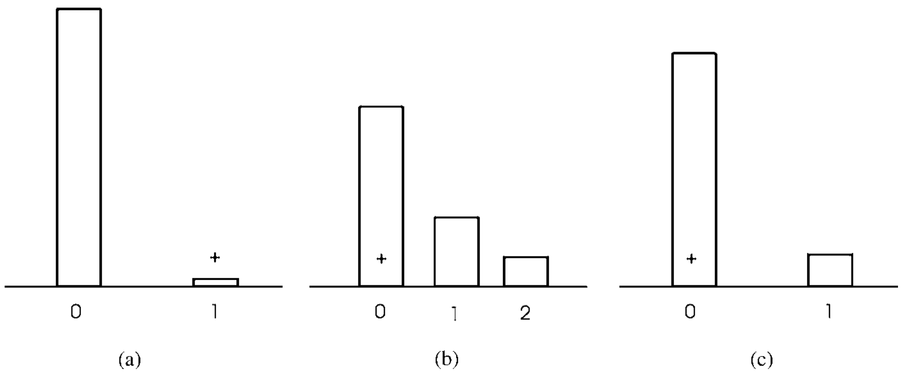


FIG. 19. Bars in the diagram indicate the population probabilities of feature values in the database for three distinct features a, b and c. The plus sign indicates the feature values (for a, b, c) for an image which is selected as an example because of $a = 1$ (i.e., for having a relevant aspect $a = 1$). We suppose b is a neutral feature, and c is a feature where $c = 1$ represents another relevant aspect, which in this case does not happen to be present in the feedback example. Since the selection is based the a-value, for both cases the b and c values of this example will be (approximately) random draws from the b and c populations. Because of the skewness they seem to favor $b = 0$ and $c = 0$ consistently, which is misleading in both cases.

When a user selects an image as feedback he generally does so based on *partial* relevance of the image. This means that he finds one or a few aspects in that image relevant; however, not all salient aspects present in the image need to be relevant, nor need all aspects of interest be present in the image. For features other than the ones by which the image was chosen, the feedback is more or less random: positive feedback is given for a certain value of the feature, where no such feedback was intended. Such examples will tend to cluster at those feature values that are most common in the database: this seems to indicate the user is interested in such values, whereas in fact he is not, thus interfering with the identification of the proper regions of relevance. See Figure 19.

For negative feedback the situation is similar. The user indicates a negative example based on one or more aspects he finds undesirable. Generally we may expect he will avoid selecting images with relevant aspects to some extent, but for neutral and other relevance inhibiting aspects the feedback information is, again, often misleading.

Considering all feedback images, we can expect to encounter situations as sketched in Figure 20. Different feature types are shown (binary, ordinal, continuous); for each case we show a situation where two of the example images possess a relevant aspect in the feature under study, whereas the other four are chosen based on other relevant aspects. Note that negatives counteract the misleading clustering of

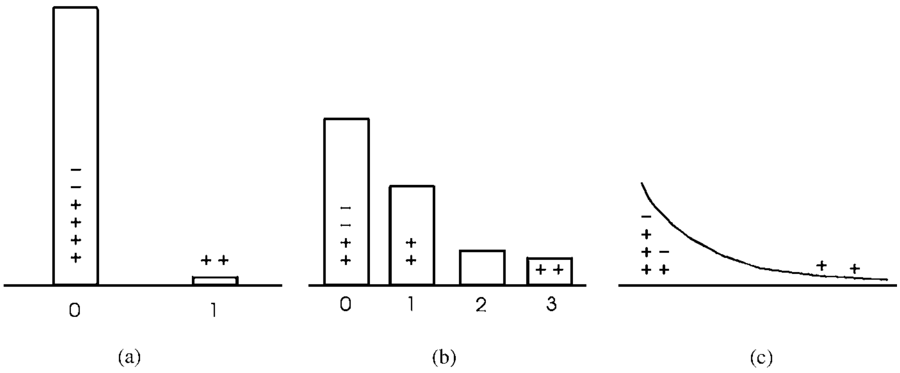


FIG. 20. Diagram shows examples of the population probability (mass/density) and potential feedback for (a) a binary feature; (b) a discrete feature; and (c) a continuous feature. For each case, two positive examples are chosen based on the feature shown; the remaining examples are chosen based on other aspects.

positives, but most learning methods will be influenced by the unintended concentration of positives.

A final issue is the lack of symmetry between positive and negative examples.Huang and Zhou [18] state as an intuition that “the positive examples are all good in the same way, but bad examples are bad in their own ways.” Though this statement may not be strictly correct in the light of the partial relevance issue discussed earlier, it is clear that the few selected negative examples are generally a very poor representation of the distribution of all negative examples. In our view no description of the class of negatives need be attempted, and examples should rather be used to purposely counteract slips of the system in seeking positive examples in misguided regions of feature space.

A useful distinction in aspects we can make in this context is that between *active* and *passive* aspects. Active aspects are aspects that the user uses explicitly in telling the system about his preferences and dislikes. What the user is looking for generally follows from a relatively small number of active enhancing aspects. Next to those there are also passive enhancing aspects, which are typically non-salient: if a user is looking for very yellow images then a “little-or-no-red” aspect would be enhancing but only in a passive sense. As follows from the lack of symmetry in positive and negative aspects, the number of inhibiting (salient) aspects is usually larger than the number of salient enhancing aspects. However, most inhibiting aspects tend to be passive. Which of the inhibiting aspects become active is not only be determined by explicit wishes of the user but also to some extent by chance, e.g., a group of red images shows up in the selection, at which the user decides to indicate he is not interested in this color.

In the following we will describe our approach to the interpretation of feedback data. We will use the feedback data first and foremost to establish which aspects matter most to the perceived relevance. For each available aspect we determine the likelihood ratio of the hypothesis that the aspect enhances relevance, relative to the hypothesis that relevance is independent of the aspect. Taking this approach has the benefit that relevance assignment is based not only on clustering behavior of positives and negatives, but is also compared to clustering behavior of random database images. This leads to a natural emphasis on salient aspects thereby solving the problems of partial relevance discussed earlier.

In addition, by taking into account the population distribution, we are not dependent on negative examples to down-weight positives that cluster at aspects with low saliency. This means negatives can be used to indicate which aspects are not desired, but are not required for the sole purpose of getting sufficient data for classification.

Finally, the use of the likelihood ratio for the evaluation of relative evidential support has a strong foundation in statistics, and allows for detailed modeling of the user's behavior in providing feedback. This, in turn, allows for inference solutions that are tailored to the needs of particular interaction schemes.

5.5 Measuring Evidential Support by Likelihood Ratios

In the following we are interested in analyzing the evidential support offered by data in comparing hypotheses. Rather than stating the problem in terms of the standard Neyman–Pearson approach in deciding between hypotheses using the likelihood ratio as a test statistic, we here emphasize a direct evidential interpretation of the likelihood ratio.

Royal [49] and Royal [50] make a strong case for the view that evidential support should not be measured for a hypothesis in isolation, but should preferably be considered relative to other hypotheses. The strength of such relative evidential support is quantified by the likelihood ratio. Hacking [17] states this so-called *Law of Likelihood* as follows:

> If hypothesis A implies that the probability that a random variable X takes the value x is $p_A(x)$, while hypothesis B implies that the probability is $p_B(x)$, then the observation $X = x$ is evidence supporting A over B if only if $p_A(x) > p_B(x)$, and the likelihood ratio, $p_A(x)/p_B(x)$, measures the strength of that evidence.

If we have a parameterized probability model for X with distributions indexed by a parameter θ, then an observation $X = x$ generates a likelihood function $L(\theta)$.

The law of likelihood then explains how to use this function: for any two parameter values θ_1 and θ_2, the ratio $L(\theta_1)/L(\theta_2)$ measures the strength of evidence, $X = x$ in support of θ_1 vis-à-vis θ_2.

In this article we propose to use feedback data (the positive and negative example images) in this way by comparing a number of hypotheses on the relation between a given aspect and image relevance. These hypotheses basically state either that an aspect is independent of image relevance (i.e., the aspect is neutral), or that the aspect is relevance enhancing or inhibiting in a certain way. Each of the hypotheses leads to a likelihood value for the feedback data. The law of likelihood quantifies the relative evidential support for any *pair* of hypotheses; in particular we will be interested if the maximum likelihood hypothesis stands out sufficiently from the alternative hypotheses.

An important question is how to interpret the likelihood ratio values. For instance, which ratios can be said to provide only weak evidence, and which strong evidence. As discussed in [49] there are various ways to develop a quantitative understanding of likelihood ratios.

The first is to compare ratios to ones obtained in canonical experiments where intuition is strong. As an example of such an experiment, suppose we have two identical urns, one containing only white balls, the other containing equal numbers of black and white balls. One urn is chosen and we draw a succession of balls from it, after each draw returning the ball to the urn and thoroughly mixing its contents. We then have two hypotheses about the contents of the urn, and the observations are the evidence.

Suppose we draw 3 balls in succession which are all white, then the likelihood ratio is $2^3 = 8$ favoring the all-white-ball hypothesis over the mixed-balls hypothesis. Similarly, if we draw n balls in succession which are all white, the likelihood ratio is 2^n. Of course there is no sharp transition between weak and strong evidence, but one may use such experiments to agree on certain benchmarks: Royall [49] proposes to use a likelihood ratio of 8 as representing “fairly strong” evidence and 32 as “strong” evidence favoring one hypothesis over the other. Note that a likelihood ratio of 32 corresponds to drawing 5 successive white balls in the canonical experiment. Similarly likelihood ratios around 1 represent “weak” evidence.

A second way of understanding likelihood ratios is by considering their effect in transforming prior into posterior odds ratios. We have

$$\frac{\Pr(A|X=x)}{\Pr(B|X=x)} = \frac{p_A(x)}{p_B(x)} \frac{\Pr(A)}{\Pr(B)}, \tag{8}$$

where $\Pr(A)$ is the prior probability that hypothesis A is true; $\Pr(A|X = x)$ the posterior, and $p_A(x)$ is the likelihood of the data x under hypothesis A.

So for each prior probability ratio $\frac{\Pr(A)}{\Pr(B)}$, the likelihood ratio $\frac{p_A(x)}{p_B(x)}$ tells us how the probability ratio changes after observing the data. For instance a likelihood ratio of 4 always produces a fourfold increase in the probability ratio.

Finally we mention a re-interpretation of likelihood values based on the observation that evidence may be *misleading*, i.e., it may happen that data represents strong evidence in favor of one hypothesis whereas, in fact, another hypothesis is true. Generally this cannot occur very often as is shown in [50] and below we will compute exact probabilities for events such as obtaining strong evidence for neutral aspects being enhancing. The expressions discussed there have strong links to standard levels of significance and hypothesis test power.

To summarize, we compare hypotheses based on their likelihood values; in the case of analyzing the partial relevance of aspects given the feedback data we compare hypotheses that the aspect is enhancing, neutral or inhibiting. Ideally we will be able to accurately model the probability of a stochastic variable representing the feedback data for each of the hypotheses, such that its measurement will provide strong evidence for one of the hypotheses most of the time. The design of such variables and models is not trivial, and will be taken up in the next section.

5.6 An Evidential Approach to Relevance Feedback by Examples

As explained, the basic setup in relevance feedback is such that a user selects images from the database to indicate his preferences and dislikes. Selection is facilitated by presenting the images in clickable selection screens each consisting of a grid of a fixed number of, say 30, thumbnail images. The number of images inspected may be larger as the user can leaf through the selection screens. Also additional selection screens may be available, for instance offering ‘informative images,’ see Section 5.6.4. The sequential ordering of the images is either random in the first cycle, or based on the relevance ranking in the subsequent cycles. The positive examples and (negative) counterexamples are collected in the collection box, consisting of the positive and negative *feedback image sets*.

At each cycle of the feedback process the user updates the examples in the feedback image sets by either:

(i) selecting new images as positive or negative examples adding them to their respective sets;

(ii) by removing images from the feedback image sets, i.e., the sets are preserved unless specific images are no longer deemed representative enough and are deleted explicitly.

As the user selects the example images based on partial relevance, i.e., based on one or a few enhancing or inhibiting aspects, it is natural to use the feedback data foremost to establish the relevance effect of the various aspects (i.e., as either enhancing, inhibiting or neutral). At the end of each cycle we thus analyze the relation between aspects and relevance by using models for the three types of effects.

The construction of the models will be based mainly on the following idea: as the user selects an image as feedback example based on one or a few enhancing or inhibiting aspects, possession of the remaining aspects will approximately follow the distribution (of aspect possession) in the database. Corresponding to the models, we formulate the following hypotheses to explain the occurrence of each of the aspects in the feedback images:

(1) the *independence*-hypothesis, or H_0-hypothesis: whether the image has the aspect is independent of whether the image is relevant;
(2) the *relevance enhancing*-hypotheses, denoted by H_K: possession of the aspect enhances image relevance;
(3) the *relevance inhibiting*-hypotheses, denoted by H_{-K}: possession of the aspect inhibits image relevance.

K denotes the number of images in the positive (enhancing case) or negative feedback image set (inhibiting case) for which the user's selection was directly influenced by the aspect under study. This means we will consider a sequence of hypotheses $\ldots, H_{-1}, H_0, H_1, H_2, \ldots$ where the actual number of hypotheses to be analyzed depends on the number of images in the feedback image set actually possessing the aspect under study. This will be explained in more detail below by discussing the models for each of the effects in turn.

5.6.1 Modeling Feedback Under the Independence Hypothesis

The H_0-hypothesis states that the relevance of the image and the aspect predicate are *independent*. In this case all feedback images have been chosen based on aspects other than the aspect under consideration, and this means that for all feedback images possession of this aspect will be distributed approximately as for a random image from the database. We thus model feedback image possession of the aspect as a Bernoulli variable with probability p_{db}, the fraction of images in the database which have the aspect.

In the following we will assume the number of positive and negative images selected to be given, and consider for each of these images if they possess the aspect or not, i.e., whether or not the associated feature values of the example images satisfy the aspect condition.

Let n^+ (n^-) be the total number of positive (negative) images selected, and N^+ (N^-) be the number of positives (negatives) that possess the aspect.

We then model both the number of positives and negatives occurring in the cell as binomial variables with probability parameter p_{db}:

$$N^+ \sim B(n^+, p_{\text{db}}) \quad \text{and} \quad N^- \sim B(n^-, p_{\text{db}}). \tag{9}$$

To see this, imagine a large urn containing a ball for each image of the database. White balls represent images which have the aspect, and black balls images which don't; the urn will then have a fraction p_{db} of white balls. If we now draw n^+ balls, the number of balls N^+ that is white is modeled well by the binomial distribution $B(n^+, p_{\text{db}})$.

The total probability mass function $p_0(x)$ for the feedback data $x = (N^+, N^-)$ is the product of the probabilities for N^+ and N^-.

5.6.2 Modeling Feedback Under Partial Relevance: Enhancing and Inhibiting Hypotheses

Relevance Enhancing Hypotheses. The total numbers of positives (n^+), negatives (n^-) are again taken to be fixed. Given that an aspect is relevant, we expect that a few, say $\widetilde{N}^+$, of the n^+ selected positive images have been chosen based to some extent on this aspect. As the remaining positives are selected because of other relevant aspects, their aspect possession will then be distributed similar as under the independence hypothesis.

The H_K hypothesis now states that the aspect is relevance enhancing, and that $\widetilde{N}^+ = K$ of the feedback images have been selected, at least partially, based on this aspect. The $(n^+ - \widetilde{N}^+)$ remaining positives are chosen independently from the aspect, so we have

$$(N^+ - K) \sim B(n^+ - K, p_{\text{db}}), \tag{10}$$

or

$$p(N^+|\widetilde{N}^+ = K) = \binom{n^+ - K}{N^+ - K} p_{\text{db}}^{(N^+-K)}(1 - p_{\text{db}})^{(n^+-N^+)}, \tag{11}$$

for $N^+ \geqslant K$, and $p(N^+|\widetilde{N}^+ = K) = 0$ for $N^+ < K$.

To obtain a model for the probability distribution of N^- we assume that negative examples have a probability to possess the aspect as under the independence hypothesis, i.e.,

$$N^- \sim B(n^-, p_{\text{db}}). \tag{12}$$

The total probability mass function $p_K(x)$ for the feedback data $x = (N^+, N^-)$ under hypothesis H_K is again the product of the probabilities for N^+ and N^-.

Relevance Inhibiting Hypotheses. The model for relevance inhibiting hypotheses is derived analogously to the enhancing case. For H_{-K} we assume that $\widetilde{N}^- = K$ of the negative images were chosen based on the aspect, giving that

$$(N^- - K) \sim B(n^- - K, p_{\text{db}}). \tag{13}$$

Assuming aspect possession for the positives as under independence leads to $p_{-K}(x)$.

5.6.3 Estimation of Relevance

Aspect Selection. Before we can estimate relevance of images in the database based on the feedback data obtained, we must first decide which aspects to take into account.

In our approach we take only those aspects for which either a relevance enhancing or a relevance inhibiting aspect is sufficiently well supported in comparison to the independence hypothesis. Here evidential support will be taken in the sense of Section 5.5, i.e., measured by means of the likelihood ratio of the respective hypotheses.

Let $p_0(x)$ be the likelihood of the feedback data under the independence hypothesis, and $p^+(x)$ and $p^-(x)$ the maximum likelihood values of the data under the enhancing and inhibiting hypotheses respectively, i.e.,

$$p^+(x) = \max_{K>0} p_K(x) \quad \text{and} \quad p^-(x) = \max_{K<0} p_K(x). \tag{14}$$

We take T to be our main decision threshold variable. If either $p^+(x)/p_0(x) \geqslant T$ or $p^-(x)/p_0(x) \geqslant T$ we accept the aspect as enhancing or inhibiting, respectively, i.e., we select such aspects to be taken into account in the relevance estimation. Note that the first likelihood ratio basically measures if the number of positives with the aspect is unexpectedly high, and the second ratio whether the number of negatives with the aspect is unexpectedly high. It may sometimes happen that both are the case; this special case will be further discussed below. Also note that the enhancing and inhibiting hypotheses model *active* aspects. Passive enhancing or inhibiting aspects will behave as neutral aspects and will thus, as intended, not be selected.

The threshold T can be directly interpreted in the sense described in Section 5.5. For instance if we have an aspect with image frequency $p_{\text{db}} = 0.5$ and take $T = 8$, one way to get the aspect to be accepted would be to select 3 out of 3 images with this aspect, i.e., by taking three positive examples each having the aspect. For more salient aspects fewer positives and a lower fraction of aspects having the aspect are required.

A more quantitative approach is to analyze probabilities of obtaining misleading evidence. In the following we discuss the comparison between neutral and enhancing aspects, but the same results apply to the neutral-inhibiting comparison.

We can compute the probability of accepting an aspect as enhancing, when the aspect is, in fact, neutral. To this end, we first note that for given (n^+, n^-) and p_{db}, each combination $X = (N^+, N^-)$ leads to a fixed likelihood ratio

$$\mathrm{LR}^+(X) = \frac{p^+(X)}{p_0(X)}. \tag{15}$$

Now assuming that an aspect is neutral means that we know the distribution of $X = (N^+, N^-)$, as $X \sim p_0$. The probability of obtaining misleading evidence indicating the aspect is enhancing for a threshold T is thus given by $\Pr(\mathrm{LR}(X) \geqslant T)$ given that X has distribution p_0. As an example, for an aspect with image frequency 0.1 (i.e., representing a quality that occurs once every 10 images), with $n^+ = 7$ we find that the probability of obtaining this type of misleading evidence is equal to 0.026 (for $T = 8$ and $T = 16$) and equal to 0.0027 for $T = 32$. This is, of course, a general pattern: when we increase T the probability of mistakenly deciding an aspect is enhancing decreases. However, there is a trade-off as increasing T also leads to an increased probability of obtaining evidence not sufficiently supporting an enhancing aspect to be enhancing. For instance, it turns out that the probability of mistakenly treating an enhancing aspect in this setup as neutral is zero for $T = 8$, or $T = 16$, but is equal to a large 0.64 for $T = 32$.

It is interesting to note that the Neyman–Pearson theorem [36] guarantees that for a given probability of falsely rejecting the neutral hypothesis, the decision based on the likelihood ratio has optimal power in deciding between the neutral hypothesis and any of the individual alternative hypotheses.

Further analysis has shown that:

- Very non-salient aspects (with $p_{\mathrm{db}} > 0.5$ say) are generally hard to discern from neutral, i.e., there is a large probability that evidence will be found for these aspects to be neutral even when they are not. These aspects might thus as well be ignored.
- $T = 8$ is a good general purpose value for the threshold.
- For very salient aspects (with $p_{\mathrm{db}} < 0.01$ say) it is possible to raise the thresholds, thereby reducing chances of mistaking neutral aspects for enhancing aspects, without running the risk of missing relevant aspects.

Of course we could also optimize the threshold for every given configuration in terms of (n^+, n^-), and p_{db}. Note however, that whether an aspect is enhancing does not correspond to a unique hypothesis. We thus need to make an additional assumption on which K-value or combination of K-values to base our distribution of X. For the experiment described above we have used the assumption that the X is distributed according to a K-value that is 2 higher than the expected number under the independence hypothesis. Using simulation studies we intend to further analyze this issue and also explore the effects of the aspect correlations mentioned below.

As a final issue it was mentioned that it may happen that both the enhancing and inhibiting hypotheses receive strong support relative to the independence hypothesis, giving a so-called *entangled* aspect. Two conflicting explanations for this state of affairs are possible:

(i) the user has strongly emphasized an aspect with his positive images without intending to do so (the aspect is neutral or inhibiting); as a reaction he has chosen an approximately equal fraction of negatives to undo this;

(ii) the user actually finds the aspect relevant, but has nevertheless selected a number of negative images with this aspect (for example because he had no other choice).

Note that this issue is not unique to our approach, but is a general problem in relevance feedback analysis. In our implementation we currently solve the problem using two strategies. The first is to demand not only sufficient support relative to the independence hypothesis but also relative to the opposing hypothesis. The second strategy is to use the direct feedback approach described in the introduction. This consists in presenting the user, at his request, with an overview of selected enhancing, inhibiting and entangled aspects. In this overview the user may confirm or reject the presented results. For each aspect he may indicate whether he considers it enhancing, inhibiting, or neutral. For enhancing and inhibiting aspects two levels of acceptance can be indicated: either selection in an AND-sense, or selection in an OR-sense. When an aspect is selected in an AND-sense this means that from then on only images will be considered (and shown) that possess that particular aspect, and images without it are no longer taken into account. The OR-sense means that the aspect is treated as any other aspect that is selected based on the statistical analysis: it will provide a contribution to the relevance as will be discussed below, but is not strictly required for an image to be considered relevant.

Relevance Estimation. For fixed aspect cells we use a matrix M with columns of boolean variables to indicate whether images have a given aspect or not. From M we compute p_{db} for each aspect as the ratio of ones in each column.

We can determine N_j^+ and N_j^- from the image index sets S^+ and S^- of positive and negative examples, using sums $\sum_{i=1}^{n^+} M(S_i^+, j)$ and $\sum_{i=1}^{n^-} M(S_i^-, j)$ respectively. Computing likelihood ratios LR_j^+ and LR_j^- for each aspect is a simple matter of substituting the values for N_j^+, N_j^- in the formulas for $p^+(x)$, $p^-(x)$ and $p_0(x)$. Also note that analyzing a number of enhancing and inhibiting hypotheses does not lead to any substantial additional computational expense, as we can compute associated likelihood values sequentially (for instance for positive K) by

$$p_K(x) = \frac{(N^+ - K + 1)}{(n^+ - K + 1)} \frac{p_{K-1}(x)}{p_{\text{db}}}, \quad 1 \leqslant K \leqslant N^+. \tag{16}$$

To obtain a prediction of image relevance to get a new selection of most relevant images we consider only the aspects that receive strongest support. Motivated by the results discussed above, we demand a likelihood ratio greater than 8. Let A^+ be the index set of accepted enhancing aspects, and A^- be the index set of accepted inhibiting aspects, then the predicted image relevance rel_i for image i is given by $\text{rel}_i = \sum_j M(i, A_j^+) - \sum_j M(i, A_j^-)$.

Note that, of course, the decision of taking into account an aspect need not so black-and-white; for instance we may down-weight the effect of aspects that show promise but for which the evidence is just not strong enough. On the other hand, one should take into consideration that the strength of evidence cannot be used directly as weighting factor in relevance prediction as for aspects receiving strong evidence for their relevance their precise strength of evidence no longer matters; in other words, weighting factors should saturate for high evidence ratios. A saturating weighting function that could be used to this end is for instance $w(\text{LR}) = 1 - \exp(-\alpha \text{LR})$, where α determines the saturation rate.

Perhaps even more importantly, however, given that typically we have a very large number of aspects, mixing the effect of many uncertain aspects may drastically deteriorate performance as the many aspect that are of no particular relevance will dominate the effect of the few truly relevant aspects. We thus choose to include only those aspects for which the evidence is sufficiently strong. Also various types of corrections for correlations between aspects are possible, (e.g., by aspect grouping, or shared scoring) these are, however, beyond the scope of this chapter.

5.6.4 Selection of Informative Images

An infamous issue in content-based image retrieval is the "page-zero problem:" a user has a certain image or class of images in mind, but due to the relative small fraction of the database shown he cannot find images to get him started. We can aid in this situation by showing the user not only a selection of images based on ranking by relevance, but additionally provide images with aspects the user has not yet seen up to that point. Such images are also informative from the inference perspective as the system cannot determine the relevance of aspects for which the user has had no chance to provide feedback.

Let S be the index set of n^S images shown up until that point from which the user has been allowed to make his selection, of which N^S images have the aspect.

To construct an informative selection, we sort the aspects based on their associated N^S value, i.e., the number of images shown to the user that possess the aspect. From this ranking we determine a set $\tilde{A}$ of most informative aspects.

Next, we rank the images that satisfy at least one of these aspects by increasing value of the total number of such aspects they have. This allows the user to provide partial relevance feedback on informative aspects, while minimizing the risk that he provides misleading feedback on other aspects.

6. Conclusion and Outlook

With the current state-of-the-art in image content description and feature extraction, meaningful retrieval in specialized image databases still depends to an important extent on explicit modeling of domain knowledge. For many low-level perceptual characteristics, e.g., with respect to color and texture, standard solutions are available. However, as the user generally thinks in terms of high level concepts, such characterizations are generally insufficient in satisfying his wishes.

Features are required that characterize the structure of the image in terms of properties and relations between the elements in the images. For instance, we have seen that for decoration designs, users may wish to find designs with certain specific types of objects, of a certain size etc. Modeling of domain knowledge should be done as generically as possible in order to avoid re-inventing the wheel for every single aspect that may play a role in the user's perception. Research is thus needed in developing flexible representations that may be re-used for various characterization tasks. In our view, a key requirement to this end is a robust system of object detection and recognition. By exploiting domain knowledge it is sometimes possible to achieve a satisfactory performance in this respect as we have seen in the example of the figure-ground segregation approach. However, further work remains to be done in particular in coping with the intricate interactions of various grouping principles and in being able to deal with the most unambiguous clues first.

Given the flexible representations, machine learning techniques can establish the precise relations required for the characterization task at hand.

Ontologies facilitate standardization by providing a vocabulary of shared terms together with a structure of feasible properties and relations by which images in a domain are described. The development of such description ontologies is also a very useful exercise in gaining insight into which aspects need to be captured automatically.

Once images can be described in terms of higher level aspects, we must still face the fact that which of these aspects matters most is generally user- and task-dependent. Using relevance feedback techniques this issue may be resolved through natural interaction with the system. To this end we described a statistically principled approach, which is directly aimed at assessing which image aspects determine relevance, and which takes into account the special structure of feedback data.

Acknowledgements

This work has been partially funded by the European Commission under the IST Programme of the Fifth Framework (Project nr. IST-2000-28427). We thank Sophis Systems NV and Pianezza Srl for supplying the decoration design test collections.

References

[1] Benitez A.B., Martinez J.M., Rising H., Salembier P., “Description of a single multimedia document”, in: Manjunath B., Salembier P., Sikora T. (Eds.), *Introduction to MPEG-7—Multimedia Content Description Interface*, John Wiley and Sons, Ltd., Chichester, England, 2002, pp. 111–138.

[2] Bigün J., Granlund G., “Optimal orientation detection of linear symmetry”, in: *Proc. 1st Internat. Conf. Computer Vision*, 1987, pp. 433–438.

[3] Bober M., Preteux F., Kim W.-Y.Y., “Shape descriptors”, in: Manjunath B., Salembier P., Sikora T. (Eds.), *Introduction to MPEG-7—Multimedia Content Description Interface*, John Wiley and Sons, Ltd., Chichester, England, 2002, pp. 231–260.

[4] Boggess A., Narcowich F.J., *A First Course in Wavelets with Fourier Analysis*, Prentice Hall, Upper Saddle River, NJ, 07458, 2001.

[5] Bovik A., Clark M., Geisler W., “Multichannel texture analysis using localized spatial filters”, *IEEE Trans. on Pattern Analysis and Machine Intelligence* **12** (12) (1990) 55–73.

[6] Choi Y., Won C.S., Ro Y.M., Manjunath B., “Texture descriptors”, in: Manjunath B., Salembier P., Sikora T. (Eds.), *Introduction to MPEG-7—Multimedia Content Description Interface,* John Wiley and Sons, Ltd., Chichester, England, 2002, pp. 213–230.

[7] Ciocca G., Schettini R., “Using a relevance feedback mechanism to improve content-based image retrieval”, in: Huijsmans D., Smeulders A. (Eds.), *Visual Information and Information Systems*, Springer-Verlag, Berlin/New York, 1999, pp. 107–114.

[8] Cox I., Miller M., Minka T., Papathomas T., “The Bayesian image retrieval system, PicHunter: Theory, implementation, and psychophysical experiments”, *IEEE Trans. Image Processing* **9** (1) (2000) 20–37.

[9] Duda R., Hart P., *Pattern Classification and Scene Analysis*, John Wiley and Sons, Inc., New York, USA, 1973.

[10] Eakins J., Boardman J., Graham M., “Trademark image retrieval by shape similarity”, *IEEE Multimedia* **5** (2) (1998) 53–63.

[11] Flickner M., Sawhney H., Niblack W., Ashley J., Huang Q., Dom B., Gorkani M., Hafner J., Lee D., Petkovic D., Steele D., Yanker P., “Query by image and video content: The QBIC system”, *IEEE Computer* **28** (9) (1995) 23–32.

[12] Freeman W., Adelson E., “The design and use of steerable filters”, *IEEE Trans. on Pattern Analysis and Machine Intelligence* **13** (9) (1991) 891–906.

[13] Freixenet J., Munoz X., Raba D., Marti J., Cufi X., “Yet another survey on image segmentation: Region and boundary information integration,” in: *Proceedings ECCV 2002 III*, 2002, pp. 408–422.

[14] Gimel’farb G., Jain A., “On retrieving textured images from an image database”, *Pattern Recognition* **29** (9) (1996) 1461–1483.

[15] Goldfarb P., Prescod P., *The XML Handbook*, third ed., Prentice Hall, NJ, 2001.

[16] Gotlieb C., Kreyszig H., “Texture descriptors based on co-occurrence matrices”, *Computer Vision, Graphics, and Image Processing* **51** (1) (1990) 70–86.

[17] Hacking I., *Logic of Statistical Inference*, Cambridge University Press, New York, 1965.

[18] Huang T., Zhou S., “Image retrieval with relevance feedback: From heuristic weight adjustment to optimal learning methods,” in: *Proc. IEEE Internat. Conf. on Image Processing (ICIP)*, October 2001.

[19] Huiskes M., Pauwels E., “Segmentation by color coalition labeling for figure-ground segregation in decoration designs,” in: *Proceedings of the 3rd International Symposium on Image and Signal Processing and Analysis (ISPA)*, 2003, pp. 84–90.

[20] Huiskes M., Pauwels E., Bernard P., Derumeaux H., Vandenborre P., Van Langenhove L., Sette S., “Metadata for decorative designs: Application of MPEG-7 in automatic design interpretation,” in: *Proceedings of the World Textile Conference and third Autex Conference*, June 2003, pp. 502–506.

[21] Kass M., Witkin A., “Analyzing oriented patterns”, in: Fischler M., Firschein O. (Eds.), *Readings in Computer Vision*, Morgan Kaufmann, Los Altos, CA, 1987, pp. 268–276.

[22] Kohonen T., *Self-Organization and Associative Memory*, third ed., Springer-Verlag, Berlin, 1989.

[23] Laaksonen J., Koskela M., Oja E., “PicSOM: Self-organizing maps for content-based image retrieval”, *Pattern Recognition Letters* **27** (13/14) (December 2000) 1199–1207.

[24] Laine A., Fan J., “Texture classification by wavelet packet signature”, *IEEE Trans. on Pattern Analysis and Machine Intelligence* **15** (11) (1993) 1186–1191.

[25] Lin H., Wang L., Yang S., “Color image retrieval based on hidden Markov models”, *IEEE Trans. Image Processing* **6** (2) (1997) 332–339.

[26] Lin H., Wang L., Yang S., “Extracting periodicity of a regular texture based on autocorrelation functions”, *Pattern Recognition Letters* **18** (1997) 433–443.

[27] Liu F., Picard R., “Periodicity, directionality, and randomness: Wold features for image modeling and retrieval”, *IEEE Trans. Pattern Analysis and Machine Intelligence* **18** (7) (1996) 517–549.

[28] Liu Y., Collins R., Tsin Y., “A computational model for periodic pattern perception based on frieze and wallpaper groups”, *IEEE Trans. Pattern Analysis and Machine Intelligence* **26** (3) (2004) 354–371.

[29] MacArthur S., Brodley C., Shyu C., “Relevance feedback decision trees in content-based image retrieval,” in: *IEEE Workshop on Content-Based Access of Image and Video Libraries*, 2000, pp. 68–72.

[30] Manjunath B., Ma W., “Texture features for browsing and retrieval of image data”, *IEEE Trans. on Pattern Analysis and Machine Intelligence* **18** (8) (1996) 837–842.

[31] Mao J., Jain A., “Texture classification and segmentation using multiresolution simultaneous autoregressive models”, *Pattern Recognition* **25** (2) (1992) 173–188.
[32] Meilhac C., Nastar C., “Relevance feedback and category search in image databases,” in: *Proc. Internat. Conf. Multimedia Computing and Systems*, 1999, pp. 512–517.
[33] Minka T., Picard R., “Interactive learning using a “society of models” ”, *Pattern Recognition* **30** (4) (1997) 565–581.
[34] Mojsilovic A., Hu J., Soljanin E., “Extraction of perceptually important colors and similarity measurement for image matching, retrieval, and analysis”, *IEEE Trans. on Image Processing* **11** (11) (November 2002) 1238–1248.
[35] MPEG7, Multimedia content description interface, http://www.chiariglione.org/mpeg/, 2003.
[36] Neyman J., Pearson E., “On the problem of the most efficient tests of statistical hypotheses”, *Philosophical Trans. of the Royal Society, Series A* **231** (1933) 289–337.
[37] Ohm J.-R., Cieplinski L., Kim H.J., Krishnamachari S., Manjunath B., Messing D.S., Yamada A., “Color descriptors”, in: Manjunath B., Salembier P., Sikora T. (Eds.), *Introduction to MPEG-7—Multimedia Content Description Interface*, John Wiley and Sons, Ltd., Chichester, England, 2002, pp. 187–212.
[38] “OWL, Web ontology language”, http://www.w3.org/2001/sw/WebOnt/, 2003.
[39] Pal N., Pal S., “A review on image segmentation techniques”, *Pattern Recognition* **26** (9) (1993) 1277–1294.
[40] Pass G., Zabih R., Miller J., “Comparing images using color coherence vectors,” in: *4th ACM Conference on Multimedia*, November 1996, pp. 65–73.
[41] Pauwels E., Huiskes M., Bernard P., Noonan K., Vandenborre P., Pianezza P., De Maddelena M., “FOUNDIT: Searching for decoration designs in digital catalogues,” in: *Proceedings of the 4th European Workshop on Image Analysis for Multimedia Interactive Services*, 2003, pp. 541–544.
[42] Peng J., Bhanu B., Qing S., “Probabilistic feature relevance learning for content-based image retrieval”, *Computer Vision and Image Understanding* **75** (1/2) (1999) 150–164.
[43] Pratt W., *Digital Image Processing*, second ed., John Wiley and Sons, New York, 1991.
[44] Randen T., Husoy J., “Filtering for texture classification: a comparative study”, *IEEE Trans. Pattern Analysis and Machine Intelligence* **21** (4) (1999) 291–310.
[45] Ranguelova E., Huiskes M., Pauwels E., “Towards computer-assisted photo-identification of humpback whales,” in: *Proceedings of ICIP 2004*, 2004.
[46] “RDF—Resource Description Frameworks”, http://www.w3.org/RDF/, 2003.
[47] Reed T., Du Buf J., “A review of recent texture segmentation and feature extraction techniques”, *Source CVGIP: Image Understanding Archive* **57** (3) (May 1993) 359–372.
[48] Rocchio J. Jr., “Relevance feedback in information retrieval”, in: Salton G. (Ed.), *The SMART Retrieval System: Experiments in Automatic Document Processing*, Prentice Hall, New York, 1971, pp. 313–323.
[49] Royall R., *Statistical Evidence: A Likelihood Paradigm, Monographs on Statistics and Probability*, Chapman and Hall, London, 1997.
[50] Royall R., “On the probability of observing misleading statistical evidence”, *Journal of the American Statistical Association* **95** (451) (2000) 760–780.

[51] Rui Y., Huang T., Ortega M., Mehrotra S., "Relevance feedback: A power tool for interactive content-based image retrieval", *IEEE Trans. Circuits and Systems for Video Technology* **8** (5) (1998) 644–655.
[52] Russ J., *The Image Processing Handbook*, second ed., CRC Press, Boca Raton, FL, 1995.
[53] Salembier P., Smith J.R., "Overview of multimedia description schemes and schema tools", in: Manjunath B., Salembier P., Sikora T. (Eds.), *Introduction to MPEG-7—Multimedia Content Description Interface*, John Wiley and Sons, Ltd., Chichester, UK, 2002, pp. 83–94.
[54] Schattschneider D., "The plane symmetry groups: their recognition and notation", *American Mathematical Monthly* **85** (1978) 439–450.
[55] Smeulders A., Worring M., Santini S., Jain R., "Content-based image retrieval at the end of the early years", *IEEE Trans. on Pattern Analysis and Machine Intelligence* **22** (12) (2000) 20–37.
[56] Soille P., *Morphological Image Analysis*, Springer-Verlag, Berlin, 1999.
[57] Sparck Jones K., "A statistical interpretation of term specificity and its application in retrieval", *Journal of Documentation* **28** (1) (1972) 11–21.
[58] Swain M., Ballard D., "Color indexing", *International Journal of Computer Vision* **7** (1) (1991) 11–32.
[59] Tieu K., Viola P., "Boosting image retrieval", *International Journal of Computer Vision* **56** (1/2) (2004) 17–36.
[60] Tong S., Chang E., "Support vector machine active learning for image retrieval," in: *Proc. of 9th ACM Internat. Conf. on Multimedia*, 2001, pp. 107–118.
[61] Van Droogenbroeck M., Talbot H., "Fast computation of morphological operations with arbitrary structuring elements", *Pattern Recognition Letters* **77** (14) (1996) 1451–1460.
[62] Vasconcelos N., Lippman A., "Learning from user feedback in image retrieval," in: *NIPS 99*, 1999, pp. 977–986.
[63] Veltkamp R., Hagedoorn M., "State-of-the-art in shape matching", in: Lew M. (Ed.), *Principles of Visual Information Retrieval*, Springer-Verlag, Berlin/New York, 2001, pp. 87–119.
[64] Wertheimer M., "Untersuchungen zur Lehre von der Gestalt. II", *Psychologische Forschung* **4** (1923) 301–350.
[65] Wu P., Manjunath B., "Adaptive nearest neighbor search for relevance feedback in large image databases," in: *Proc. of 9th ACM Internat. Conf. on Multimedia*, 2001, pp. 89–97.
[66] Wu Y., Tian Q., Huang T., "Discriminant EM algorithm with application to image retrieval," in: *IEEE CVPR*, 2000, pp. 1222–1227.
[67] Zhang H., Su Z., "Relevance feedback in CBTR", in: Zhou X., Pu P. (Eds.), *Visual and Multimedia Information Systems*, Kluwer Academic Publishers, Dordrecht/Norwell, MA, 2002, pp. 21–35.
[68] Zhou X., Huang T., "Relevance feedback in image retrieval: a comprehensive review", *ACM Multimedia Systems Journal* **8** (6) (April 2003) 536–544.

Defect Analysis: Basic Techniques for Management and Learning

DAVID N. CARD

Q-Labs
115 Windward Way
Indian Harbour, FL 32937
USA
card@computer.org

Abstract
This chapter provides an overview of both the state of the art and the state of the practice of defect analysis, with an emphasis on the latter. Defect analysis is defined as the study of the properties of defects themselves, as opposed to methods for predicting the number and nature of defects based on other properties of software, such as complexity models. Defect data often is one of the first types of data that software organizations collect. Success in analyzing defect data often determines whether or not a software organization's measurement program flourishes or withers. This chapter discusses three practical purposes of defect analysis: modeling/predicting software quality, monitoring process performance, and learning from past mistakes about how to improve process performance and software quality. The techniques supporting these purposes range from simple arithmetic to complex statistical analysis. While this chapter provides an overview of many defect analysis approaches, it focuses on the simplest, those that are most readily applied in practice.

ADVANCES IN COMPUTERS, VOL. 65
ISSN: 0065-2458/DOI 10.1016/S0065-2458(05)65006-1

1. Introduction

There are many different notions of software quality. ISO/IEC Standard 9126 [26] defines six: reliability, maintainability, portability, functionality, efficiency, and usability. Many of these are difficult to measure directly. Regardless of the notion of quality that a project or organization adopts, most practitioners would agree that the presence of defects indicates a lack of quality, whatever it is. Consequently, managing and learning from defects is profoundly important to software practitioners. This chapter provides an overview of the techniques that are commonly used in defect analysis.

Defect analysis is defined as the investigation of the properties of defects themselves, usually with the aim of minimizing them or managing their consequences. Historically, academic researchers have focused relatively more attention on predicting defects from other properties of software, such as complexity and structure, rather than on the direct analysis of defects. The related field of reliability modeling has a long history of both academic research and practical application. It deals with "failures," a behavior of software closely related to defects. While the study of failures gives us approximate information about the defectiveness of software, it is not a direct analysis of defects.

For purposes of this chapter we define a defect to be a segment of software or documentation that must be changed in order to satisfy the project requirements. Usually defects are the consequence of mistakes or omissions by human beings during the course of developing the software or documentation. Defects may be found in inspections or reviews as well as resulting from the debugging of failures detected in testing. Sometimes failures are equated with defects. However, a failure may be the consequence of more than one defect. The kind of information typically recorded about defects includes:

- Type of defect.
- Phase/activity in which inserted.
- Phase/activity in which detected.
- Effort to fix.

Defect analysis has become important to software practitioners because defects are both a convenient source of information and a significant impact to project success.

They are convenient in the sense that they are abundant and easily counted. They are significant in that more defects means lower customer satisfaction, more rework effort/cost, and potential delays while repairs are made. Consequently minimizing, or at least managing, defects is a basic concern of software managers.

This chapter discusses three basic purposes of defect analysis:

- Modeling (or predicting) software quality—to determine whether or not an acceptable level of quality will be achieved at delivery. This involves developing a plan for quality.
- Monitoring process performance—to recognize when process performance departs from expectations with respect to defectiveness. Changes in process performance may threaten the achievement of the quality plan.
- Learning from past mistakes—to figure out how to improve process performance and software quality. The need or opportunity to learn often is signaled by the results of quality modeling and process monitoring.

Together, the techniques that implement these objectives define a feedback system that helps to control and improve an industrial software engineering process.

Defects also represent work items from a project management perspective. That is, the effort to fix defects must be considered in project planning and tracking. Moreover, the number of defects outstanding may be a factor in the acceptance of the software by the customer. These issues are briefly discussed along with the defect-based models of software quality. However, a thorough treatment of defects as work items falls outside the scope of this chapter.

The techniques supporting these three basic purposes of defect analysis range from simple arithmetic to complex statistical computations. While this chapter provides an overview of many defect analysis approaches, it focuses on the simplest, those that are most readily applied in practice.

The next three sections (2 through 4) review techniques appropriate for each of the three purposes of defect analysis. Section 5 discusses the relationship of these techniques to some of the popular software process improvement paradigms. Finally, Section 5 provides the summary and conclusions.

2. Modeling for Quality Management

The practical purpose of modeling software quality with defects is to determine whether or not the level of defects in the software at different stages of development is acceptable and consistent with project plans. This involves developing a plan for quality. There are two general approaches to modeling software quality with defects.

One involves the analysis of defect levels across the project lifecycle. The other compares defect discoveries by multiple agents, perhaps within the same lifecycle phase. The following sections describe several techniques supporting each approach.

2.1 Life Cycle Defect Profiles

Remus and Ziles [45] proposed a simple two phase model of defect discovery in projects. However, that approach only supports estimation late in the project's life. This section discusses developing a similar, but more detailed, life-cycle defect profile as described in [10]. A defect profile serves as a "quality budget." It describes planned quality levels at each phase of development, just as a budget shows planned effort (or cost) levels. Actual defect levels can be measured and compared to the plan, just as actual effort (or cost) is compared to planned effort (or cost). Investigating departures from the plan leads to corrective actions that optimize project outcomes.

Software development is made up of a set of processes, usually organized into sequenced phases, each of which has some ability to insert and detect defects. The quality plan is an estimate of the number of defects expected to be inserted and detected in each phase. However, only the number of detected defects in any phase can be known with confidence prior to the completion of the project. The number of defects inserted in each phase cannot be known until all defects have been found. We approach that number after the system has been fielded. Consequently, this approach focuses on defects detected as the knowable quantity.

The techniques presented here depend on two key assumptions:

- Size is the easily quantifiable attribute of software that is most closely associated with the number of defects [17]. The basic test of the effectiveness of complexity models and other indicators of defect-proneness is "Does this model show a significantly higher correlation with defects than just size (e.g., lines of code) alone?" [11].
- Defect insertion and detection rates tend to remain within relatively consistent ranges as long as the software processes of the project remain stable. They are not exactly constant, but they perform within a recognizable range.

The first assumption appears to be inherent to the nature of software. High maturity organizations actively work to make the second assumption come true. That is, they are acting to bring their processes under control. However, that assumption does not require the demonstration of statistically stable processes, just repeatable processes that are consistently performed.

Either of two strategies may be adopted in developing a defect profile: analysis of empirical data or reliance on an analytical model [10]. The following subsections discuss these strategies.

2.1.1 An Empirical Model

The simplest approach to generating a defect profile is to collect actual data about the insertion and detection rates in each phase from projects within the organization to which the defect profile is intended to apply. The development of an empirical profile can be accomplished in four steps, as follows:

- Historical data are collected. Table I shows a simple spreadsheet used to tabulate defect discovery and detection data using example data. In addition the size of the project from which the defect data is collected must be known. The selected size measure must be applied consistently, but it this approach does not require the use of any specific measure. Lines of code, function points, number of classes, etc., may be used as appropriate. (The data in Table I is simulated, not real.)
- An initial profile of the number of defects found in each phase is generated as shown in Figure 1. The bars in that figure represent the row totals found in the last column of Table I.
- This initial profile is scaled to account for differences between the size of the project(s) from which the profile was developed and the size of the project to which it is applied. This is accomplished by multiplying by the ratio of the project sizes. For example, if the defect profile in Figure 1 were to be used to develop a defect profile for a project twice the size of the project providing the data that went into Figure 1, then the bars of the profile representing the new project would be twice the size of those in Figure 1. This implies that twice as many defects are likely to be found in each phase of the new project.
- The scaled defect profile is adjusted further to reflect the planned performance of the project. For example, if the project plan called for the automatic generation of code from design instead of hand coding as previously done, then

TABLE I
EXAMPLE OF EMPIRICAL DEFECT PROFILE (SIMULATED DATA)

Phase detected	Phase inserted					
	Analysis	Design	Code	Developer test	System test	Total
Analysis	0					0
Design	50	200				250
Code	50	100	300			450
Developer test	25	50	150	0		225
System test	18	38	113	0	0	169
Operation	7	12	37	0	0	56
Total	150	400	600	0	0	1150

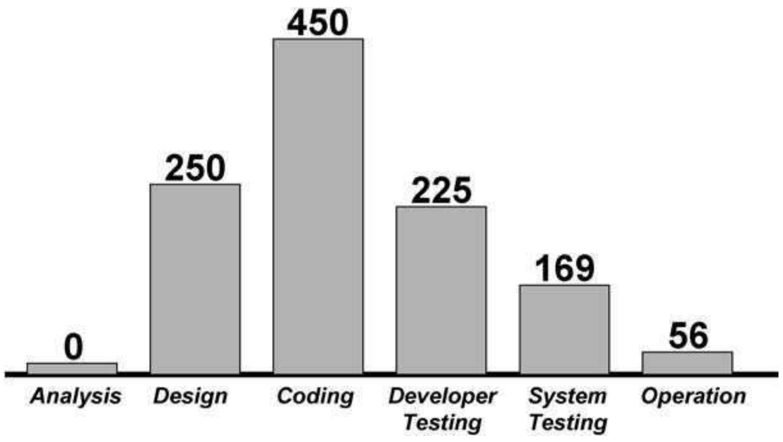


FIG. 1. Example of defect profile with data from Table I.

> the number of defects inserted in the implementation phase would be adjusted downwards to reflect this change in the coding process. Also, changes in the project's process (such as increased or improved inspections) may be induced in order to reach a specified target in terms of delivered quality, if previous performance did not yield the required level of quality, as specified by a customer requirement or an organizational goal.

Actual defect counts can then be compared with this final plan (defect profile) as the project progresses. Suggestions for this activity are provided later in this article. Note that the defect profile does not address defect status (i.e., "open" versus "closed" problems/defects). All defects detected, regardless of whether or not they ever get resolved, are included in the defect counts.

Figure 2 shows as actual defect profile as prepared two-thirds of the way through software integration testing. About two-thirds of the predicted number of defects actually have been detected. This illustrates that the real value of the defect profile lies in its ability to make quality visible during development, not as a post-mortem analysis technique.

Figure 2 shows an example of a defect profile developed empirically for an actual military project [9]. This figure shows the predicted number of defects to be injected and detected in each phase, based on previous projects. However, only actual counts are shown for the number of detected is shown, because the actual number injected can't be determined with any confidence until after delivery of the software.

The project in Figure 2 actually was completed after this graph was prepared. The planned and actual defect levels never differed by more than 10 percent. The project

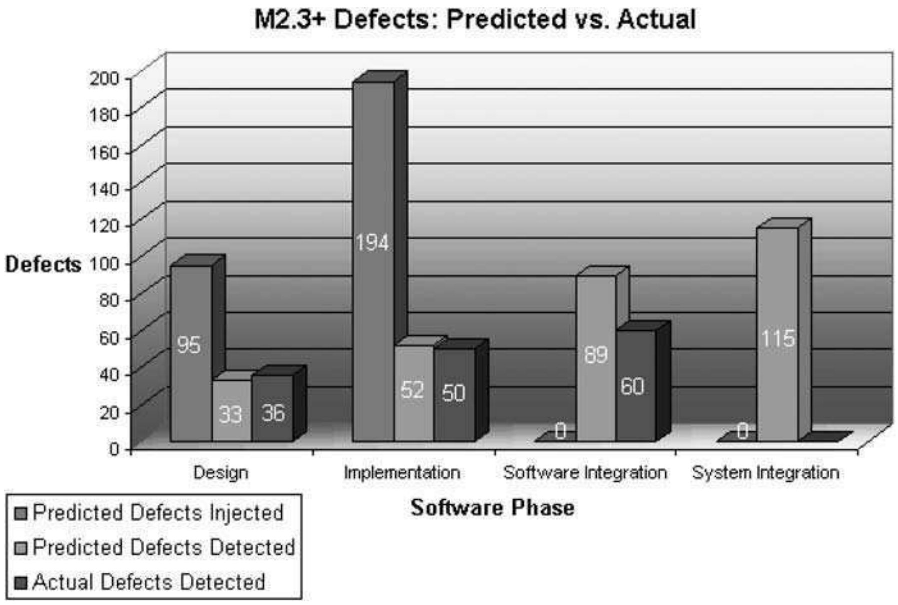


FIG. 2. Example of empirical defect profile (actual project data). (See Color Plate Section, Plate 30.)

team handed their product over to the customer with a high degree of confidence that it met the targeted level of quality.

Weller [49] provides a case study of successfully applying this approach to a large commercial software system. Again, the defect profile provided useful insight into the quality of the developing software system.

2.1.2 An Analytical Model

Defect profiles may also be generated using mathematical models. Many early studies of defect occurrence suggested that it followed a Rayleigh curve, roughly proportional to project staffing. The underlying assumption is that the more effort expended, the more mistakes that are made and found. Putnam [42] described the implications of the Rayleigh curve for estimating project effort.

Gaffney [21] developed a similar model for defects:

$$V_t = E\left(1 - e^{-Bt^2}\right)$$

where:

- V_t = Number of defects discovered by time t;

- E = Total number of defects inserted;
- B = Location parameter for peak.

The time periods t can be assumed to be equal to life-cycle phase transition boundaries in order to apply the model to project phases rather than elapsed time. The location parameter B fixes the time of the maximum (or peak) of the distribution (discovery of defects). For example, $B = 1$ means that the peak occurs at $t = 1$.

The analytical approach involves applying regression analysis to actual phase-by-phase defect data to determine the values of B and E that produce a curve most like the input data. While specialized tools are available, e.g., SWEEP [48], to perform this analysis—it can easily be implemented in Microsoft Excel using the Data Analysis Add-Ins.

The effectiveness of the analytical approach using the Rayleigh model depends on the satisfaction of additional assumptions, including:

- Unimodal staffing profile.
- Life-cycle phases of similar (not exactly equal) duration (not effort).
- Reasonably complete reporting of defects.
- Use of observable/operational defects only.

To the extent that these assumptions are satisfied, this model gives better results. Analytical models such as this are useful when the organization lacks complete life-cycle defect data or desires to smooth existing data to provide an initial solution for new projects without prior historical data. The defect profile obtained from the actual data can be easily adjusted to fit projects with different numbers of life cycle phases and varying process performance by selecting appropriate values of E and B.

Figure 3 provides an example of a defect profile for another actual military project generated by SWEEP. The light bars in Figure 2 represent the expected number of defects for each phase, based on the model. For this specific project, the actual number of defects discovered is substantially lower than planned during Design. Consequently, additional emphasis was placed on performing rigorous inspections during Code, with the result that more defects than originally anticipated were captured during Code, putting the project back on track to deliver a quality project as shown at Post Release.

Harbaugh [23] provides a detailed analysis of applying the Rayleigh model to a large software project. Kan [33] presents some similar approaches to analytical defect modeling. Rallis and Lansdowne [44] have developed a very different analytical modeling approach that uses data from sequential independent reviews to estimate the number of defects remaining and eventual reliability. However, this approach does not appear to have been tried out in industry, yet.

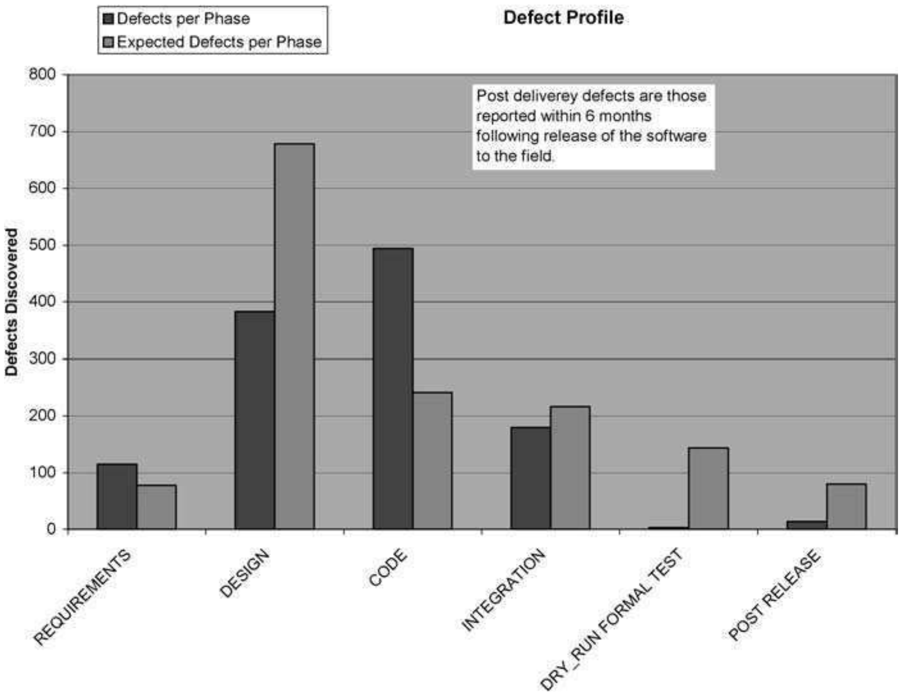


FIG. 3. Example of analytical defect profile (actual project data). (See Color Plate Section, Plate 31.)

2.1.3 *Interpreting Differences*

During project execution planned defect levels are compared to actual defect levels. Typically, this occurs at major phase transitions (milestones). However, if a phase extends over more than six months, then additional checkpoints may be inserted into some phases (as in the example of Figure 2 where analyses were conducted at the completion of each third of integration testing). Analyzing the performance of the project relative to the planned defect profile involves three steps:

- Determine if the differences are significant and/or substantive. (We never get exactly what we plan in the real world.) Assessment of the magnitude of the differences might be accomplished by seeking visually large differences, establishing thresholds based on experience, or applying statistical tests such as the Chi-Square [25].
- Determine the underlying cause of the difference. This may require an examination of other types and sources of data, such as process audit results as well

as effort and schedule data. Techniques for causal analysis are discussed in Section 4.

- Take action as appropriate. This includes corrective actions to address problems identified in the preceding step, as well updates to the defect profile to reflect anticipated future performance.

Differences between planned and actual defect levels do not always represent quality problems. Potential explanations of departures from the plan (defect profile) include:

- Bad initial plan (assumptions not satisfied, incomplete or inappropriate data).
- Wrong software size (more or less than initial estimate).
- Change in process performance (better or worse than planned).
- Greater or lesser complexity of software than initially assumed.
- Inspection and/or test coverage not as complete as planned.

Analyzing departures from the defect profile early in the lifecycle gives us feedback on our understanding of the size and complexity of the software engineering task while there is still time to react. Phan et al. [41] showed that the costs of removing defects increase as the project moves through its lifecycle. Westland [50] also argued the cost benefits of early defect detection and removal. Defects represent rework that must eventually be accomplished in order to make the software acceptable for delivery. Consequently, understanding the defect content of the software early in the life cycle has important business consequences. The defect profile makes those defects visible to management.

2.1.4 Orthogonal Defect Classification

Defect models can become very rich. Chillarge [13], for example, developed an approach called Orthogonal Defect Classification that involves tracking and analyzing many attributes of defects. These attributes include type of defect, detecting activity, and inserting activity. Eight types of defects are defined within ODC:

- Interface;
- Function;
- Build/package/merge;
- Assignment;
- Documentation;
- Checking;
- Algorithm;
- Timing/Serialization.

Different types of defects tend to be found at different stages of development. For example, function defects are relatively easy to find during inspections, so they tend to be found and removed early. Timing/serialization defects, on the other hand, are difficult to find during inspections, so they tend to turn up during testing. A separate defect profile can be developed for each type of defect to reflect its propensity for detection during each phase. These defect classifications and profiles also facilitate the causal analysis process when potential problems are identified.

Bhandari [3] reports on the successful application of ODC to three different kinds of projects. Bridge et al. [4] describe the application of ODC to a mobile telephone project. In this case the categories of the existing defect classification system were mapped to the ODC structure, resulting in a somewhat smaller set of categories. The feedback on occurrences of defect types was used to guide process improvement and monitor quality progress.

This discussion has shown that relatively simple models of software quality based on defect profiles are becoming increasingly popular in the software industry as organizations mature. These models establish a “quality budget” that helps to make tradeoffs among cost, schedule, and quality visible and reasoned, rather than choices by default. Defect profiles present quality performance to the project manager in a form that he/she understands. Thus, the consequences of a decision such as “reducing inspection and testing effort to accelerate progress” can be predicted. Unintended departures from planned quality activities can be detected and addressed.

2.2 Capture/Recapture

Another way of estimating defect content from observed defects involves comparing the results of different inspectors operating in parallel. Capture–recapture models are well-established as methods of estimating animal populations in biological field studies. This approach involves capturing animals in the wild, tagging them, and then releasing them. The proportion of tagged animals subsequently recaptured gives insight into the size of the total population. If a high percentage of animals captured in the second pass are tagged, then the captured animals represent a larger proportion of the total population than if a small proportion of recaptures are tagged.

Eick et al. [16] first proposed applying capture-recapture methods to software engineering. Petersson et al. [40] summarize ten year’s of research and experience with applying capture-recapture methods to software inspections. Here the defects captured by independent inspectors are compared. Many different algorithms have been proposed during this time. El Emam and Laitenberger [18] performed an extensive simulation study of capture-recapture models attempting to identify the best algorithms and assumptions. Probably the most enduring criticism of the capture-recapture approach has been that the number of defects associated with any particular software artifact is so low that the resulting confidence intervals for the estimates of the number of defects are very wide [29]. Petersson et al. [40], report only a single published industrial case study of the application of capture/recapture models. Most work thus far has been carried out with student-based experiments.

Both defect profiling and capture-recapture techniques use information about the defects that have been discovered to assess the current level of quality, so that appropriate action can be taken. The appropriate action to be taken is decided upon via the defect causal analysis techniques discussed in Section 4.

3. Monitoring Process Performance

The defect estimation methods described earlier work best when the underlying processes that insert and detect defects are performed consistently. Good process definitions, quality assurance, and other regulatory mechanisms help to ensure that happens. However, process performance also can be monitored directly to determine if processes are stable. Statistical Process Control (SPC) was developed by Shewhart [47] to manage manufacturing processes. Many good textbooks (e.g., [51]) explain the principles and techniques of SPC. Recently, textbooks have appeared that discuss the application of SPC techniques [19], and statistics in general [5], to software.

3.1 SPC Concepts

SPC is a problem-solving approach and a set of supporting techniques. It focuses on managing the variation inherent in all processes. Controlling variation results in processes that are stable and predictable. The SPC toolkit includes run charts, control charts, histograms, Pareto charts, scatter diagrams, and more. The SPC problem-solving approach involves four steps:

(1) develop a “control” plan,
(2) perform a process as defined,
(3) measure the results, and
(4) use the results to stabilize and, then, improve the process.

Stability is the result of control. Achieving control requires real-time feedback on process performance. This means that the performance of the process must be evaluated as it is being executed, not just at the end. For example, controlling software or systems design means recognizing whether or not that design subprocess (or key process elements within it) is performing as expected before all executions of the subprocess (or process element) have been completed, so that appropriate action can be taken. Put in other terms, process control requires in-phase monitoring of the performance of design activities, not just an end-of-phase review of design performance.

Software and systems engineering processes are composed of many subprocesses and process elements. It is impractical to attempt to control all those components of the process. However, the stability of the overall end-to-end process can be promoted by focusing attention on key process elements.

Each time a process is repeated, a slightly different result is obtained. Regardless of how hard one tries to repeat the process exactly as intended, the output is, at least, slightly different. This is due to variations in the input as well as other factors influencing process performance. Variations in performance may be described as due to one of two types of causes (or factors):

Special Cause. These are factors that affect a specific execution of a process element. Often, they result from failing to execute the process element as intended. Dealing with special causes is a control and stability issue. Special causes also may be referred to as "assignable causes."

Common Cause. These are the factors that operate continually. Every time the process element is executed, these factors affect it. Dealing with common causes is an improvement issue.

Whether a cause is "special" or "common" cannot be determined until the cause has been found. Consequently, even when focusing on control, opportunities for improvement will be encountered.

To help illustrate the difference between special and common causes, consider the following example. The definition of a project's inspection process states that at least two (working) days prior to holding an inspection meeting, the inspection materials must be sent to inspection participants; within 2 weeks (10 working days) after holding an inspection, all actions (including resolving problems and filling out the inspection report) must be completed. Based on this definition and knowledge of human nature, the typical time required for one execution of the inspection process element will be 12 working days. Most people tend to get things done around the deadline. Thus, they are likely to send out the inspection materials only 2 days in advance and take the full 2 weeks to close out the inspection. The process definition and human nature are both examples of "common causes." They operate every time you execute this process element. Both could be changed, *if that were judged likely to lead to improved performance*: staff could be incentivized to finish early; the process definition could be rewritten to allow more or less time.

Now, imagine that during an inspection of a design unit, an inconsistency was discovered in an interface specification. In order to get the design unit right, the interface specification has to be corrected first. That could easily take more than the 2 weeks typically required for closing an inspection. This is an example of a "special cause." It affects only this particular execution of the instantiated process element.

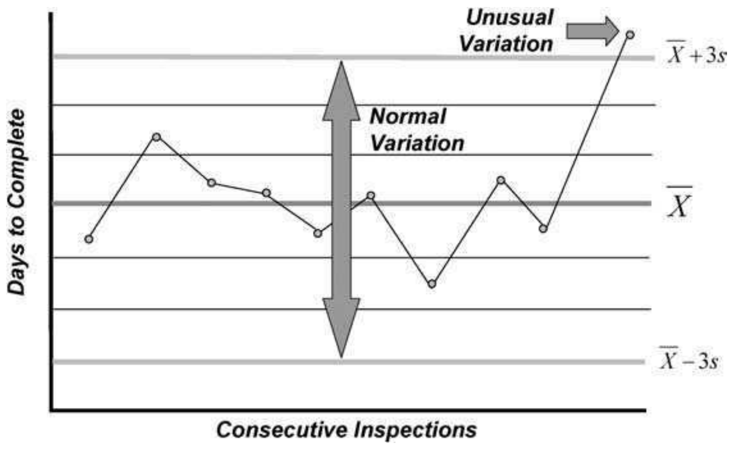


FIG. 4. Illustration of common and special causes. (See Color Plate Section, Plate 32.)

The process manager needs to recognize that this situation has occurred and take additional steps to resolve the situation in a timely manner.

Figure 4 illustrates these concepts. Measuring the "days to complete" for a sequence of design inspections might yield the data plotted in the figure. This data could be used to develop a statistical model of the performance of this process element. For this data, assume a normal distribution. Then, $\overline{X}$ and s are the average and the standard deviation, respectively, computed from the data using an appropriate technique. More than 99% of values from a normal distribution fall within three standard deviations of the mean, so data within this range represents normal variability, or common causes and random error. The last point plotted falls outside this range, indicating an unusual source of variation, or a special cause. The analysis technique represented by Figure 4 is called a control chart. It provides a graphical mechanism for studying process variation. The points that are plotted represent subgroups. Each subgroup may contain multiple observations.

Efforts to stabilize and control processes typically depend on "small-scale" studies. They focus at the level of process elements, employing one or two key measures per process element. An individual statistical model is developed for each performance measure. However, the results of these individual analyses are considered together in deciding on the *need* for corrective or improvement action. Because control studies typically involve only one or a few measures, they do not provide enough information to decide *how* a process should be corrected or improved. Control studies depend on subsequent causal analysis (see Section 4) to define appropriate actions once the need for action has been identified.

3.2 Does SPC Apply to Software Processes?

When modeling the performance of a complex system, the analyst has a choice between

(1) developing a comprehensive multi-factor model that encompasses all the important performance parameters, or,
(2) isolating individual factors for study separately.

The first choice may be implemented via analysis of variance or regression. The second choice may be accomplished via rational subgrouping and control charting (i.e., SPC). The SPC approach has the advantages of conceptual and mathematical simplicity. However, that does not mean that SPC is the best approach to all process analysis problems.

Some critics have argued that the appropriate conditions for applying SPC never obtain in software, especially due to the small quantities and human factors involved. Nevertheless, there are many examples of success, e.g., [20] and [49]. Pyzdek [43, p. 660] explains the issue as follows:

> Another example of a single-piece run is software development. The “part” is this case is the working copy of the software delivered to the customer. Only a single unit of product is involved. How can we use SPC here?
> Again, the answer comes when we direct our attention to the underlying process. Any marketable software product will consist of thousands, perhaps millions of bytes of finished machine code. This code will be compiled from thousands of lines of source code. The source code will be arranged in modules; the modules will contain procedures; the procedures will contain functions; and so on. Computer science has developed a number of ways of measuring the quality of computer code. The resulting numbers, called computer metrics, can be analyzed using SPC tools just like any other numbers. The processes that produced the code can thus be measured, controlled, and improved. If the process is in statistical control, the process elements, such as programmer selection and training, coding style, planning, procedures, etc., must be examined. If the process is not in statistical control, the special cause of the problem must be identified.

Clearly, SPC can be applied effectively to some software processes, even if it is not appropriate for analyzing all software process problems. Given that, let’s discuss some common control charting techniques and, then, consider the kinds of things that can go wrong in implementing SPC for software.

3.3 Control Charting Techniques

A control chart is a model of the expected performance of a subprocess or process element. A statistical model of data is called a probability distribution. Thus, control

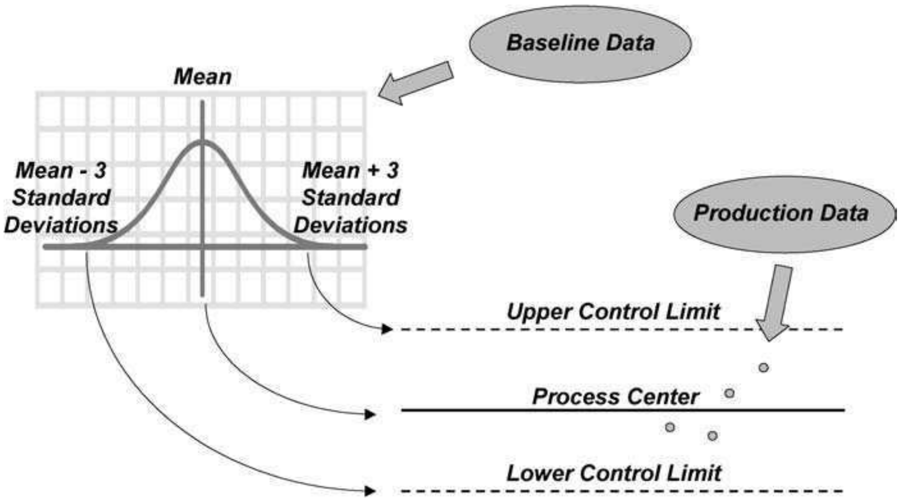


FIG. 5. Generation of a control chart from a probability distribution. (See Color Plate Section, Plate 33.)

charting involves defining a probability distribution from a baseline set of data that can be used to predict and evaluate future performance of a subprocess (or process element). Figure 5 shows how a probability distribution is transformed into a control chart for monitoring successive executions of a subprocess.

The two most common methods of control charting defect data (e.g., defects, defects per line, defects per hour) are the “U” chart and the “XmR” chart. These control charting techniques assume different underlying distributions. Often, the nature of the data distribution can be discovered by examining a histogram of the data. When significant amounts of the data generated by a process hover around some natural limit (e.g., defect rate can’t be less than zero), that usually is an indication of a Poisson (or related) probability distribution. In comparison, the Normal distribution is unimodal and symmetric.

3.3.1 U Chart

The “U” chart applies to Poisson-distributed data, with a potentially varying area of opportunity for the subject event (i.e., defect detection) to occur. Each execution of the process element under study results in some number of observations of an event or characteristic of interest (e.g., defect). The area of opportunity is a measure of the amount that the process has been exercised in order to observe the corresponding events (e.g., hours spent inspecting or lines inspected). The "U" chart tracks the rate of occurrence of the events from subgroup to subgroup.

The first step in building a "U" chart is to compute the centerline, $\bar{u}$, the average rate of occurrence across all the subgroups in the baseline data. The centerline is calculated as follows (where C is the number of occurrences and N is the area of opportunity in the baseline):

$$\bar{u} = \frac{C}{N}.$$

This estimate of $\bar{u}$ is the control parameter for the control chart. The standard deviation (s_i) of a sample with area of opportunity, n_i, drawn from a Poisson distribution with a mean of $\bar{u}$ is a function of $\bar{u}$, as follows:

$$s_i = \sqrt{\frac{\bar{u}}{n_i}}.$$

Because the standard deviation is a function of the mean, only $\bar{u}$ needs to be control-charted. This standard deviation is used to compute the control limits around the process center. The formulas for the control limits follow:

$$UCLi = \bar{u} + 3s_i, \qquad LCLi = \bar{u} - 3s_i.$$

The rate of occurrence for each subgroup (u_i) is plotted against this process center and control limits. Note that separate control limits must be computed for each subgroup (denoted by i). The process center and standard deviation are used to evaluate the stability of the process (using the baseline data from which they were calculated), as well as to determine whether the process element remains in control as it continues to be executed.

Managing processes that exhibit Poisson behavior poses some special challenges. Changing the mean changes the variation in the process. In particular, reducing the mean requires reducing the variability in the process. Because it seems more convenient to seek process improvements that adjust only the mean, analysts sometimes are tempted to substitute a technique based on the Normal distribution (i.e., "XmR" charts) to compute a standard deviation that appears to be independent of the mean. As a physical fact, whenever the measure of process performance involves a natural lower limit (e.g., zero defects), the standard deviation must be reduced in order to reduce the mean. Changing the statistical model does not change actual process behavior. The choice of statistical model (and control chart) should be driven by the nature of the data, not the preferences of the analyst.

3.3.2 XmR Charts

The Individuals and Moving Range ("XmR") charts apply to normally distributed data when each subgroup represents a single observation. Thus, the individual observations must be organized into time order. Periodic data often is a good candidate for analysis with the "XmR" charts. Because a Normal distribution is assumed, two control charts are required: one for the mean and one for the standard deviation, as both may vary over time. The average of the individual observations defines the process center for the first control chart. It is defined as follows:

$$\bar{X} = \frac{\sum_1^k X_i}{k}.$$

In order to calculate control limits, the standard deviation must be estimated. The population standard deviation cannot be estimated from a single observation, so the variation between successive observations is used. This is called the "moving range." The most common approach (the one discussed here) considers the variation between two successive observations, although calculations incorporating additional adjacent points can be used. The moving range based on two successive observations is calculated as follows:

$$\bar{R}^* = \frac{\sum_2^k |X_{i-1} - X_i|}{k - 1}.$$

The control limits for the mean depend on the calculated $\bar{R}^*$. However, $\bar{R}^*$ is a biased estimator of the population standard deviation. The following formulas for computing control limits include a bias adjustment factor:

$$UCL = \bar{X} + 2.66\bar{R}^*, \qquad LCL = \bar{X} - 2.66\bar{R}^*.$$

The second control chart has the estimate of the standard deviation, $\bar{R}^*$, at its center. The control limits around $\bar{R}^*$, based on two observations, are computed as follows:

$$UCL = 3.27\bar{R}^*, \qquad LCL = 0.$$

The coefficients given in the formulas for computing control limits around $\bar{X}$ and $\bar{R}^*$ assume that the moving range is computed from two consecutive observations.

Kan [33] suggests that because of the limited amounts of data and time pressure of software projects, control charts often cannot be used in the real-time mode of operation for which they are intended, but often must be used somewhat retrospectively—to analyze what has recently happened rather than what is happening at this moment.

3.4 Issues in Control Chart Application

Three basic issues confront the practitioner attempting to apply control charts to defect data:

(1) where to get the data,

(2) what data to analyze, and
(3) which chart to use in the analysis.

This section discusses those issues. For most software engineering organizations, the inspection (or peer review) process best meets the criteria for application of SPC. It is a relatively small subprocess that gets executed repeatedly and is relatively easy to instrument with data collection. Moreover, inspections can be implemented relatively early in the life cycle when measurement often is most difficult.

Because inspection events usually are not directly comparable (involving different amounts of effort and review material) simply charting the number of defects from inspections could be misleading. Consequently, the number of defects usually is normalized by dividing it either by the number of preparation/review hours or the size of the review package (e.g., lines or pages). The variable, *defects per hour*, has an important advantage. It applies universally—regardless of the artifact being inspected the hours can be counted in the same way. On the other hand, *defects per unit of size* requires the definition of a different size measure for each artifact type. Often getting these different definitions understood and applied consistently in practice becomes a challenge. Some organizations use "estimated lines of code" so that the same normalization factor can be applied across the life cycle. The problem here is that another source of uncertainty or variability (the estimation process) has been introduced into our data by using this estimate. Now, out of control situations can result from poor size estimates, as well as changes in defect insertion and detection performance. When size measures are used as the basis of normalization or inspection results, they should be measures of the actual artifacts being inspected (e.g., lines of text for requirements documents) rather than estimates of the completed software size.

Both the "U" chart and "XmR" charts have been applied to defect data from software inspections. The choice of control chart should depend on the underlying distribution of the data and its organization into subgroups [51]. It has been established for quite some time that software defects and failures exhibit Poisson behavior. That is the basis for most efforts at modeling software reliability, see for example [35]. Consequently, the earliest efforts to apply control charts to software, i.e., [22] and [12], made use of "U" charts. More recently, Weller [49] employed "U" charts in his analysis of data from a large commercial software project. Nevertheless, some recent work has applied the "XmR" method to defect data, e.g., [30] and [20].

Jalote and Saxena [31] argue that the control limits for defect rates produced by the "XmR" method are too wide. He suggests using control limits set to less than three standard deviations. The other obvious alternative is to use the "U" chart, which produces narrower limits for Poisson distributed data with a low rate of occurrence. For those processes with a high rate of occurrence, the Poisson and Normal distributions appear similar.

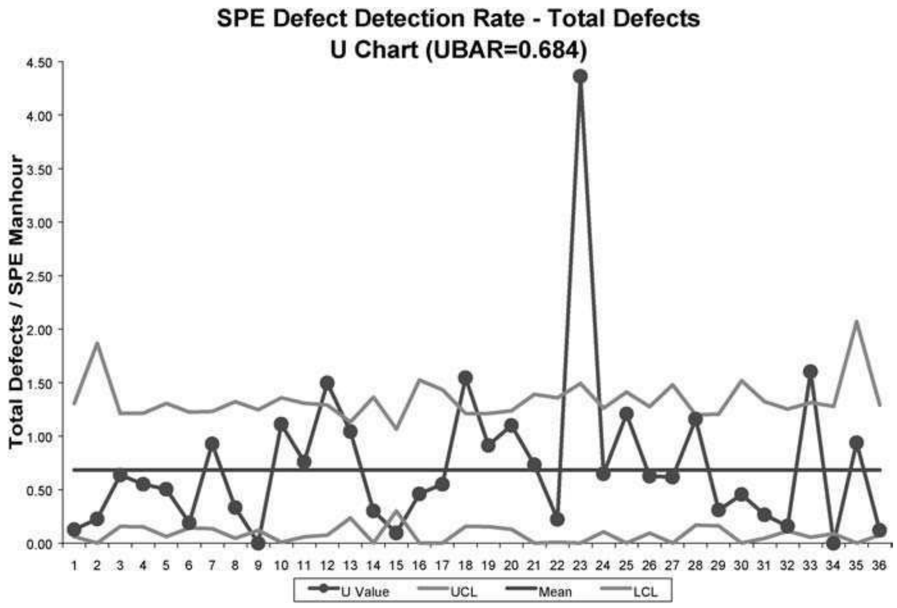


FIG. 6. U chart applied to defect data. (See Color Plate Section, Plate 34.)

The difference between the results of the two methods is illustrated in Figures 6 and 7. The same set of data from a real industry project is plotted in both figures. Each point plotted represents data from one inspection (or peer review), referred to as SPEs by this organization. The "U" chart (Figure 6) produces different control limits for each inspection, because the area of opportunity for finding defects (labor hours) can be different for each inspection. The result is control limits that are generally narrower than the limits in Figure 7. Note that some points fall out of control both above and below the limits in Figure 6. Thus, an inspection that consumed a lot of effort, but did not find any (or very few defects) could be considered out of control (on the low side). From a practical viewpoint, an inspection that found no defects sounds like an anomaly that deserves investigation, regardless of the statistics.

In contrast, Figure 7 only flags the single largest spike in the upper direction as out of control. Those situations in which significant amounts of effort were expended and no defects were found are omitted from further consideration. This is desirable if your goal is not to gain control of your process and reduce variability, but rather to demonstrate that your process is stable and that no further work is required. (This is a temptation for those whose efforts to apply SPC are driven primarily by a desire to achieve an external certification.) As Jalote and Saxena [31] point out, the cost of

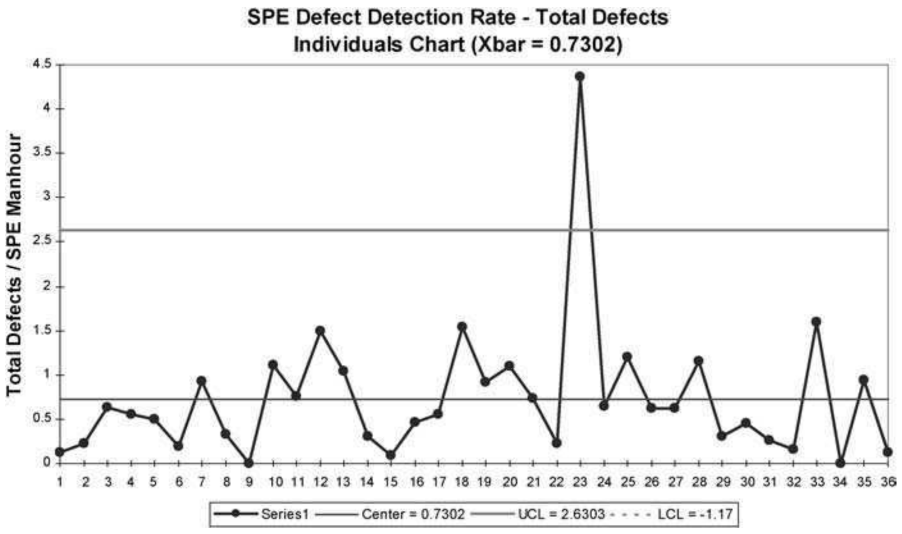


FIG. 7. Individuals chart applied to defect data. (See Color Plate Section, Plate 35.)

allowing a software process run out of control is so much greater than the effort to investigate problems and gain control, that the process manager should move towards tighter limits. The "U" chart accomplishes this in most practical situations.

Note that Figure 7 only shows the "individuals" chart of the "XmR" pair of charts. The "moving range" chart also should be constructed and analyzed for out of control situations. Those who are interested in *demonstrating* control, rather than *achieving* real control and reducing variability often just work with the individuals chart.

3.5 Common Mistakes in Control Charting

Two mistakes are commonly made in control charting. The first is combining data from multiple projects into the same control chart. The second is mixing different instantiations of the same defined process on the same control chart.

3.5.1 Control Charts at the Organizational Level

The term "organizational baseline" in the CMM and CMMI frequently is misunderstood to imply that data from multiple projects should be combined into one control chart to establish a common process performance baseline. The real intent is to develop a process performance summary, not to control process execution. Typically, this includes project process performance baselines, as well as other summary statistics. The organizational baseline is intended to support planning, not control of processes. Naively combining data from disparate projects does not facilitate effective planning.

Control charts based on multiproject data at the organizational level violate the rules of rational subgrouping as defined by Nelson [36]. Some of the specific reasons why this is not appropriate are as follows:

- This data is not available in real time so it cannot really be used to "control" anything. Communicating data from the projects to the organization and then returning the analysis results typically involves a substantial delay.
- If there is an out-of-control signal, it is difficult to identify its source because each project's process may be tailored and instantiated uniquely.
- Projects are not comparable to the degree that successive executions of a process element within a project are comparable. Different types of projects may exhibit substantial differences in performance in terms of productivity and quality, for example. A result that is unusual for one project may not be unusual for another project.
- Changes in the process of one project do not affect the results of other projects. Thus signals may be hidden by project results that cancel each other out or created by accidentally re-enforcing trends among different projects.
- The order of execution of activities among projects often cannot be determined unambiguously (a basic assumption of control charting) because they often overlap in time. Moreover, there often is little communication among the processes of different projects, so changes in one subprocess do not affect the coincident observations (a criteria for rational subgrouping).

The use of control charts to establish organizational process baselines for project processes is misleading and generally should be discouraged.

However, there are two situations in which organizational control charts make sense. First, organization-wide processes (e.g., training, quality assurance, and process definition maintenance) may be candidates for SPC; these are individual processes that are applied across multiple projects. Second, an organization that consists of many small projects working in the same application domain, and using the same personnel, methods, and tools often may be treated as one large project.

3.5.2 Different Instantiations of a Process

Process instantiations often are not distinguished from process descriptions. Only the performance of executing processes (instantiations) can be controlled and improved. "Improving" the clarity and consistency, for example, of the process definition may be a means of improving the performance of its instantiations. However, each different instantiation may lead to a different level of performance. Data from different instantiations of process elements should not be combined into one control chart. When performing optimization studies, use an analysis technique that allows for multiple instantiations (or levels of factors), for example, Analysis of Variance.

One problematic situation involves combining data from multiple types of inspections onto one control chart, even if all the inspections pertain to the same project. (Combining data from multiple projects only compounds the problem, as discussed earlier.) All these inspections may follow the same process definition (e.g., documented procedure). However, their performance may be very different because of differences in the teams (e.g., systems engineers, software engineers), artifacts inspected (e.g., requirements, design diagrams, code) and practice of inspection (e.g., amount of preparation, degree of training). Analyze each instantiation of the inspection process separately, unless the data indicates otherwise.

4. Learning and Improvement

The techniques of quality modeling and process control discussed in the previous sections help to predict quality performance within an activity or across the life cycle, and to recognize when anomalies have occurred that require further investigation. This section explains how defect information is used to understand the factors that determine the performance of a process and change that performance in the desired direction, either in an attempt to gain control of a process or to improve the performance of a stable process. Defect analysis helps an organization to learn from its past performance (especially mistakes), so that its performance can be improved [6]. One of the first systematic investigations of this type was undertaken by Basili and Weiss [1]. However, they focused on the problem of measuring defects rather than the nature of causal systems and the process of causal analysis.

4.1 Causal Systems

Searching for the cause of a problem (“laying the blame”) is a common human endeavor that wouldn’t seem to require much formalism. However, causal investigations often go wrong—beginning with a misunderstanding of the nature of causality. A causal system is an interacting set of events and conditions that produces recognizable consequences. Causal analysis is the systematic investigation of a causal system in order to identify actions that influence a causal system, usually in order to minimize undesirable consequences. Causal analysis may sometimes be referred to as root cause analysis or defect prevention. Causal analysis focuses on understanding cause-effect relationships.

Regardless of the specific terms and techniques employed causal analysis is likely to fail if the underlying concept of causality is not understood. Three conditions must be established in order to demonstrate a causal relationship (following Babbie [2]):

- First, there must be a correlation or association between the hypothesized cause and effect.
- Second, the cause must precede the effect in time.
- Third, the mechanism linking the cause to the effect must be identified.

The first condition implies that when the cause occurs, the effect is also likely to be observed. Often, this is demonstrated through statistical correlation and regression. While the second condition seems obvious, a common mistake in the practice of causal analysis is to hypothesize cause-effect relationships between factors that occur simultaneously. This is an over-interpretation of correlational analysis.

Figure 8 shows a scatter diagram of two variables measuring inspection (or peer review) performance. These two variables frequently demonstrate significant correlations. This diagram and a correlation coefficient computed from the data often are taken as evidence that preparation causes detection. (Preparation includes the effort spent reviewing and red-lining materials prior to the actual inspection meeting.)

However, most inspection defects are discovered *during* preparation. Both meters are running simultaneously. Thus, preparation performance cannot substantially influence detection performance. They are measures of the same activity. Rather, the

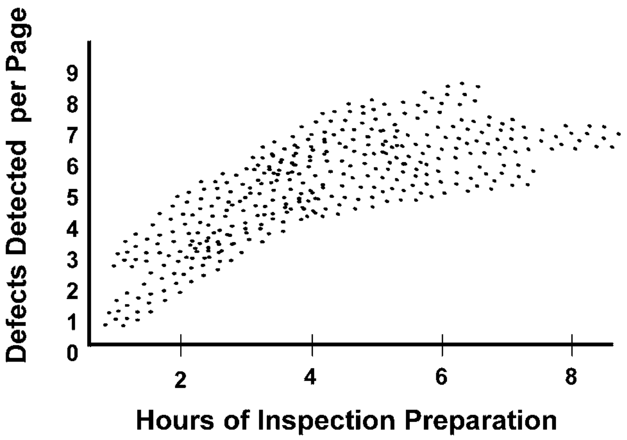


FIG. 8. Example of correlation between variables. (See Color Plate Section, Plate 36.)

correlation suggests that some other factor affects both preparation and detection. Issuing a mandate (as a corrective action) to spend more time in preparation may result in more time being charged to inspections, but it isn't likely to increase the defect detection rate. The underlying cause of low preparation and detection rates may be a lack of understanding of how to review, schedule pressure, or other factors that affect both measures. That underlying cause must be addressed to increase both the amount of preparation and detection rate. Recognition of the correlation relationship helps to narrow the set of potential causes to things that affect both preparation and detection performance. It does not demonstrate a causal relationship.

The relationship between the height and weight of adult human beings provides a good analogy to the situation described in Figure 8. Taller people tend to weigh more than shorter people. (Obviously, other factors intervene, as well.) While this is a necessary relationship, it is not a causal relationship. It would be a mistake to assume that increasing someone's weight would also increase his/her height. Both variables are determined by other causes (for example, genetics and childhood nutrition). Those underlying causes are the ones that need to be identified and manipulated in this causal system.

Some of the responsibility for this kind of misinterpretation can be attributed to statisticians. The horizontal and vertical axes of Figure 8 are typically referred to as the "independent" and "dependent" variables respectively. While these terms are simple labels, not intended to imply a causal relationship, the terminology is often misunderstood. Pearl [39] discusses the mathematical and statistical aspects of causal modeling in more detail.

Satisfying the third condition requires investigating the causal system. Many good examples of causal analysis efforts in software engineering have been published (e.g., [34,15,52,32]). However, these efforts have adopted different terminology and approaches. In particular, the elements of a "causal system" have not been defined in a consistent way. The differences between these analyses obscure the commonality in the subject matter to which the procedures are applied.

One of the consequences of a poor understanding of the nature of causal systems and causal analysis is that causal analysis sessions become superficial exercises that don't look deeply enough to find the important causes and potential actions that offer real leverage in changing performance. This reduces the cost-benefit of the investment in causal analysis expected of mature software organizations. This section describes a model of causal analysis and a set of supporting terms that has evolved from extensive experience with the software industry. Some of these experiences with causal analysis were summarized in [7]. This experience encompasses scientific data processing software, configuration management, and other software-related processes.

4.1.1 Elements of a Causal System

A cause-effect relationship may be one link in an effectively infinite network of causes and effects. A richer vocabulary than just “causes” and “effects” is needed to help us determine appropriate starting and stopping points for causal analysis. The model and terminology described in this section facilitate reasoning about causal systems and planning for causal analysis. Figure 9 describes the key elements of a causal system. Most of the approaches to causal analysis previously cited don’t explicitly address all these elements of a causal system.

As indicated in the figure, an investigation of a causal systems include three classes of elements:

- Objectives—our purposes in investigating the causal system.
- Observations—the events and conditions that comprise the causal system.
- Actions—our efforts to influence the behavior of the causal system.

Observations are events and conditions that may be detected. Building an understanding of a causal system requires identifying these events and conditions, as well as discovering the relationships among them. Observations include:

- Symptoms—these are undesirable consequences of the problem. Treating them does not make the problem go away, but may minimize the damage.
- Problem—this is the specific situation that, if corrected, results in the disappearance of further symptoms.

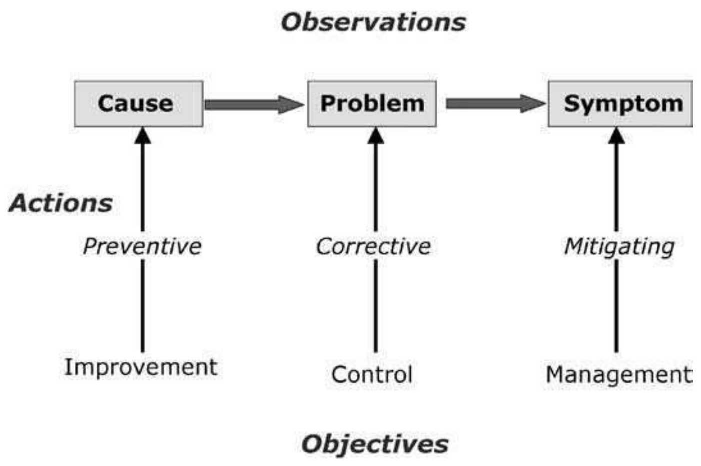


FIG. 9. Elements of a causal system. (See Color Plate Section, Plate 37.)

- Causes—these are the events and conditions that contribute to the occurrence of the problem. Addressing them helps prevent future similar problems.

Note that both problems and symptoms are effects of one or more underlying causes. Once a causal system is understood, action can be take to change its behavior and/or impact on the organization. Actions may be of three types:

- Preventive—reducing the chances that similar problems will occur again.
- Corrective—fixing problems directly.
- Mitigating—countering the adverse consequences (symptoms) of problems.

The corrective type usually includes actions to detect problems earlier, so that they can be corrected before they produce symptoms. The optimum mix of preventive, corrective, and mitigating actions to be applied to a causal system depends on the cost of taking the actions as well as the magnitude of symptoms produced. Attacking the cause, itself, may not be the course of action that produces the maximum cost-benefit in all situations. Potential symptoms and mitigations may be addressed as part of risk management activity.

Three objectives or motivations for undertaking causal analysis are common:

- Improvement—triggered by recognition of an opportunity or identification of a performance improvement goals.
- Control—triggered by an outlier or usual result relative to past performance (e.g., out of control situation from a control chart, as discussed previously).
- Management—triggered by a departure from plans or targets (e.g., analysis of defect profile, as previously discussed).

Regardless of the motivation for causal analysis, all elements of the causal system (as described above) should be considered.

Most real causal systems are more complex than Figure 8 suggests. That is, a specific problem may produce multiple symptoms. Moreover, many causes may contribute to the problem. Consequently, many different actions of all types may be possible. The “fish bone” diagram [28] is a popular tool for describing and reasoning about causal systems.

These general concepts of causal systems can be applied to the investigation of any undesirable situation, not just to the investigation of defects. A good understanding of the basic concepts and terminology of causal systems helps to overcome the difficulties inherent in implementing a practice that seems “obvious.”

Effective causal analysis is becoming ever more important to the software industry as process maturity increases and new forces, such as Six Sigma [24] focus increasing attention on quality improvement. Academic researchers, especially those conducting empirical studies, also may benefit from thinking a little more systematically about causal systems. Application of the concepts and terminology presented here help ensure that causal systems get fully investigated and effective actions are taken.

4.2 Defect Causal Analysis

As stated previously many different approaches to causal analysis have been applied to software defects. Mays et al. [34] published one of the first approaches. Card [7] elaborated on the Mays approach. More recently, Rooney and Heuvel [46] described a generic method of causal analysis, based on experience in the nuclear power industry. The Mays and Card procedures are similar to the Rooney and Heuvel procedure. Yu [52] and Leszak [32] published results of causal analyses of industry data, but did not describe re-useable procedures in any detail.

This section summarizes the basic concepts of causal analysis as applied to defects, based on the procedures of Mays, Card, and Rooney and Heuvel. Three common principles drive these DCA-based approaches to quality improvement:

- *Reduce defects to improve quality*—while there are many different ideas about what quality is, or which "-ility" is more important (e.g., reliability, portability), everyone can probably agree that if there are lots of defects, then there is probably a lack of quality—whatever that is. Thus, we can improve software quality by focusing on the prevention and early detection of defects, a readily measureable attribute of software.
- *Apply local expertise*—the people who really understand what went wrong are the people who were present when the defects were inserted—the software engineering team. While causal analysis can be performed by outside groups such as quality assurance, researchers, or a software engineering process group, those people generally don't know the detailed circumstances that lead to the mistake or how to prevent it in the future.
- *Focus on systematic errors*—most projects have too many defects to even consider conducting a causal analysis of all of them. However, some mistakes tend to get repeated. These "systematic errors" account for a large portion of the defects found in the typical software project. Identifying and preventing systematic errors can have a big impact on quality (in terms of defects) for a relatively small investment.

There are relatively few pre-requisites for implementing DCA. They include the following:

- Mistakes must have been made (this is not usually a difficult pre-requisite to satisfy).
- Mistakes must have been documented through problem reports, inspection results, etc.
- Desire must exist to avoid mistakes (or at least avoid the negative feedback associated with mistakes).
- Basic software process must be defined to provide a framework for effective actions.

As will be discussed later, the software engineering process does not have to be fully defined in detail in order for DCA to provide value. However, without at least a basic process framework it becomes very difficult to understand where and how defects enter the software and which actions are most likely to prevent them or find them earlier.

4.2.1 Overview of DCA Procedure

The usual approach to software defects is to fix them and then forget them. DCA provides a mechanism for learning from them. DCA may be triggered by an anomaly in a defect profile or an out of control situation on a control chart. In these cases, the data associated with the anomaly becomes the subject of the causal analysis. When the objective is continuous process improvement, samples of defects are drawn, periodically, for periodic causal analysis meetings. The meetings produce recommendations for an action team. The proposals may be either short-term (i.e., for immediate action by the project) or long-term (i.e., for the benefit of future projects). The longer-term proposed action should be integrated into the organization's other process improvement efforts. Usually, the impacts of causal analysis and resulting actions can be measured within a few months of start.

The Causal Analysis Team is the focus of the DCA process. Most of its work is performed during a causal analysis meeting. During this activity the software team analyzes problems and makes recommendations for improvements that will prevent defects or detect defects earlier in the process. The causal analysis team should be made up of the software producers (developer and maintainers) who have the greatest intimacy with the product and process. Meetings typically last about two hours. The causal analysis meeting should be lead by a designated moderator or facilitator. The role of the facilitator is to hold the team to its agenda while preserving group integrity. The facilitator could be a respected member of the team, a member of the software engineering process group, a tester, or a member of another causal analysis team.

The typical agenda for a DCA meeting includes six steps:

1. *Select problem sample*—When DCA is triggered by an anomaly in a defect profile or an out of control situation on a control chart, the data associated with the anomaly becomes the subject of the causal analysis. Causal analysis for continuous improvement should consider the general population of defects. Since most projects have more problems than they can afford to analyze, a sample must be selected for consideration during the causal analysis meeting. Few teams can handle more than 20 problems in a two hour meeting. The problems should be as representative of the team's work as possible.

2. *Classify selected problems*—Classifying or grouping problems helps to identify clusters in which systematic errors are likely to be found. Three dimensions are especially useful for classifying problems:

- When was the defect that caused the problem inserted into the software?
- When was the problem detected?
- What type of mistake was made or defect was introduced?

The first two classification dimensions correspond to activities or phases in the software process. The last dimension reflects the nature of the work performed and the opportunities for making mistakes. Some commonly used types of errors include interface, computational, logical, initialization, and data structure. Depending on the nature of the project, additional classes such as documentation and user interface may be added. ODC [13] offers a comprehensive classification scheme for DCA purposes. Tables or Pareto charts may be produced to help identify clusters of problems. Systematic errors are likely to be found in the most common defect types.

3. *Identify systematic errors*—A systematic error results in the same or similar defects being repeated on different occasions. Usually systematic errors are associated with a specific activity or part of the product. Ignore the effects of individual defects in seeking out systematic errors. Card [7] showed several examples of actual systematic errors. In each of these cases, 20 to 40 percent of the defects in the samples examined during the causal analysis meeting were associated with the systematic error. The fact that large numbers of defects result from these mistakes provides the motivation for changing the process.

4. *Determine principal cause*—Many factors may contribute to a systematic error. Usually it is uneconomical to attempt to address them all, so attention must be concentrated on the principal cause. It is during this stage that mastery of the concepts of causality described in Section 4.1 becomes essential to success. "Fish bone" (or Ishikawa) diagrams may be developed at this stage. Causes usually fall into one of four categories [28]:

- *Methods* (may be incomplete, ambiguous, wrong, or unenforced).
- *Tools/environment* (may be clumsy, unreliable, or defective).
- *People* (may lack adequate training or understanding).
- *Input/requirements* (may be incomplete, ambiguous, or defective).

The causal categories help to group related items, as well as to identify general areas of the software process that may need attention. Like the defect classification scheme, the causal categories should be adapted as necessary to support the analysis and reporting needs of the organization implementing DCA.

5. *Develop action proposals*—Once the principal cause of a systematic defect has been found, action proposals must be developed that will promote either prevention or earlier detection of the systematic defect. Only a small number of relevant actions is needed. Too many actions can overwhelm management's ability to process them. One good action proposal that gets implemented is worth more than any number waiting in a queue. Action proposals should be specific and address the systematic error. DCA differs from generic process assessment methods in that its focus is on identifying specific actions rather than suggesting broad areas for increased process improvement attention.

6. *Document meeting results*—Records of meeting results are necessary to ensure that actions get implemented. In most cases a simple form-based report will suffice. The causal analysis results are provided to an action team.

Most recommendations produced by the causal analysis team can't be implemented without management support. To get any benefit from DCA, an action team must be formed that meets regularly to consider proposed actions. Multiple causal analysis teams may feed into one action team. The role of the action team includes:

- Prioritizing action proposals. (Few organizations can afford to implement all proposals at once.)
- Resolving conflicts and combining related proposals (especially if multiple causal analysis teams are operating).
- Planning and scheduling the implementation of proposals.
- Allocating resources and assigning responsibility for implementation of proposals.
- Monitoring the progress of implementation and effectiveness of actions.
- Communicating actions and status to the causal analysis teams.

The benefits of DCA are lost without timely action. It's usually advisable to identify the members of the action team before conducting any causal analysis meetings.

Because DCA is a commonsense idea, organizations sometimes attempt to "just do it," without adequate preparation. Card [7] provides a more detailed description of the causal analysis procedure, as well as recommendations for successfully implementing it.

4.2.2 Benefits of DCA

Defect causal analysis links the analysis of defect profiles and control charts to actions. Only actions produce benefits. Measuring and analyzing seldom produce improvement by themselves. Thus the benefits of defect analysis, in general, must be viewed in terms of the actions resulting from causal analysis. The simplest ways of determining the effectiveness of improvement actions on quality are to track the overall defect rate and the distribution of error types.

Mays et al. [34] and Leszak et al. [32] described the application of DCA to projects involving hundreds of staff. Dangerfield et al. [15] and Yu [52] described experiences with smaller projects. In all cases, defect rates declined substantially, often by as much as 50 percent over a two-year period. Of course, not all defects are equal in terms of their impact on the user and the success of the software product. For example, certain parts of a system may be more critical to its operation. Nevertheless, reducing defects in general reduces critical defects, too. Moreover, reducing systematic sources of defects usually also provides benefits in terms of reducing development cost and cycle time.

The distribution of error types may be a more sensitive indicator of process changes in the short-term than the overall defect rate. For example, if a systematic error had been found among problems classified as "interface," and an effective action for preventing this systematic error had been implemented, then the proportion of problems in this class should go down over time. This change in distribution often can be detected earlier than the change in defect rate.

Other, more subjective, benefits of DCA include the following:

- *An awareness of quality*. Participating in causal analysis and action meetings makes software quality tangible to managers and producers alike. They gain a practical understanding of the consequences of quality.
- *A commitment to the process*. Evidence accumulated through DCA helps show producers the value of conforming to the process. For example, Dangerfield et al. [15] reported that about two-thirds of all systematic errors were associated with software methods. Moreover, the majority of this problem type were caused by a failure to follow some element of the defined process or to communicate important information. Conducting DCA convinces the producers of the value of an effective process.

- *An understanding of quality measurement.* When producers begin analyzing problems and implementing improvements via DCA, they begin to understand the value of quality data. Their fears about misuse of data decline, because they are users.

DCA isn't expensive, but it does require some investment. The cost of operating a DCA program, from causal analysis through implementing actions, ranges from 0.5 percent [34] to 1.5 percent [15] of the software budget. In addition to this operational cost, investment must be made to start up the DCA program. This includes providing training, defining classification schemes, setting up procedures, and establishing mechanisms for data collection. This implementation cost depends on what relevant resources the organization already possesses as well as how widely DCA is to be implemented. Nevertheless, the experiences cited here show that if quality is important to your organization, then DCA is well worth the investment.

DCA is a low cost strategy for improving software quality. It has proven itself effective in many different organizational settings. While the benefits of DCA take time to realize, implementing the project-level approach described here typically produces measureable results within a matter of months. Of course, without timely follow through by the action team, no benefits will be obtained at all.

Remember that this approach to DCA is based on sampling, not an exhaustive analysis of all problems reported. A systematic error is likely to be represented in this sample, or the next. Actions should focus on the systematic errors—a subset of the sample. There are examples of organizations that have conducted causal analyses of 200 problems and produced 500 proposed actions. That quantity of recommendations is likely to overwhelm management's ability to react. Action is more likely if fewer high-leverage actions are proposed.

DCA is readily adapted to other tasks besides software production. For example, the goal of testers is to find all defects before the system gets fielded. Any problem reported from the field represents a defect for the testers. This information can be used in the DCA process to identify improvements to the testing process that could find those defects before they get fielded.

Many of the actions proposed by the DCA teams will be small incremental improvements rather than revolutionary ideas. Unfortunately, much of our professional lives are occupied in dealing with problematic trivia—small things that go wrong, but that take up lots of time. Reducing these hindrances allows an organization to perform at its true potential.

5. Summary and Conclusions

Recent industry trends are promoting the increasing use of defect analysis and statistical methods, to industrial software processes [8]. These trends include the adoption of the Capability Maturity Model for Software [37], Capability Maturity Model—Integrated [14], Six Sigma [24], and ISO Standard 9001 [27]. The techniques that have proven important to industry in this context include defect profiles, statistical process control, and defect causal analysis. The preceding discussion identified numerous successful examples of the application of each. Together, these techniques provide a feedback system that helps to control and improve an industrial software engineering process.

Table II summarizes the state of industry adoption and the level of research activity for the defect analysis techniques discussed earlier in this chapter. The segment of industry considered for this comparison does not include all software engineering organizations, only mature organizations that are actively trying to perform defect analysis. Recent data from the Software Engineering Institute suggest that about 20 percent of organizations pursuing CMM or CMMI-based process improvement fall into the high maturity category, and of these, 100 percent are measuring and analyzing defects [38].

Despite this industry interest, the background research for this chapter identified much less published academic research activity in defect analysis techniques than in other techniques such as software reliability or complexity modeling. Many of the references cited for this chapter are industry case studies, rather than true research efforts. Table II shows the author's assessment of the level of research in the topics discussed in this chapter. One of the difficulties of summarizing the status of defect analysis is that much of the work is being done in industry where it is less likely to get published and formally critiqued.

While this chapter is intended to provide an overview of the concepts and a guide to the literature for those interested in applying defect analysis in industry, it may also help researchers to identify important new areas of focus. Because of its economic

TABLE II
RESEARCH AND APPLICATION OF DEFECT ANALYSIS TECHNIQUES

Technique	Industry adoption	Research activity
Empirical defect profiles	Wide	Nearly none
Analytic defect profiles	Limited	Low
Orthogonal defect classification	Limited	Low
Capture-recapture models	None	Moderate
Control charts	Wide	Low
Causal analysis methods	Wide	Nearly none

importance, defect analysis needs to be approached more rigorously and objectively than it often has been in practice. Academic researchers can help to fill that gap by studying the underlying principles and evaluating the statistical techniques of defect analysis in the software engineering context. The industrial feedback process described in this chapter offers many opportunities for further research.

References

[1] Basili V.R., Weiss D.M., "A methodology for collecting valid software engineering data", *IEEE Transactions on Software Engineering* **10** (6) (1984).

[2] Babbie E., *The Practice of Social Research*, Wadsworth Publishing, Belmont, CA, 1986.

[3] Bhandari I., "In-process improvement through defect data interpretation", *IBM Systems Journal* **33** (1) (1994).

[4] Bridge N., et al., "Orthogonal defect classification using defect data to improve software development", *ASQ Software Quality Newsletter* (March 1998).

[5] Burr A., Owen M., *Statistical Methods for Software Quality*, International Thompson Computer Press, 1996.

[6] Card D., "Defect causal analysis drives down error rates", *IEEE Software* (July 1993).

[7] Card D., "Learning from our mistakes with defect causal analysis", *IEEE Software* (January 1998).

[8] Card D., "Sorting out six sigma and the CMM", *IEEE Software* (July 2000).

[9] Card D., "Quantitatively managing the object-oriented design process", in: *Canadian National Research Council Conference on Quality Assurance of Object-Oriented Software*, 2000.

[10] Card D., "Managing software quality with defects", *Crosstalk* (March 2003).

[11] Card D., Agresti W., "Resolving the software science anomaly", *Journal of Systems and Software* (1990).

[12] Card D., Berg R.A., "An industrial engineering approach to software development", *Journal of Systems and Software* (October 1989).

[13] Chillarge R., et al., "Orthogonal defect classification", *IEEE Transactions on Software Engineering* (November 1992).

[14] Chrissis M.B., et al., *CMMI—Guidelines for Process Integration and Product Improvement*, Addison–Wesley, Reading, MA, 2003.

[15] Dangerfield O., et al., "Defect causal analysis—a report from the field", in: *ASQC International Conference on Software Quality*, 1992.

[16] Eick S.G., Loader S.R., Long M.D., Votta L.G., Vander Wiel S.A., "Estimating software fault content before coding", in: *Proceedings of IEEE 14th International Conference on Software Engineering*, 1992.

[17] El Emam K., "The confounding effect of class size on the validity of object oriented metrics", *IEEE Transactions on Software Engineering* (2001).

[18] El Emam K., Laitenberger O., "Evaluating capture–recapture models with two inspectors", *IEEE Transactions on Software Engineering* (September 2001).

[19] Florac W., Carleton A., *Measuring the Software Process: Statistical Process Control for Software Process Improvement*, Addison–Wesley, Reading, MA, 1999.
[20] Florac W., et al., "Statistical process control: Analyzing a space shuttle onboard software process", *IEEE Software* (July 2000).
[21] Gaffney J., "On prediction of software-related performance of large-scale systems", in: *CMG XV*, 1984.
[22] Gardiner J.S., Montgomery D.C., "Using statistical control charts for software quality control", *Quality and Reliability Engineering International* (1987).
[23] Harbaugh S., "Crusader software quality assurance process improvement", Integrated Software, Inc., Technical Report, 2002.
[24] Harry M., Schroeder R., *Six Sigma*, Doubleday, New York, 2000.
[25] Hays W., Walker R., *Statistics: Probability, Inference, and Decision*, Holt, Rinehart, and Winston, New York, 1970.
[26] International Organization for Standardization, "ISO/IEC Standard 9126, Information Technology—Software Quality, Part 1", 1995.
[27] International Organization for Standardization, "ISO Standard 9001: Quality Management Systems", 2000.
[28] Ishikawa K., *Guide to Quality Control*, Asian Productivity Organization Press, 1986.
[29] Isoda S., "A criticism on the capture-and-recapture method for software reliability assurance", *Journal of Systems and Software* **43** (1998).
[30] Jacob A.L., Pillal S.K., "Statistical process control to improve coding and code review", *IEEE Software* (May 2003).
[31] Jalote P., Saxena A., *IEEE Transactions on Software Engineering* (December 2002).
[32] Leszak M., et al., "Classification and evaluation of defects in a project perspective", *Journal of Systems and Software* (April 2002).
[33] Kan S.H., *Models and Metrics in Software Quality Engineering*, Addison–Wesley, Reading, MA, 1995.
[34] Mays R., et al., "Experiences with defect prevention", *IBM Systems Journal* (January 1990).
[35] Musa J.D., Iannino A., Okumoto K., *Software Reliability, Measurement Prediction and Application*, McGraw–Hill, New York, 1987.
[36] Nelson L.S., "Control charts: Rational subgroups and effective applications", *Journal of Quality Technology* (January 1988).
[37] Paulk M., et al., *Capability Maturity Model*, Addison–Wesley, Reading, MA, 1994.
[38] Paulk M., Goldensen D., White D., *The 1999 Survey of High maturity Organizations*, Software Engineering Institute, 2002.
[39] Pearl J., *Causality: Models, Reasoning, and Inference*, Cambridge University Press, Cambridge, UK, 2000.
[40] Petersson H., et al., "Capture–recapture in software inspections after 10 years research—theory, evaluation, and application", *Journal of Systems and Software* **72** (2004).
[41] Phan D.D., et al., "Managing software quality in a very large software project", *Information and Management* **29** (1995).
[42] Putnam L.H., "A generic empirical solution to the macro sizing and estimation problem", *IEEE Transactions on Software Engineering* **4** (4) (1978).

[43] Pyzdek T., “The Six Sigma Handbook”, McGraw–Hill, New York, 2003.
[44] Rallis N.E., Lansdowne Z.F., “Reliability estimation for a software system with sequential independent reviews”, *IEEE Transactions on Software Engineering* (December 2001).
[45] Remus H., Ziles S., “Prediction and management of program quality”, in: *Proceedings of IEEE Fourth International Conference on Software Engineering*, 1979.
[46] Rooney J.J., Vanden Heuval L.N., “Root cause analysis for beginners”, *ASQ Quality Progress* (July 2004).
[47] Shewhart W.A., *Economic Control of Manufactured Product*, 1931.
[48] Software Productivity Consortium, “SWEEP users’ guide, SPC-98030-MC”, 1997.
[49] Weller E.F., “Practical applications of statistical process control”, *IEEE Software* (June 2000).
[50] Westland J.C., “The cost behavior of software defects”, *Decision Support Systems* **37** (2004).
[51] Wheeler D., Chambers D.S., *Understanding Statistical Process Control*, SPC Press, 1992.
[52] Yu W., “A software fault prevention approach in coding and root cause analysis”, *Bell Labs Technical Journal* (April 1998).

Function Points

CHRISTOPHER J. LOKAN

School of Information Technology and Electrical Engineering
UNSW@ADFA
Northcott Drive
Canberra ACT 2600
Australia
c.lokan@adfa.edu.au

Abstract
Functional size measurement—measuring the functionality delivered by a software application to its users—is vital to software project managers. Its uses include estimation, managing scope in project planning and tracking, and measuring productivity.

There is no direct scale for measuring software functionality. One must instead decide which aspects of software to measure, that seem to capture functionality; how to measure those aspects; and how to combine the measurements into an overall measure of application size.

Many methods for how to do this have been proposed, beginning with Allan Albrecht's invention of Function Point Analysis in 1979. The methods vary in what things are measured and how the measuring is done; and also in usefulness, applicability, and industry acceptance. Two methods have achieved significant industry acceptance, and a third is on its way.

In this chapter we trace the evolution of functional size measurement, from Albrecht's original ideas through to the recent development of ISO standards. We describe Albrecht's method, and what has been learned about it through extensive experience and empirical research. We present some interesting variants on Albrecht's method that did not achieve widespread success, as well as the two other methods that can now be described as mainstream. We describe research concerning function points, and conclude with thoughts on the current status of functional size measurement and some expectations for future work.

ADVANCES IN COMPUTERS, VOL. 65
ISSN: 0065-2458/DOI 10.1016/S0065-2458(05)65007-3

1. Introduction

Throughout the history of software measurement, one attribute of computer software has always been seen as fundamental: its size.

Software is written to solve problems. Size measurement can focus on either the problem, or the program:

- The problem is expressed in the Software Requirements Specification. This document states the requirements that the delivered software must satisfy. It defines the *functionality* to be delivered by the software to the user.

  Problem size can be interpreted as the amount of functionality delivered by the software. Intuitively, software that lets the user do more things has greater functionality, and solves a larger problem.
- The program that solves the problem has a physical size. Various measures of program *length* capture this aspect of size.

Length and functionality are different aspects of size (although they tend to increase together). Programs with the same functionality can differ widely in length, and programs of the same length can deliver very different functionality.

Measuring length is easy (though there are issues to be careful of). Measuring functionality is harder. This chapter aims to give the reader an understanding of how software functionality is measured. We begin with a brief comparison of length and functionality measures, to demonstrate the need for measures of functionality.

1.1 Length vs. Functionality

The oldest measure of software size is simply the number of lines of code (“LOC”).

Which lines to count (all lines? executable lines only? should comments be included?) is a matter of argument. The definition most commonly adopted is

> any line of program text that is not a blank line or comment, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, and executable and non-executable statements [18].

This is simply the physical length of the program, as a compiler sees it.

Physical LOC is determined by program layout, which can be arbitrary in many languages. Other measures have been proposed that appear less arbitrary. The main ones are the number of tokens that make up the program text, and the number of statements, or logical lines, in the program. In principle, these are much better measures of length: they measure the semantic units in the program, and are less prone to the vagaries of individual programming styles. In practice, these measures are all highly correlated with each other, and with physical LOC. The theoretical deficiencies of physical LOC are not such a problem in practice, with respect to other length measures.

Length measures are valuable to developers. There is a long history of using them to predict things of interest about a software development project (like number of errors, development effort, maintenance effort, etc.). They are genuine measures of packaging requirements for finished software [33], and the storage required for source code [25]. Finally, they are easy to measure: many tools exist to count program length.

Length measures have been widely criticized (see for example Jones [58]). Criticisms include:

- sensitivity to individual programming style;
- lack of standard definitions;
- language dependent;
- difficult to apply to 4GLs, spreadsheets, application generators, etc.;
- difficulty of dealing with multiple languages;
- inability to measure specifications;
- not available until late in the life cycle, and difficult to estimate earlier;
- not relevant to users.

These criticisms are all well founded, to differing degrees. The first turns out not to matter so much in practice: in programming teams, individual variations tend to average out, and coding standards and pretty-printing software can also reduce variation. The next four can be largely overcome by applying some common sense. But the last three represent a real problem, corresponding to a shift in perspective from that of the programmer to that of the user. Program length means nothing to users.

The alternative is to try to define "functional measures" that measure the functionality of software, from a user's point of view. This involves identifying things in the problem specification that represent functionality, and somehow measuring those things.

If successful in their aims, functional measures bring several benefits [58]:

- understandable to users, and based on things that users care about;
- available early, at the specification stage;
- independent of programming language and development technology;
- reflective of real economic productivity.

Effective functional measures provide a more meaningful basis than length measures for things like measuring or benchmarking software development productivity, and quantifying an organization's software portfolio. They are also used effectively in software acquisition contracts, enabling the scope of a requirement—or change to a requirement—to be properly assessed.

Numerous functional size measures have been proposed, varying in what things are measured and how they are measured. The rest of this chapter describes the most important of those proposals.

1.2 Outline of this Chapter

When people talk about functional size measures, they usually mean "function points"—more specifically, "IFPUG function points" ("IFPUG" stands for International Function Point Users Group). Section 2 describes IFPUG function points and their evolution.

Section 3 describes what has been learned, through experience and empirical research, about IFPUG function points. This section identifies the strengths, weaknesses, and limitations of IFPUG function points. It sets the scene for following sections that describe other proposals for functional size measurement.

Section 4 describes Mark II Function Points—until recently, the only alternative to IFPUG function points with significant industry acceptance. Section 5 presents a few other proposals, which featured important ideas but for one reason or another never achieved wide usage. Section 6 describes COSMIC Full Function Points, emerging now as a significant alternative.

Object orientation is important in today's software development world. Applying function points to object oriented software is an area of current research interest. Section 7 describes the main activity in this area.

Section 8 looks briefly at recently-approved ISO standards for functional size measurement in general, and some function point approaches in particular.

Section 9 draws everything together and presents some thoughts on future directions.

2. Albrecht/IFPUG Function Points

Function points were introduced by Allan Albrecht, of IBM.

Albrecht's aim was to measure application development productivity in IBM's DP Services organization. His landmark paper from 1979 [8] is really about measuring productivity. He begins with a description of the application development process used by the DP Services organization. He comments on why to measure productivity, and some things to be careful about when measuring productivity. He describes the measures he uses for product and cost. He presents and interprets trends in productivity in IBM's environment.

It is a good paper about measuring productivity. But it is remembered for one of the measures he used. He states:

> To measure productivity we had to define and measure a product and a cost. The product that was analyzed was function value delivered . . . The cost used was the work-hours contributed.

For the product measure, Albrecht wanted to avoid measures "such as lines of code that can have widely differing values depending on the technology used." His objective was

> to develop a relative measure of function value delivered to the user that was independent of the particular technology or approach used.

He explained how his measure of function value was constructed, and gave it a name: *function points*.

2.1 The 1979 Definition

Albrecht defined a two-stage process for calculating function points.

1. Count certain factors that are the outward manifestations of an application, and weight those counts by numbers designed to reflect the function value to the customer.
2. Adjust that result for the effect of other factors relating to the technical complexity of the application.

The result is labelled as "function points."

The keys to this process are: which factors are considered to represent the outward manifestations of the application; and what weights to give to each factor.

Albrecht listed four factors, or types of component: external user inputs, outputs, inquiries, and master files. These four component types were identified from five years of experience with projects in IBM's systems development environment.

The relative weights given to inputs, outputs, inquiries, and master files were 4, 5, 4, and 10 respectively. These weights were determined "by debate and trial."

For the technical complexity adjustment phase, ten factors were identified. These were the extent to which the design or implementation of the application involved: reliable backup, data communications, distributed processing functions, performance considerations, a heavily used operational configuration, online data entry, data entry involving complex transactions, online update of master files, complex data and transaction components, and complex internal processing. Each of these ten factors could affect the function point value by 5%. The maximum possible adjustment was 50%, expressed as plus or minus 25% so that average technical complexity resulted in no change to the original function point value.

Validation of function points took the form of showing empirical relationships between function points and other measures of interest. Relationships were demonstrated between function points and effort, and between function points and LOC for three languages [8,10].

Some essential features, common to all function point approaches since, are visible here. The focus is on externally-visible aspects of an application, that a user can see. A measure is sought that is independent of the technology or approach used for implementation. The structure involves identifying, counting, and weighting important elements.

Note also that this definition of function points is oriented towards file-oriented data processing; and that there are several subjective aspects involved in the calculation.

2.2 The 1984 Revision

Function points did not last long in the form that Albrecht initially defined them. Albrecht's 1983 paper with Gaffney [10] described a revised structure, which was documented fully in an IBM report published in November 1984 [9].

This revised formulation of function points is much more elaborate than the 1979 definition. It still has subjective aspects, though guidelines were provided to try to reduce this. The method is still geared towards file-oriented data processing.

There are three main changes from 1979:

- There are five component types, not four, as "Master files" are separated into internal and external files.
- Instead of a single overall variation of plus or minus 2.5% to recognize simple or complex files and transactions, each component of the application is classified individually as having low, average or high functional complexity. The classification is based on numbers of file types, record types, and data element types involved.
- The technical complexity adjustment factor, previously plus or minus 25%, became plus or minus 35% based on a different set of adjustment factors (three were removed, and seven added). Guidelines were defined to reduce the subjectivity in scoring the adjustment factors.

Since this definition is still at the core of "Albrecht/IFPUG function points," the most widely-used function point approach today, it is worth presenting it now in detail.

2.2.1 Unadjusted Function Points

The first (and most important) phase of Function Point Analysis ("FPA") involves identifying certain components of the system that provide functionality to the user.

Functionality is defined with reference to the “system boundary:” the interface between the external user and the application.

Albrecht defined the five component types as follows [9]:

- *External inputs*: “data or user control input . . . that enters the external boundary of the application being measured, and adds or changes data in a logical internal file type.”
- *External outputs*: “data or control output . . . that leaves the external boundary of the application being measured.”
- *External inquiries*: “unique input/output combination, where an input causes and generates an immediate output.”
- *Internal logical files*: “logical group of user data or control information in the application . . . from the viewpoint of the user, that is generated, used, and maintained by the application.”
- *External interface files*: “files passed or shared between applications . . . logical group of user or control information that enters or leaves the application.”

Figure 1 depicts the five components.

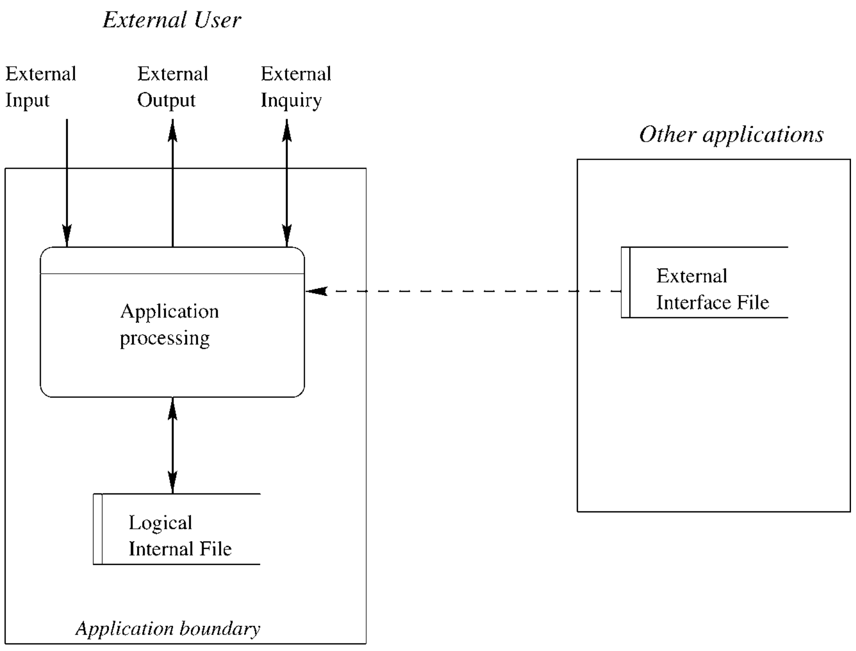


FIG. 1. Components in Albrecht function points.

The number of function points awarded to a component depends on its “functional complexity.” This is determined from the numbers of file types, record types, and data element types involved. Tables I, II and III respectively show how to determine the functional complexity of an input, output or inquiry, and internal or external file. Table IV shows the number of function points awarded for each type of component and each functional complexity level.

To recap, the first phase of FPA involves defining the system boundary; identifying the inputs, outputs, inquiries, internal logical files, and external interface files; award-

TABLE I
FUNCTIONAL COMPLEXITY OF EXTERNAL INPUTS

File types	Data element types		
referenced	1–4	5–15	16+
0–1	Low	Low	Average
2	Low	Average	High
3+	Average	High	High

TABLE II
FUNCTIONAL COMPLEXITY OF EXTERNAL OUTPUTS, INQUIRIES

File types		Data element types	
referenced	1–5	6–19	20+
< 2	Low	Low	Average
2–3	Low	Average	High
4+	Average	High	High

TABLE III
FUNCTIONAL COMPLEXITY OF FILES

Record		Data element types	
types	1–19	20–50	51+
0–1	Low	Low	Average
2–5	Low	Average	High
6+	Average	High	High

TABLE IV
FUNCTION POINTS PER COMPONENT

	Functional complexity		
Component	Low	Average	High
Input	3	4	6
Output	4	5	7
Inquiry	3	4	6
Internal file	7	10	15
External file	5	7	10

ing a number of function points to each individual input, output, inquiry, internal file and external file according to its functional complexity; and summing all of those function point values. This results in a total unadjusted function point ("UFP") value, which is seen as capturing the user-oriented functionality inherent in the system.

2.2.2 Adjusted Function Points

The second phase of FPA involves calculating a Value Adjustment Factor ("VAF"), based on fourteen general system characteristics ("GSCs"), and using it to adjust the UFP value.

The general system characteristics are supposed to capture other aspects of the functionality of the system: "pervasive general factors that are not sufficiently represented by the previously discussed transactional and data functions" [30]. They are evaluated for the system as a whole, rather than applying to individual elements.

The fourteen characteristics are:

- *Data communications*: the degree to which the application communicates directly with the processor. (Batch applications are at one extreme; real-time, telecommunication, or process control systems at the other.)
- *Distributed data processing*: the degree to which distributed data or processing functions are a characteristic of the application, within the application boundary.
- *Performance*: the degree to which response time and throughput performance must be considered in developing the application.
- *Heavily used configuration*: a heavily used operational configuration requires special design considerations.
- *Transaction rate*: a high transaction rate influences the design, development, installation and support of the system.
- *Online data entry*: online data entry and control functions are provided in the application.
- *End user efficiency*: the degree to which online functions emphasize a design for end user efficiency.
- *Online update*: the application involves online update of internal files.
- *Complex processing*: the degree to which processing logic influenced the development of the application.
- *Reusability*: the degree to which the application and the code in the application have been specifically designed, developed, and supported to be reusable.
- *Ease of installation*: the degree to which conversion from previous environments influenced the development of the application; conversion and installation ease are characteristics of the application.

- *Ease of operation*: the degree to which the application attends to operational aspects, such as startup, backup and recovery processes, without requiring manual intervention.
- *Multiple sites*: the degree to which the application is designed, developed and supported to be installed at multiple sites for multiple organizations.
- *Facilitate change*: the degree to which the application is designed, developed and supported for easy modification of processing logic or data structure.

Seven of these characteristics were present in Albrecht's initial formulation of function points. Three of the original ten were dropped (element complexity—now assessed separately for each element; reliable backup; and whether online data entry involves complex transactions). Seven characteristics were added: transaction rate, end user efficiency, reuse, ease of installation, ease of operation, multiple sites, facilitate change.

The GSCs are used to adjust the UFP value as follows. The degree of influence of each characteristic is scored on a scale from 0 to 5; the sum of fourteen scores is a result from 0 to 70; this is converted linearly to a value from 0.65 to 1.35. That value (the VAF) is multiplied by the UFP to give the adjusted function point ("AFP") value.

2.3 IFPUG

In 1986, several users of function points formed a non-profit organization called the International Function Point Users Group ("IFPUG").

IFPUG's aims are to spread the use of function points, and to standardize the way people count and use function points. It does this by holding regular conferences, publishing guides and manuals, and running exams (people who pass IFPUG's exam are styled "Certified Function Point Specialists").

IFPUG has made minor changes to terminology, but the basic structure of function points has remained essentially unchanged since 1984.

IFPUG has several committees, the most important being the Counting Practices Committee. This committee defines how function points should be counted in various situations—creating standard guidelines and resolving inconsistencies. The results are published in the Counting Practices Manual. This document has gone through several Releases, including 3.0 [34] in 1990, 4.0 [36] in 1994, 4.1 [40] in 1999, and 4.2 [41] in 2004.

Another part of IFPUG's activity is to make recommendations on how types of software and aspects of software that were not considered in 1984 can be mapped into the original function point structure [35,37–39].

Function points as Albrecht defined them and IFPUG maintains them are referred to hereafter as "IFPUG function points." Where "function points" are referred to without qualification, it is IFPUG function points we mean.

3. Experience with IFPUG Function Points

Much has been learned over the years about the strengths and weaknesses of function points.

The strength of IFPUG function points is pragmatic: they have been used successfully in practice for many years.

But they have been criticized on several grounds. These include theoretical problems concerning their construction; doubts about the detail of various steps; disagreements about exactly what they measure; subjectivity of measurement; some important aspects of software or types of software are not measured.

This section describes some of the experience and knowledge gained about function points. We begin with their successful use in practice, and then look at their problems and limitations.

3.1 Success of Function Points

We have seen that function points were developed in the context of application development in IBM in the 1970s and early 1980s. They are geared towards data-rich business applications, a dominant application domain in IBM then.

Functional size measurement has some key characteristics that are easy to promote. The orientation towards users, and early availability (measurable at the specification stage) are genuinely important and attractive features.

Albrecht/IFPUG function points "got in early," at a time when the need for functional size measurement was beginning to be appreciated and there was nothing much else around in the area. Since they were indeed found to be a useful measure for MIS software, they became a pragmatic success.

Function points were marketed heavily by IBM, IFPUG, and several consultants. Estimation tools based on function points were developed successfully. Books were written describing FPA (e.g., [23,29,30,58]) and experience with it in different industry sectors (e.g., [58]), and many articles written promoting its use (e.g., [28]).

By the mid-1990s, it was forecast that "function points would become the primary means for measuring application size, reaching a penetration of approximately 50 percent of development organizations by the year 2000" [32] (quoted in [28]). FPA was probably used to some extent by the information systems departments of most major companies and government departments in North America, much of Western Europe, and other parts of the world [98].

Though their use may have declined [98], IFPUG function points are still very much the dominant functional size measure in the world today. They have almost complete hold of the functional sizing market everywhere except the UK, where IFPUG function points and Mark II function points (described in Section 4) have about half the market each. COSMIC-FFP (described in Section 6) is gaining industry acceptance, but is still in its early days.

The key word is "pragmatic." Practitioners have found function points to be a useful measure, even while recognizing some of the criticisms. "Are function points a perfect metric? No. Are they a useful metric? In my experience, yes" [28]. "If [a measure] proves to be empirically correlated to some software phenomena we wish to forecast and control . . . why should we refuse it only because the measure is not formally validated?" [79].

A considerable body of knowledge and experience has accumulated. Practitioners have learned how to use function points effectively, what variations to expect between different industry sectors, and even how to exploit some of the structural characteristics of function points that worry researchers. This is most true in the MIS domain; function points are used much less in real-time software and communications software.

3.2 Theoretical Problems of Construction

Abran and Robillard investigated the whole underlying structure of function point analysis, considering all the measurement scales and mathematical operations used throughout the process [5,6].

They observe that the process begins with measurement on an absolute scale (identifying components, recognizing their types, counting the data and record types they involve). Abran called the collection of operations to this point the Function Point Measurement Model ("FPMM") [3]; the rest of the process he called the Function Point Productivity Model ("FPPM").

The process then moves to ordinal scales (classifying components into three levels of functional complexity), losing information in the process but still a valid operation according to measurement theory. Thereafter, things go awry. The assignment of points to components of different levels takes measurement from the ordinal scale back to the ratio scale—adding information, but without a theoretical underpinning.

Calculation of the VAF is similarly flawed. Invalid operations are performed, as ordinal scale measurements are converted to absolute scale values with no underpinning theory to justify the conversion. Effectively, arithmetic is performed on ordinal scale values as though they were measured on a ratio scale.

Hence, from a theoretical point of view function points cannot be considered a proper measure according to measurement theory. “The mapping, or measurement space, of FPA undoubtedly needs to be clarified if it is to be trusted as a valid measurement system” [6].

Empirical research showed that for the purposes of understanding productivity—the original purpose of function points, after all—none of the steps of the FPPM added significant information. Effort could be explained as well from the elementary measurements from the FPMM as from the final function point value resulting from the full process [6].

Kitchenham et al. proposed a framework for software measurement validation in 1995 [67]. They too were critical of the construction of IFPUG function points. They noted the scale transformations, and invalid operations performed on measurements from different scales. Further criticisms were that function points are discontinuous (no value of 1 or 2 FPs is possible), and there is no “unit” value. They are not fully based on a ratio scale. The result is a measure that is not valid in various ways. The risk is that it may behave in unexpected ways, for example asserting incorrectly that one program is larger than another [64].

3.3 Weighting the Components

The origin, validity and objectivity of the weights (function points per component type) have long been questioned.

Albrecht saw the weights as measures of the relative value of the function to the user/customer. They were chosen by debate and trial within IBM.

Symons published the first criticisms of function points [99]. He had several concerns about the weights. He questioned whether they would be valid in all circumstances. He felt that they contained some anomalies: components containing very many data elements receive at most twice the function points of a component containing only one data element. He desired some more objective assessment of the weights.

The concept of function value to the user has troubled others as well. Skeptics of this subjective concept view the weights as being really related to effort. For example:

> in the absence of any convincingly objective measure of user function or value of a transaction . . . independent of the cost of implementation, we are forced to relate function or value to some cost-related reference measure [106].

In devising Mark II function points (Section 4), Symons equated value with production effort.

Symons has reported Albrecht as saying the weights were chosen “on the basis of the relative numbers of data elements appearing on each of these four component

TABLE V
WEIGHTS DERIVED BY WITTIG

Component	Functional complexity		
	Low	Average	High
Input	4	7	13
Output	3	5	10
Inquiry	3	5	9
Internal file	3	5	12
External file	2	5	9

types, based on an examination of a number of systems which were being studied at that time in IBM" [100]. If the weights reflect the amount of data present in a component, there is some link with effort: programming effort is required to validate input data elements, and generate and format output elements.

There has been one attempt to derive weights that really do reflect perceived value to the user. Wittig et al. asked users to give their perceptions of amount of functionality provided by various FPA components. Numerous pairwise comparisons were made in which users assessed which of two components was larger, and by how much. With sufficient samples of all combinations of component types and functional complexity levels, it becomes possible to develop a set of weights empirically (using the Analytic Hierarchy Process [92] process) that reflect the users' perceptions of relative functionality. Initial results from a pilot study of 23 projects [111] were confirmed by further study of another 22 projects [90]. They are presented in Table V.

These differ from Albrecht's weights in several respects. Albrecht weights files more heavily than transactions, but here it is generally the other way around. With Albrecht's weights, the maximum weight for a component type is always about twice the minimum; here the maximum is about three times the minimum for transaction types, and four times the minimum for file types. Inputs have the least weight under Albrecht, and the most weight here.

Wittig's results provide an empirical basis for changing the weights in FPA so that function points really do reflect functionality to the user. The research has gone no further, though, and the new weights have not been adopted by IFPUG.

3.4 What Is Actually Measured?

Kitchenham et al. noted that it is not clear what function points actually measure: functionality is one concept, product size as perceived by a user is a different concept [67].

Whether IFPUG function points measure size alone is a point of strong debate. The aim is to measure the problem, independent of the technology used to implement the solution. Critics argue that function points don't actually work that way. Technology dependence is implicit in the choice of weights for function point components [99]. Technology dependencies can occur in all aspects of the FPA approach [106].

Further, several of the characteristics measured in the adjustment phase of FPA are really effort drivers, not size drivers [72]. Including them in the sizing process corrupts function points as a "pure" measure of size.

The problem is that organizations use size measures for more than just sizing the problem. They are also used for estimating effort, a task requiring technology dependence. A size measure that is truly unrelated to technology is no help for estimation [80]. Right from Albrecht's initial papers, validation of function points has commonly been done by relating size in FPs to size in LOC or to development effort, thus relating functional measurement to technical measurement.

Proponents of FPA (e.g., [28,30,58]) assert that function points *are* purely a measure of functionality, independent of technology, unrelated to effort. The argument amounts to a philosophical divide.

3.5 Subjectivity

IFPUG's work in developing successive releases of the Counting Practices Manual is aimed at standardizing function point counting and reducing subjectivity. Even so, counting function points is a subjective task, involving judgements by the person doing the count. This means that two analysts counting the same specification are unlikely to arrive at the same answer.

Early studies showed that within organizations, the variation between different function point counters was up to 30%; between organizations, it exceeded 30% [77]. Later studies showed the variation between counters to be about 12% [63], or "as little as plus or minus 11 percent" [28].

Training and experience is the key. Rule reports that the variation between recently trained project staff is typically around 23% [91]. For experienced specialists, following rigorous standards, the variation can be less than 5% [15,91].

This does not necessarily mean that counters will agree with each other to within a few percent on all aspects of a function point count. In practice there can be many differences in identifying and counting individual components, but they often cancel each other out when they are aggregated [83].

3.6 What Is Missed Out?

Function points are oriented towards data-strong systems, typified by business software. Processing in these systems is simple. Most effort goes into defining data structures.

Not all systems fit this pattern. Scientific and engineering software is often function-strong: dominated by the internal processing required to transform inputs to outputs. Real-time software is often control-strong: dominated by behavioural or control issues. Hybrid systems have elements of each of these types of software.

Traditional function points are less effective for software from these other domains.

Further, software development is very different now to when function points were first proposed. Then, bespoke MIS software was much more important. Now, software development is much more likely to involve package customization. The nature of software is often different: real-time, multi-layer, client-server software, rather than mainframe- and database-oriented software.

Although IFPUG has developed guidelines to help in several situations (e.g., [24, 73]), it can take some effort to relate the five component types that suit data-strong software to other types of software.

This is the prime motivation behind most alternatives to IFPUG function points.

3.7 The General Systems Characteristics and VAF

The adjustment phase of FPA is widely used by practitioners within the IFPUG world. This is partly because of a view that the general systems characteristics are genuinely important aspects of software projects, and partly for compatibility with the large number of past FP measurements that used it.

However, the GSC's and (especially) the VAF are probably the most criticized aspects of function points.

3.7.1 General Systems Characteristics

Symons observed that it is confusing to consider internal processing complexity as one of the GSCs, and also to give it a role in allocating function points to a component according to the numbers of file and element types referred to in the component [99].

Symons also felt that the 14 specific characteristics were unlikely to be satisfactory for all time. He felt that more than 14 were needed (in Mark II function points he extended it to 19), and that the particular set of 14 might need to change over time. This last point is supported by empirical evidence that two characteristics (online entry, and data communications) are inevitably scored as 5 in today's online world. They should be redefined if they are to keep any discriminative value [71].

Empirical study has also shown that patterns can be observed in the GSCs for different types of software [29,71]. For example, management information systems and decision support systems tend to be low in complexity, and place high importance on facilitating change by the user. Transaction/production systems and office information systems have more constraints on performance, and are less concerned with facilitating change by the user.

The definitions of some GSCs overlap. Relationships between them have been studied several times, invariably finding that several GSCs tend to vary together: they tend to be either low together or high together. This suggests that instead of 14 separate things being captured by the GSCs, a smaller number of underlying dimensions is involved, with some being captured several times over by different GSCs. The number of underlying dimensions might be 5 [71], 6 [66] or 7 [21], depending on the data set, but it is clear that there are not 14 independent factors.

As noted above, several GSCs (for example, performance requirements, reusability, heavily used configuration) really do not belong in a size measure. They are effort drivers, not size drivers [72]. Including them in the sizing process corrupts function points as a size measure.

3.7.2 Value Adjustment Factor

The VAF has received heavy criticism.

It involves inadmissible transformations according to measurement theory: arithmetic should not be performed on ordinal scale values [67].

Other criticisms are that it is not right to give all adjustment factors the same weight [21,99]; a multiplicative model would make more sense than an additive model, and the range of possible technical complexity adjustments is too narrow [21].

Tellingly, the VAF turns out not to provide any significant value. It makes no difference to the accuracy of effort estimates. The relationship between adjusted function points and development effort is no stronger than that between unadjusted function points and development effort [3,54,66,71]. This is probably because in most systems the VAF turns out to be very close to 1.0 [71]. The VAF is not even reliable as a simple indicator of whether effort is likely to be more or less than “average” for a given UFP [71].

Outside the IFPUG community, use of the VAF is declining. An adjustment phase is no longer a recommended part of Mark II function points, it has never been part of COSMIC-FFP, and it is not part of the recently-adopted standards for functional size measurement. Symons describes the VAF as “totally irrelevant to modern software development” [98].

For a while IFPUG described the VAF as an optional part of the process, in “Release 4.1 Unadjusted” of the Counting Practices Manual [40]. This meant that Release 4.1 Unadjusted was able to comply with the new ISO standard for functional size measurement [44], and could itself be approved as a standard [48]. But the latest Release (4.2 [41]) of the IFPUG Counting Practices Manual no longer describes the VAF as optional, and Release 4.1 is no longer marketed on IFPUG’s web site. It is clear that in IFPUG's eyes the VAF is still an integral part of Function Point Analysis.

3.8 Relationships Between FP Component Types

The relative proportions of each of the five component types, and the three levels within each component type, turn out to be remarkably stable.

This is worrying to researchers, concerned about the validity of the internal structure of function points: strong correlations between the different component types suggests that some things are effectively counted more than once. On the other hand, there are advantages for practitioners, who can make use of this knowledge in several ways.

3.8.1 Function Point Breakdown

Bock and Klepper [15] observed that the proportion of low, average and high components of each type was stable in their environment. They used this knowledge to simplify the function point counting process. Instead of separately determining the functional complexity of each individual component, they simply gave each component the average number of function points (determined by multiple regression) for that component type in their environment. Those averages are shown in Table VI.

Similar results (also shown in Table VI) have been found by the International Software Benchmarking Standards Group (ISBSG) [42].

The most time-consuming aspect of function point analysis is counting all the element types and file types referenced in a function point component, in order to determine whether the component has low, average or high functional complexity. Assigning average numbers of function points has the potential to speed up the counting process, "with no significant reduction in accuracy" [15].

We can compare the averages in Table VI with Albrecht's weights. For example, Albrecht's gives an output 4, 5, or 7 function points respectively for low, average and

TABLE VI
AVERAGE FUNCTION POINTS PER COMPONENT

Component	Bock and Klepper	ISBSG
Input	3.18	4.3
Output	5.33	5.4
Inquiry	3.92	3.8
Internal file	8.41	7.4
External file	5.54	5.5

TABLE VII
CONTRIBUTION OF COMPONENT TYPES TO TOTAL UFP

Component	New	Enhance
Input	37%	36%
Output	24%	32%
Inquiry	13%	12%
Internal file	22%	15%
External file	4%	5%

high functional complexity. As the average is about 5.3 to 5.4, outputs tend to be "average." Other transactional components (inputs and inquiries) also tend to have low to average functional complexity. Data components (internal logical files, and external interface files) tend to have low functional complexity. High functional complexity is rare. These observations suggest there could be some benefit in recalibrating the tables used to determine functional complexity, particularly for files.

The relative contribution to total FP from each of the component types has also been analyzed by ISBSG [42]. For new developments there tends to be about 3 inputs, 1.5 outputs, and 1 inquiry per internal logical file. For enhancement projects there tends to be about 4 inputs, 3 outputs, and 1.4 inquiries per internal logical file.

The contribution of each component type to overall FP is shown in Table VII. New developments and enhancement projects are tabulated separately, because there are statistically significant differences in their breakdowns. Not surprisingly, enhancement projects involve less change to data and more creation of extra outputs.

Knowing the typical breakdown of FP components can be helpful early in a project, when a quick estimate of the total size in function points can be very useful. If one of the five component types can be counted early, and the relationship between that component and total function points is stable enough, an early estimate of total function points might be possible.

In many projects, a data model is available right from the start, or is the first part of the system to be documented. This means that the number of internal logical files can often be counted very early. Strong linear relationships have repeatedly been found between the number of internal files and total UFP, with approximately 30 UFP per file [70]. In the ISBSG data set, even stronger relationships have been found between the number of inputs and total function points. This is less useful to an estimator, though, because the number of inputs is not usually known as early in a project.

Knowing the typical breakdown of FP components can also be useful for validation of a function point count. One form of validation can be to check that the FP breakdown is close to that expected [85]. While any given project may well vary from the average, large departures from expectation can indicate that a counting error has been made. Things to check include the relationship between internal logical files and UFP; the ratios of numbers of component types to each other; the percentage contribution of each component type to total UFP; and the average functional complexity of each function type. Checklists can indicate errors that might have been made if particular deviations from average are seen.

3.8.2 Correlations Between Component Types

Three studies have been made into the correlations between the five component types. The first two agreed on some findings and disagreed on others [54,65]. The third confirmed the agreements of the previous two, and suggested explanations for the disagreements [69].

It is clear that correlations between the five component types are inherent in FPA. Although the strengths of the correlations might vary from one data set to another, some common patterns are observed:

- Inputs, inquiries, and internal files are always correlated.
- External files are rarely correlated with the other components.
- Most correlations are weak, but some are strong enough (0.7 or greater) to indicate that some things are effectively counted more than once in FPA.

The strength and statistical significance of correlations between the components are greater for projects developed using 4GLs and application generators. They are weaker and vary more in projects developed using 3GLs. The correlations are also stronger in new developments than in maintenance projects [69].

Since 4GLs are most used in the domain where FPA is most widely accepted, this might imply that some simplification of the FPA process is possible in the areas where it works.

3.9 Backfiring

One aspect of function points that has been studied empirically from the very beginning—even by proponents of function points—is the relationship between function points and lines of code. This relationship is an interesting one. Function points and LOC are supposed to measure different things: respectively, the size of the problem independent of any solution technology, and the size of the solution.

But from the beginning, strong correlations have been observed between function points and lines of code for a given language [10]. Albrecht and Gaffney suggested this as the basis of a two-step process for estimating effort, using observed linear relationships to estimate lines of code from function points, and then effort from lines of code. A different formula was needed for each different language, but the correlation between lines of code and function points was very strong for each language.

Many thousands of projects have been sized now with both lines of code and function points. Formulas relating function points and program length are built into several commercial software estimation tools.

The concept of "backfiring" is simply the reverse of estimating length from function points: function points are estimated for an application by dividing its length by the average number of statements per function point for the language concerned [57]. The value of backfiring lies in providing a quick way to estimate the total size in function points of an organization's software portfolio, as a step in planning the maintenance of that portfolio [30]. Backfiring too is supported by commercial estimation tools.

In 1995 Capers Jones described backfiring as a "useful technique," that provides "a powerful way of sizing, or predicting, source-code volume for any known programming language or combination of languages" [57]. He noted that "the margin of error in converting LOC data into function points or back is high, but it's improving in both directions as more data becomes available."

In 1996, the second edition of Jones's book *Applied Software Measurement* [58] included a table identifying the number of logical source statements needed on average to encode one function point in each of 464 programming languages. He noted that code complexity could affect the relationship for any given program, but wrote about backfiring in matter-of-fact terms as a practical thing to do.

Backfiring presents an ironic problem. A basic tenet is that functionality and length are *different*; and that function points, being user-oriented, are a much better measure than lines of code, which are subject to variations between programmers and languages. From this viewpoint, Jones has written famously that using lines of code as a software size measure should be considered professional malpractice [59]. The argument is undermined if it turns out that function points and length are strongly related.

Jones is now much less positive about backfiring. His advice is to "strongly discourage its use in virtually every conceivable circumstance," and that "if it seems too good to be true, it probably is. Sooner or later, real counting will need to be done" [61]. Nevertheless, the "Programming Languages Table" is still maintained and is still available for purchase from Jones's company [61], albeit with warnings about its use.

Fundamentally, backfiring probably can be used effectively to estimate function points from program length or vice versa. The correlations between function points and length are undeniable. But two important qualifications must be noted.

First, as Jones points out, "local development practices and the way languages deliver functionality differently will always make backfiring a troublesome, dangerous exercise—especially if the backfiring is not meticulously calibrated to local conditions." In other words, knowledge is needed of relationships between function points and length in one's own environment; generic tables such as Jones's Programming Languages Table should not be relied on.

Second, backfiring should only be used to estimate portfolios of multiple projects. For a single project, the risk of an inaccurate estimate is high. Across a set of projects, there is some chance that individual errors will even out and averages will be reasonable.

3.10 Summary

The section has presented criticisms and research that suggest the need for other approaches to functional size measurement. Indeed, the rest of this chapter describes the evolution of several other approaches.

The criticisms and research findings described above have met with various responses.

Some have been disputed or ignored. Challenges to function points as purely a *size* measure are disputed on philosophical grounds. Problems of construction are overlooked, because function points have been found to be useful despite them.

Others have led to some evolution of how IFPUG function points are counted. Successive releases of IFPUG's Counting Practices Manual aim to improve standardization, reduce subjectivity, and indicate how things that were not initially considered should be counted within the core FP structure. IFPUG works hard to ensure the relevance of function points in a changing software world.

IFPUG's changes are all concerned with how to make counting decisions. The underlying structure of function points, and the process of calculating function points, has remained unchanged.

Several people have attempted to overcome perceived problems with IFPUG function points, by proposing extensions, or alternatives, to them. We turn now to the most significant of those other proposals.

4. Mark II Function Points

The first significant alternative to Albrecht's function points was put forward in the late 1980s by Charles Symons, from the United Kingdom.

While praising Albrecht for breaking important ground, Symons saw several weaknesses in Albrecht's approach. Symons' criticisms were published in 1988 [99], along with an outline of an alternative formulation for function points: dubbed "Mark II Function Points." Full details of the new approach were set out in a book, published in 1991 [100].

Although Symons described Mark II function points as an evolutionary step, the aims and approach are quite different to Albrecht function points.

Mark II function points gained sufficient industry acceptance (though scarcely used outside the UK) to join IFPUG function points as a mainstream function point approach.

4.1 Symons' Criticisms

Symons criticized several aspects of Albrecht function points (most of these have been discussed in Section 3):

- Classifying each component as having low, average or high functional complexity was too much of a simplification.
- The origin, validity and objectivity of the weights (function points per component type) were doubtful.
- The treatment of internal processing complexity was confusing.
- The general systems characteristics had several flaws.
- Function points are not summable in the way one would expect. If several separate collaborating systems are replaced by one integrated system, the integrated system has fewer function points than the component systems. In effect, the whole is less than the sum of the parts.

Symons' main point was that Albrecht's approach was developed in a particular environment; Symons questioned the suitability of the method, and particularly the weights, for general application.

In 1991 Symons added two more criticisms [100]. First, some types of software cannot be sized reliably using Albrecht function points. Examples are expert systems and system software: software that features complex internal processing, or complex algorithmic processing. Second, the count of files should be weighted by usage. A file that is used in many transactions should be counted multiple times, not just once.

The latter point reflects a crucial change in philosophy, that informed Symons' new formulation of Mark II function points. Albrecht sought to measure things of value to the user, and measured them once each. Symons retained the aims of technology independence and user focus, but moved away from the subjective concept of value to the user. Instead he sought explicitly to relate the system size measure to the effort involved in developing the system. This was more objective than trying to measure value to the user, and directly suits the aims of having a size measure suitable for measuring productivity and estimating effort. From this point of view, "The size of the databases, measured in function points ... is not directly relevant to the size of the applications ... What matters in measuring the size of systems for performance measurement and estimating purposes, is the usage of the files by the transactions within the applications" [100].

Symons took most of these observations into account when formulating Mark II function points. He did away with the low/average/high classification, and simply counted the numbers of data elements. He preferred the relational database concept of “entity” to the ambiguous concept of “logical file,” and changed the basic components to be counted in a way that meant files were counted as many times as they were used. He determined the weights for different element types through a calibration process relating specifically to project effort, and proposed that the weights might change over time to retain the relationship with effort. He modified the adjustment process based on the general systems characteristics.

The one thing he did not try to do was measure the size of an “algorithm,” regarding that as a problem for the future. Thus, Mark II function points have the same problems as IFPUG function points in terms of range of applicability. They are still geared towards file-oriented business systems.

4.2 Mark II Function Points

In Mark II function points, the system is regarded as consisting of a collection of “logical transactions.” Each has input, processing, and output components. The size of a logical transaction is the sum of the sizes of the input, processing, and output components. The size of the system is the sum of the sizes of the logical transactions.

The sizes of the input and output components of a logical transaction are proportional to the numbers of data element types they involve. As the number of element types goes up, the size goes up; there is no simplification to just low, average or high. The idea is that the effort to format and validate an input, and to format an output, is proportional to the number of elements involved.

For the size of the processing component of a transaction, Symons looked to McCabe’s cyclomatic complexity [78] and Jackson’s mapping of data structure to code logic [53]. Symons proposed that a measure of processing complexity is to count the number of data entity types (data entity types are the same as entities in relational data modelling) referenced (created, read, updated, deleted) by the transaction.

The size of a logical transaction, expressed in Unadjusted Function Points, is thus:

$$\text{UFP} = N_i W_i + N_e W_e + N_o W_o$$

where

N_i = number of input data element types,

W_i = weight of an input data element type,

N_e = number of entity types referenced,

W_e = weight of an entity type,

N_o = number of output data element types,

W_o = weight of an output data element type.

In 1988 Symons' initial calibration yielded weights of $W_i = 0.44$, $W_e = 1.67$, $W_o = 0.38$. By 1991 they had become 0.58, 1.66, and 0.26 respectively. The weights have not changed since 1991.

Mark II function points include an adjustment phase based on some general systems characteristics. The final calculation of the adjustment factor is similar to IFPUG: scoring the characteristics on a scale of 0 to 5, adding the scores, scaling the result, and adding it to 0.65 to produce a Technical Complexity Adjustment factor. The unadjusted size is multiplied by the TCA to produce the system size in adjusted function points.

There are two differences between the IFPUG and Mark II calculations of the TCA. First, the impact of each factor is halved in Mark II when calculating the TCA. Second, Mark II adds five extra system characteristics (interfaces to other applications, special security features, direct access for third parties, documentation requirements, special user training facilities) to the 14 of IFPUG, making 19 in total. The possibility of adding further client-defined characteristics was also envisaged.

Symons saw little incentive to put much work into redefining the adjustment process. He observed that the range of TCA values was small in practice, and seemed already to be smaller than in 1984. He expected the TCA to continue to decline in significance. Although it still appears in the Mark II FPA Counting Practices Manual [104] as an optional part of the Mark II FPA process, its use is no longer recommended.

4.3 Experience with Mark II Function Points

Kitchenham et al. consider that Mark II function points satisfy the properties of a valid measure, but only if they are regarded as an effort measure, not a size measure [67].

A natural question is whether or not there is a relationship between IFPUG function points and Mark II function points. Symons reported initially that they did not correlate well, and the scatter was random, so it would not be possible to predict Mark II size from IFPUG size; this implies that they measure different aspects of size [100]. He found that the Mark II method gives a higher UFP score than the IFPUG method as system size increased.

Dolado [22] found the opposite. He found a strong correlation between IFPUG and Mark II function points. In his data set, the IFPUG method gave a higher UFP score than the Mark II method on all projects but one.

Symons later reported conversion formulas between IFPUG function points and Mark II function points, for new developments, for two ranges of project size [101]. He noted again that the Mark II method gives a higher UFP score than the IFPUG method as system size increased, as expected since files may be counted multiple times. Up to 1500 IFPUG UFP's, a quadratic formula relates the two sizes.

$$\text{MkII} = 0.9 \times \text{IFPUG} + 0.0005 \times \text{IFPUG}^2.$$

This formula was derived empirically, from a collection of systems counted with both methods. Above 1500 IFPUG UPFs, no empirical data was available. Some approximate MkII/IFPUG ratios were derived by extrapolation.

The UK Software Metrics Association maintains the Mark II FPA Counting Practices Manual. Mark II function points have achieved a strong share of the function point market in the UK, but are little used elsewhere.

5. Some Other Early Variations

Mark II function points are by no means the only alternative proposed to IFPUG function points. Many variants exist in the literature (see [76] for a broader survey than is presented here). Apart from the main approaches identified in this chapter, none has achieved much currency.

Most have been proposed because someone thought something was missing from IFPUG FPA, and thought they knew how to fill the gap.

Three early alternatives or variations to function points are described now. The first is interesting because it appeared at about the same time as function points. The others extend function points in different ways, attempting to improve the measurement of different types of software.

5.1 DeMarco's Function Weight

A proposal with similar aims to function points, called "System Bang," was published in 1982 by DeMarco [19]. It too was proposed as an implementation-independent indication of the function to be delivered, as perceived by the user.

DeMarco based his measure on things that can be counted from a specification model: the function, data and state models of the Structured Analysis and Design Technique. He identified twelve primitive things that should be counted as soon as the specification model was complete. The primitives included processes on data flow diagrams, objects and relationships on entity-relationship diagrams, and states and transitions on state-transition diagrams.

DeMarco did not suggest calculating Bang as a weighted sum of all twelve primitives. He suggested instead that projects should be divided into a small number of domains, and a different formulation of Bang should be developed for each domain.

His main classification of systems was function strong, data strong, and hybrid.

Function strong systems "can be thought of almost entirely in terms of the operations they perform upon data." It is the internal processing required to transform inputs into outputs that matters, and the data is fairly simple. For a function strong system, DeMarco suggested that the principal component of Bang is Functional Primitives (lowest level pieces on data flow diagrams). Each functional primitive was given a weight depending on the number of input and output "tokens" (data items that need not be subdivided) it worked with. The weighted sum of Functional Primitives was called "Function Bang" (since renamed "Function Weight").

A data strong system "is one with a significant database, and most of the effort for this system is allocable to tasks having to do with implementing the database itself." For such a system, the principal component of Bang is the number of objects in the data model. Each object receives a weight depending on the number of relationships it participates in. "Data Bang" is the weighted sum of objects.

Hybrid systems fall between function strong and data strong systems. DeMarco suggested calculating both Function Bang and Data Bang for such systems, and using the two values separately as predictors when forecasting cost.

DeMarco also classified systems on a second dimension, based on the relative importance of data movement compared to computation. Commercial systems have more data movement, and scientific systems more computation. This classification did not affect how Bang was computed, but rather how the value was used. Bang was intended to be a predictor of project cost, and DeMarco warned that different projections should be used for projects from different domains.

It is interesting to draw some comparisons between function points and Bang. Data is essentially measured the same way: DeMarco's objects correspond to files in function points. Processing is counted differently: Bang counts processes on data flow diagrams, which is a low-level unit, while function points count flows across the system boundary. Function points are geared towards data strong systems, while Bang recognizes function-strong systems as needing to be counted differently. Neither approach considers systems where states and transitions are significant. Function points do not consider them at all; DeMarco identifies states and transitions as things to count, but then gives them no further consideration.

DeMarco has cited anecdotal evidence of the usefulness of his method for estimating effort [20]. No formal validations have been published. Its only other appearance in the literature is in a simulation study from 1993 [87]. Bang never achieved the critical mass of users to become a mainstream functional sizing method.

5.2 3D Function Points

3D function points were developed by Whitmire at Boeing in the early 1990s [108, 110]. The motivation was his perception that traditional function points did not properly measure scientific and real-time software.

3D function points follow DeMarco in regarding all applications as having three dimensions: data, function, and control. “Each dimension contains characteristics that contribute to overall problem complexity or size, and these characteristics can be measured directly” [108].

Traditional function points were accepted as capturing the data dimension. They are incorporated directly into 3D function points to measure that dimension.

Like DeMarco, Whitmire viewed function strong systems as being dominated by the processing required to transform inputs into outputs. But where DeMarco counts processes, weighted according to the number of input and output tokens they work with, 3D function points measure processing differently.

A “transformation” is defined to be “the set of process steps and governing semantic statements [predicates that must remain invariant throughout the sequence of operations, or the pre- and post-conditions defined for each operation] required to transform one set of input data to output data” [108]. Depending on the number of processing steps and the number of semantic statements involved, a transformation is classed as having low, average or high functional complexity (see Table VIII), and then receives 7, 10 or 15 3D function points respectively.

The third dimension is control. This dimension is measured by counting the states and transitions on a Finite State Machine. Initial and terminal states are not counted, and the transition count is reduced by the number of states so that only multiple paths leading out of a state are counted. The resulting number of transitions is added to the 3D function points (i.e., each transition is counted separately, with a weight of 1 3D function point).

Whitmire recommended representing the 3D function points of an application as a triple (data, function, control) rather than just adding them together. The single combined value is also useful, but considering the three values separately helps spot characteristics of an application that are hidden by a single value [107].

TABLE VIII
FUNCTIONAL COMPLEXITY OF TRANSFORMATIONS IN 3D FP

Processing	Semantic statements		
steps	1–4	5–15	16+
0–1	Low	Low	Average
2	Low	Average	High
3+	Average	High	High

DeMarco suggested counting the control dimension of an application, but didn't actually do it. 3D function points were the first proposal to include all three dimensions. They have never become a mainstream sizing method though. Symons [98] reported in 2001 that 3D function points were still used successfully within Boeing, but no further information has been published outside Boeing.

5.3 Feature Points

Feature Points are an extension of Albrecht's 1984 definition of function points. They were proposed by Capers Jones in 1986, in an attempt to measure software that is high in algorithmic complexity (for example systems software, real-time software) [56,58].

A sixth component was added to the five standard function point component types. An "algorithm" was defined as "the set of rules which must be completely expressed in order to solve a significant computational problem" [58]. Examples are a square root extraction routine or a Julian date conversion routine.

Algorithms receive 3 points each in the feature points method. There is no classification of low, average and high functional complexity for any type of component (continuing with a change to the function point approach that Jones had already made, in an attempt to simplify the process, in his SPQR/20 estimation model). The weight given to internal logical files is reduced from 10 points to 7, to reflect the reduced significance of data files in systems software.

Jones reported that for classical MIS projects—the natural domain of IFPUG function points—feature points and function points often gave almost identical results. But for harder forms of systems software, feature point counts were significantly higher.

Feature points were the first attempt to capture the processing aspects of an application in a tool and method that was marketed for widespread use. Although they were supported in the SPQR/20 estimation tool, they were still described as "experimental" in 1996 [58]. They never got beyond that stage. Although they are still documented on SPR's web site [56], they are no longer supported by SPR.

5.3.1 How to Count Algorithms?

The significant contribution of feature points was the idea of counting algorithms. But that was also its weakness. Whitmire felt that the definition of algorithms was not sufficient for counting purposes [108], and that they were deficient in considering processing steps but not semantic statements. Symons noted the "inherent difficulty of agreeing standard ways of defining and assigning a weight to algorithms of increasing size and complexity" [98], and considers that no functional sizing method has made any progress towards handling algorithmic complexity [97].

One recent proposal [88,89] involves sizing an algorithm by regarding it as consisting of data to be operated on (an input), a local storage area to hold intermediate results (one or more internal logical files), and a result (an output). Their functional complexity is determined using normal IFPUG methods; a design aim of the approach is that it should fit directly within IFPUG methods. The function point value for the algorithm is the sum of the values for the input, storage, and output. The approach has been demonstrated on two examples. Whether this approach scales up has yet to be seen, and there remains the crucial question of deciding which algorithms to count.

5.4 Summary

Each of the three variants discussed in this section has interesting aspects. But none has achieved widespread success.

Bang and feature points were proposed by consultants, who used them in their own practices. Neither supports them any more. Presumably the methods were not successful enough or promoted enough to provide sufficient business value to their creators. 3D function points have only ever been used within Boeing, and have not been promoted in the literature for over 10 years. With no promotion outside Boeing, they have always been noted as an important but localized idea.

The problem that these three approaches tried to address was measuring real-time software and measuring algorithms. Measuring algorithms is still an unsolved problem. Measuring real-time software may have been solved: we turn now to COSMIC-FFP.

6. COSMIC

All of the function point approaches discussed above were the ideas of individuals. COSMIC is different.

COSMIC stands for Common Software Measurement International Consortium. Established in 1998, it is a consortium of academics and practitioners with an interest in functional sizing of software. Alain Abran and Charles Symons were (and remain) joint project leaders. The original participants came from Canada, Europe and Australia. Most were participants in the ISO Working Group that was then engaged in developing a standard for functional size measurement.

COSMIC's aim was to develop and promote a new approach to functional size, useable for performance measurement and estimation, applicable to as wide a range of software domains as possible. Priority was given to business software, real-time software, and hybrids of the two, but not software of high algorithmic complexity.

The resulting measure built on work that was already under way on “Full Function Points.”

6.1 Full Function Points

In 1997, Abran and his colleagues proposed an extension to IFPUG function points, with the aim of better measuring real-time software [96].

They felt that an extension to IFPUG function points was needed, because IFPUG function points had two problems when dealing with the control aspect of real-time software. First, real-time software frequently involved single-occurrence groups of data; these are difficult to map to internal logical files and external interface files. Second, real-time processes varied widely in their number of sub-processes; this could not be captured properly by measurement at the process level which can only give a narrow range of function points (e.g., 3 to 6 points for an input).

They proposed that the functional user requirements of real-time software included data-rich files and transactions, plus control data and control transactions. The data-rich aspects could be measured using IFPUG function points. The control aspects needed to be measured separately, for which they defined new control data and transaction types.

The sum of the two measurements represented the total system size, measured in Full Function Points (“FFP”).

For control data they defined control groups. A control group is a group of control data used by the application, identified from a functional perspective; it exists for more than one transaction. They drew a distinction between read-only control groups, that are used but not changed by the application, and updated control groups that do get changed.

For control transactions, they defined four function types. Each represents a type of data movement. An external control *entry* represents a group of control data coming in from outside the application boundary. An external control *exit* represents a group of control data going outside the application boundary. An internal control *read* represents a group of control data being read from internal storage. An internal control *write* represents a group of control data being written to internal storage.

The principle behind measuring a process is that it has a separate sub-process for each data movement, and the functional size is directly proportional to the number of sub-processes. Control groups are not sized directly; only the data movements contribute to the functional size.

There is a very different measurement perspective involved here, compared to the IFPUG process. IFPUG sizes a transaction by counting the data elements involved, converting that to one of three functional complexity levels, and giving a consequent number of between 3 and 7 function points. Files are also sized, once each, using a similar procedure that gives each file between 5 and 15 function points. FFP also measures size by counting data, but it is data movement rather than static data structure that is measured. Moreover, FFP sizes a control process by simply counting the data movements, which can be anything from 2 upwards with no upper limit.

The two measurement methods are different enough that it was necessary to justify the validity of adding the two results to give a single number [86].

Validation of FFP took the form of industrial field trials, to check the relevance and useability of the approach and to compare the results from IFPUG and FFP measurement. Early trials showed that for MIS software, IFPUG and FFP measurements were similar, while for real-time software FFP measurements were much higher.

By the time version 2.0 of the FFP measurement manual was issued in July 1999 [105], FFP had dropped the IFPUG part of the process. *All* data and transactions, not just control data and transactions, were measured in terms of entry, exit, read and write sub-processes. (Simpler terminology was possible as a result: for example, "entry" was now used, instead of "external control entry.")

Another fundamental difference from the IFPUG point of view had emerged by then: "layers" were being considered [84]. IFPUG views functionality as entirely based on business functions seen by external users. The developers of FFP considered that other aspects of software should be included as well, perhaps involving different layers of functionality.

FFP and IFPUG function points were now entirely different measures.

6.2 COSMIC Full Function Points

By the end of 1998, COSMIC had been formed. The aim was to develop, and gain acceptance as an industry standard, a new approach to functional size measurement.

COSMIC sought to develop a measure that should: be useful to software project managers; be widely applicable, with priority given to business and real-time domains; conform to the emerging ISO standard for functional size measurement. They began by reviewing the main existing function point methods, and came up with a set of principles. They moved on to defining their own functional size measure.

The resulting measure drew heavily on ideas from Mark II and FFP. FFP version 2.0 was adapted to suit COSMIC's measurement principles, and rebadged as COSMIC Full Function Points ("COSMIC-FFP"). The first Measurement Manual was issued in October 1999. The most recent version is dated January 2003 [4].

The principles behind COSMIC-FFP are that:

- The functional user requirements of software are defined as a set of *functional processes*. A functional process is "an elementary component of a set of Functional User Requirements comprising a unique cohesive and independently

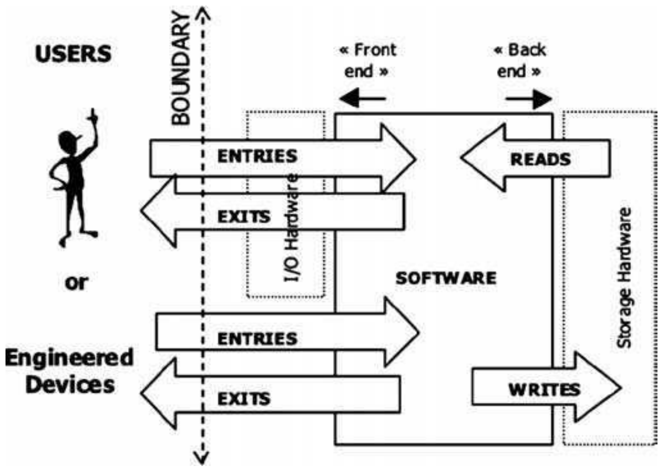


FIG. 2. Data movements in COSMIC-FFP.

executable set of data movements. It is triggered by one or more triggering events . . . It is complete when it has executed all that is required to be done in response to the triggering event" [4]. Triggering events occur outside the software boundary.

- Software manipulates pieces of information, designated as data groups, which consist of data attributes. Figure 2 depicts the flow of data groups.[1]
- Functional processes involve sub-processes, concerned with movement (entries, exits, reads and writes) and transformation of data groups.
- The functional size of a functional process is directly proportional to its number of data movements.
- The functional size of an application is the sum of the sizes of its functional processes.

At heart, sizing an application with COSMIC-FFP involves identifying the functional processes; identifying data groups; and identifying and counting the movements of data groups.

Note that data transformations are not considered. Some processing is assumed to be associated with each data movement (validating inputs, formatting outputs, etc.). The processing involved in data transformations is assumed to be covered by, or at least proportional to, this baseline processing. That won't be true for soft-

[1] Figure 2 is a reproduction of Figure 2.4.1.1 from [4, p. 9]. Reproduced with permission.

ware involving complex algorithmic transformations; COSMIC does not claim that COSMIC-FFP applies to such software.

Each functional process involves at least two data movements. There must be at least one entry, and one exit or write. There is no upper limit on data movements; instances of functional processes with over 100 data movements have been observed.

The unit of measurement in COSMIC-FFP is not called a "function point." It is the "COSMIC functional size unit," or "Cfsu." It is defined to be equivalent to one single data movement type at the sub-process level.

COSMIC-FFP excludes a technical complexity adjustment. It does not attempt to take into account the effect on size of technical or quality requirements. Those are recognized as important in a project, but should not be included when measuring functional size.

COSMIC-FFP has two important innovations: layers, and measurement viewpoints.

Layers. The concept of layers in COSMIC-FFP is similar to the concept of a layered software architecture. A layer is the result of functional partitioning, such that all included functional processes perform at the same level of abstraction. A layer provides functional services to its users; subordinate layers provide services to software in higher layers; software in a subordinate layer can perform without assistance from software in layers that use its services.

The point is that functional user requirements may be assigned to multiple layers in an application. This matters from a sizing point of view. If just the external behaviour is measured, the allocation of functionality to layers is not relevant. If layers are measured separately, overall functional size goes up, as inter-process communication is measured. For example, in a non-layered view, reads and writes are only measured once as internal operations. If they are delegated to a subordinate layer in a layered model, an extra level of entry/write and exit/read is measured. Writes from layer 1 become entries in layer 2, exits from layer 2 become reads in layer 1, and all are measured. Communication between peers in the same layer can also be measured.

Measurement Viewpoints. Regarding software as a black box, with only externally visible behaviour, is one viewpoint: suited to an end user, who cares only about what they may do with the software. This is the "end user measurement viewpoint." Regarding software as a collection of functions, spread across different layers, each providing separate functionality to the user, is a different viewpoint: suited to a developer, who cares about the totality of the software that they must write to meet the user's needs. This is the "developer measurement viewpoint." The contrast between these viewpoints looks a lot like the functionality/length difference—except that layers in COSMIC-FFP still represent functionality that must be provided, rather than lines of code that must be written.

The difference between viewpoints is important. Different viewpoints lead to different sizes being measured. The end user viewpoint corresponds to the IFPUG view of functionality. End users are not interested in a layered view, for example. Developers, and managers of development projects, are concerned more with the major components to be developed: these might reflect different layers, or peers within the same layer.

It is important to identify the viewpoint for measurement, if comparisons are to be attempted between developments.

6.3 Experience with COSMIC

COSMIC-FFP were tested in industrial field trials [7], to test that the documentation about COSMIC-FFP is understandable across different domains; that the method can be applied with reasonable effort; and that the sizes reflected the functionality as perceived by experts.

Results showed that experience with both COSMIC-FFP and the application domain was required to ensure repeatability. The COSMIC-FFP approach was seen as easy to apply. The effort involved was comparable to that for other functional sizing methods (though if only poor or incomplete documentation is available, IFPUG is probably slightly easier to apply than COSMIC).

The field trials also produced some data about the relationship between Cfsu and effort. The initial data set was small, so the results were not definitive, but strong relationships were observed.

The COSMIC Measurement Manual comments on conversions between COSMIC-FFP and other function point measures. There are few projects yet that have been sized with both COSMIC-FFP and another method (though one case study exists in which the same system was measured with five different approaches [26]), so most of the comments are based on theory and impressions rather than statistics. A small number of projects have been sized with both IFPUG and COSMIC FFPs. It seems that, on average, IFPUG and COSMIC sizes (if both are measured from the same end-user viewpoint) are roughly similar for new development projects of over about 100 Cfsu. For enhancement projects, and projects measured from different viewpoints, poorer correlations are expected. Direct data is not available relating COSMIC sizes to Mark II or early FFP sizes, but it is expected that on average they should all give roughly similar sizes.

In Section 3.8 we saw that in the IFPUG world, knowing the balance to expect between the five component types is useful for early estimation and for validating a count. Similar value can be expected in the COSMIC world. Study of 52 projects from the ISBSG Data Repository found the proportions of the four data movement types (entries, exits, reads, writes) to total functional size to be 33%, 35%, 19% and 13% respectively. Entries and exits contribute about equally, with a big drop to reads and a further drop to writes [43].

The use of COSMIC-FFP is increasing, particularly for real-time software, though its take-up is slower in the USA where IFPUG holds sway.

7. Function Points for Object-Oriented Software

From a user's point of view, object oriented software development is an implementation approach. In principle, it should be possible to apply function points as readily to OO software as to software developed using any other technology.

Specification documents include some form of model of the system, from which function points are counted. Traditional function points are counted from a data model and descriptions of the transactions. Those items may not be represented directly in an object oriented development, where the system model is probably expressed in UML. If function points are to be counted for an OO system, the issue is how to relate things in a UML model to the files and transactions required for a function point count.

Garmus and Herron [30] include a chapter on counting an object-oriented application. But there is almost nothing OO-specific in the chapter, beyond noting that a different approach is often needed because the documents are different. Their advice is to first become familiar with the entire system model, including functional description, object models and system diagrams. Once you understand what the system provides for the user, you can then count function points normally.

Several authors have proposed methods for adapting function points to object oriented software. Some retain a focus on traditional function points as the output from a count. They concentrate on relating OO concepts to FP components. Others develop new measures, tailored to OO software but analogous to FPs in some aspects of their construction.

In contrast to traditional methods, where IFPUG, Mark II and COSMIC dominate, the range of proposals for OO software is wide. None has yet achieved broad recognition. This indicates that measurement in the OO domain is not as well understood.

7.1 Mapping OO Concepts to Function Points

If traditional function points are to be counted from a UML model, the issue is how to relate OO concepts to FP components.

Most proposals are based on class diagrams. A draft proposal [38] by IFPUG treats classes as files, and methods as transactions. Longstreet [75] treats classes that represent non-transient data as files, and describes sending and receiving messages as outputs and inputs respectively.

Longstreet [74] also sees a natural link between use cases and FPA. He gives some small examples but not much other detail.

There are four main proposals for identifying IFPUG function point components from UML models. Three give particular sets of rules and guidelines. The fourth draws on the others, seeking to determine empirically how best to make various counting decisions. All of these proposals aim to replicate (in most cases automatically) the result that would be produced by an expert human counter of IFPUG function points.

Fetcke [27] defined a set of rules for mapping a use case model and class model to concepts from the IFPUG Counting Practices Manual [36]. Some actors are identified as users or external systems. Use cases that interact directly with users or external systems, and other use cases that extend those with direct external interaction, are candidates for transactional components. Domain objects (specifically, entity objects, if that level of analysis is provided) correspond to files. Rules for handling aggregation and inheritance are given. Fetcke concentrated on demonstrating that his method was unambiguous enough to be applied in practice; how close the results were to a normal FP count was not studied.

Uemura et al. [103] considered class diagrams and sequence diagrams, obtaining most information from sequence diagrams. Classes that might represent data files are identified from the full set of sequence diagrams. Those whose data are changed by at least one transaction are internal files, the others are external files. Class attributes map to DETs, and RETs are essentially ignored (assumed always to be 1). Transactional components are identified from sequences of messages, initiated by external actors. Rules are given to decide the type of transactional component (input, output, inquiry). Message arguments map to DETs, and the number of FTRs is 1 or 2 depending on which rule is applied. Provided the class and sequence diagrams meet certain conditions, these rules can all be automated; their paper reports early success with a tool they have developed.

Abrahão et al. [2] use yet another combination of UML diagrams. They consider that an OO conceptual model has four aspects: data (object model: class diagram), behaviour (dynamic model: state transition diagrams and interaction diagrams), process (functional model: describing the semantics of changes to an object's state), and presentation (presentation model: describing user interaction with the system). Classes and legacy views correspond to internal and external files. Services that change the state of a file correspond to inputs. Outputs and inquiries are recognized from the presentation model. Rules are given for identifying and counting each type of function point component, and the final result is intended to match an IFPUG function point count. Results from a case study are promising [1], and they too have built a tool to automate their process.

Cantone et al. [16] are building on the work of Fetcke, Uemura, and also Antoniol et al. [14]. They consider class diagrams, use case diagrams, and sequence diagrams. They do not specify a particular set of rules for what to count and how. They provide some guidelines and heuristics, and give their comments on the appropriateness of rules proposed by earlier researchers. They have also identified many situations where different decisions can be made on how to treat certain aspects of an OO model. They are conducting empirical work to see how different decisions affect the outcome. The aim is to develop a tool that can automatically analyze a UML model and generate a function point count that is as close as possible to that given by an expert in FPA.

Jenner has initiated work on mapping UML concepts to the data movement subprocesses in COSMIC-FFP [55]. The main resource is the sequence diagram. Discrete interactions between actors and the system correspond to functional processes; messages that cross the system boundary correspond to entries and exits; messages sent to objects correspond to reads or writes. The method can be automated if the sequence diagrams are fully specified and optional return arrows from read processes are not included in the diagram.

7.2 Function Point-Like Measures for OO

Another set of proposals is somewhat different. Measures are defined that are tailored to OO software, and are analogous to function points in some aspects of their construction. In most cases the same sorts of things are counted: objects for files, and methods or messages for transactions. But there is no expectation that the result would closely match an IFPUG FPA count.

Whitmire [109] considered each class as an internal file; messages sent across the system boundary were treated as transactions.

Schooneveldt [94] treated classes as files, and considered services delivered by objects to clients as transactions.

Sneed [95] proposed *object points* as a measure of size for OO software. Object points are derived from the class structures, the messages, and the processes or use cases, weighted by complexity adjustment factors.

Predictive Object Points (POPs) [82] are based on counts of classes and weighted methods per class, with adjustments for the average depth of the inheritance tree and the average number of children per class. Methods are weighted by considering their type (constructor, destructor, modifier, selector, iterator) and complexity (low, average, high), giving a number of POPs in a way analogous to traditional FPs.

Graham [31] proposed *task points* as a size measure that can be computed at the requirements analysis stage. A task point is an atomic task that the system will carry out in support of user requirements. Task points are equivalent to the leaf nodes in the Task Object Model—a hierarchical model of the business tasks to be supported by the system.

Two proposals are worth a slightly more detailed look, because they have more presence in the literature.

7.2.1 Object Oriented Function Points

Antoniol et al. [14] introduced *Object Oriented Function Points* ("OOFPs"). All measurement was based on the class diagram.

Though drawing heavily on IFPUG function points for inspiration, the philosophy behind OOFPs was not to find a way to count IFPUG function points. It was to develop a measure based on OO concepts that is useful for estimating project attributes such as LOC and effort.

OOFPs map classes to files, and methods to transactions. No attempt was made to distinguish between inputs, outputs and inquiries: all were simply regarded as "service requests." For classes, attributes and associations were mapped to DETs, and multi-valued attributes/associations to RETs. For methods, simple and compound arguments were mapped to DETs and RETs respectively.

Each class and method was given a number of points, based on the numbers of DETs and RETs, in the same way as traditional function points. As an initial formulation, OOFPs adopted the classification and weighting tables from the traditional FP method.

OOFPs are similar in structure to most other approaches, in mapping classes to files and services or messages to transactions. But OOFPs had the advantage that they could be counted automatically. Some design decisions were made to ensure that was the case. Tools were constructed to automate the process.

Pilot studies indicated that OOFPs were good predictors of size in LOC [13], though other primitive OO measures did as well.

An interesting outcome from the empirical research conducted with OOFPs was concerned with different ways to handle generalization and aggregation. With aggregation, is it best to count an entire aggregation structure as a single logical file, recursively joining lower level aggregations, or should the classes stay separate? With generalization, is it best to count each individual class with only its own attributes and associations, or should one consider as a different logical file the whole collection of classes comprised in the entire path from the root superclass to each leaf subclass? These decisions affect the number of classes counted, and the number of attributes within each class. The best approach, judging by how accurately

OOFPs predicted LOC, was to perform full inheritance before counting files (so all inherited attributes are counted for a file, and only leaf classes are counted), but not to aggregate classes (so just treat aggregation as equivalent to association),

Development of OOFPs has lapsed. Some of the ideas in OOFPs have fed into current work by Cantone et al. [16].

7.2.2 Use Case Points

Use case points have been proposed as an early predictor of software development effort. They are calculated from a use case model.

The components that are measured are different to the components of FPA (actors and use cases, rather than data and transactions). Another key difference from FPA is that the resulting number is seen as an indicator of effort, not size. But the stages involved in calculating use case points are modelled on the calculation of IFPUG function points.

The method was proposed by Karner in 1993 [62] (accessible descriptions are in [93] and [68]).

The method is based on identifying and classifying actors and use cases, and then adjusting for various technical and environmental factors that would influence development effort.

There are three stages to the process.

1. Each actor in the use case model is classified as simple, average, or complex. The classification is based on the nature of the interface between the actor and the application. The idea is that more complex interfaces involve more programming effort, so more use case points are given to them. Simple, average, and complex actors receive 1, 2 and 3 points respectively.
2. Each use case in the model is classified as simple, average, or complex. The classification is based on the number of “transactions” involved in the use case. A transaction is an atomic set of activities occurring between an actor and the system, occurring entirely or not at all. Again, the idea is that the more tasks that are involved in a use case the more programming effort there will be, so more use case points are awarded to use cases with more transactions. Simple, average, and complex use cases receive 5, 10 and 15 points respectively.

   The total number of points for all use cases and actors is summed, giving Unadjusted Use Case Points (“UUCP”). UUCP are meant to account for effort that is related to the inherent size of the task.
3. The use case points are adjusted to take into account 13 technical factors and 8 environmental factors that are expected to influence effort. Technical factors relate to aspects of the application itself. Environmental factors relate to the development team and environment.

> Each of these 21 factors is scored from 0 to 5: 0 means the factor is irrelevant for the project; 5 means the factor is essential. The scores for the technical factors are used to compute the Technical Complexity Factor ("TCF"), a number ranging from 0.6 to 1.35. The scores for the environmental factors are used to compute the Environment Factor ("EF"), a number ranging from 0.425 to 1.7.
>
> The UUCP is multiplied by both of the adjustment factors to give Adjusted Use Case Points ("UCP").

The analogies with IFPUG function points are obvious. Relevant components are classified to one of three levels and given use case points according to level; adjustment factors are calculated and applied in a manner similar to FPA; several of the technical adjustment factors are even the same as in FPA.

A difference is that the adjustment factors do not all have the same weights. The Environmental Factor is also new. It would be clearly inappropriate in a purported size measure, but it makes sense for an indicator of effort.

All the criticisms about inappropriate scale transformations and operations that apply to IFPUG FPA apply to use case points too. But the criticism that adjustment factors are effort drivers, not size drivers, no longer matters. Uncertainty about what the component weights represent is also removed: use case points are unequivocally intended to predict effort.

Some case studies (e.g., [12]) have found use case points to be good predictors of effort; about 15 to 30 staff-hours per use case point seems typical. How use case points compare with other estimation approaches has not been evaluated though, apart from a study in which they outperformed expert judgement [11].

Use case points are sensitive to an issue which is common in the world of use case modelling: writing the use cases and their constituent transactions at a standard level of granularity. Standards are needed on how to write use cases if comparisons across organizations are to be attempted. This is improving, with the publication of books like [17].

As with any other approach of this sort, manual counting of use case points is undesirable. It is time-consuming, and there is some subjectivity. Tools exist to identify actors and use cases, and to handle the calculation steps, but there remains the problem of classifying actors and use cases. Kusumoto et al. [68] have proposed heuristics for doing this automatically, and have developed a prototype tool.

7.3 Summary

Several proposals have been described above. The various methods may be compared on several criteria, including structure, how much information is considered, and when the information is available; objectivity and the potential for automation; and value to a user.

There is a trend towards considering ever more information from a UML model in order to come up with a size measure. This may give more useful results. It is also likely to improve the identification of FP component types, if that is the aim, as more information is available. The down-side is that some methods can only be applied later in the life cycle, perhaps well into the design stage.

In the end, the value of each method depends on how useful the measures turn out to be for project management. This is difficult to judge from the literature, because the proposals have been validated in different ways. For example, Fetcke concentrated on demonstrating that his method was unambiguous enough to be applied in practice. Schooneveldt showed in a case study that his method gave a result similar to a traditional FP count. Graham developed a tool to estimate effort from task points, but its accuracy is not reported in the literature. OOFPs were compared with LOC, not function points or effort. Use case points are validated by relating them to effort.

This is an area of active research. Among current activity, the work of Abrahão's group and Cantone's group look most promising, along with development of use case points. We can expect new proposals for mapping UML concepts to IFPUG and COSMIC components, and new FPA-like proposals for OO and other software. We can also expect more work on validation of different proposals, including comparative evaluations.

8. Function Point Standards

Different functional sizing methods take different interpretations on the concepts of software functionality. This leads to inconsistencies between methods, and also between different counters using the same method.

To resolve these inconsistencies, and define more rigorously what functional size measurement means, a standardization project was established within ISO/IEC. A working group (WG12) was put together in 1993, under ISO/IEC JTC1 Subcommittee 7 (whose area of work is software and systems engineering). ISO/IEC JTC1/SC7/WG12 had representatives from 12 countries, and also from the main function point users groups. Its task was to develop standards for functional size measurement.

The first task was to define standards for functional size measurement as a general concept. It was 10 years before a series of five standards were all approved, that collectively make up ISO/IEC 14143 *Functional Size Measurement*:

1. *Definition of concepts* (1998) [44]. This part identifies the common fundamental characteristics of functional size measurement methods, and defines a set of generic mandatory requirements for a method to be called a Functional Size Measurement Method ("FSMM").

2. *Conformity evaluation of software size measurement methods to ISO/IEC 14143-1:1998* (2002) [45]. This part is used by people needing to verify that a given software FSMM complies with the requirements of 14143-1.
3. *Verification of functional size measurement methods* (2003) [50]. This part is used by people needing to check the effectiveness of a particular FSMM as a measurement technique.
4. *Reference model* (2003) [51]. This provides a collection of reference user requirements, that can be used to test the effectiveness of a particular FSMM for different software types in different environments. It also provides the means to compare measurement results between FSMMs. methods.
5. *Determination of functional domains for use with functional size measurement* (2004) [52]. An important issue with FSMMs is their applicability to the functional domain of the software they are measuring. This part describes how to define functional domains, and provides guidance for classifying functional requirements over functional domains.

A sixth part (*Guide for the use of ISO/IEC 14143 series and related international standards*) provides guidance on how to select a suitable functional size measurement method. This part is presently at the balloting stage of the approval process.

Following the publication of the ISO/IEC 14143 series, four major function point approaches have been evaluated against that series and have gained recognition as ISO/IEC standards themselves:

- IFPUG's Counting Practices version 4.1 [40] unadjusted (i.e., without the VAF) is published as ISO/IEC 20926 [48].
- Mark II function points [104] is published as ISO/IEC 20968 [46].
- The Netherlands Software Metrics Association's variant of IFPUG function points is published as ISO/IEC 24570 [49].
- The COSMIC-FFP method is published as ISO/IEC 19761 [47].

It is important to note that a Value Adjustment Factor is not part of functional size measurement as defined by ISO/IEC 14143-1. COSMIC-FFP never included an adjustment phase anyway. The others have had to drop that component in order to conform to ISO/IEC 14143-1. Many people may still use a VAF, but that part of their counting does not comply with ISO/IEC's standard definitions. For example, Release 4.2 of IFPUG's Counting Practices Manual reinstates the VAF that had been optional in Release 4.1 Unadjusted, and so does not comply with ISO/IEC 14143-1.

ISO/IEC has not attempted to decide between the relative merits of different sizing methods. It is up to the market to decide.

9. Conclusions

Many approaches to functional size measurement have been proposed over the 25 years since function points were first described. We have described the main proposals and a few other variants.

There are now three major methods in use: IFPUG function points, 25 years old; Mark II function points, 15 years old; and COSMIC-FFP, 5 years old.

IFPUG has history and marketing behind it. A large base of experience permits benchmarking in many industry sectors. IFPUG holds the dominant share of the functional sizing market today, and will for some time.

IFPUG function points are not likely to develop further. Though IFPUG considers possible research areas (see, for example, [102]), it is a basic tenet that new ideas must require no change to the basic IFPUG structure.

Mark II is in a similar, though less dominant, position. Further development is unlikely, because Mark II is essentially considered to be replaced by COSMIC-FFP.

COSMIC-FFP is likely to increase its share of the functional sizing market, particularly in the real-time domain where other approaches have never been very successful.

With all of these approaches, we can expect to see continued refinement of guidelines, to reduce subjectivity and improve consistency between counters. There will also be some interest in mappings between other software models (such as UML) and the components counted in these approaches.

With IFPUG and Mark II, most work is likely to be based around exploiting the large experience base that has accumulated: for example, in benchmarking [42, 60], and development of estimation techniques [81]. The sizing methods are mature (COSMIC would say dated), so emphasis will be on using the measurements rather than developing the measure and developing the experience base.

Work on COSMIC-FFP will also involve refinement of measurement guidelines (for example, the definition of "layers" was updated in August 2004). The main effort in the short term will be based on promoting and expanding its use, developing tools to support its use, and building an experience base.

Business software and real-time software are now well addressed by functional sizing methods. The problem of how to size scientific software, featuring complex algorithms, remains.

ACKNOWLEDGEMENTS

Thanks to Alain Abran and Pam Morris for their reviews of an earlier version of this chapter.

References

[1] Abrahão S., Poels G., Pastor O., "Comparative evaluation of functional size measurement methods: An experimental analysis", Technical Report Working Paper 2004/234, Ghent University, March 2004.

[2] Abrahão S., Poels G., Pastor O., "Functional size measurement method for object-oriented conceptual schemas: Design and evaluation issues", Technical Report Working Paper 2004/233, Ghent University, March 2004.

[3] Abran A., "Analysis of the measurement process of function point analysis", PhD thesis, École Polytechnique de Montréal, March 1994.

[4] Abran A., Desharnais J.-M., Oligny S., St-Pierre D., Symons C., *COSMIC-FFP Measurement Manual, Version 2.2, The COSMIC Implementation Guide for ISO/IEC 19761:2003*, École de technologie supérieure, Université du Québec, Montréal, 2003, http://www.lrgl.uqam.ca/cosmic-ffp/manual.jsp.

[5] Abran A., Robillard P.N., "Function points: A study of their measurement processes and scale transformations", *Journal of Systems and Software* **25** (2) (May 1994) 171–184.

[6] Abran A., Robillard P.N., "Function points analysis: An empirical study of its measurement processes", *IEEE Transactions on Software Engineering* **22** (12) (December 1996) 895–910.

[7] Abran A., Symons C., Oligny S., "An overview of COSMIC-FFP field trial results", in: *Proc. ESCOM 2001, London,* April 2001.

[8] Albrecht A.J., "Measuring application development productivity", in: *Proc. IBM Applications Development Symposium*, IBM, October 1979, pp. 83–92. Reprinted in *Programming Productivity: Issues for the Eighties*, second ed., IEEE Computer Society, Los Alamitos, CA, 1986.

[9] Albrecht A.J., "AD/M productivity measurement and estimate validation", Technical Report CIS & A Guideline 313, IBM, November 1984.

[10] Albrecht A.J., Gaffney J., "Software function, source lines of code and development effort prediction: A software science validation", *IEEE Transactions on Software Engineering* **9** (6) (June 1983) 639–648.

[11] Anda B., "Comparing effort estimates based on use case points with expert estimates", in: *Proc. EASE2002 (Empirical Assessment in Software Engineering), Keele, UK*, April 2002.

[12] Anda B., Dreiem H., Sjøberg D.I.K., Jørgensen M., "Estimating software development effort based on use cases—experiences from industry", in: Gogolla M., Kobryn C. (Eds.), *UML 2001—Proc. 4th International Conference on the UML*, in: *Lecture Notes in Computer Science*, vol. 2185, Springer-Verlag, Berlin/New York, October 2001, pp. 487–504.

[13] Antoniol G., Fiutem R., Lokan C., "Object-oriented function points: An empirical validation", *Empirical Software Engineering* **8** (3) (September 2003) 225–547.

[14] Antoniol G., Lokan C., Caldiera G., Fiutem R., "A function-point like measure for object oriented software", *Empirical Software Engineering* **4** (3) (September 1999) 263–287.

[15] Bock D.B., Klepper R., "FP-S: A simplified function point counting method", *Journal of Systems and Software* **18** (3) (July 1992) 245–254.

[16] Cantone G., Pace D., Calavaro G., "Applying function point to Unified Modeling Language: Conversion model and pilot study", in: *Proc. 10th International Symposium on Software Metrics*, IEEE, September 2004.

[17] Cockburn A., *Writing Effective Use Cases*, Addison–Wesley, Reading, MA, 2000.

[18] Conte S.D., Dunsmore H.E., Shen V.Y., *Software Engineering Metrics and Models*, Benjamin–Cummings, Redwood City, CA, 1986.

[19] DeMarco T., *Controlling Software Projects*, Yourdon Press, 1982.

[20] DeMarco T., "An algorithm for sizing software products", *Performance Evaluation Review* **12** (2) (1984) 13–22.

[21] Desharnais J.-M., Hudon G., "Adjustment model for function point scope factors—a statistical study", in: *Proc. Spring Conference, Florida*, IFPUG, 1990.

[22] Dolado J.J., "A study of the relationships among Albrecht and Mark II function points, lines of code 4GL and effort", *Journal of Systems and Software* **37** (2) (May 1997) 161–173.

[23] Dreger J.B., *Function Point Analysis*, Prentice Hall, New York, 1989.

[24] European Function Point Users Group, "Function point counting practices for highly constrained systems", 1993.

[25] Fenton N.E., Pfleeger S.L., *Software Metrics: A Rigorous and Practical Approach*, second ed., Thomson, Washington, DC, 1997.

[26] Fetcke T., "The warehouse software portfolio—a case study in functional size measurement", Technical Report Research Report 1999-20, TU Berlin, 1999.

[27] Fetcke T., Abran A., Nguyen T.-H., "Mapping the OO-Jacobson approach into function point analysis", in: *Proc. TOOLS-23'97*, IEEE, August 1997.

[28] Furey S., "Why we should use function points", *IEEE Software* **14** (2) (March/April 1997) 28–30.

[29] Garmus D., Herron D., *Measuring the Software Process—A Practical Guide to Functional Measurements*, Prentice Hall, New York, 1996.

[30] Garmus D., Herron D., *Function Point Analysis: Measurement Practices for Successful Software Projects*, Addison–Wesley, Reading, MA, 2001.

[31] Graham I., "Making progress in metrics", *Object Magazine* **6** (8) (1996) 68–73.

[32] Hotle M., "Understanding and improving the AD estimating process", Technical report, Gartner Group, November 1996.

[33] Humphrey W.S., *A Discipline for Software Engineering*, Addison–Wesley, Reading, MA, 1995.

[34] IFPUG, *Function Point Counting Practices Manual, Release 3.0*, International Function Point Users Group, Westerville, Ohio, 1990.

[35] IFPUG, *Function Point Counting Practices: Case Study 2 (Analysis—ERD, DFD Construction—DB2 Data Base, Graphical User Interface GUI Windows)*, International Function Point Users Group, Westerville, Ohio, 1994.

[36] IFPUG, *Function Point Counting Practices Manual, Release 4.0*, International Function Point Users Group, Westerville, Ohio, 1994.

[37] IFPUG, *Function Point Counting Practices: Case Study 1 (Analysis—ERD, Process hierarchical model Construction—IMS data base, Text Base Screen Implementation)*, International Function Point Users Group, Westerville, Ohio, 1996.

[38] IFPUG, *Function Point Counting Practices: Case Study 3 (Object-Oriented Analysis, Object-Oriented Design)*, International Function Point Users Group, Westerville, Ohio, 1996.

[39] IFPUG, *Function Point Counting Practices: Case Study 4 (TRACS—A Traffic Control System with Real-Time Components)*, International Function Point Users Group, Westerville, Ohio, 1998.

[40] IFPUG, *Function Point Counting Practices Manual, Release 4.1*, International Function Point Users Group, Westerville, Ohio, 1999.

[41] IFPUG, *Function Point Counting Practices Manual, Release 4.2*, International Function Point Users Group, Princeton Junction, New Jersey, 2004.

[42] ISBSG, *Worldwide Software Development—the Benchmark. Release 5*, International Software Benchmarking Standards Group, 1998.

[43] ISBSG, "Projects sized using COSMIC full function points", in: *The Benchmark. Release 8*, International Software Benchmarking Standards Group, 2004, Chapter 8.

[44] SO/IEC, "14143-1:1998, Functional size measurement—part 1: Definition of concepts", 1998.

[45] ISO/IEC, "14143-2:2002, Functional size measurement—part 2: Conformity evaluation of software size measurement methods to ISO/IEC 14143-1:1998", 2002.

[46] ISO/IEC, "20968:2002, Mk II function point analysis—counting practices manual", 2002.

[47] ISO/IEC, "19761:2003, COSMIC-FFP—a functional size measurement method", 2003.

[48] ISO/IEC, "20926:2003, IFPUG 4.1, unadjusted functional size measurement method—counting practices manual", 2003.

[49] ISO/IEC, "24570:2003, NESMA functional size measurement method version 2.1", 2003.

[50] ISO/IEC, "TR 14143-3:2003, Functional size measurement—part 3: Verification of functional size measurement methods", 2003.

[51] ISO/IEC, "TR 14143-4:2003, Functional size measurement—part 4: Reference model", 2003.

[52] ISO/IEC, "TR 14143-5:2004, Functional size measurement—part 5: Determination of functional domains for use with functional size measurement", 2004.

[53] Jackson M., *Principles of Program Design*, Academic Press, San Diego, CA, 1975.

[54] Jeffery D.R., Stathis J., "Function point sizing: Structure, validity and applicability", *Empirical Software Engineering* **1** (1) (March 1996) 11–30.

[55] Jenner M.S., "Automation of counting of functional size using COSMIC-FFP in UML", in: *Proc. 2002 International Workshop on Software Measurement, Magdeburg, Germany*, October 2002.

[56] Jones C., "What are feature points?", Software Productivity Research, 1992.

[57] Jones C., "Backfiring: Converting lines of code to function points", *Computer* **28** (11) (November 1995) 87–88.

[58] Jones C., *Applied Software Measurement*, second ed., McGraw–Hill, New York, 1996.

[59] Jones C., "Should the 'lines of code' metric be viewed as professional malpractice?", *Voice* **1** (2) (1997) 10–14.

[60] Jones C., *Software Assessments, Benchmarks, and Best Practices*, Addison–Wesley, Reading, MA, 2000.

[61] Jones C., "Programming languages table", http://www.spr.com/products/programming.htm, August 2003.

[62] Karner G., "Metrics for objectory", Diploma thesis, University of Linköping, Sweden, No. LiTH-IDA-Ex-9344:21, December 1993.

[63] Kemerer C.F., "Reliability of Function Points measurement: A field experiment", *Communications of the ACM* **36** (2) (February 1993) 85–97.

[64] Kitchenham B., "The problem with function points", *IEEE Software* **14** (2) (March/April 1997) 29–31.

[65] Kitchenham B., Känsälä K., "Inter-item correlations among function points", in: *Proc. 15th International Conference on Software Engineering*, IEEE, May 1993, pp. 477–480.

[66] Kitchenham B.A., "Empirical studies of assumptions that underlie software cost-estimation models", *Information and Software Technology* **34** (4) (April 1992) 211–218.

[67] Kitchenham B.A., Pfleeger S.L., Fenton N., "Towards a framework for software measurement validation", *IEEE Transactions on Software Engineering* **12** (12) (December 1992) 929–944.

[68] Kusumoto S., Matukawa F., Inoue K., Hanabasa S., Maegawa Y., "Effort estimation tool based on use case points method", in: *Proc. 10th International Symposium on Software Metrics*, IEEE, September 2004.

[69] Lokan C.J., "An empirical study of the correlations between function point elements", in: *Proc. 6th International Symposium on Software Metrics*, IEEE, November 1999, pp. 200–206.

[70] Lokan C.J., "Statistical analysis of ISBSG data and function point analysis", in: *Proc. Australian Software Metrics Conference*, Australian Software Metrics Association, November 1999.

[71] Lokan C.J., "An empirical analysis of function point adjustment factors", *Information and Software Technology* **42** (9) (June 2000) 649–659.

[72] Lokan C.J., Abran A., "Multiple viewpoints in functional size measurement", in: *Proc. 1999 International Workshop on Software Measurement, Lac Superieur, Quebec, Canada*, September 1999, pp. 121–131.

[73] Longstreet D., "Function points applied to new and emerging technologies", http://www.softwaremetrics.com, 2000.

[74] Longstreet D., "Use cases and function points", http://www.softwaremetrics.com, 2000.

[75] Longstreet D., "OO and function points", http://www.softwaremetrics.com, 2001.

[76] Lother M., Dumke R., "Points metrics—comparison and analysis", in: *Current Trends in Software Measurement, Aachen, Germany*, Shaker Publishing, 2001, pp. 228–267.

[77] Low G.C., Jeffery D.R., "Function points in the estimation and evaluation of the software process", *IEEE Transactions on Software Engineering* **16** (1) (January 1990) 64–71.

[78] McCabe T.J., "A complexity measure", *IEEE Transactions on Software Engineering* **2** (4) (1976) 308–320.

[79] Meli R., "Functional metrics: Problems and possible solutions", in: *Proc. FESMA'98, Antwerp*, 1998, pp. 503–514.

[80] Meli R., “Functional and technical software measurement: Conflict or integration”, in: *Proc. FESMA 2000*, 2000.

[81] Meli R., Santillo L., “Function point estimation methods: A comparative overview”, in: *Proc. FESMA’99, Amsterdam*, 1999, pp. 271–286.

[82] Minkiewicz A., “Measuring object-oriented software with predictive object points”, in: *Proc. ASM’97—Applications in Software Measurement, Atlanta*, October 1997.

[83] Morris P., “Function point audits”, in: *Proc. Australian Software Metrics Conference*, Australian Software Metrics Association, September 2004.

[84] Morris P., Desharnais J.-M., “Measuring ALL the software not just what the business uses”, in: *Proc. Fall Conference, Orlando*, IFPUG, September 1998.

[85] Morris P., Desharnais J.-M., “Function point analysis: Validating the result”, Technical Report Version 1.3, Total Metrics Pty Ltd, February 1999.

[86] Oligny S., Abran A., “On the compatibility between full function points and IFPUG function points”, in: Kusters R., Cowderoy A., Heemstra F., van Veenendaal E. (Eds.), *Project Control for Software Quality (Proc. ESCOM’99)*, Shaker Publishing, 1999.

[87] Rask R., Laamanen P., Lyytinen K., “Simulation and comparison of Albrecht’s function points and DeMarco’s function bang metrics in a CASE environment”, *IEEE Transactions on Software Engineering* **19** (7) (July 1993) 661–671.

[88] Redgate N., Tichenor C.B., “Measure size, complexity of algorithms using function points”, *Crosstalk* (February 2001) 12–15, http://www.stsc.hill.af.mil/crosstalk/.

[89] Redgate N., Tichenor C.B., “Measuring calculus integration formulas using function point analysis”, *Crosstalk* (June 2002) 24–27, http://www.stsc.hill.af.mil/crosstalk/.

[90] Rudolph E.E., Wittig G.E., Finnie G.R., Morris P.M., “Verifying function point values”, in: *Proc. FESMA’98, Antwerp*, 1998.

[91] Rule P.G., “The importance of the size of software requirements”, in: *Proc. NASSCOM Conference, Mumbai, India*, February 2001.

[92] Saaty T.L., *The Analytic Hierarchy Process*, McGraw–Hill, New York, 1980.

[93] Schneider G., Winters J.P., *Applying Use Cases*, second ed., Addison–Wesley, Reading, MA, 2001.

[94] Schooneveldt M., Hastings T., Mocek J., Fountain R., “Measuring the size of object-oriented systems”, in: *Proc. 2nd Australian Conference on Software Metrics*, Australian Software Metrics Association, November 1995, pp. 83–93.

[95] Sneed H., “Estimating the development costs of object-oriented software”, in: *Proceedings of 7th European Software Control and Metrics Conference, Wilmslow, UK*, May 1996.

[96] St-Pierre D., Maya M., Abran A., Desharnais J.-M., “Adapting function points to real-time software”, in: *Proc. Fall Conference, Scottsdale*, IFPUG, 1997.

[97] Symons C., Personal communication. December 2003.

[98] Symons C., “Come back function point analysis (modernised)—all is forgiven!”, in: *Proc. 4th European Conference on Software Measurement and ICT Control, Heidelberg, Germany*, FESMA–DASMA, May 2001, pp. 413–426.

[99] Symons C.R., “Function point analysis: Difficulties and improvements”, *IEEE Transactions on Software Engineering* **14** (1) (January 1988) 2–11.

[100] Symons C.R., *Software Sizing and Estimating: Mk II FPA*, Wiley, New York, 1991.

[101] Symons C.R., "Conversion between IFPUG 4.0 and MkII function points, version 3.0. Software Measurement Services", http://www.gifpa.co.uk, 1999.

[102] Tichenor C.B., "Recommendations for further function point research", http://www.softwaremetrics.com, 2000.

[103] Uemura T., Kusumoto S., Inoue K., "Function-point analysis using design specifications based on the Unified Modelling Language", *Software Maintenance and Evolution: Research and Practice* **13** (4) (July/August 2001) 223–243.

[104] UK Software Metrics Association, "MkII FPA counting practices manual version 1.3.1", http://www.gifpa.co.uk, October 1998.

[105] UQAM Software Engineering Management Research Laboratory, "Full function points measurement manual version 2.0", 1999.

[106] Verner J.M., Tate G., Jackson B., Hayward R.G., "Technology dependence in function point analysis: A case study and critical review", in: *Proc. 11th International Conference on Software Engineering*, IEEE, 1989, pp. 375–382.

[107] Whitmire S.A., "Posting to function point mailing list", 21 September 1995.

[108] Whitmire S.A., "3D function points: Scientific and real-time extensions to function points", in: *Proc. 10th Pacific Northwest Software Quality Conference, Portland, Oregon*, 1992, Pacific Agenda.

[109] Whitmire S.A., "Applying function points to object oriented software", in: Keyes J. (Ed.), *Software Engineering Productivity Handbook*, McGraw–Hill, New York, 1993, pp. 229–244, Chapter 13.

[110] Whitmire S.A., "An introduction to 3D function points", *Software Development* (April 1995) 43–53.

[111] Wittig G.E., Finnie G.R., Rudolph E.E., Morris P.M., "Project research design to validate FPA coefficients using AHP", in: *Proc. 3rd Australian Software Metrics Conference*, Australian Software Metrics Association, November 1996.

The Role of Mathematics in Computer Science and Software Engineering Education

PETER B. HENDERSON

Department of Computer Science and Software Engineering
Butler University
Indianapolis, IN
USA
phenders@butler.edu

Abstract

Continuous mathematics is an important foundation for many science and engineering disciplines. Similarly, discrete mathematics and logic are foundations for computer based disciplines such as computer science, software engineering and information systems. However, these essential foundations are often taught independently and relevant connections to computing, required to motivate the mathematics, are usually not made.

Mathematics is a natural complementary discipline for learning, understanding and appreciating many fundamental computer science concepts. Accordingly, for the students benefit, foundational mathematics should be introduced early and integrated throughout the curriculum. This chapter provides motivation, specific and general guidelines, curriculum structures and a representative first course for significantly enhancing the mathematical reasoning skills of computer science and software engineering graduates. Over twenty years teaching foundational computing, talking to and surveying students, alumni, educators and corporate people have convinced the author that graduates of mathematically oriented programs will be better general problem solvers and software practitioners.

ISSN: 0065-2458/DOI 10.1016/S0065-2458(05)65008-5

1. Introduction

Based on my 35 years of experience teaching electrical engineering, computer science and software engineering, I would affirm that “Mathematics and logical reasoning are important for success in almost all disciplines, but especially quantitative areas such as the sciences and engineering.” I could simply stop here, claiming I have stated my position, but the editors wish more, and I should make a strong case for this position. You may agree, disagree, or be tentative about this statement. If you agree, I hope further reading will help you to crystallize your thinking and provide solid arguments for discussions with others. If you disagree or are unsure, I hope further reading will help open your mind to thinking carefully about these issues.

Too often in the education of computer science and software engineering undergraduate students mathematics is relegated to the background. This is particularly true in the United States [48], the educational system with which I am most familiar. The foundational mathematics for computer science and software engineering is frequently introduced too late in the curriculum and important connections with relevant CS/SE concepts are not made. Ideally, mathematics should be integrated throughout the CS/SE curriculum [25]. I would like to claim I have solved this problem and can provide empirical evidence supporting this claim. I have not. However, what I attempt to offer is a seed for change in educational philosophy with the realization that this will take time. With this in mind, I hope you will continue reading and will not be disappointed.

In this chapter I provide motivation, specific and general guidelines, curriculum structures and a representative first course for significantly enhancing the mathematical reasoning skills of computer science and software engineering graduates. Mathematics is a natural complementary discipline for learning, understanding and appreciating many fundamental computer science concepts.

2. Mystery Novels and John Wooden

I like to think that this chapter will be crisp and sharp, leading you along a carefully orchestrated journey of understanding. But I fear it might be more like a mystery novel where various pieces of a puzzle are presented and you must draw your own conclusions. Those preferring the abbreviated version can skip to the conclusions, but will miss many of the key ideas.

As educators, we should always be seeking role models. One of the best, in my opinion, is John Wooden, the former basketball coach of the University of California at Los Angeles (UCLA). I have watched his TV show "The Pyramid of Success," read his book [49], and try to use his philosophy of life in my classrooms and interactions with students. Education is about life, not just the pursuit of knowledge. My favorite adaptation of Wooden's philosophy is "Success is the peace of mind that comes from striving to do your best." You will see similar expositions of this exceptional educational philosophy sprinkled throughout this chapter. I hope you find them as useful as I have.

To begin the journey, let me share with you an email I recently received from a former student who took my freshmen foundations of computer science course in 1997.

> "Dear Prof. Henderson, It's been a while since I've taken your discrete mathematics course (CSE 113) at {SUNY} Stony Brook, and I wanted to track you down to say thank you! That class, although I failed it as a freshman, was one of the most important and useful classes I have ever taken.
> I am no longer studying comp. sci. academically but am currently completing a double masters program in business and finance. But your class has provided me tools that have not only been beneficial for computers, but for every aspect of my life. I actually believe that everyone regardless of their major should be forced to take a discrete mathematics/logic course as a freshman."

Most educators have received similar complements from former students, but what struck me about this one is that this student came to understand, even after failing the class, the fundamental educational lessons I had intended to convey.

3. Computer Science and Software Engineering

To appreciate the role of mathematics for students studying computer science (CS) and software engineering (SE), it is necessary to understand the relationship between these two disciplines. Basically this reduces to the distinction between science and engineering [44]. Most people understand the differences and relationships between chemistry and chemical engineering, or between physics and electrical or mechanical engineering. One is science, primarily advancing knowledge of the discipline, and the other engineering, which is primarily the application of this knowledge to achieving the technological needs of mankind.

As currently taught, it might be difficult to distinguish the attributes of a computer science graduate from those of a software engineering graduate. The latter might have a few extra courses designated software engineering; however, both are primarily educated for entry level programming positions. In my mind, this is neither science nor engineering.

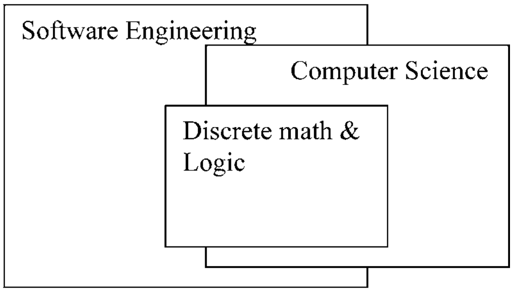
In the United States accreditation of computer science and software engineering undergraduate programs is under the preview of ABET, the Accreditation Board for Engineering and Technology [1]. Below is the definition of engineering from the Accreditation Board for Engineering and Technology in the U.S. [1]:

> "Engineering is the profession in which a knowledge of the mathematical and natural sciences, gained by study, experience, and practice, is applied with judgment to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind."

Software engineering does not fit this definition very well. What is the corresponding "natural science," or "materials and forces of nature?" I believe that computer science, a man made science, is the foundational science for software engineering, that the "materials" are mainly conceptual rather than physical, and the "forces of nature" are really "laws of the universe." Accordingly, I offer the following alternative definition:

> "Engineering is the profession in which knowledge of the foundational mathematics and sciences, gained by study, experience, and practice, are applied with judgment to develop ways to utilize, economically, the materials, concepts and laws of the universe for the benefit of mankind."

"Foundational mathematics and sciences" seems to be a better, more general way to express these relationships and dependencies. For software engineering, they might be roughly illustrated by the following picture with the foundational mathematics being discrete mathematics including logic. The view I see evolving is undergraduate computer science primarily as preparation for graduate studies for advancing knowledge, and software engineering as the professional track for the



design/development of software systems. Graduates of both programs should be competent programmers, but this would not be their primary career path. Programmers are fundamentally technicians, professionals who know how to achieve a goal without comprehensive knowledge of the underlying foundational mathematics or science.

4. Foundational Mathematics

“The person who knows *HOW* will always have a job, but the person who knows *WHY* will always be his/her boss [42].” Training versus education! In my experience most students are striving to get the knowledge they need to get a job (HOW) rather than to understand the foundations of that knowledge (WHY). This is the “filling a vessel” view of education [12] which is also one feature of the knowledge survey of software practitioners [34].

Knowledge, rather than understanding, is often the driving force behind curriculum development for many disciplines, primarily due to pressure from employers, administrators, and students. Also, knowledge based courses are easier to teach and to assess student performance (e.g., “What year was the war of 1812 fought?”).

How does this relate to mathematics? For the sciences and engineering, mathematics is a prerequisite for understanding. A personal example will help to make the connection. I was an amateur radio operator at age 12. This required a broad range of knowledge of electronics. I *knew* but did not *understand* the required knowledge such as ohms laws, standing wave ratio, voltage decay of a resistor–capacitor circuit, etc. The “understanding” came in my first electronic circuits course which I took in the fourth semester of my undergraduate electrical engineering program, after Calculus I, Calculus II, Differential Equations, Physics I, Physics II, Chemistry I, Chemistry II and other foundational engineering courses. For instance, the formula that models the behavior of voltage decay of a resistor–capacitor (RC) circuit is derived from calculus and differential equations. Electronic technicians, hobbyists and amateur radio operators know HOW to use such formulas, electrical engineers understand WHY they apply because these foundational principles are used, reinforced and expanded upon in all technical courses in the curriculum. They are an integral component of engineering courses; upon graduation, mathematical and scientific foundations are intrinsic in an engineers reasoning.

Personally, I have always found this “grounding” very comforting because I felt I could understand something from fundamental principles when necessary. This is the same feeling I would like my CS and SE students to acquire. However, “job” translates to “knowledge (HOW)” rather than “understanding (WHY).” (“Teach me how to program, not the principles underlying why a concept works!”) Students learn to develop algorithms, to trace them, to develop test scenarios, and to implement, test and debug them. They may be introduced to formal techniques, but are rarely required to use them extensively. Indeed, most CS/SE graduates are unable to give a coherent argument that a simple algorithm/program works correctly based upon a set of specifications. Contrast this with engineering graduates who have learned to reason from first principles. Of course, this raises the question “What are the first principles of CS/SE?” I claim they are mathematics, primarily discrete, and logic.

5. Models

Webster’s dictionary defines a model as “a miniature representation of something; *or* a pattern of something to be made.” Engineering employs both physical models and mathematical models, often comparing one to the other as a method of checking consistency during early stages of analysis and development. Indeed, the discipline of mathematics evolved from mankind’s need to create models of nature and hence is a key tool for modeling in most disciplines. Evidence of this in undergraduate education comes from the Mathematics Association of America CUPM Curriculum Foundations Project [10] in which only one topic, ‘modeling,’ was found to be universally significant for all client disciplines—engineering, economics, computer science, physics, chemistry, biology, business, manufacturing, statistics, mathematics, etc. Because of software’s abstract nature, tangible models are impossible or difficult to construct; accordingly, software engineers are constrained to mathematical models. Also, there are few good tools for building models of software systems, the mathematical backgrounds required for modeling is often weak, and deadline and economic pressure frequently preclude building and checking viable system models.

It is impossible to develop a correct software system without an accurate model. There must be something guiding the development process. In contrast to traditional practicing engineers, however, software practitioners don’t usually construct formal models for checking system specifications and constraints. The model usually used is an unchecked, unverified mental model often expressed as a set of requirements and specifications. The resulting software system is usually the first complete checkable “model” of the desired artifact. Only now can it be validated against the desired mental model, the potentially inaccurate requirements/specifications, etc. using extensive testing. This deviates significantly from the traditional engineering process where the model is constructed and validated during the early stages. Could this be a reason why software systems are so susceptible to errors and problems? As with traditional engineering, mathematics is an important tool for modeling software systems.

Until recently, automated tools for modeling and checking software systems were unavailable or complex [9,29–31,33]. Now, however, undergraduate courses are developing around these concepts [37,24].

A significant component of undergraduate CS and SE education is devoted to implementing software systems rather than learning and applying fundamental mathematics. The picture below shows how students learned to use algebra as a basic mathematical modeling tool:

$$\begin{tikzcd}
\text{Problem domain} \arrow[r] & \text{Algebraic model} \arrow[d, "\text{Solving}"] \\
\text{Problem domain} & \text{Algebraic model solution} \arrow[l]
\end{tikzcd}$$

The terms “Algebraic model,” “Solving” and “Algebraic model solution” can be replaced with any meaningful context. When teaching logic, I tell students that developing a precise logical expression for a logical word problem is modeling, or when arguing a logical conclusion for word problems the following modeling diagram is representative:

$$\begin{tikzcd}
\text{Problem domain} \arrow[r] & \text{Math logic model} \arrow[d, "\text{Argument}"] \\
\text{Problem domain} & \text{Logical conclusion} \arrow[l]
\end{tikzcd}$$

There are several points I wish to make here regarding the education of CS and SE students; each of these points will be presented and discussed in more detail subsequently. First, mathematics should be presented as a tool for building and reasoning about models, including numerous examples. Second, every algorithm, structure, computer program, and/or software system is simply a ‘model’ of a process—an executable model. Third, a good grasp of logic and discrete mathematics is fundamental for constructing and reasoning about such models. And, last, but most important, all these ideas should be introduced as early as possible, ideally during the first semester of the freshman year or preferably before. In traditional engineering education, continuous mathematics is required early, and a significant component of pre-college mathematics throughout the world is focused on preparing students for calculus.[1]

6. General Mathematical Reasoning

The concepts of mathematical/abstract reasoning developed when our ancestors started counting stones and fashioning tools. Books and articles have been written about the relevance of mathematical reasoning for the general population [15]. The Working Group on Integrating Mathematical Reasoning into Computer Science Curricula defines mathematical reasoning as “Applying mathematical techniques, concepts and processes, either explicitly or implicitly, in the solution of problems—in other words, mathematical modes of thought that help us to solve problems in any domain. In its most general interpretation, every problem-solving activity requires mathematical thinking. For example, basic logic, be it used explicitly or implicitly is required for all problem-solving activities” [50].

One of the best expositions on the importance of mathematics for computer scientists and software engineers was written by Keith Devlin in two viewpoint articles “Do software engineers need mathematics?” [12] and “The real reason why software engineers need math” [13]. Here is one quote from the first article:

> “. . . software engineering is all about abstraction. Every single concept, construct, and method is entirely abstract. Of course, it doesn’t feel that way to most software engineers. But that’s my point. The main benefit they got from the mathematics they learned in school and at university was the experience of rigorous reasoning with purely abstract objects and structures.
> Moreover, mathematics was the only subject that gave them that experience. It’s not what was taught in the mathematics class that was important; it’s the fact that it was mathematical. In everyday life, familiarity breeds contempt. But when it comes to learning how to work in a highly abstract realm, familiarity breeds a sense of, well, familiarity—meaning that what once seemed abstract starts to feel concrete, and thus more manageable.”

This message is reinforced by many students and graduates with whom I have communicated. It is not the explicit knowledge or facts they recall and use every day, but rather the general mathematical way of thinking. This view is not universally held by software professionals. My good friend Bob Glass, author of “A New Answer to ‘How Important is Mathematics to the Software Practitioner?’ ” [22], and I have had numerous personal and public discussions regarding these issues [5]. Indeed,

[1] We should strive for more balance between continuous and discrete mathematics in pre-college.

the study “What Knowledge is Important to a Software Professional?” [34] he cites supports this view. This study measures mathematical knowledge, not reasoning, and I do agree that there are good software engineers who are not good at math; however, in general, they must be good abstract thinkers.

Here is a simple example of general mathematical thinking I use in my foundations of computing course. To the untrained eye, this has little to do with mathematics or computing.

> One morning, exactly at sunrise, a Buddhist monk began to climb a tall mountain. The narrow path, no more that a foot or two wide, spiraled around the mountain to a temple at the summit. The monk ascended the path at various rates of speed, stopping many times along the way to rest and to eat. He reached the temple shortly after sunset. The next day be began his journey back along the same path, starting at sunrise and again walking at various speeds with many pauses along the way. His average speed descending was, of course, greater than his average climbing speed. Show there is a spot along the path that the monk will occupy on both trips at precisely the same time of day.

Verbal arguments often fall short whereas *graphical* views (plotting distance versus time) or *abstract* views (envisioning a monk and his clone on the same day) provide a representation/context for the argument others can more readily understand. Important problem solving techniques also manifest themselves in this example. For instance, students should learn to ignore irrelevant details such as the width of the path, rates of ascent and descent, etc. given in the problem statement. Educators often underestimate how much such irrelevant details inhibit student’s problem solving abilities.

Another simple example is having students solve such problems as: What is the next number in each of the following sequences?

(a) 1, 2, 4, 7, 11, 16, ? (b) 1, 1, 2, 3, 5, 8, 13, ? (c) 1, 3, 7, 15, 31, ?

7. Patterns, It Is All About Patterns

In the preceding problem, the astute reader will see a closer connection between mathematics and computing. But what is the lesson students learn from such problems? Hint: Patterns—indeed one definition of mathematics is that it is “the study of patterns”—patterns of life, nature and the universe [15]. Engineering is the application of patterns to solving problems—hence another connection between mathematics and engineering. In this section we will explore some of the earlier themes—foundational math, models, general mathematical thinking, declarative versus imperative reasoning—in this context. In subsequent sections, the influences on and relationships to computer science and software engineering undergraduate curricula will be made.

Finding the next number in a sequence such as 1, 1, 2, 3, 5, 8, 13, ? is one example of discovering a pattern and applying it. Specifying the pattern discovered is mathematical:

$$\begin{aligned}
&f(1) = 1,\\
&f(2) = 1,\\
&f(n) = f(n-1) + f(n-2) \quad \text{for } n > 2.
\end{aligned}$$

Evaluating $f(n)$ for specified values of n is computational in nature. Even for this simple evaluation there are numerous computational approaches, or algorithms. One is recursive, applying the definition directly, another is iterative, compute the sequence $f(1), f(2), \ldots, f(n)$.

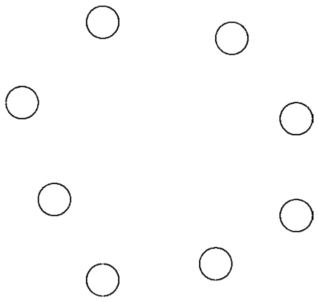
A simplistic view of developing software to solve a given problem involves discovering pattern(s), specifying and checking the pattern(s) discovered, creating a computational process (e.g., algorithm), and encoding/testing this process. One approach to educating CS and SE students is to give them lots of patterns, each with corresponding algorithmic templates, and numerous practice problems. This is the way many of our students learn to develop software, and it is also the way practicing engineers work most of the time. However, engineers have the advantage of understanding the “patterns” based upon foundational mathematical and scientific principles, and this helps them to be creative problem solvers, especially when there is no known or familiar “pattern.”

Pattern discovery requires matching (have you seen this or a similar pattern before) or inductive reasoning (finding a pattern from examples). Completely and accurately specifying a pattern requires mathematics as does checking/validating the pattern. Composing a useful algorithm from these pattern(s) is not always a direct encoding. This requires logical reasoning, familiarity with numerous similar algorithms/software components, knowledge of the application domain, and a puzzle solving mentality. General mathematical reasoning skills will help with all of these aspects.

Here is a simple problem illustrating pattern discovery:

> Eight people meet in a room. Each shakes hands with every other person once. How many total handshakes are there? If $n > 1$ people met, how many handshakes would there be?

One concrete approach a student might use to answer the first question is to perform an experiment using 8 people. This is one “brute force” approach. A second might be to create an abstract representation of the problem (recall the monk problem) perhaps by drawing 8 circles connected by straight lines and counting the lines.



Computer scientists and software engineers are usually seeking solutions for general problems (e.g., "If $n > 1$ people met, ..."). Accordingly, a third approach would be to find a general pattern. Students are encouraged to construct a table starting with small values of n and look for patterns; for example, if one knows the number of handshakes for $n - 1$ people how can the number for n people be determined? Note that this simple observation constitutes the conceptual basis for recursive and iterative problem solving.

The key observation from Table I is that $\#\text{shakes}(n) - \#\text{shakes}(n-1) = n - 1$, which students discover after careful thought and perhaps some hints. An alternate discovery approach is to "abstractly" draw $n-1$ circles. Then add an nth circle noting that $n-1$ lines, each representing one handshake, must be drawn to the existing $n-1$ circles. The result can be expressed

$$\#\text{shakes}(2) = 1,$$
$$\#\text{shakes}(n) = \#\text{shakes}(n-1) + (n-1) \quad \text{for } n > 2.$$

From this general mathematical solution, students could compose a computational solution by programming this, either recursively or iteratively. The connection between the mathematical specification and the algorithmic solution is important and will be discussed subsequently.

TABLE I

n	#shakes
2	1
3	3
4	6
5	10
6	15

The astute student will note that this computation is inefficient for larger values of n. Ideally, we seek an expression directly relating #shakes(n) with n which does not require computing intermediate values. With some hints students can find the relationship

$$\#\text{shakes}(n) = 1 + 2 + 3 + \cdots + (n - 1)$$

and ultimately the efficient "closed form" expression

$$\#\text{shakes}(n) = n(n - 1)/2.$$

Here, students used inductive reasoning to discover the relationship

$$1 + 2 + 3 + \cdots + (n - 1) = n(n - 1)/2 \quad \text{for all } n > 1.$$

Students who are familiar with such relationships might use the "have seen this pattern before" approach. However, I feel it is important that all students be able to apply fundamental principles to discover such patterns. This is not just for those special cases where they can't identify a familiar pattern, but such mathematical reasoning skills are important for developing, verifying and debugging computational solutions.

I believe current CS and SE graduates are generally not proficient with this kind of reasoning. This comes back to the "how" and "why," and the expectations CS and SE educators have of their graduates. There is a tendency to equate good software development skills with good reasoning skills.[2] I don't believe this inference is necessarily valid and hence is one of the key reasons students should learn fundamental, general, math based reasoning skills early—not just algorithmic problem solving.

For the handshaking problem the lesson does not end here. Discovering a potential relationship is just the beginning—indeed, it may not be valid for all cases. A good example is the polynomial $n^2 + n + 41$ which Euler found is prime for $n = 0, 1, 2, \ldots, 39$. Thus, checking or verifying the relationship discovered is extremely important. This is where proof techniques, such as mathematical induction, come into play. As noted earlier, discrete mathematics courses are often viewed by students as a collection of topics with little or no connection to computer science. Mathematical induction is hard, and without a motivation for learning it, students don't take it seriously.

Below is an interesting variation of this handshaking problem which could be used as a follow up:

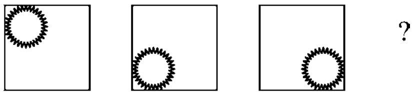
> A social psychologist was interested in the custom of handshaking. He noticed that some people are more inclined than others to shake hands when they are

[2] Recall the debate regarding "programming = computer science."

introduced. One evening when he and his wife had joined four other married couples at a party, he took advantage of the occasion to collect data. He asked each of the other nine people at the party how many people they had shaken hands with during the introductions. He received a different answer, zero through eight, from each of the nine people. You can assume that husbands and wives don't shake hands with each other during introductions, and of course, people don't shake hands with themselves. Given this information, find out how often the psychologist's wife shook hands.[3] (Note: At first it may seem that you do not have sufficient information to solve this problem. You do!! One key is finding a good scheme for representing all the information given.)

8. Inductive Thinking and Generalization

The concept of inductive thinking was introduced in the previous section. One dictionary definition of the noun 'induction' is: "reasoning from detailed facts to general principles." An associated noun and noun phrase are: 'generalization' and 'inductive reasoning.' In the latter, 'inductive' is an adjective. 'Inductively' would be the adverb. "Induction is an inference drawn from all the particulars."—Sir W. Hamilton. What is the next symbol in the following sequence?



The detailed facts (observations) are presented, the general principles is a pattern discovered (expressed as a hypothesis), and the answer is obtained by applying the general principles. Once the general principle (hypothesis) is confirmed and "mathematically" specified, more general questions can be answered. In computer-based problem solving, the process of discovering the general principles is the challenging, and the most interesting and creative, aspect.

Students are usually introduced to the term *induction* in the context of mathematical induction[4] or inductive proofs. Here a collection of examples and exercises such as "prove that $2+4+6+\cdots+2n = n(n+1)$ for all $n \geqslant 0$" are used. The important concept of inductive reasoning is lost and its relevant connections with computer science, such as discovering and reasoning about algorithms, developing test cases, and debugging are missed.

Inductive reasoning (generalization) is an important part of the discovery process for general algorithmic solutions. Novice students can usually solve specific in-

[3] See Appendix E for a reference to a solution.

[4] Mathematical induction is simply a tool for confirming (proving) that a general principle is true.

stances of problems (e.g., find the sum $4 + 6 + (-3) + 18$), but experience difficulty generalizing[5] (e.g., give an algorithm for finding the sum of any given sequence of numbers). They learn to generalize by seeing/solving many similar programming problems.

I believe that this is rote learning, and that students educated this way tend to be limited thinkers. Why? Because they don't understand or appreciate the fundamental underlying principles of inductive reasoning based problem solving. They learn to use a limited form of inductive reasoning in a very limited context: writing computer programs.

By introducing students to general forms of inductive reasoning early, I believe they will become more general thinkers, better algorithmic problem solvers, and hence, better software developers. In computer-based problem solving, inductive reasoning leads to the discovery of patterns. Mathematics can be used to describe and confirm the patterns discovered. Accordingly, mathematics is a natural complementary discipline for introducing and reinforcing inductive reasoning.

Sequences of numbers, symbols, pictures, etc. are good primers for getting students' inductive reasoning "mental juices" flowing. Have students informally specify the pattern they discover and use it to solve "what are the next n (numbers, symbols, pictures, etc.) questions." Then, simple inductive reasoning exercises such as the following can be presented.

Four match sticks can be used to make a square (Figure 1), seven to make a row of two squares (Figure 2), ten to make a row of three squares, and so on and so on. . .[6]

(a) How many match sticks are required to make a row of 12 squares? How about 37? Now try 679?

(b) In general, to make a row of $N \geqslant 1$ squares, how many match sticks are required? Express your answer in terms of N.

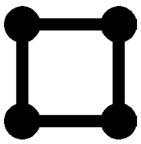


FIG. 1.

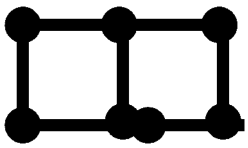


FIG. 2.

[5] One of my favorite examples of this cognitive mental block is the card stacking game (see http://www.math-in-cs.org/examples.html) where students can easily solve specific problem instances, but find it difficult to discover a general algorithmic solution.

[6] See Appendix E for solutions to parts (b), (d) and (e) below.

(c) We define a *joint* to be a point where two or more match sticks touch. For example, there are 4 joints in a single square, and 6 joints in a row of two squares. How many joints are there in a row of 6 squares? How about 22? Try 65?

(d) In general, how many joints are there in a row of $N \geqslant 1$ squares? Express your answer in terms of N.

(a) Given a row of $N \geqslant 1$ squares, let M be the number of match sticks, and J the number of joints. What is the mathematical relationship, if any, between the values of N, M and J? Note the relationship to Euler's formula on planar graphs.

A subsequent exercise could be a generalization to an N by N grid of such squares, with similar questions. There are many such mathematical inductive reasoning exercises, and recreational mathematics provides a rich source of these [4,32]. With a basic understanding of inductive reasoning, students will become more effective problem solvers and will more easily grasp important principles such as of mathematical induction, iteration invariants, algorithm correctness, formal specification, etc.

Recall induction is "reasoning from detailed facts to general principles." Since software systems usually solve problems based upon general principles, inductive reasoning plays a very important role in understanding, specifying, designing, implementing, testing, debugging and validating software.

For instance, consider the algorithm below. Inductive reasoning questions appropriate for an introductory computer science course might involve finding the values of 'j' and 'k' after the 1st, 2nd, 9th, 50th and 479th iterations and, upon termination, to give a general expression relating these values to the number of iterations, and finally to identify the total number of iterations. This is one form of algorithm analysis.

```
$j \leftarrow 0$
$k \leftarrow 2$
WHILE $j < 1000$ DO
    $j \leftarrow j + 2$
    $k \leftarrow k + j$
```

Now let's examine the role of inductive reasoning in the development of an algorithm from first principles.[7] This problem would be appropriate for an introductory computer science course, especially one requiring a discrete mathematics course as a prerequisite.

[7] Rather than the traditional rote 'template matching' approach many students try to use.

Problem: Consider a sequence of $n > 0$ integers $i_1, i_2, i_3, \ldots, i_n$ for which we wish to find the maximum value.

Understanding the Problem: Have students complete some examples to ensure an accurate understanding of the problem. For instance,

3, 4, 2	maxValue = 4
8	maxValue = ?
−6, −3	maxValue = ?
−4, 0	maxValue = ?
−3, 11, 2	maxValue = ?
3, 2, 3	maxValue = ?
empty	maxValue = ?

Inductive Reasoning from an Example: Consider the sequence 8, 3, −4, 16, 10, 34, 67, −3, 59. One pedagogical aid I have students use is to think of a window through which only one element in the sequence can be viewed at a time. With a sequence the order of viewing the elements is predetermined.[8] The first element viewed[9] is $\boxed{8}$.

Continuing this leads naturally to the table below, which students complete.

Window	maximum value seen so far
8	8
3	8
−4	?
16	?
10	?
34	?
:	:
59	?

Students begin to see the pattern of information being gathered at each step, that is, the ‘maximum value seen so far.’

Generalized Inductive Reasoning: Repeat the above reasoning for a general sequence $i_1, i_2, i_3, \ldots, i_n, n > 0$. Identify and specify the pattern of information gathered at each step as shown below. Here ‘max : int × int → int’ is a function that returns the maximum value of two integer arguments.

[8] One of several advantages of a sequence “structure” over a random access array “structure.”
[9] There must always be such a first element since the sequence in non-empty.

Window	maximum value seen so far
i_1	i_1
i_2	$\max(i_1, i_2)$
i_3	$\max\big(\max(i_1, i_2), i_3\big)$
i_4	$\max\big(\max\big(\max(i_1, i_2), i_3\big), i_4\big)$
:	:
etc.	etc.

Give this table to students and ask them to identify all occurrences of the same subexpression (e.g., $\max(i_1, i_2)$, $\max(\max(i_1, i_2), i_3)$, etc.). Using different highlighting colors for this task is ideal. Now ask students to express in writing insights gained from this activity. They should identify the pattern for computing the current maximum value of the sequence from the previous one—the generalized pattern which is, informally,

max(previous maximum value seen so far, current value in window).

Generalized Pattern: Having computed the maximum of the first $k - 1$ integers in the sequence $i_1, i_2, i_3, \ldots, i_{k-1}$ how is the maximum of the sequence $i_1, i_2, i_3, \ldots, i_k$ computed? This is the essence of a solution based upon inductive reasoning. To proceed, let maxOfSeq : integer sequence $\rightarrow$ int be a function representing a solution to the problem, and ask students to complete the following equality which is valid for each $k > 1$

$$\text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_k) = \max\big(\text{maxOfSeq}(\underline{\qquad\qquad}), i_k\big).$$

The answer students discover is:

$$\text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_k) = \max\big(\text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_{k-1}), i_k\big),$$
$$k > 1.$$

with the initial “base” case:

$$\text{maxOfSeq}(i_1) = i_1, \quad k = 1.$$

In other words, the next maximum value is the maximum of the current maximum value and the current value (e.g., i_k). This may seem like an arduous set of activities, but with sufficient practice students will learn to be more precise when solving algorithmic problems. Notice that we have not precisely specified pre and post conditions for this problem. Nor have we identified candidate iteration invariants, or written down any part of the algorithm yet. But with the understanding gained, these should be relatively easy. Also, observe that this equality represents the foundations for both recursive and iterative solutions. The alternative view below leads directly to a recursive solution

$$\text{maxOfSeq}(i_1) = i_1, \quad k = 1,$$
$$\text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_k) = \max\bigl(i_1, \text{maxOfSeq}(i_2, i_3, \ldots, i_k)\bigr), \quad k > 1.$$

Below a simple recursive solution is presented using the pattern matching features of the functional language Standard ML. Here [ ] is the empty list, $[i]$ a list of one integer value i, and for the nonempty list $[i_1, i_2, i_3, \ldots, i_n]$ and the pattern *first*::*rest*, *first* $= i_1$ and *rest* $= [i_2, i_3, \ldots, i_n]$. That is, identifier *first* is bound to integer value i_1 and identifier *rest* is bound to the integer list $[i_2, i_3, \ldots, i_n]$. This function definition is declarative and can be viewed as an executable model for this problem.

(* precondition: $[i_1, i_2, i_3, \ldots, i_n]$, $n > 0$ is a list of integers *)

```
fun maxOfSeq([ ]) = raise maxException  (* empty sequence has no max value *)
| maxOfSeq([i]) = i                     (* sequence with one int value i *)
| maxOfSeq(first::rest) = max(first, maxOfSeq(rest));
```

(* postcondition: Let max = maxOfSeq($[i_1, i_2, i_3, \ldots, i_n]$), then
for all $k = 1, 2, \ldots, n$, max $\geqslant i_k$ and there exists $k = 1, 2, \ldots, n$, max $= i_k$ *)

Please note that no variables are used, and that this function definition is in one-to-one correspondence with the solution developed above. The latter point is extremely important, since we should strive to minimize the "semantic gap" between an abstract solution and its implementation. Unfortunately, modern introductory programming languages do just the opposite, forcing students to understand, create and use unnecessary artifacts to fill this semantic gap. Simple conceptual ideas get lost a mid all the additional baggage/notation/jargon/constraints—learning these seem to be part of the attraction of programming. Unfortunately this also distracts students from understanding the important fundamental concepts.

A final very important point is that the correctness of this function definition can be argued directly using the principle of mathematical induction. That is, the function works correctly for a list with one element since the value i is returned for the "base" case maxOfSeq($[i]$). Also, if the function works correctly for the list $[i_2, i_3, \ldots, i_n]$, then it is easy to see that it works correctly for the list $[i_1, i_2, i_3, \ldots, i_n]$ since for this "inductive" case the value returned is $\max(i_1, \text{maxOfSeq}([i_2, i_3, \ldots, i_n]))$. A symmetric inductive argument works to demonstrate the validity of the original derived view

$$\text{maxOfSeq}(i_1) = i_1, \quad k = 1,$$
$$\text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_k)$$
$$= \max\bigl(\text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_{k-1}), i_k\bigr), \quad k > 1,$$

which is the basis for the traditional iterative algorithm. More about mathematical induction later.

9. Declarative Versus Imperative Reasoning

CS/SE graduates are trained and learn the tools necessary to be good software developers. This is primarily imperative or operational reasoning. However, graduates generally don't learn, or appreciate, the tools necessary to reason about software systems. This requires declarative or assertive reasoning. I hope to convince you that one can't be a good software developer without good declarative reasoning skills, and that most programmers use such skills extensively, but, not usually in a concise manner. In other words, it is impossible to specify, design, compose, test, and debug a software system without thinking declaratively.

To clarify this point, let me continue the example from the previous section. We will develop an imperative solution to the problem of computing the maximum value of a sequence $i_1, i_2, i_3, \ldots, i_n, n > 0$. "Abstracting" towards an algorithmic solution let's introduce the variable ***maxValue***[10]

$$\text{maxValue} = \text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_k).$$

The meaning of the variable ***maxValue*** is precisely defined *before* composing an algorithm. Contrast this with the "shotgun" approach to "creating" variables which students habitually learn to use. Now, clearly,

$$\begin{aligned}
&\text{maxValue} = \max\big(\text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_{k-1}), i_k\big) \quad \text{or}\\
&\text{maxValue} = \max(\text{maxValue}', i_k) \quad \text{where}\\
&\text{maxValue}' = \text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_{k-1}).
\end{aligned}$$

Here, the meta variable ***maxValue***$'$ is introduced for explanatory purposes only and will not be used in the algorithm. Recall that the key insight is "the next maximum value is the maximum of the current maximum value and the current value." At first glance this statement is imperative—"how" to compute the next value. However, in its original forms

$$\begin{aligned}
&\text{maxOfSeq}(i_1) = i_1, \quad k = 1,\\
&\text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_k)\\
&\quad = \max\big(\text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_{k-1}), i_k\big), \quad k > 1,\\
&\text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_k) = \max\big(i_1, \text{maxOfSeq}(i_2, i_3, \ldots, i_k)\big), \quad k > 1,
\end{aligned}$$

[10] Variables were not required in the recursive solution thus reducing its cognitive complexity.

it can be viewed declaratively as an equality relationship between functions of two sequences and the value i_k or i_1. This declarative view was implemented by the function maxOfSeq in the previous section.

The preconditions, post conditions and iteration invariant are illustrated in $\{\ldots\}$ in the algorithm below. To specify the predicate corresponding to the iteration invariant $I(k)$ the function maxOfSeq : integer sequence $\rightarrow$ int is used

$$I(k) \equiv \text{maxValue} = \text{maxOfSeq}(i_1, i_2, i_3, \ldots, i_k).$$

Note that this predicate is declarative, and corresponds with the well defined meaning of the variable ***maxValue*** which was derived directly from the preceding inductive analysis.

```
{pre-condition: integer sequence $i_1, i_2, i_3, \ldots, i_n, n > 0$}
    maxValue $\leftarrow i_1$
    $k \leftarrow 1$
    {$I(k)$ is true}
    while $k < n$ do
        {$I(k)$ and $k < n$ are true}
        $k \leftarrow k + 1$
        maxValue $\leftarrow$ max(maxValue, $i_k$)
        {$I(k)$ is true}
{post-condition: $I(k)$ and $k = n \rightarrow$ maxValue $=$ maxOfSeq$(i_1, i_2, i_3, \ldots, i_n)$}
```

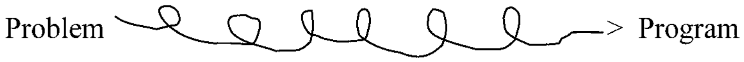
Note that the iteration invariant is true before and after the iteration, and that the body of the iteration maintains the validity of the invariant. Hopefully you see that the assertions—the stuff within the braces in this algorithm—are required to derive this algorithm, and that a purely operational view is insufficient to capture the pattern of information valid for all possible sequences.

Much of mathematics is declarative in nature. For example, basic algebra is an equality declaration between two expressions. However, most of the mathematics pre-college students are expected to know is operational in nature (e.g., how to manipulate algebraic expressions). In addition, it focuses on preparing students for continuous rather than discrete mathematics. The latter fails to prepare students properly for undergraduate CS and SE studies, and the former provides the wrong mindset. I also believe that the overemphasis on imperative thinking of current curricula actually narrows the reasoning skills of graduates. This is directly and indirectly supported by a number of studies cited and discussed in the subsequent sections.

Until pre-college mathematics preparation, and its focus on operational techniques, changes CS/SE programs must adapt to establish a more declaratively focused curriculum, starting with the first course. Ways for achieving this will also be presented in subsequent sections.

10. Algorithmic Problem Solving

Most introductory CS courses expose students to algorithmic problem solving via programming. The approaches used generally reinforce operational/imperative thinking and deemphasize declarative thinking and mathematics as problem solving tools. Although the objectives of the course are usually achieved, the residual impact on student perceptions of the discipline are further solidified—mathematics and assertive reasoning are not relevant, and computer programming is basically translating word problems to programs.



In the paper "Embedding Instructive Assertions in Program Design" [20] David Ginat addresses this issue by proposing an approach using informal assertions. The paper starts by observing that most programmers employ an operational perspective for algorithmic problem solving and that this can lead to incorrect or inefficient solutions. Numerous examples are presented using assertive reasoning to derive correct, efficient solutions. His conclusion provides the following insight:

> "... while the role of the operational way is fundamental, it is insufficient by itself and should be combined with an assertional perspective."

When solving problems,[11] there are very important synergistic relationships between the operational and declarative perspectives. Both are usually required. Two characteristics of an expert problem solver are the ability to know when to use the proper perspective and how to effectively switch between perspectives. David's paper illustrates the seeds for conveying this in an introductory CS course.

Most introductory computer science text books and courses convey an informal, superficial view of algorithmic problem solving. This is appealing to students' intuition and educators who don't fully appreciate mathematically based approaches. This usually sets the tone for the entire curriculum making it difficult to motivate more formal mathematics based methodologies and problem solving views.

A deeper view of algorithmic problem solving requires mathematical thinking. That is, the ability to first understand a problem, to precisely identify the key concepts, to specify these concepts unambiguously and to use them to develop an algorithmic solution. We have already seen how careful inductive reasoning leads to

[11] Not necessarily algorithmic ones.

identifying patterns. Two other central aspects are abstraction, hiding unimportant details, and case analysis, identifying precisely all relevant cases. Case analysis includes situations leading to invalid results (i.e., exceptions) and breaking the problem into basic computationally identifiable cases (e.g., special, base, inductive, etc.). For example, when searching a list the current item is either the desired one or it is not. Students usually learn to implement this using sequential control features such as "if–then," "if–then–else," "case," "switch," etc. In fact, implementing an algorithm in an imperative programming language requires additional programming language artifacts, further widening the semantic gap for students and thus reducing the possibility of getting the implementation correct initially.

Motivated by the maximum of a sequence example and using the terminology already introduced, the algorithmic problem solving process using these two views might be further decomposed:

- Pattern & Case discovery: is primarily operational;
- Pattern & Case specification: is primarily declarative;
- Functional/recursive model: is primarily declarative;
- Identify variables: both operational & declarative;
- Specify invariants: is primarily declarative;
- Identify constructs: is primarily operational;
- Complete constructs: both operational & declarative;
- Compose constructs: both operational & declarative;
- Testing: both operational & declarative.

Indeed, when developing imperative algorithms/computer programs for problems with which I am unfamiliar, I first develop understanding and declarative relationships using inductive reasoning and case analysis, and then implement/verify/test these by building a model—often an executable one using a recursive functional language which supports pattern matching. Creating a model first leads to a more precise understanding of the problem (i.e., exceptions, pre and post conditions, etc.) and potential correct algorithmic solutions (i.e., cases, invariants, etc.). Contrast this process with the way most students learn to program—imprecise problem understanding, a vague mental model, artifact heavy implementation language, extensive program/debug/test cycles. Structured programming, patterns, object oriented methods, etc. help somewhat to ease the chaos for students, but the underlying mathematical reasoning is lost as depicted by the picture below:

Problem
Program

Operational reasoning alone will not suffice.

After reading this section, I believe some readers will feel I am missing the point because it narrowly focuses on basic algorithms and doesn't address data structures, objects, UML, events, parallelism or other models of computation. One of the primary concerns I often hear is that such mathematically based techniques are simply toys and don't scale up to real systems. In response, I have several comments. First, and most important, if graduates are not able to think this way using fundamental principles, then I don't see how they will mature to become effective, general, creative problem solvers. Second, these techniques do work well at the component, or small scale, level. Third, the field is maturing rapidly with the development of new and more powerful formal tools which graduates will have to use at some time in the future. These include evolving modeling and model checking tools such as Alloy, formal specification languages—Z, etc., and compilers for languages like Eiffel and SPARK Ada [7] supporting formal assertions which can be analyzed and checked. Indeed, I predict that future software development environments will provide a sophisticated artificial intelligence-theorem prover based assistant which will look for logic errors based upon formal assertions and provide suggestions to the developer. Simplified versions of such a tool could be developed for teaching programming.

11. Recursive Thinking

One of my recent Butler graduates was discussing a data organization problem with his co-workers, who could not find a good solution. He went to his office and after 15 minutes returned with a simple, clean, recursive programmed solution. They were astounded. He had been exposed to recursive programming his first two years at the University of Arizona and extensively to recursive thinking/problem solving in our Foundations of Computing I and II courses upon transferring to Butler for his final two years.

Many graduates don't feel comfortable with this level of recursive thinking/problem solving. A study by David Ginat "Do Senior Students Capitalize on Recursion?" [21] confirms this. He states,

> "The student solutions and explanations demonstrate very limited capitalization on recursion as a problem solving means."

I would be so bold as to claim that this evidence generalizes to other important mathematical concepts. Below are a few potential titles for future papers:

- "Do Senior Students Capitalize on Logic?"
- "Do CS Graduates use Mathematical Induction?"

- “Do Senior Students Understand Functions?”
- “Sets as Foreign Objects”

Of course I am being partly facetious here to make my point. Recursive thinking, like other mathematically based concepts, must be introduced early and conceptually, and be continually reinforced, both conceptually and in practice, throughout the entire curriculum. Immersion is important! With respect to mathematics and its role, CS and SE educators need to further appreciate the importance of mathematics and mathematical reasoning, better understand the mathematical foundations of CS and SE, and learn how to integrate mathematically based concepts into courses. The journey is just beginning.

Now, back to recursion. Composing several recursive programs in languages which “indirectly” support recursion[12] does not suffice for students to learn recursive problem solving. This “operational” approach to teaching recursion fails to ensure that students understand the concept of recursion and its important relationships with inductive thinking and mathematical induction. In addition, some CS and SE educators believe that recursion, as an implementation tool, is inefficient and hence should not be emphasized. I maintain that students should learn, understand and appreciate the concept of recursion, and be able to use it as a problem solving and reasoning tool. Fortunately, there has been significant interest and progress in this endeavor to which I hope this chapter contributes.

Eric Roberts book “Thinking Recursively [45],” early functional languages such as LISP and Scheme, and the seminal text “The Structure and Interpretation of Computer Programs [2],” along with many other texts, materials, languages, innovative courses, etc. have helped pave the way for more mathematical approaches to teaching recursive reasoning. But there is still a long way to go as many graduates view key concepts such as recursion and mathematical induction as “magical incantations.” With respect to recursive problem solving, David Ginat’s article [21] provides reasons for this and some specific guidelines for improvements—many of which can be adapted for other mathematically based concepts. In this chapter, I will provide general guidelines, curriculum structures and introductory level courses for significantly enhancing the mathematical reasoning skills of CS and SE graduates.

To extend these ideas further it is important to note the connections between research in mathematics education and computer science education [3]. Here is a relevant quote from the abstract of the paper “Mathematical reasoning in task solving” [36]:

[12] C, C++, Ada, Pascal, Java, C#, ... Functional languages such as Standard ML and Haskel more directly support recursive reasoning IMHO.

The results indicate that focusing on what is familiar and remembered at a superficial level is dominant over reasoning based on mathematical properties of the components involved, even when the latter could lead to considerable progress.

This is one reason David Ginat points out why senior students have difficulty with recursive problem solving. It also helps to clarify why my graduate could use recursion effectively. But more importantly, it is a statement about the deficiency of the education of CS and SE students as supported by several studies regarding students' mathematical abilities and perceptions [48,34].

Continuing on this theme, the "languages" one thinks and solves problems in are the ones someone is most familiar with. If a CS/SE graduate only learns C++ and Java, these are the "languages" they will think in. Learning a new language or paradigm is primarily based upon conceptual knowledge rather than specific knowledge of other languages. Indeed, previous knowledge often constrains learning as evidenced by a software professional's difficulty to learn a new paradigm—for example, procedural to object oriented. I would argue that much of this conceptual knowledge is based on mathematics. A review of topics in text books for programming language courses helps to support this, and at least one embraces a more mathematically rigorous perspective [43]. These are all steps in the right direction, but there is still a long way to go, especially in the area of teaching mathematical foundations early and integrating mathematics throughout the curriculum.

12. Mathematical Induction

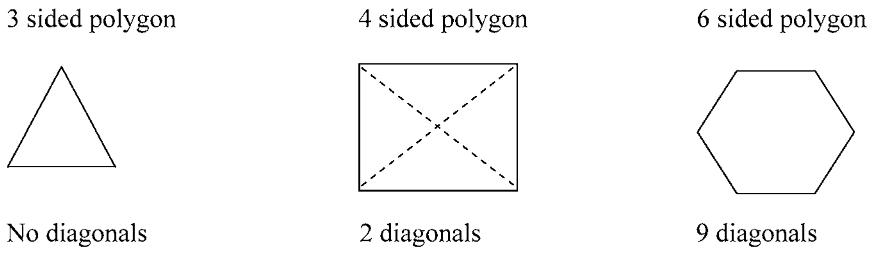
Students are usually introduced to the principle of mathematical induction in their first discrete mathematics course. Recursion and iteration are fundamental conceptual and implementation ideas in computer science. Yet most graduates don't have a good grasp of the relationships between mathematical induction, recursion and iteration. A majority of the introductory CS textbooks and data structures/algorithms texts provide minimal coverage of mathematical induction [48] primarily because discrete mathematics is not typically a prerequisite for these courses. To help make these essential connections, some curricula require discrete mathematics as a corequisite with the data structures course. To be successful, this requires careful, deliberate coordination between the instructors of these two courses.

Discrete mathematics courses for CS/SE students are often taught by mathematics faculty, who may not have a good understanding of the connections, and/or may believe that discrete mathematics is an advanced topic requiring several prerequisite courses. For CS/SE students already a bit math phobic this creates an even deeper chasm. These issues have been recognized by the Mathematical Association of America (MAA) and addressed in their 2004 curriculum guidelines [38].

Mathematical induction has been mentioned in previous sections of this paper in the context of arguing the validity of a recursive or iterative pattern discovered. I believe that such links to fundamental computer science principles are crucial for motivating students and that they can and must be made in the first year.

It is generally agreed that mathematical induction is a difficult concept for students. This has been studied by math educational researchers [16]. The traditional approach to teaching mathematical induction is to introduce it as a modular unit in the first discrete mathematics course using numerous numerical examples and exercises, and then reinforce these ideas in the context of other concepts in the course, such as binary trees, etc. For CS/SE majors mathematical induction is usually touched on in other courses (e.g., algorithm analysis, theory of computation, etc.). I believe this is insufficient for students to understand and be able to use induction, and to make the necessary links with CS.

Besides the links, motivation and continued reinforcement, “setting the students’ mental stage” before formally introducing mathematical induction is very important. In many text books and courses the topic “pops up” with little preparation or forethought. To “get the inductive mental juices flowing” I have students solve numerous simple inductive reasoning and generalization problems, such as presented in Section 8, starting at the beginning of the course. Many of these require pattern identification and specification, and are of the general form: “If you are given that predicate $P(n-1)$ is true for $n-1$ items, find $P(n)$ for n items.” For example, consider a polygon with n sides (3, 4 and 6 sided polygons are shown below. Diagonals are illustrated as dotted lines).



(a) How many diagonals are there in a 7 sided polygon?
(b) How many diagonals are there in an 11 sided polygon?
(c) How many diagonals are there in an 44 sided polygon?
(d) If there are k diagonals in an $(n-1)$-sided polygon, then how many diagonals are there in an n-sided polygon? Give your answer in terms of k and n.
(e) Give an expression, in terms of n, for the number of diagonals in an n-sided polygon.

This way, the ideas underlying mathematical induction don't suddenly "pop up." Indeed, I have found that after the course, most students have a good grasp of the concept of induction. This is verified by the collective responses over the past 15 years to the following question on my post course survey "This course has increased my understanding of induction [1–5]" (where 1 = Strongly Agree to 5 = Strongly Disagree). The average response is about 2.1, slightly below "Agree."

Unfortunately understanding induction and being able to use it are different. Many students understand the concept but have difficulty applying it to solve problems and doing inductive arguments. This is often due to a lack of understanding of the application domain and/or weak mathematical skills. For example, students struggle with the following numeric inductive argument primarily due to faltering algebraic skills.

$$1 + 4 + 9 + \cdots + n^2 = n(n+1)(2n+1)/6 \quad \text{for all } n \geqslant 0.$$

Continual reinforcement, motivation and connections are all extremely important. In CS/SE courses, recursion and iteration are rich pedagogical sources for reinforcing arguments based upon mathematical induction. Again, this mathematical based way of thinking must be established as early as possible, preferably upon entry. The seeds can be planted in introductory computer science courses through inductive reasoning discussion and exercises. The second best option would be a first course emphasizing recursive thinking, perhaps using a functional based language such as Scheme, Standard ML, Miranda or Haskel. The best choice is requiring students to take the introductory discrete mathematics or a foundations of computing course upon entry. A prototype entry level foundations course is presented in Section 15. This course includes hands on computer based problem solving exercises which strengthen key mathematical concepts such as mathematical induction. One way the latter can be achieved is with structural induction—using induction to argue the correctness of recursively function definitions on recursive structures such as lists and binary trees. Function ***maxOfSeq*** presented at the end of the "Inductive Thinking and Generalization" section is a representative example. The objective is to ensure graduates can effectively use induction as a reasoning and checking tool.

13. Why Mathematics?

Until now, I have presented general beliefs, topic heading and examples attempting to build motivation for mathematics in the education of CS and SE students. Now it is time to make these points more concrete. The section is the title of a paper from the Communications of the ACM dedicated to the reasons why CS and SE students need math [14]. Many of the items below appeared in my paper "Mathematical

Reasoning in Software Engineering Education" in this issue [23]. They are included here for completeness with expanded justifications. I would like to think they are comprehensive, but realize there are numerous views regarding these issues.

(1) Software is abstract.
(2) Notations, symbols, abstractions, and precision are features common to both mathematics and software engineering.
(3) Mathematics is important for modeling and specifying software system behavior.
(4) Many application domains (engineering, science, economics, etc.) are mathematically based.
(5) Mathematical reasoning is important for most problem solving, especially software systems development.

(1) *Software is abstract*: One of the first abstractions our ancestors developed was counting, a basic form of mathematics. From this start, mathematics has become the primary tool mankind uses to deal with abstractions. Constructing non-physical artifacts, like software, requires abstract reasoning for which there is no better tool than mathematics. Accordingly, a software system can be viewed as a mathematically precise model of some desired process or computation. Software practitioners generally agree that software is abstract, but seem to prefer using alternative, non-mathematical tools for specifying, designing, implementing, testing, debugging and maintaining software systems. I believe this will slowly change as systems become larger and more complex, more mathematically based tools become available (e.g., modeling and model checking), and graduates are more mathematically savvy and have a better appreciation for the power of mathematics as a reasoning tool.
(2) *Notations, symbols, abstractions, precision*: Software relies heavily upon all four of these. Notations and symbols are abstractions for common objects and concepts. This is why $y = ax + b$ is familiar from algebra, and **count == 0** from programming. Both are well understood and precise in their context of use. Students are motivated to learn the notations, symbols, and precise syntax and semantics of a programming language. In reality, this is no different than learning these for mathematics, which is often easier. However, students perceive mathematics as static and rote. Programming is perceived as dynamic and exhilarating, appealing to our operational oriented minds. There are more mathematically based computer languages and tools which provide similar levels of exhilaration if educators will discover and embrace them. These include functional languages such as Standard ML, Miranda and Haskel, model building and checking languages, and mathematics programming languages such as Maple, Mathematica and Axiom. All make extensive use of notations,

symbols, abstractions and precision and are more suited to students learning a declarative style of thinking.

(3) *Modeling software systems*: As discussed in Section 5, a model, even a mental one, must be created before beginning construction of any artifact. Today, software development is more of an art where an initial vision slowly takes form—like molding a chunk of clay, and this is part of its attraction. However, such ad hoc methods are not acceptable for projects where a more precise understanding of the desired artifact is required before construction. Building, analyzing, and checking a "mathematical" model first is one solution. New software system modeling tools, languages, and techniques are evolving whose use will one day become the norm [9,29,31,33,24]. Currently, system specification languages such as Z, Larch, etc. are available to more precisely specify the desired behavior [27].

(4) *Application domains*: Mathematics is a rich, inclusive, universal language for communication between diverse groups. Accordingly it provides software practitioners with a tool for communicating effectively with clients and colleagues from any discipline (engineers, scientists, mathematicians, statisticians, actuaries, and economists) with mathematical foundations.

(5) *Mathematical reasoning*: This topic has been addressed in other sections of this chapter.

14. Curricula Issues

Computer science and software engineering are young, maturing disciplines. Guidance for education in these disciplines comes from a variety of sources, primarily the professional organizations, in particular the ACM and IEEE. Computer science has had a long history of curricula recommendations [46] and the first "official" recommendation for software engineering was in 2004 [47]. Both current recommendations [46,47] include requirements for computer based mathematics in the form of discrete mathematics/structures topics and suggest early introduction of these topics. This is a good start to ensuring mathematics proficiency of our graduates. But there are many holes yet to be filled, specifically making mathematics more pervasive by integrating and using fundamental concepts throughout the entire curriculum. This will help to ensure mathematical maturity, proficiency and use of mathematics as a problem solving tool.

A mathematically based curriculum must start with a mathematically based course. In Section 15 "Foundations of Computing—a First Course," details of such a course for CS/SE majors which I have developed over 20+ years, are presented. One graduate commented that: "{this course} is a lot closer to the ideal {first CS course} than any introductory programming course could be." However, this is only one potential model for a first course. In due time, I believe that Discrete Mathematics I and II, offered by the mathematics department, will become the standard freshman sequence, similar to the way Calculus I and II are for most science and engineering majors. In other words, discrete mathematics will acquire equal stature with calculus for entering students. Progress toward this is offered by: (1) discrete mathematics being recommended earlier with almost 40% of the CS/SE programs requiring it in the freshman year in 2001 [28], and (2) the recent curriculum recommendations of the Mathematics Association of America [38].

Instead of independent discrete mathematics courses, some educators suggest integrating these topics and concepts into existing introductory computer courses, perhaps using a just-in-time pedagogy to motivate pertinent connections. In theory, this approach makes sense and has merits, but I don't think it is viable as an expedient way to promote change. Instructors would be reluctant to make room for additional material and would have little motivation to change their way of covering the material. Students would direct the course toward HOW and discourage the WHY. Indeed, at several institutions I have visited over the past several years, students have been successful in deferring planned changes toward a more mathematically (WHY) oriented curriculum. In my opinion, this is very short sighted on the part of the students, their parents and the administration. Interestingly, the current downturn in computer science studies provides a good climate for change since students might conceivably be more interested in learning the WHY than the HOW, to improve their employment marketability.

An entry level discrete mathematics course follows the successful engineering model, makes effective use of faculty resources, is therefore attractive to administrators, and addresses the emerging need for discrete concepts in other disciplines such as engineering, biology, economics, media arts, etc.

The ultimate goal is a curriculum in which foundational mathematical concepts become pervasive throughout all courses. Here is one potential model for the first three semesters:

$$\begin{tikzcd}
\text{CS 0} \arrow[dr] & & \\
& \text{CS I} \arrow[r] & \text{CS II} \\
\text{DM I} \arrow[r] & \text{DM II} \arrow[ur] &
\end{tikzcd}$$

CS 0 is an overview of the discipline with an introduction to programming, and would be taken by students with no programming experience or who wish to refresh their knowledge. Such an Introduction to Computer Science course is often required by other majors, so it can serve multiple purposes.

Another recent trend is the introduction of CS III to ensure programming competency and breadth of knowledge before advanced courses. One concern here, as in the above model, is the length of the pre-requisite chain. On the surface this seems to be an issue, but in practice more material can be covered with a deeper level of understanding using the model above rather than the traditional CS I $\rightarrow$ CS II followed by CS III sequence, with DM I and II intermixed somewhere.

First, CS I is supercharged by CS 0 and DM I. This means the instructor does not have to start from scratch. Second, CS 0 and DM I provide a "reality check" for students so the overall quality/maturity of students in CS I is significantly higher. Third, the overall logic, mathematical foundations, problem solving, conceptual (e.g., recursion, induction) and abstract reasoning skills of the CS I students are significantly enhanced. For example, students don't struggle with negating a Boolean conditional since they know and can apply DeMorgan's Laws.

What are these enhanced CS I and CS II courses? I don't have a definitive answer. However, from 1996 to 1998 I developed and taught at SUNY Stony Brook a mathematically oriented CS I course following the foundations course. It made extensive use of the mathematical foundations, especially predicate logic and mathematical induction. These included formal pre and post conditions, assertions (especially iteration and class invariants), and formal arguments of the correctness of imperative and recursive algorithms. Object oriented software development using Bertrand Meyers' "successive opening of black boxes" pedagogical approach [40] and "programming by contract" [41] in Modula 3 was used, and students were required to provide extensive semi-formal heading comments for object classes and methods. A significant amount of exemplarity Modula-3 software was written and provided for learning and completing projects. Unfortunately, the course was not a resounding success—the "WHY" students really liked it and the "HOW" students didn't. I felt it was headed in the right direction but would have taken several more years to develop.[13]

With respect to curriculum reform, in my opinion, the two most important aspects are introducing mathematical foundations early and integrating/reinforcing/enhancing these foundations in all courses. Progress has been made on the former and in identifying the foundational mathematics. This will eventually precipitate the latter as educators learn to effectively use the mathematical preparation of students and new text books evolve. Doug Baldwin and Greg Scragg's new book "*Algorithms and Data Structures: The Science of Computing*" is a representative CS II text [6].

One impediment to integration is the prerequisite mathematics instructors presume. Currently for many CS/SE courses, it is very little, so if an instructor wishes to use mathematical concepts they reserve course time to teach these. For example, when teaching a course on formal specifications or modeling, they would cover the basic principles of propositional and predicate logic first, or logic and relations are often taught before introducing relational algebra in a database course. With a consistent math early prerequisite structure, review, rather than (re)teaching would become

[13] A one year leave of absence and my move to Butler impeded further development.

the norm and more relevant course material and reinforcement of key mathematics and computer science concepts could be achieved.

15. Foundations of Computing—A First Course

One of the fundamental themes echoed through this chapter is the need for an early introduction to the mathematics of computing and its continual reinforcement throughout the curriculum. I believe that if students are properly prepared with strong foundations, the rest of their academic career will be significantly easier. The entry level course for CS/SE students should be more mathematically based than the traditional CS-I course. This might be achieved by requiring more rigorous and precise reasoning, or using more mathematically oriented declarative logic or functional languages. As mentioned in the preceding section, I think that eventually entering CS/SE students with proper preparation will take discrete mathematics. This math early model has withstood the test of time in science and engineering disciplines.

To support my position, I offer as an example the foundations of computing course that I developed. This course has evolved over the past 20+ years at SUNY Stony Brook and Butler University as the first course for majors. My objectives for this course include

- Logic and discrete mathematics;
- Math maturity and math thinking;
- General Problem solving skills;
- Appreciation for mathematics and math based reasoning;
- Declarative reasoning;
- Inductive reasoning, recursion, mathematical induction;
- Connections to fundamental computer science concepts;
- Hands on activities to reinforce concepts;
- Cooperative and communication skills.

The results of a 1999 alumni satisfaction survey for this course, CSE-113 at SUNY Stony Brook, are presented in Appendix A. More details can be found at [26]. Below is one representative alumni quote:

> "Not only did the course {CSE-113} help me to refine my problem solving skills, but it also provided a fantastic preview for all the material I studied in subsequent courses. In fact, during every other computer science class I took, there were always references back to material that was introduced in CSE-113. More importantly, however, the skills which are developed in CSE-113 help students tackle real world programming problems."

A course description and syllabus for this course in its current form at Butler University are presented in Appendix B. This is a two dimensional view of a multidimensional course. The two text books currently used are “Discrete Mathematics with Applications, third ed.” by Susanna Epp [17] and “Effective Problem Solving, second ed.” by Marvin Levine [35]. Some of the main themes are:

- Logic is the foundational glue for all topics;[14]
- General problem solving skills are built;
- Power of arguments;
- Deemphasize terminology, definitions and rote learning;
- Emphasize conceptual understanding;
- Difficult concepts and topics are introduced informally or through examples and are presented from numerous perspectives throughout the course (spiral approach);
- Zingers and chaos;
- Hands on, laboratory component;
- A relatively small number of topics are covered;
- Challenge.

The need for a foundational theme for introductory level discrete mathematics courses has lead to logic. Text books are introducing logic first and using it as the foundation for subsequent topics [17,19,18]. In addition, logic is the basis for formal techniques—specifications, modeling, model checking, program verification, etc.

“Effective Problem Solving” helps students gain the skills and confidence to tackle problems. One of the most important problem solving strategies students learn is engagement, do something, anything to get started. For example, a student who had just completed this course told me he gained the confidence to try to fix a malfunctioning television set at his girlfriend’s house.

Most CS/SE graduates are good algorithmic problem solvers, but extensive focus on the imperative approach cognitively narrows a students’ general problem solving skills [39]. Students can only learn problem solving by solving a variety of “challenging” problems which often require extensive trial and error. It is important to include problems with no solution and multiple solutions; ones requiring different abstract representations and solution strategies. Given only one semester, the level and focus of the problem solving aspects of the course are unfortunately restricted with more attention to case analysis and inductive/recursive pattern discovery.

For traditional engineering, the focus is primarily on computational mathematics—using mathematics to compute the voltage at a point in a circuit or the stress on a

[14] Recall many discrete mathematics courses are perceived as a collection of disjoint topics.

bridge component. In contrast, CS/SE principally centers on logical reasoning based mathematics—e.g., precisely specifying the behavior of a system, arguing the correctness of an algorithm, or reversing engineering logic when maintaining or debugging a software system. This is the motivation CS/SE educators give as to why students need to learn logic and understand basic proof techniques. Discrete mathematics is ideally suited to this, and freshman, or even pre-college students, can learn basic proof techniques. However, there are two points I would like to make in this regard.

(1) I find that using the term “argument” rather than “proof” is conceptually less intimidating for students. For example, when demonstrating the correctness of a recursively defined list processing function, I ask students to do a mathematical induction argument.
(2) Becoming proficient with logic and using argument/proof techniques takes a long time. It requires continual use and reinforcement throughout the curriculum. One or two courses alone will not suffice.

Bloom’s cognitive learning taxonomy [8] specifies five levels: knowledge/recall, comprehension, application, analysis, synthesis and evaluation. Ensuring that CS/SE graduates become critical thinkers starts with conceptual understanding, applying concepts to solve problems, and basic analysis and synthesis skills. However, the focal point of courses and text books is too often knowledge (definitions, terminology, etc.) rather than comprehension/application. Creative assignments and open book/notes exams are ways to help correct this. A representative first exam is presented in Appendix C.

There are numerous topics and concepts with which students struggle. These include, among others, basic propositional logic (e.g., implication and arguments), predicate logic (e.g., quantifiers, negation, vacuous truth, arguments), recursion, inductive reasoning, and mathematical induction. Some topics such as sets and functions are familiar, but from a limited context. To ensure understanding of conceptually difficult topics they must be introduced early, intuitively, slowly and must be continually reinforced. As mentioned earlier, mathematical induction is introduced after students have devoted 6–8 weeks solving numerous inductive reasoning problems. Students are then required to use it to argue the correctness of various recursive function definitions developed in the laboratory component of the course.

Students like predictable rote learning courses (i.e., comfort zones), but these are not always conducive to learning. I use zingers, problems with a twist, frequently in class, and on assignments, quizzes and exams to make or reinforce important points and to help break the student’s reliance on intuition. My favorite zingers deal with difficult concepts such as vacuous truth. Some chaos, or doing the unexpected, helps to keep students alert and prepare them for the enviable confusion of life.

Of course there are complaints from students who feel they are not learning "programming." A very general definition of programming is "creating a sequence of instructions to enable the computer to do something." "Instructions" can be *imperative*, a students' typical perception,[15] or *declarative*. Asking a student to create a sequence(set) of logically precise statements and a conclusion from a logical word problem, and presenting these to a theorem prover to determine if the conclusion is true, is "programming." So is having students define relationships such as ancestor, sibling, uncle, etc. in ProLog or defining recursive functions such as maxOfSeq in Standard ML. This helps to correct student misconceptions about "programming" and at the same time provides them with a level intellectual playing field.[16]

Some examples laboratory exercises in support of these objectives are presented in Appendix D. These include:

- Propositional logic resolution theorem prover for logic word problems.
- Predicate logic resolution theorem prover for logic word problems.
- Prolog for solving problems.
- Introduction to basic functional problem solving using Standard ML.
- Recursive list processing using Standard ML.

Such hands-on computer experiences are extremely important for reinforcing key concepts and making connections between mathematics and computer science. Here minimizing the "semantic gap" between the concept and the implementation tool/language is extremely important for maximizing student learning. Straightforward theorem provers with a natural interface and logical expression language are a nice match for reinforcing propositional and predicate logic. Prolog expands these ideas further supporting a more powerful query language and recursion, and contrasting a pure logical solution strategy with a goal directed problem-solving strategy, both employing backtracking. Functional languages such as Standard ML are ideal for reinforcing the mathematical concept of a function, developing problem analysis, decomposition and synthesis skills, and promoting recursive problem solving. These laboratories are completed by the students in groups of 2 or 3. Each group submits an extensive laboratory report following a prescribed format with individual discussion and conclusion sections worth about 1/3 of the report grade. One key to the success of these hands-on laboratories is a well orchestrated sequence of activities building from basic tutorials and simple canned exercises through to more creative, challenging ones.

Reviewing the syllabus of a typical introductory discrete mathematics course often reveals a long list of topics such as propositional and predicate logic, sets, sequences,

[15] The word "sequence" seems to imply the imperative interpretation.

[16] Students with programming background versus those with little or no background.

proof techniques, functions, number theory, mathematical induction, counting, combinatorics, relations, graphs and trees, algorithms, etc. Many include explicit application areas such as digital logic circuits, algorithm correctness and analysis, etc. This is frequently necessary since the CS curriculum, especially at many liberal arts colleges, only has room for one discrete mathematics course. Unfortunately this tight packing of concepts is not a productive learning environment for many students. My preference and experience is to reduce the number of topics in favor of understanding. Also, I strongly believe that either two discrete or foundation of computing courses are required or lacking that, that one course followed by a rigorous introduction of additional topics in computer science courses is needed.

Students need to be challenged to achieve their full potential. I find this course is a good reality check for students planning on studying CS or SE. In particular, students who enjoy intellectual challenges are encouraged to major in CS/SE. Perhaps, this is one reason women are attracted to the course and another might be the emphasis on concepts rather than programming applications and hacking [11]. The results of my unpublished studies over the five year period 1990–1995 shows that women worked longer (43 more average minutes per week), smarter (77% versus 64% participated in study groups), and performed better (cumulative grade point average[17] of 2.67 versus 2.59) than men in this course. At Butler, the course has also become popular as preparation for the logical/analytical components of the Law School Admissions Test (LSAT).

Using the model described in Section 14, Foundations of Computing I is followed by Foundations of Computing II. This course covers relations, trees, graphs, counting, finite state machines, algorithm correctness, and further reinforces inductive reasoning, recursive problem solving and mathematical induction with a sequence of laboratory assignments which includes developing a simple propositional logic theorem prover, arithmetic expression evaluator, and a finite state machine guessing game.

16. Conclusions

Mathematics is a natural complementary discipline for learning, understanding and appreciating many fundamental computer science concepts, and mathematical reasoning is intrinsic to computer science and software engineering. Accordingly, it should become integral to both curricula. Progress in this direction has been made in recent curricula recommendations and in understanding/appreciating the role of mathematics. However, there is still a long way to go to before all CS/SE graduates can reason clearly and precisely about algorithms and software systems.

[17] $A = 4$, $B = 3$, $C = 2$, $D = 1$, $F = 0$.

Computer based disciplines use discrete mathematics and logic mainly in a declarative mode, whereas traditional science and engineering disciplines primarily employ continuous mathematics in a calculational mode. Pre-college mathematics primarily focuses on the preparation for the latter, often setting the wrong mindset for entering CS/SE students. This mindset is further reinforced by the operational emphasis of current CS/SE curricula. I believe in time the continuous and discrete mathematical emphasis of pre-college will reach a natural balance as discrete and logical thinking become more pervasive in other disciplines such as the science, economics and engineering.

There may be some concern that this chapter is too superficial, or that it didn't go deep enough or lay out an aggressive, specific strategy for advancing the role of mathematics in CS/SE undergraduate education. There are several reasons for this. First, I have presented, for the most part, what I have had experience and feel comfortable with. Second, curricula transformations are often slow in educational disciplines, so presenting the seeds for change (with which everyone may not agree) will hopefully have more immediate impact. Grand visions rarely do. Third, I strongly believe that mathematically oriented SE/CS curriculum can and will be achieved. This evolution will take time as mathematics and CS/SE educators gradually gain an understanding, and an appreciation for, the role of mathematics. Also, I firmly believe that graduates of mathematically oriented curricula will be significantly better computer scientists and practicing software engineers.

ACKNOWLEDGEMENTS

I would like to thank the reviewers and in particular Judy Gersting for comments and suggestions which greatly improved the readability and quality of this chapter.

Appendix A: CSE-113 Foundations of Computer Science I

Stony Brook Alumni Survey Summary (Spring 1999)

(121 respondents - 86 male, 35 female)

CSE-113 was good preparation for computer science courses.

	All	Male	Female
Strongly Agree	94%	91%	100%
Agree	6%	9%	-
Neutral	-	-	-
Disagree	-	-	-
Strongly Disagree	-	-	-

CSE-113 was good preparation for math courses.

	All	Male	Female
Strongly Agree	74%	72%	77%
Agree	23%	23%	23%
Neutral	3%	5%	-
Disagree	-	-	-
Strongly Disagree	-	-	-

CSE-113 was good preparation for my other courses.

	All	Male	Female
Strongly Agree	62%	57%	74%
Agree	34%	37%	26%
Neutral	3%	3%	-
Disagree	1%	1%	-
Strongly Disagree	-	-	-

CSE-113 was one of the most important courses I took at Stony Brook.

	All	Male	Female
Strongly Agree	77%	74%	83%
Agree	19%	21%	14%
Neutral	3%	4%	3%
Disagree	1%	1%	-
Strongly Disagree	-	-	-

CSE-113 should be kept as the first course for CS majors at Stony Brook.

	All	Male	Female
Strongly Agree	96%	94%	100%
Agree	4%	6%	-
Neutral	-	-	-
Disagree	-	-	-
Strongly Disagree	-	-	-

CSE-113 should be replaced with a more programming oriented course.

	All	Male	Female
Strongly Agree	-	-	-
Agree	-	-	-
Neutral	7%	9%	3%
Disagree	24%	19%	34%
Strongly Disagree	69%	71%	63%

CSE-113 helped me improve my general problem solving skills.

	All	Male	Female
Strongly Agree	92%	89%	100%
Agree	8%	10%	-
Neutral	1%	1%	-
Disagree	-	-	-
Strongly Disagree	-	-	-

CSE-113 helped me improve my logical thinking skills.

	All	Male	Female
Strongly Agree	88%	84%	97%

Agree	12%	16%	3%
Neutral	-	-	-
Disagree	-	-	-
Strongly Disagree	-	-	-

CSE-113 helped me improve my ability to communicate with others.

	All	Male	Female
Strongly Agree	22%	19%	32%
Agree	44%	39%	57%
Neutral	29%	38%	11%
Disagree	5%	4%	-
Strongly Disagree	-	-	-

CSE-113 helped me improve my confidence to overcome a challenge.

	All	Male	Female
Strongly Agree	69%	66%	80%
Agree	27%	30%	17%
Neutral	3%	3%	3%
Disagree	1%	1%	-
Strongly Disagree	-	-	-

Appendix B: Butler University, Foundations of Computing I

CS151. Foundations of Computing 1 (3 credit hours): Introduction to mathematical problem-solving, with emphasis on techniques for designing computer-based solutions. Concepts include problem solving principles, logic, proof techniques, sets, sequences, functions, and inductive and recursive thinking.

Syllabus:

Prepositional Logic

1. Propositions and Statements
2. Truth Tables
3. Logical Operators
4. Logical Implication and Equivalence
5. Logical Proof Techniques
6. Computer Modeling of Logical Reasoning
   a. Propositional Logic Theorem Prover

Predicate Logic

1. Predicate Variables, Quantifiers
2. Inference and Methods of Proof
3. Computer Modeling of Logical Reasoning
4. Logic Programming Paradigms
   a. Predicate Logic Theorem Prover
   b. Prolog

Sets

1. Representation
2. Subsets
3. Set Operations
4. Power Sets and Counting

Functions, Recursion, and Mathematical Induction

1. Function Mappings
2. Function Composition
3. Recursively Defined Functions
4. Mathematical Induction
5. Functional Programming Paradigms
   a. Standard ML
   b. Recursive Problem Solving—Lists

General Problem Solving

1. Goals, Givens, and Rules
2. Inference and States
3. Action Sequences
4. Subgoals, Divide and Conquer
5. Contradiction and Working Backward
6. Backtracking

Computer Based Problem Solving

1. Problem Decomposition
2. Effective use of Abstraction
3. Information Structures
4. Programming Paradigms (Logic and Functional Programming)
5. Expressive Power of Languages

Appendix C: Sample First Exam for Foundations of Computing I (100 minutes)

1. (5 pts) What is the next number in the following sequence?
   4, 5, 7, 11, ?? Answer:
2. (5 pts) Construct the truth table for the following logical expression $(p \vee q) \rightarrow {\sim}p)$.

3. (9 pts) In English, give the converse, inverse, and contra positive of the following statement.

   "A positive integer x is divisible by 10 implies it is divisible by 2."
4. (10 pts) Is the following argument valid? If so, give the steps of the argument. If not, explain why.

   $$P \to Q, \quad \sim P \to \sim R, \quad R \vee \sim R.$$

   Therefore we can conclude Q is true.
5. (5 pts) Is the following argument valid? Explain your answer. "If John gets a great job he can buy a new car. John gets a job. Therefore, John can buy a new car."
6. (8 pts) One end of an elastic string that is four inches long is connected to a wall, and a ladybug is at the middle of this string. It begins crawling towards the wall, and after the ladybug travels an inch, the string is pulled to stretch to double its length, to 8 inches. After the ladybug travels another inch toward the wall, the string again stretches to double its length. Each time the ladybug travels one inch, the string doubles in length. After the string stretches for the 37th time, where is the ladybug? Answer:
7. (10 pts) If Lisa plays golf and Lynn plays tennis, then the kids will not go swimming.

   Either Lynn plays tennis or she makes dinner for the kids.

   If the temperature is over 90 degrees, then the kids go swimming.

   Lisa plays golf.

   The temperature is over 90 degrees.

   Conclusion: Lynn makes dinner for the kids.

   Let G represent: "Lisa plays golf." Let T represent: "Lynn plays tennis." Let S represent: "The kids will go swimming." Let D represent: "Lynn makes dinner for the kids." Let N represent: "The temperature is over 90 degrees."
   (a) Express these statements in prepositional logic.
   (b) Give a proof by contradiction to show the conclusion is true.
8. (12 pts) Let male(p) be a predicate true if person p is male, and let eats(p, f) be a binary predicate over people and food (true if person p eats food f). For each of the following sentences, give a predicate logic expression that represents its meaning:
   (a) Fred eats apples.
   (b) All people eat fruit.
   (c) Someone doesn't eat liver.
   (d) Some men eat quiche.
   (e) Not every man eats quiche.
   (f) Some man who eats pizza also eats apples.

9. (6 pts) “There are two bugs in a gallon jar. Every minute, the number of bugs doubles. If the jar is filled in 30 minutes, how long will does it take for the jar to become half filled with bugs?” Answer:

Consider this same problem, but starting with four bugs instead of two. How long will does it take for the jar to become half filled with bugs? Answer:

10. (15 pts) Consider the following clauses and conclusion.

$$Q \vee P, \quad Q \vee R \vee {\sim}W, \quad P \vee R, \quad {\sim}R, \quad W.$$

Conclusion: Q and P.
(a) Draw a resolution tree to demonstrate the conclusion is true.
(b) How many total resolutions are required for this proof? Answer:
(c) Is any backtracking required in this proof? Explain your answer.

11. (15 pts) An N-bit “Gray Code” is a sequence of all the N-bit binary numbers, ordered in such a way that each binary number differs from its predecessor and its successor by exactly 1 bit (and the first and last differ by 1 bit also). For example:
 1-bit Gray Code: 0, 1,
 3-bit Gray Code: 000, 001, 011, 010, 110, 111, 101, 100 (differs from 000 by 1 bit).
 However, the sequence 00, 11, 01, 10 is not a 2-bit Gray code since the first and second elements differ by two bits.
(a) Give a 2 bit Grey Code sequence.
(b) In terms of N, give an expression for the number of binary numbers in an N-bit Gray Code sequence.
(c) If you were given a Gray Code sequence for $N - 1$ bits, explain how you could construct a Gray Code sequence for N bits from it.

Appendix D: Representative List Processing Lab Exercises Using Standard ML

1. Complete the definition of the function **sum: real list → real** that returns the sum of all the elements in the argument list. It should return zero if the list is empty. Rewrite this definition using **if–then–else** rather than pattern matching.

```
fun sum([ ]) = 0.0
|sum(fst::rst) = fst + ______ :
```

Create a function **reverse: ′a list → ′a list** that reverses the elements of its argument list. Use wishful thinking by assuming you are given the reverse of the rest of the list.

    ****** Use induction to prove that your definition of this function works as specified.

2. Compose a function that counts the number of occurrences of a specified item in the list. For example, 2 appears twice in the list [2, 5, 6, 8, 2, 8, 9] and **true** appears 0 times in the list **[false, false]**. Make up a meaningful name for this function.

    ****** Use induction to prove that your definition of this function works as specified.
3. Define a function **find: ′a list * ′a** 391 **→ int** that takes arguments **list** and an element **elt**, and returns the position of the first (reading from left to right) occurrence of **elt** in **list**. For example, if **list** = [5, 3, 7, 3, 4] and **elt** = 3, then **find(list, elt)** should return 2. If the element does not occur in the list, then function **find** should return zero.
4. Develop a function **select: int * ′a list → ′a** that returns the element in specified integer position of the list. For example, select(2, [3.4, 6.2, 1.7]) = 6.2. Frequently a function cannot be defined for all possible argument values (i.e., it is a partial function). For function **select**, if the integer argument does not correspond to the position of any element in the list you should indicate an exceptional condition which forces the function to fail. We have predefined such an exceptional condition and named it SelectError. The statement raise SelectError will force a failure with the SelectError message.
5. Develop a function **firstN: int * ′a list → ′a list** which returns the first n elements of the argument list. If $n \leqslant 0$, then the empty list is returned, and if n is greater than or equal to the length of the argument list, then the entire argument list is returned. For instance,

    firstN(4, [2, 4, 13, 5, 8, 10, 13]) returns [2, 4, 13, 5]

    ****** Use induction to prove that your definition of this function works as specified.
6. Use functions **reverse** and **firstN** to define the function **lastN**, which returns the last n elements of the argument list.
7. Define a function **ordered: int list → boolean** which returns true if the integer list is in ascending order and false otherwise (e.g., **ordered([2,4,6])** returns true, but **ordered([2,6,4])** returns false).

    Vacuously, the empty list is ordered because there is nothing to keep it from being ordered.

****** Use induction to prove that your definition of this function works as specified.

8. Develop a function **merge: int list * int list → int list** which merges two ordered (sorted) integer lists into a single ordered integer list. For instance,

   merge([2, 4, 5, 8, 10, 13], [1, 2, 3, 6, 9, 11, 13, 15, 16, 20])

   returns the list[1, 2, 2, 3, 4, 5, 6, 8, 9, 10, 11, 13, 13, 15, 16, 20]

9. Compose a function that determines the count of all clusters in an integer list. A cluster is a sequence of one or more non-zero values. In the list [0, 0, 3, 4, 12, 0, 0, 0, 0] the subsequence 3, 4, 12 is a cluster of three non-zero elements. If your function is applied to this list it would return the list [3], indicating a single cluster with three non-zero elements. For the list [0, 0, 4, 0, 3, 4, 0, 0, 0, ~7, 8, ~10, 12, 0, 0, 0, 4, 0, 0] your function should return the list [1, 2, 4, 1] specifying the size of each of the four clusters when scanning the list from left to right. (Note: this is a challenging problem so think carefully. ~ is unary minus in Standard ML.)
10. Create a function **Exists: ('a→ boolean) * 'a list → boolean** which takes a boolean valued function (a predicate) and a list, and returns true if the predicate argument is true for at least one element in the list, and false otherwise. For example, let **even: int → bool** be a function that returns true if the argument is an even integer. Now, Exists(even, [3, 7, 13, 12, 15, 5, 7, 4]) returns **true** since at least one integer in the list is even.

Appendix E: Solutions for Problems Cited

1. "A social psychologist was interested in the custom of handshaking, . . ." A solution is presented in the Ask Marilyn column in Parade Magazine, August 28, 2004 (http://archive.parade.com/askmarilyn_archive/2004/0829.html) for the following equivalent problem:

   A woman and her husband hosted a party for four other couples. The hostess asked everyone to shake hands and introduce themselves to each other. Of course, no one shook hands with his or her spouse. At some point, the hostess stopped them and asked each person how many hands he or she had shaken. Each person gave a different response. What was the response of her husband?
2. Match Stick Problem
   (b) Number of Match Sticks(N squares) $= N + 3$;
   (d) Number of Joints(N squares) $= 3N - 1$;
   (e) Constructing a table helps students identify the pattern

$$N = M - J + 1.$$

REFERENCES

[1] Accreditation Board for Engineering and Technology, Inc., http://www.abet.org/.

[2] Ableson H., Sussman G.J., Sussman J., *The Structure and Interpretation of Computer Programs*, second ed., MIT Press, Cambridge, MA, 1996.

[3] Almstrum V., “Import and export to/from computing science education: The case of mathematics education research”, Panel presentation, ITiCSE 2002, Aarhus, Denmark.

[4] Averbach B., Chein O., *Problem Solving Through Recreational Mathematics*, Freeman, New York, 1980.

[5] Baldwin D., Henderson P.B., “The importance of mathematics to the software practitioner”, *IEEE Software* (March–April 2002) 112, 110, 111.

[6] Baldwin D., Scragg G.W., *Algorithms and Data Structures: The Science of Computing*, Charles River Media, 2004.

[7] Barnes J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison–Wesley, Reading, MA, 2004.

[8] Bloom B., et al., *Bloom’s Taxonomy of the Cognitive Domain*, 1956.

[9] Clarke E.M., Grumberg O., Peled D., *Model Checking*, MIT Press, Cambridge, MA, 1999.

[10] “MAA CUPM—CRAFTY Curriculum Foundations Project”, http://www.mathsci.appstate.edu/~wmcb/CFF/.

[11] De Palma P., “Why women avoid computer science”, *Communications of the ACM* **44** (6) (June 2001) 27–29.

[12] Devlin K., “Do software engineers need mathematics?”, MAA OnLine, http://www.maa.org/devlin/devlin_10_00.html, October 2000.

[13] Devlin K., “The real reason why software engineers need math”, *Communications of the ACM* **44** (10) (October 2001) 21–22.

[14] Devlin K. (Ed.), Special issue on “Why universities require computer science students to take math”, *Communications of the ACM* **46** (9) (September 2003).

[15] Devlin K., *Mathematics: The Science of Patterns*, Freeman, New York, 1994.

[16] Dubinsky E., “Teaching mathematical induction I”, *Journal of Mathematical Behavior* **6** (1) (1987) 305–317.

[17] Epp S., *Discrete Mathematics with Applications*, third ed., Brooks/Cole Publishing, 2004.

[18] Epp S., “The role of logic in teaching proof”, *American Mathematical Monthly* **110** (10) (December 2003) 886–899.

[19] Gersting J.L., *Mathematical Structures for Computer Science*, fifth ed., Freeman, New York, 2003.

[20] Ginat D., “Embedding instructive assertions in program design”, ITiCSE 2004, Leeds, UK.

[21] Ginat D., “Do senior students capitalize on recursion?”, ITiCSE 2004, Leeds, UK.

[22] Glass R.L., “A new answer to ‘How important is mathematics to the software practitioner’?”, *IEEE Software* (November/December 2000) 135–136.

[23] Henderson P.B., “Mathematical reasoning in software engineering education”, *Communications of the ACM* **46** (9) (September 2003) 45–50.

[24] Henderson P.B., "The role of modeling in software engineering education", in: *Proceeding of the Frontiers in Education Conference, Boulder, CO*, 2003.

[25] Henderson P.B., et al., "Striving for mathematical thinking", *SIGCSE Bulletin—Inroads* **33** (4) (December 2001) 114–124.

[26] Henderson P.B., "Foundations of CS I Stony Brook alumni survey", http://www.ic.sunysb.edu/cse113/survey/.

[27] Hinchey M.G., Bowen J.P. (Eds.), *Applications of Formal Methods*, Prentice Hall, New York, 1995.

[28] Hitchner L., "Survey of discrete structures/discrete math courses", http://blue.butler.edu/~phenders/WP2002/HitchnerStudy.txt, 2001.

[29] Huth M., Ryan M., *Logic in Computer Science: Modeling and Reasoning about Systems*, Cambridge University Press, Cambridge, UK, 2001.

[30] Huth M., "Mathematics for the exploration of requirements", *ACM SIGCSE Bulletin Inroads* **36** (2) (June 2004).

[31] Jackson D., "The alloy analyzer", http://alloy.mit.edu/.

[32] Ashbacher C. (Ed.), *Journal of Recreational Mathematics*, Baywood Publishing.

[33] Lamport L., *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison–Wesley, Reading, MA, 2002.

[34] Lethbridge T., "What knowledge is important to a software professional?", *IEEE Computer* **33** (5) (May 2000) 44–50.

[35] Levine M., *Effective Problem Solving*, second ed., Prentice Hall, New York, 1994.

[36] Lithner J., "Mathematical reasoning in task solving", *Educational Studies in Mathematics* **41** (2000) 165–190.

[37] Liu H., Gluch D.P., "A proposal for introducing model checking into an undergraduate software engineering curriculum", in: *16th Southeastern Small College Computing Conference*, November 2002.

[38] *Undergraduate Programs and Courses in the Mathematical Sciences: CUPM Curriculum Guide 2004*, The Mathematics Association of America, 2004.

[39] Mayer R.E., Dyck J.L., Vilbert W., "Learning to program and learning to think: What's the connection?", *Communications of the ACM* **29** (7) (July 1986) 605–610.

[40] Meyer B., "Toward an object oriented curriculum", *The Journal of Object Oriented Programming* **6** (2) (May 1993) 76–81.

[41] Meyer B., *Object Oriented Software Construction*, Prentice Hall, New York, 1997.

[42] Morissette A. (Canadian singer) Quote.

[43] Noonan R., Tucker A., *Programming Languages: Principles and Paradigms*, McGraw–Hill, New York, 2002.

[44] Parnas D.L., "Software engineering programmes are not computer science programmes", *Annals of Software Engineering* **6** (1–4) (1998) 19–37.

[45] Roberts E., *Thinking Recursively*, Wiley, New York, 1986.

[46] Roberts E., Shackelford R., et al., "Computing curricula 2001: Computer science volume", http://www.acm.org/sigcse/cc2001/, December 15, 2001.

[47] Sobel A. (Ed.), "Computing Curricula Software Engineering Volume", http://sites.computer.org/ccse/, May 24, 2004.

[48] Tucker A.B., Kelemen C.F., Bruce K.B., "Our curriculum has become math-phobic!", in: *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, February 2001, pp. 243–247.
[49] Wooden J., Jamison S., *Wooden*, Contemporary Books, 1997.
[50] The Working Group on Integrating Mathematical Reasoning into Computer Science Curricula, http://www.math-in-cs.org.

Author Index

Numbers in *italics* indicate the pages on which complete references are given.

Subject Index

Contents of Volumes in This Series