Academic Press is an imprint of Elsevier

32 Jamestown Road, London, NW1 7BY, UK
Radarweg 29, PO Box 211, 1000 AE Amsterdam, The Netherlands
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
525 B Street, Suite 1900, San Diego, CA 92101-4495, USA

First edition 2009

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-374812-6

ISSN: 0065-2458

Printed and bound in USA

09 10 11 12 10 9 8 7 6 5 4 3 2 1

Contributors

Prof. Robert Aalberts is the Lied Professor of Legal Studies at the University of Nevada, Las Vegas. He received his Juris Doctor from Loyola University and an M.A. from the University of Missouri-Columbia. Prior to his academic career, he was an attorney for the Gulf Oil Company. His primary research interests include real estate law, cyber law, and employment law. He has also published over 105 articles in legal and business periodicals. He is currently the Editor-in-Chief of the *Real Estate Law Journal* where he has served for the past 16 years. He is also coauthor of the textbook, *Law and Business: The Regulatory Environment*, 1994, published by the McGraw-Hill Book Company and *Real Estate Law*, 7th edition, 2009 published by Southwestern/Cengage Learning.

Christopher Ackermann is a Scientist at the Fraunhofer Center for Experimental Software Engineering, Maryland and is pursuing a Ph.D. at the University of Maryland, College Park. He received his Bachelor's Degree from the University for Applied Sciences, Mannheim, Germany in 2006 and earned his Master's Degree from the University of Maryland in 2008. He has been active in the fields of software architectures, software testing and verification, model-based development, empirical software engineering, software visualization, and change impact analysis. His current research interests include software architecture analysis, testing and verification, and model-based development.

Prof. Eric Allender received a B.A. from the University of Iowa in 1979, majoring in Computer Science and Theatre, and a Ph.D. from Georgia Tech in 1985. He has been at Rutgers University since then, serving as department chair from 2006 to 2009. He is a Fellow of the ACM and serves on the editorial boards of the *ACM Transactions on Computation Theory*, *Computational Complexity*, and *The Chicago Journal of Theoretical Computer Science*. He has chaired the Conference Committee for the annual IEEE Conference on Computational Complexity, and he serves on the Scientific Board for the Electronic Colloquium on Computational Complexity (ECCC).

Prof. Hany Farid received his undergraduate degree in Computer Science and Applied Mathematics from the University of Rochester in 1989. He received his Ph.D. in Computer Science from the University of Pennsylvania in 1997. Following a 2-year postdoctoral position in Brain and Cognitive Sciencesat MIT, he joined the Dartmouth faculty in 1999. He is the David T. McLaughlin Distinguished Professor of Computer Science and Associate Chair of Computer Science. He is also affiliated with the Institute for Security Technology Studies at Dartmouth. He is the recipient of an NSF CAREER award, a Sloan Fellowship, and a Guggenheim Fellowship. He can be reached at farid@cs.dartmouth.edu, and more information about his work can be found at www.cs.dartmouth.edu/farid.

Prof. David Hames is an Associate Professor of Management at the University of Nevada, Las Vegas. He earned his B.A. from Albion College, his M.L.I.R. from Michigan State University, and his Ph.D. from the University of North Caroling at Chapel Hill. His research, which focuses on employment law, human resource management, and labor–management relations, has been published in journals such as *Group and Organization Management*, *Human Resource Management Review*, *Leadership and Organization Development Journal*, *Employee Responsibilities & Rights Journal*, *Labor Law Journal*, and *Risk Management & Insurance Review*.

Prof. Andrew Johnson is an Associate Professor in the Department of Computer Science and member of the Electronic Visualization Laboratory at the University of Illinois at Chicago. His research focuses on the development and effective use of advanced visualization displays, including virtual reality displays, autostereo displays, high-resolution walls and tables, for scientific discovery and in formal and informal education.

Prof. Jason Leigh is an Associate Professor of Computer Science and Director of the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago. He is a cofounder of VRCO, the GeoWall Consortium, and the Global Lambda Visualization Facility. He currently leads the visualization and collaboration research on the National Science Foundation's OptIPuter project, and has led EVL's Tele-Immersion research since 1995. His main area of interest is in developing ultra-high-resolution display and collaboration technologies for supporting a wide range of applications ranging from the remote exploration of large-scale data, education, and interactive entertainment.

Dr. Mikael Lindvall is a Senior Scientist and the Director of the Software Architecture and Embedded Systems division at Fraunhofer Center for Experimental Software Engineering, Maryland. He is interested in best practices and

methodologies for software engineering, in general, and specializes on software architecture evaluation and software evolution. He received his Ph.D. in Computer Science from Linköpings University, Sweden, in 1997. His Ph.D. work focused on evolution of object-oriented systems and was based on a commercial development project at Ericsson Radio in Sweden.

Prof. Jörn Loviscach is a Professor for Computer Graphics, Animation, and Simulation at Hochschule Bremen (University of Applied Sciences) in Bremen, Germany. He is interested in 2D and 3D graphics algorithms and systems, human–computer interaction, audio and music computing, in particular concerning applications that require signal processing and/or the development of specialized electronics. A regular contributor to conferences such as SIGGRAPH, Eurographics, and the AES Convention, he has published numerous chapters in book series such as *Game Programming Gems* and *ShaderX Programming*. Before becoming a professor in 2000, he was Deputy Editor-in-Chief of the popular German computer magazine "c't," the editorial staff of which he joined soon after earning his doctorate degree in physics.

Dr. Jürgen Münch is Division Manager for Software and Systems Quality Management at the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern, Germany. Before that, he was Department Head for Processes and Measurement at Fraunhofer IESE and an executive board member of the temporary research institute SFB 501, which focused on software product lines. He received his Ph.D. degree (Dr. rer. nat.) in Computer Science from the University of Kaiserslautern, Germany, at the chair of Prof. Dr. Dieter Rombach. His research interests in software engineering include (1) modeling and measurement of software processes and resulting products, (2) software quality assurance and control, (3) technology evaluation through experimental means and simulation, (4) software product lines, and (5) technology transfer methods. He has significant project management experience and has headed various large research and industrial software engineering projects, including the definition of international quality and process standards. His main industrial consulting activities are in the areas of process management, goal-oriented measurement, quality management, and quantitative modeling. He has been teaching and training in both university and industry environments. He has coauthored more than 80 international publications, and has been co-organizer, program cochair, or member of the program committee of numerous high-standard software engineering conferences and workshops. He is a member of ACM, IEEE, the IEEE Computer Society, and the German Computer Society (GI).

Prof. Percy Poon is an Associate Professor of Finance at the University of Nevada, Las Vegas. He received his Ph.D. in Finance from Louisiana State University.

His primary research interests are in the investment area. He has published numerous articles in both finance and other business periodicals, including the *American Business Law Journal*, *Journal of Finance*, *Journal of Banking and Finance*, *CACM*, *Financial Review*, and *Financial Practice and Education*. His research on portfolio diversification has been cited by the *Wall Street Journal* and the *Investor's Business Daily*. He has served as an *ad hoc* reviewer for various academic financial periodicals, including *Financial Management*, *Financial Review*, and *Financial Practice and Education*. He also offered his expertise to the business community, including seminars to a utilities company on the uses of options and futures to hedge energy costs.

Prof. Luc Renambot received a Ph.D. at the University of Rennes-1 (France) in 2000, conducting research on parallel rendering algorithms for illumination simulation. Then holding a postdoctoral position at the Free University of Amsterdam, until 2002, he worked on bringing education and scientific visualization to virtual reality environments. In 2003, he joined EVL/UIC first as a PostDoc and now as Research Assistant Professor, where his research topics include high-resolution displays, computer graphics, parallel computing, and high-speed networking.

Prof. Günther Ruhe holds an Industrial Research Chair in Software Engineering at University of Calgary. His main results and publications are in software engineering decision support, software release planning, software project management, measurement, simulation, and empirical research. From 1996 until 2001, he was Deputy Director of the Fraunhofer Institute for Experimental Software Engineering. He is the author of two books, several book chapters, and more than 120 publications. Dr. Ruhe is a member of the ACM, the IEEE Computer Society, and the German Computer Society (GI).

Dr. M. Omolade Saliu is a Decision Support Architect at Online Business Systems in Calgary, Canada. He is currently involved in developing decision support solutions for performance management. He received his Ph.D. in Computer Science from the University of Calgary, Canada in 2007. His Ph.D. research was sponsored by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Alberta Informatics Circle of Research Excellence (iCORE). Saliu's Ph.D. research focussed on decision support for planning the releases of evolving software systems. His research interests include software architecture evaluation, software release planning, and decision support.

Prof. Paul D. Thistle is Professor of Finance at the University of Nevada, Las Vegas. He earned his B.B.A. from the University of Portland and his M.S. and Ph.D.

in Economics from Texas A&M University. He has taught at the University of Arizona, University of Alabama, and Western Michigan University, and was a Heubner Postdoctoral Fellow at the Wharton School. While his primary research interest is in insurance and risk management, he has published extensively in economics, finance, insurance, real estate, and management information systems. His research was supported by the Nevada Insurance Education Foundation.

Dr. Laurence Tratt is a Senior Lecturer at Bournemouth University and a Software Consultant. He is an Associated Editor-in-Chief of IEEE Software. He received the Ph.D. degree from King's College of London. He is the chief designer of the Converge programming language. His research interests include programming languages, domain-specific languages, and software modeling.

Dr. Adam Trendowicz received a degree in Computer Science (B.Sc.) and in Software Engineering (M.Sc.) from the Poznan University of Technology, Poland, in 2000. He is currently a researcher at the Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, Germany in the Processes and Measurement department. Before that, he worked as a software engineering consultant at Q-Labs GmbH, Germany. His research and industrial activities include software cost modeling, measurement, data analysis, and process improvement.

Preface

This is volume 77, the last volume in the 50th year of publication of the *Advances in Computers*. Since 1960, annual volumes have been produced containing chapters authored by some of the leading experts in the field of computers. For 50 years, these volumes offer ideas and developments that are changing our society. This volume presents eight different topics covering many different aspects of computer science. I hope you find them of interest. The first three chapters provide insights into the different ways individuals can interact with electronic devices. First we look at digital photography. Then we look at other display devices and in Chapter 3 at other game interfaces that have been developed.

Today, the ubiquitous film camera has all but disappeared from view to be replaced by ever cheaper and larger digital chips of memory. While allowing a huge number of pictures to be taken essentially for free, there is a cost in security of the pictures. Digital images and associated software allow the photographer (or almost anyone else for that matter) to manipulate the bits of the image and hence change the picture. How do we discover such tampering and how do digital forensics work to uncover fakery? Hany Farid in "Photo Fakery and Forensics" in Chapter 1 of this volume discusses methods for detecting inconsistencies in lighting and pixel correlations to detect forgeries.

Not only have cameras changed, but so too have all other visual devices connected to the computer. Jason Leigh, Andrew Johnson, and Luc Renambot in Chapter 2's "Advances in Computer Displays" discuss a wide variety of visual display technology—from the old-fashioned cathode ray tube (CRT) to more modern plasma displays, steroscopic displays, and wall displays. They discuss what the environment of the future—whether at work or at home—is likely to contain.

In Chapter 3, Jörn Loviscach in "Playing with All Senses: Human–Computer Interface Devices for Games" discusses mechanisms for interacting with games on a computer. After quickly passing through the usual mouse, keyboard, and joystick, he discusses pen and touch input devices, sensors, and cameras. Inertial sensors allow the user to move the device and the computer to interpret that motion, such as in Nintendo's successful Wii machine. Incorporating all of these into the next

generation of game allows the user to experience a multimedia approach toward a game—a far cry from the early Pong, which was a simple paddle for hitting an image of a ball back across the screen.

One of the long-standing unsolved questions in computer science theory is the resolution of the NP-completeness problem—simply stated as "Does P equal NP?" I have always been interested in this question since I was a graduate student in the late 1960s when the problem was first posed. To date it has not yet been solved, but I was very interested in seeing what has been learned over the past 40 years. In Chapter 4, Eric Allender in "A Status Report on the P Versus NP Question" discusses what the question is and what has happened in this 40-year period.

Chapter 5 by Laurence Tratt is entitled "Dynamically Typed Languages." Historically, most programming languages were compiled into executable machine code by a compiler. For efficiency these languages such as FORTRAN, Algol, Pascal, C were statically typed. That is, the data type (e.g., *integer*) was specified so the compiler could generate efficient code for it. Today, there is more interest in dynamically typed languages where data are processed while the program executes and the distinction between compilation and execution is getting blurred. Languages like Perl and Python, as well as the older LISP, are examples of this. In this chapter, Dr. Tratt discusses the advantages of using such dynamically typed languages.

In Chapter 6, Adam Trendowicz and Jürgen Münch's "Factors Influencing Software Development Productivity—State-of-the-Art and Industrial Experiences" look at the continuing evolution of software engineering development practices. Of particular interest is how one measures the productivity since scheduling of appropriate resources is critical for completing a project with least organizational impact? Over estimate productivity and you need to overspend to put more people on project and under estimate productivity and you have employees with little to do. In this chapter, the authors look at various factors that have been reported in the literature as most important aspects affecting productivity.

In Chapter 7 "Evaluating the Modifiability of Software Architectural Designs" by M. Omolade Saliu, Günther Ruhe, Mikael Lindvall, and Christopher Ackermann, the authors present an architectural design evaluation technique called EBEAM. Since software undergoes change as it evolves and since this becomes a dominant cost factor over time, it is important to understand how modifiable the software is as it develops. EBEAM is described and its use on an experimental system shows its value.

As the Internet becomes more invasive in our lives, its impact, in terms of costs and money transferred, is well into the multibillions of dollars (or euros or pounds) per year. Although our legal system has been developing for hundreds of years, this new technology is radically different from older systems. When you send a document by email (e.g., the contents of a CD containing music), no object actually is

sent— only the electronic bits that describe the document are sent. So who owns this document? This is only one simple example. Can our 1000-year-old system based upon English common law adapt to this new technology? In the final chapter, "The Common Law and Its Impact on the Internet," Robert Aalberts, David Hames, Percy Poon, and Paul D. Thistle discuss how the legal system is adapting to this new cyberworld.

I hope that you find these chapters of use to you in your work. If you have any topics you would like to see in these volumes, let me know. If you would like to write a chapter for a forthcoming volume, also let me know. I can be reached at mvz@cs.umd.edu.

Marvin Zelkowitz
College Park, Maryland

Photo Fakery and Forensics

HANY FARID

Department of Computer Science, Dartmouth College, Hanover, New Hampshire 03755, USA

Abstract

Photographs can no longer be trusted. From the tabloid magazines to the fashion industry, mainstream media outlets, political campaigns, and the photo hoaxes that land in our email inboxes, doctored photographs are appearing with a growing frequency and sophistication. I will briefly describe the impact of all of this photographic tampering and recent technological advances that have the potential to return some trust to photographs. Specifically, I will describe a representative sample of image forensics techniques for detecting inconsistencies in lighting, pixel correlations, and compression artifacts.

ADVANCES IN COMPUTERS, VOL. 77
ISSN: 0065-2458/DOI: 10.1016/S0065-2458(09)01201-7

1. Photo Fakery

History is riddled with the remnants of photographic fakery. Stalin, Mao, Hitler, Mussolni, Castro, and Brezhnev each had photographs manipulated in an attempt to alter history. Cumbersome and time-consuming darkroom techniques were required to alter history on behalf of Stalin and others. Today, powerful and low-cost digital technology has made it far easier for nearly anyone to alter digital images. And the resulting fakes are often very difficult to detect. This photographic fakery is having a significant impact in many different areas.

1.1 Media

For the past decade, Adnan Hajj has produced striking war photographs from the ongoing struggle in the Middle East. On 7 August 2006, the Reuters news agency published one of Hajj's photographs showing the remnants of an Israeli bombing of a Lebanese town. In the week that followed, hundreds of bloggers and nearly every major news organization reported that the photograph had been doctored with the addition of more smoke. The general consensus was one of outrage and anger—Hajj was accused of doctoring the image to exaggerate the impact of the Israeli shelling. An embarrassed Reuters quickly retracted the photograph and removed from its archives nearly 1000 photographs contributed by Hajj. The case of Hajj is, of course, by no means unique. In 2003, Brian Walski, a veteran photographer of numerous wars, doctored a photograph that appeared on the cover of the Los Angeles Times. After discovering the fake, the outraged editors of the LA Times fired Walski. The news magazines Time and Newsweek have each been rocked by scandal after it was revealed that photographs appearing on their covers had been doctored. And, in the past few years, countless news organizations around the world have been shaken by similar experiences.

1.2 Science

Those in the media are not alone in succumbing to the temptation to manipulate photographs. In 2004, Professor Hwang Woo-Suk and colleagues published what appeared to be groundbreaking advances in stem cell research. This paper appeared in one of the most prestigious scientific journals, *Science*. Evidence slowly emerged that these results were manipulated and/or fabricated. After months of controversy, Hwang retracted the *Science* paper and resigned his position at the University. An independent panel investigating the accusations of fraud found, in part, that at least nine of the 11 customized stem cell colonies that Hwang had claimed to have made were fakes. Much of the evidence for those nine colonies, the panel said, involved

doctored photographs of two other, authentic, colonies. While this case garnered international coverage and outrage, it is by no means unique. In an increasingly competitive field, scientists are succumbing to the temptation to exaggerate or fabricate their results. Mike Rossner, the managing editor of the *Journal of Cell Biology* estimates that as many as 20% of accepted manuscripts to his journal contain at least one figure that has to be remade because of inappropriate image manipulation [1].

1.3 Law

The child pornography charges against its Police Chief shocked the small town of Wapakoneta, OH. At his trial, the defendant's lawyer argued that if the State could not prove that the seized images were real, then the defendant was within his rights in possessing the images. In 1996, the Child Pornography Prevention Act (CPPA) extended the existing federal criminal laws against child pornography to include certain types of ''virtual porn.'' In 2002, the United States Supreme Court found that portions of the CPPA, being overly broad and restrictive, violated First Amendment rights. The Court ruled that ''virtual'' or ''computer-generated'' images depicting a fictitious minor are constitutionally protected. The burden of proof in this case, and countless others, shifted to the State who had to prove that the images were real and not computer generated. Given the sophistication of computer-generated images, several state and federal rulings have further found that juries should not be asked to make the determination between real and virtual. And at least one federal judge questioned the ability of even expert witnesses to make this determination. This example highlights the general complexities that exist at the intersection of digital technology and the law.

1.4 Politics

''Fonda Speaks to Vietnam Veterans at Anti-War Rally'' read the headline with an accompanying photograph purportedly showing Senator John Kerry sharing a stage with the then controversial Jane Fonda. The faux article was also a fake—a composite of two separate and unrelated photographs. And just days after being selected as a running mate to U.S. presidential hopeful John McCain, doctored images of a bikini clad and gun-toting Sarah Palin were widely distributed on the Internet. The pairing of one's political enemies with controversial figures is certainly not new. It is believed that a doctored photograph contributed to Senator Millard Tydings' electoral defeat in 1950. The photo of Tydings conversing with Earl Browder, a leader of the American Communist party, was meant to suggest that Tydings had communist sympathies. Recent political ads have seen a startling number of doctored photographs pitting candidates in a flattering or damaging light.

1.5 National Security

With tensions mounting between the United States and Iran, the Iranian Government announced the successful testing of ballistic missiles. As evidence, the government released a photograph showing the simultaneous launch of four missiles. Shortly after its worldwide publication, it was revealed that the image had been doctored. In reality, only three missiles had launched, while the fourth missile, which failed to launch, was digitally inserted. This example highlighted the importance of image authentication and showed the potential implications of photo tampering on a geopolitical stage.

While historically they may have been the exception, doctored photographs today are increasingly impacting nearly every aspect of our society. While the technology to distort and manipulate digital media is developing at breakneck speeds, the technology to detect such alterations is lagging behind. To this end, I will describe some recent innovations for detecting photo tampering that have the potential to return some trust to photographs.

2. Photo Forensics

Digital watermarking has been proposed as a means by which an image can be authenticated (see, e.g., [2, 3] for general surveys). The drawback of this approach is that a watermark must be inserted at the time of recording, which would limit this approach to specially equipped digital cameras. This method also relies on the assumption that the digital watermark cannot be easily removed and reinserted—it is not yet clear whether this is a reasonable assumption (e.g., [4]). In contrast to these approaches, we have proposed techniques for detecting tampering in digital images that work in the absence of any digital watermark or signature.

Given the variety of images and forms of tampering, the forensic analysis of images benefits from a variety of tools that can detect various forms of tampering. Over the past 8 years my students, colleagues, and I have developed a suite of computational and mathematical techniques for detecting tampering in digital images. Our approach in developing each forensic tool is to first understand how a specific form of tampering disturbs certain statistical or geometric properties of an image, and then to develop a computational techniques to detect these perturbations. Within this framework, I describe five of such techniques.[1]

[1] Portions of this chapter have appeared in [5–8].

Specifically, I will describe three techniques for detecting inconsistencies in lighting, the first two of which estimate the direction to a light source, and the third of which estimates a more complex lighting environment consisting of multiple light sources. The fourth technique exploits pixel correlations that are introduced into an image as a result of the specific design of digital camera sensors. And the final technique leverages the artifacts introduced by the JPEG compression algorithm. These techniques were chosen as a representative sample of a larger body of image forensic techniques.

2.1 Lighting Direction (2D)

Consider the creation of a forgery showing two movie stars, rumored to be romantically involved, walking down a sunset beach. Such an image might be created by splicing together individual images of each movie star. In so doing, it is often difficult to exactly match the lighting effects due to directional lighting (e.g., the sun on a clear day). Differences in lighting can, therefore, be a telltale sign of digital tampering. To the extent that the direction of the light source can be estimated for different objects/people in an image, inconsistencies in the lighting direction can be used as evidence of digital tampering.

The standard approaches for estimating light source direction begin by making some simplifying assumptions (1) the surface of interest is Lambertian (the surface reflects light isotropically), (2) the surface has a constant reflectance value, (3) the surface is illuminated by a point light source infinitely far away, and (4) the angle between the surface normal and the light direction is in the range 0–90°. Under these assumptions, the image intensity can be expressed as

$$I(x,y) = R\left(\vec{N}(x,y)\cdot\vec{L}\right) + A, \tag{1}$$

where R is the constant reflectance value, $\vec{L}$ is a 3-vector pointing in the direction of the light source, $\vec{N}(x,y)$ is a 3-vector representing the surface normal at the point (x, y), and A is a constant ambient light term [9] (Fig. 1, left). If we are only interested in the direction of the light source, then the reflectance term, R, can be considered to have unit value, understanding that the estimation of $\vec{L}$ will only be within an unknown scale factor. The resulting linear equation provides a single constraint in four unknowns, the three components of $\vec{L}$ and the ambient term A.

With at least four points with the same reflectance, R, and distinct surface normals, $\vec{N}$, the light source direction and ambient term can be solved for using standard least-squares estimation. To begin, a quadratic error function, embodying the imaging model of Equation 1, is given by

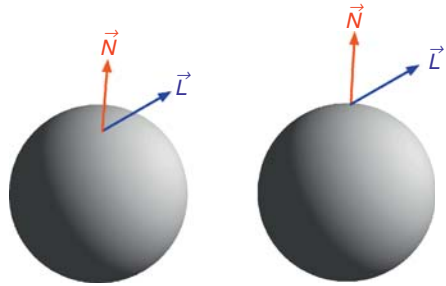

FIG. 1. Schematic diagram of the imaging geometry for 3D surface normals (left) and 2D surface normals (right). In the 2D case, the z-component of the surface normal ($\vec{N}$) is zero.

$$E\left(\vec{L},A\right) = \left|\left| M \begin{pmatrix} L_x \\ L_y \\ L_z \\ A \end{pmatrix} - \begin{pmatrix} I(x_1,y_1) \\ I(x_2,y_2) \\ \vdots \\ I(x_p,y_p) \end{pmatrix} \right|\right|^2 = \left|\left| M\vec{v} - \vec{b} \right|\right|^2, \tag{2}$$

where $\|\cdot\|$ denotes vector norm, L_x, L_y, and L_z denote the components of the light source direction $\vec{L}$, and

$$M = \begin{pmatrix} N_x(x_1,y_1) & N_y(x_1,y_1) & N_z(x_1,y_1) & 1 \\ N_x(x_2,y_2) & N_y(x_2,y_2) & N_z(x_2,y_2) & 1 \\ \vdots & \vdots & \vdots & \vdots \\ N_x(x_p,y_p) & N_y(x_p,y_p) & N_z(x_p,y_p) & 1 \end{pmatrix}, \tag{3}$$

where $N_x(x_i,y_i)$, $N_y(x_i,y_i)$, and $N_z(x_i,y_i)$ denote the components of the surface normal $\vec{N}$ at image coordinate (x_i,y_i). The quadratic error function above is minimized by differentiating with respect to the unknown, $\vec{v}$, setting the result equal to zero, and solving for $\vec{v}$ to yield the least-squares estimate:

$$\vec{v} = \left(M^{\mathrm{T}}M\right)^{-1}M^{\mathrm{T}}\vec{b}\,. \tag{4}$$

Note that this solution requires knowledge of 3D surface normals from at least four distinct points ($p \geq 4$) on a surface with the same reflectance. With only a single image and no objects of known geometry in the scene, it is unlikely that this will be possible. Most approaches to overcome this problem rely on acquiring multiple images [10] or placing an object of known geometry in the scene (e.g., a sphere) [11]. For forensic applications, these solutions are not practical.

In [12], the authors suggest a clever solution for estimating two components of the light source directionv (L_x and L_y) from only a single image. While their approach clearly provides less information regarding the light source direction, it does make the problem tractable from a single image. The authors note that at the occluding boundary of a surface, the z-component of the surface normal is zero, $N_z = 0$. In addition, the x- and y-components of the surface normal, N_x and N_y, can be estimated directly from the image (Fig. 1, right).

With this assumption, the error function of Equation 2 takes the form

$$E(\vec{L}, A) = \left|\left| M \begin{pmatrix} L_x \\ L_y \\ A \end{pmatrix} - \begin{pmatrix} I(x_1, y_1) \\ I(x_2, y_2) \\ \vdots \\ I(x_p, y_p) \end{pmatrix} \right|\right|^2 = \left|\left| M\vec{v} - \vec{b} \right|\right|^2, \tag{5}$$

where

$$M = \begin{pmatrix} N_x(x_1, y_1) & N_y(x_1, y_1) & 1 \\ N_x(x_2, y_2) & N_y(x_2, y_2) & 1 \\ \vdots & \vdots & \vdots \\ N_x(x_p, y_p) & N_y(x_p, y_p) & 1 \end{pmatrix}. \tag{6}$$

This error function is minimized, as before, using standard least squares to yield the same solution as in Equation 4, but with the matrix M taking the form given in Equation 6. In this case, the solution requires knowledge of 2D surface normals from at least three distinct points ($p \geq 3$) on a surface with the same reflectance.

The intensity, $I(x_i, y_i)$, at a boundary point, (x_i, y_i), cannot be directly measured from the image as the surface is occluded. The authors in [12] note, however, that the intensity can be extrapolated by considering the intensity profile along a ray coincident to the 2D surface normal. They also found that simply using the intensity close to the border of the surface is often sufficient.

We extend this basic formulation in two ways. First, we estimate the two-dimensional light source direction from local patches along an object's boundary (as opposed to along extended boundaries as in [12]). This is done to relax the assumption that the reflectance along the entire surface is constant. Then, a regularization (smoothness) term is introduced to better condition the final estimate of light source direction.

The constant reflectance assumption is relaxed by assuming that the reflectance for a local surface patch (as opposed to the entire surface) is constant. This requires us to estimate individual light source directions, $\vec{L}^i$, for each patch along a surface. Under the infinite light source assumption, the orientation of these estimates should not vary, but their magnitude may (recall that the estimate of the light source is only within a scale factor, which depends on the reflectance value R, Equation 1).

Consider a surface partitioned into n patches, and, for notational simplicity, assume that each patch contains p points. The new error function to be minimized is constructed by packing together, for each patch, the 2D version of the constraint of Equation 1:

$$E_1\left(\vec{L}^1,\ldots,\vec{L}^n,A\right) = \left|\left| M\begin{pmatrix} L_x^1 \\ L_y^1 \\ \vdots \\ L_x^n \\ L_y^n \\ A \end{pmatrix} - \begin{pmatrix} I\left(x_1^1,y_1^1\right) \\ \vdots \\ I\left(x_p^1,y_p^1\right) \\ \vdots \\ I\left(x_1^n,y_1^n\right) \\ \vdots \\ I\left(x_p^n,y_p^n\right) \end{pmatrix} \right|\right|^2 = \left|\left| M\,\vec{v} - \vec{b} \right|\right|^2, \tag{7}$$

where

$$M = \begin{pmatrix} N_x\left(x_1^1,y_1^1\right) & N_y\left(x_1^1,y_1^1\right) & & 0 & 0 & 1 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ N_x\left(x_p^1,y_p^1\right) & N_y\left(x_p^1,y_p^1\right) & & 0 & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & & N_x\left(x_1^n,y_1^n\right) & N_y\left(x_1^n,y_1^n\right) & 1 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & & N_x\left(x_p^n,y_p^n\right) & N_y\left(x_p^n,y_p^n\right) & 1 \end{pmatrix}. \tag{8}$$

The above quadratic error function is minimized, as before, using least squares with the solution taking on the same form as in Equation 4. In this case, the solution provides n estimates of the 2D light directions, $\vec{L}^1,\ldots,\vec{L}^n$, and an ambient term A. Note that while individual light source directions are estimated for each surface patch, a single ambient term is assumed.

While the local estimation of light source directions allows for the relaxation of the constant reflectance assumption, it could potentially yield less stable results. Note that under the assumption of an infinite point light source, the orientation of the n light directions should be equal. With the additional assumption that the change in reflectance from patch to patch is relatively small (i.e., the change in the magnitude of neighboring $\vec{L}^i$ is small), we can condition the individual estimates with the following regularization term:

$$E_2\left(\vec{L}^1,\ldots,\vec{L}^n\right) = \sum_{i=2}^{n} \left|\left| \vec{L}^i - \vec{L}^i - 1 \right|\right|^2. \tag{9}$$

This additional error term penalizes neighboring estimates that are different from one another. The quadratic error function $E_1(\cdot)$ (Equation 7) is conditioned by

combining it with the regularization term $E_2(\cdot)$, scaled by a factor λ, to yield the final error function:

$$E\left(\vec{L}^1,\ldots,\vec{L}^n,A\right)=E_1\left(\vec{L}^1,\ldots,\vec{L}^n,A\right)+\lambda E_2\left(\vec{L}^1,\ldots,\vec{L}^n\right). \tag{10}$$

This combined error function can still be minimized using least-squares minimization. The error function $E_2(\cdot)$ is first written in a more compact and convenient form as

$$E_2\left(\vec{v}\right)=\left\|C\vec{v}\right\|^2, \tag{11}$$

where the $2n-2\times 2n+1$ matrix C is given by

$$C=\begin{pmatrix} -1 & 0 & 1 & 0 & \ldots & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & \ldots & 0 & 0 & 0 & 0 & 0 \\ \vdots & & & & \ddots & & & & & \vdots \\ 0 & 0 & 0 & 0 & \ldots & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \ldots & 0 & -1 & 0 & 1 & 0 \end{pmatrix}, \tag{12}$$

with $\vec{v}=\left(L_x^1\ L_y^1\ L_x^2\ L_y^2\ldots L_x^n\ L_y^n\ A\right)^{\mathrm{T}}$. The error function of Equation 10 then takes the form

$$E\left(\vec{v}\right)=\left\|M\vec{v}-\vec{b}\right\|^2+\lambda\left\|C\vec{v}\right\|^2. \tag{13}$$

Differentiating this error function yields

$$\begin{aligned} E'\left(\vec{v}\right) &= 2M^{\mathrm{T}}M\,\vec{v}-2M^{\mathrm{T}}\vec{b}+2\lambda C^{\mathrm{T}}C\vec{v} \\ &= 2(M^{\mathrm{T}}M+\lambda C^{\mathrm{T}}C)\,\vec{v}-2M^{\mathrm{T}}\vec{b}. \end{aligned} \tag{14}$$

Setting this result equal to zero and solving for $\vec{v}$ yields the least-squares estimate:

$$\vec{v}=\left(M^{\mathrm{T}}M+\lambda C^{\mathrm{T}}C\right)^{+}M^{\mathrm{T}}\vec{b}, \tag{15}$$

where $+$ denotes pseudoinverse. The final light direction estimate is computed by averaging the n resulting light direction estimates, $\vec{L}^1,\ldots,\vec{L}^n$.

The light direction estimation requires the localization of an occluding boundary. These boundaries are extracted by manually selecting points in the image along an occluding boundary. This rough estimate of the position of the boundary is used to define its spatial extent. The boundary is then partitioned into approximately eight small patches. Three points near the occluding boundary are manually selected for each patch, and fit with a quadratic curve. The surface normals along each patch are then estimated analytically from the resulting quadratic fit.

Shown in Fig. 2 are eight images of objects illuminated by the sun on a clear day. To determine the accuracy of our approach, a calibration target, consisting of a flat surface with a rod extending from the center, was placed in each scene. The target was approximately parallel to the image plane, so that the shadow cast by the rod indicated the direction of the sun. Errors in our estimated light source direction are given relative to this orientation. The average estimation error is 4.8° with a minimum and maximum error of 0.6° and 10.9°. The image returning the largest error is the parking meters. There are probably at least three reasons for this larger error, and for errors in general. The first is that the metallic surface violates the Lambertian assumption. The second is that the paint on the meter is worn in several spots causing the reflectance to vary, at times, significantly from patch to patch. And the third is that we did not calibrate the camera so as to remove luminance nonlinearities (e.g., gamma correction) in the image.

The creation of a digital forgery often involves combining objects/people from separate images. In so doing, it is difficult to exactly match the lighting effects due to directional lighting (e.g., the sun on a clear day). At least one reason for this is that such a manipulation may require the creation or removal of shadows and lighting gradients. And while large inconsistencies in lighting direction may be fairly obvious, there is evidence from the human psychophysics literature that we are surprisingly insensitive to differences in lighting across an image [13, 14]. To the extent that the direction of the light source can be estimated for different objects/people in an image, inconsistencies in lighting can be used as evidence of digital tampering.

2.2 Lighting Direction (3D)

In the previous section, we described how to estimate the light source direction in 2D. While this approach has the benefit of being applicable to arbitrary objects, it has the drawback that it can only determine the direction to the light source within 1 degree of ambiguity. Next we describe how the full 3D light source direction can be estimated by leveraging a 3D model of the human eye. Specifically, we describe how to estimate the 3D direction to a light source from specular highlights on the eyes.

The position of a specular highlight is determined by the relative positions of the light source, the reflective surface and the viewer (or camera). In Fig. 3, for example, is a diagram showing the creation of a specular highlight on an eye. In this diagram, the three vectors $\vec{L}$, $\vec{N}$, and $\vec{R}$ correspond to the direction to the light, the surface normal at the point at which the highlight is formed, and the direction in which the highlight will be seen. For a perfect reflector, the highlight is seen only when the

FIG. 2. Shown are eight images with the extracted occluding boundaries (black), individual light source estimates (white), and the final average light source direction (large arrow). In each image, the cast shadow on the calibration target indicates the direction to the illuminating sun, and has been darkened to enhance visibility.

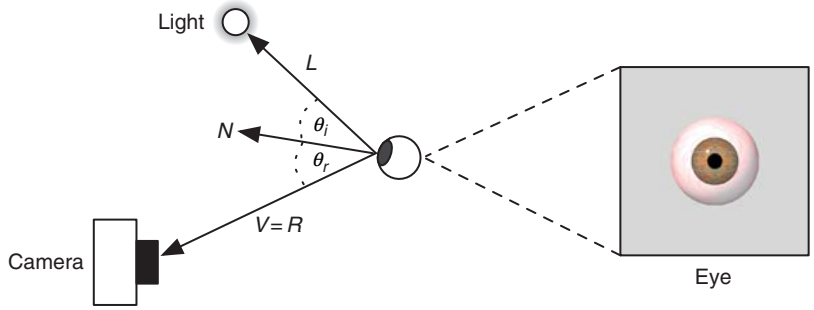

FIG. 3. The formation of a specular highlight on an eye (small white dot on the iris). The position of the highlight is determined by the surface normal $\vec{N}$ and the relative directions to the light source $\vec{L}$ and viewer $\vec{V}$.

view direction $\vec{V} = \vec{R}$. For an imperfect reflector, a specular highlight can be seen for viewing directions $\vec{V}$ near $\vec{R}$, with the strongest highlight seen when $\vec{V} = \vec{R}$.

An algebraic relationship between the vectors $\vec{L}$, $\vec{N}$, and $\vec{V}$ is first derived. We then show how the 3D vectors $\vec{N}$ and $\vec{V}$ can be estimated from a single image, from which the direction to the light source $\vec{L}$ is determined.

The law of reflection states that a light ray reflects off of a surface at an angle of reflection θ_r equal to the angle of incidence θ_i, where these angles are measured with respect to the surface normal $\vec{N}$ (Fig. 3).

Assuming unit length vectors, the direction of the reflected ray $\vec{R}$ can be described in terms of the light direction $\vec{L}$ and the surface normal $\vec{N}$:

$$\begin{aligned} \vec{R} &= \vec{L} + 2\Big(\cos(\theta_i)\vec{N} - \vec{L}\Big) \\ &= 2\cos(\theta_i)\vec{N} - \vec{L}. \end{aligned} \tag{16}$$

By assuming a perfect reflector ($\vec{V} = \vec{R}$), the above constraint yields

$$\begin{aligned} \vec{L} &= 2\cos(\theta_i)\vec{N} - \vec{V} \\ &= 2\Big(\vec{V}^T\vec{N}\Big)\vec{N} - \vec{V}. \end{aligned} \tag{17}$$

The light direction $\vec{L}$ can, therefore, be estimated from the surface normal $\vec{N}$ and view direction $\vec{V}$ at a specular highlight. Note that the light direction is specified with respect to the eye, and not the camera. In practice, all of these vectors will be placed in a common coordinate system, allowing us to compare light directions across the image.

To estimate the surface normal $\vec{N}$ and view direction $\vec{V}$ in a common coordinate system, we first need to estimate the projective transform that describes the transformation from world to image coordinates. With only a single image, this

calibration is generally an underconstrained problem. In our case, however, the known geometry of the eye can be exploited to estimate this required transform. Throughout, uppercase symbols will denote world coordinates and lowercase will denote camera/image coordinates.

The limbus, the boundary between the sclera (white part of the eye) and the iris (colored part of the eye), can be well modeled as a circle [15]. The image of the limbus, however, will be an ellipse except when the eye is directly facing the camera. Intuitively, the distortion of the ellipse away from a circle will be related to the pose and position of the eye relative to the camera. We therefore seek the transform that aligns the image of the limbus to a circle.

In general, a projective transform that maps 3D world coordinates to 2D image coordinates can be represented, in homogeneous coordinates, as a 3×4 matrix. We assume that points on a limbus are coplanar, and define the world coordinate system such that the limbus lies in the $Z = 0$ plane. With this assumption, the projective transformation reduces to a 3×3 planar projective transform [16], where the world points $\vec{X}$ and image points $\vec{x}$ are represented by 2D homogeneous vectors.

Points on the limbus in our world coordinate system satisfy the following implicit equation of a circle:

$$f\left(\vec{X}; \vec{\alpha}\right) = (X_1 - C_1)^2 + (X_2 - C_2)^2 - r^2 = 0, \tag{18}$$

where $\vec{\alpha} = (C_1 \quad C_2 \quad r)^{\mathrm{T}}$ denotes the circle center and radius.

Consider a collection of points, $\vec{X}_i, i = 1, \ldots, m$, each of which satisfy Equation 18. Under an ideal pinhole camera model, the world point $\vec{X}_i$ maps to the image point $\vec{x}_i$ as follows:

$$\vec{x}_i = H\vec{X}_i, \tag{19}$$

where H is a 3×3 projective transform matrix.

The estimation of H can be formulated in an orthogonal distance fitting framework. Let $E(\cdot)$ be an error function on the parameter vector $\vec{\alpha}$ and the unknown projective transform H:

$$E\left(\vec{\alpha}, H\right) = \sum_{i=1}^{m} \min_{\vec{X}} \left|\left| \vec{x}_i - H\vec{X} \right|\right|^2, \tag{20}$$

where $\vec{\hat{X}}$ is on the circle parametrized by $\vec{\alpha}$. The error embodies the sum of the squared errors between the data, $\vec{x}_i$, and the closest point on the model, $\vec{\hat{X}}$. This error function is minimized using nonlinear least squares via the Levenberg–Marquardt iteration [17].

Once estimated, the projective transform H can be decomposed in terms of intrinsic and extrinsic camera parameters [16]. The intrinsic parameters consist of the camera

focal length, camera center, skew, and aspect ratio. For simplicity, we will assume that the camera center is the image center and that the skew is 0 and the aspect ratio is 1, leaving only the focal length f. The extrinsic parameters consist of a rotation matrix R and translation vector $\vec{t}$ that define the transformation between the world and camera coordinate systems. Since the world points lie on a single plane, the projective transform can be decomposed in terms of the intrinsic and extrinsic parameters as

$$H = \lambda K \left(\vec{r}_1 \; \vec{r}_2 \; \vec{t} \right), \tag{21}$$

where the 3×3 intrinsic matrix K is

$$K = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{22}$$

where λ is a scale factor, the column vectors $\vec{r}_1$ and $\vec{r}_2$ are the first two columns of the rotation matrix R, and $\vec{t}$ is the translation vector.

With a known focal length f, and hence a known matrix K, the world to camera coordinate transform $\hat{H}$ can be estimated directly:

$$\begin{aligned} \frac{1}{\lambda} K^{-1} H &= \left(\vec{r}_1 \; \vec{r}_2 \; \vec{t} \right) \\ \hat{H} &= \left(\vec{r}_1 \; \vec{r}_2 \; \vec{t} \right), \end{aligned} \tag{23}$$

where the scale factor λ is chosen so that $\vec{r}_1$ and $\vec{r}_2$ are unit vectors. The complete rotation matrix is given by

$$R = \left(\vec{r}_1 \; \vec{r}_2 \; \vec{r}_1 \times \vec{r}_2 \right), \tag{24}$$

where $\times$ denotes cross product. If the focal length is unknown, it can be directly estimated as described in [6].

Recall that the minimization of Equation 20 yields both the transform H and the circle parameters $\vec{\alpha}$ for the limbus. The unit vector from the center of the limbus to the origin of the camera coordinate system is the view direction, $\vec{v}$. Let $\vec{X}_c = (C_1 \; C_2 \; 1)$ denote the estimated center of a limbus in world coordinates. In the camera coordinate system, this point is given by

$$\vec{x}_c = \hat{H} \vec{X}_c. \tag{25}$$

The view direction, as a unit vector, in the camera coordinate system is then given by

$$\vec{v} = -\frac{\vec{x}_c}{\| \vec{x}_c \|}, \tag{26}$$

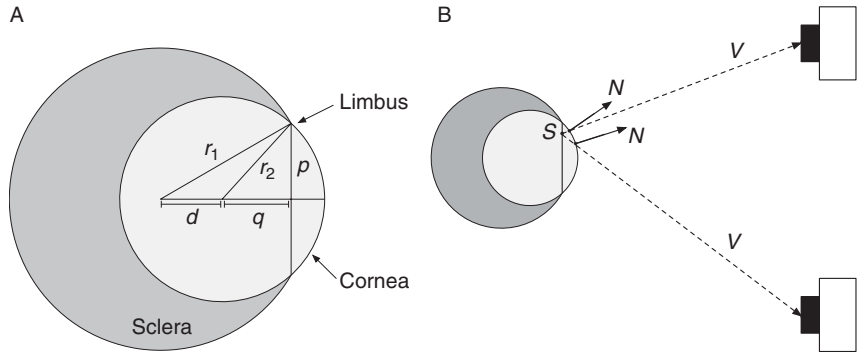

FIG. 4. (A) A side view of a 3D model of the human eye. The larger sphere represents the sclera and the smaller sphere represents the cornea. The limbus is defined by the intersection of the two spheres. (B) The surface normal at a point $\vec{S}$ in the plane of the limbus depends on the view direction $\vec{V}$.

where the negative sign reverses the vector so that it points from the eye to the camera.

The 3D surface normal $\vec{N}$ at a specular highlight is estimated from a 3D model of the human eye [18]. The model consists of a pair of spheres as illustrated in Fig. 4A. The larger sphere, with radius $r_1 = 11.5$ mm, represents the sclera and the smaller sphere, with radius $r_2 = 7.8$ mm, represents the cornea. The centers of the spheres are displaced by a distance $d = 4.7$ mm. The limbus, a circle with radius $p = 5.8$ mm, is defined by the intersection of the two spheres. The distance between the center of the smaller sphere and the plane containing the limbus is $q = 5.25$ mm. These measurements vary slightly among adults, and the radii of the spheres are approximately 0.1 mm smaller for female eyes [18, 19].

Consider a specular highlight in world coordinates at location $\vec{S} = (S_x\ S_y)$, measured with respect to the center of the limbus. The surface normal at $\vec{S}$ depends on the view direction $\vec{V}$. Fig. 4B is a schematic showing this relationship for two different positions of the camera. The surface normal $\vec{N}$ is determined by intersecting the ray leaving $\vec{S}$, along the direction $\vec{V}$, with the edge of the sphere. This intersection can be computed by solving a quadratic system for k, the distance between $\vec{S}$ and the edge of the sphere,

$$\begin{aligned}(S_x + kV_x)^2 + (S_y + kV_y)^2 + (q + kV_z)^2 &= r_2^2,\\ k^2 + 2(S_xV_x + S_yV_y + qV_z)k + \left(S_x^2 + S_y^2 + q^2 - r_2^2\right) &= 0,\end{aligned} \tag{27}$$

where q and r_2 are specified by the 3D model of the eye. The view direction $\vec{V} = (V_x\ V_y\ V_z)$ in the world coordinate system is given by

$$\vec{V} = R^{-1}\ \vec{v}, \tag{28}$$

where $\vec{v}$ is the view direction in camera coordinates and R is the estimated rotation between the world and camera coordinate systems. The surface normal $\vec{N}$ in the world coordinate system is then given by

$$\vec{N} = \begin{pmatrix} S_x + kV_x \\ S_y + kV_y \\ q + kV_z \end{pmatrix}, \tag{29}$$

and in camera coordinates: $\vec{n} = R\ \vec{N}$.

Consider a specular highlight $\vec{x}_s$ specified in image coordinates and the estimated projective transform H from world to image coordinates. The inverse transform H^{-1} maps the coordinates of the specular highlight into world coordinates:

$$\vec{X}_s = H^{-1}\vec{x}_s. \tag{30}$$

The center $\vec{C}$ and radius r of the limbus in the world coordinate system determine the coordinates of the specular highlight, $\vec{S}$, with respect to the model:

$$\vec{S} = \frac{p}{r}\left(\vec{X}_s - \vec{C}\right), \tag{31}$$

where p is specified by the 3D model of the eye. The position of the specular highlight $\vec{S}$ is then used to determine the surface normal $\vec{N}$. Combined with the estimate of the view direction $\vec{V}$, the light source direction $\vec{L}$ can be estimated from Equation 17. To compare light source estimates in the image, the light source estimate is converted to camera coordinates: $\vec{l} = R\ \vec{L}$.

To test the efficacy of this light estimation, synthetic images of eyes were rendered using the pbrt environment [20]. The shape of the eyes conformed to the 3D model described above and the eyes were placed in 1 of 12 different locations. For each location, the eyes were rotated by a unique amount relative to the camera. The eyes were illuminated with two light sources: a fixed light directly in line with the camera and a second light placed in one of four different positions. The 12 locations and 4 light directions gave rise to 48 images (Fig. 5). Each image was rendered at a resolution of 1200 × 1600 pixels, with the cornea occupying less than 0.1% of the entire image. Shown in Fig. 5 are several examples of the rendered eyes, along with a schematic of the imaging geometry.

The limbus and position of the specular highlight(s) were automatically extracted from the rendered image. For each highlight, the projective transform H, the view direction $\vec{v}$ and surface normal $\vec{n}$ were estimated, from which the direction to the light source $\vec{l}$ was determined. The angular error between the estimated $\vec{l}$ and actual

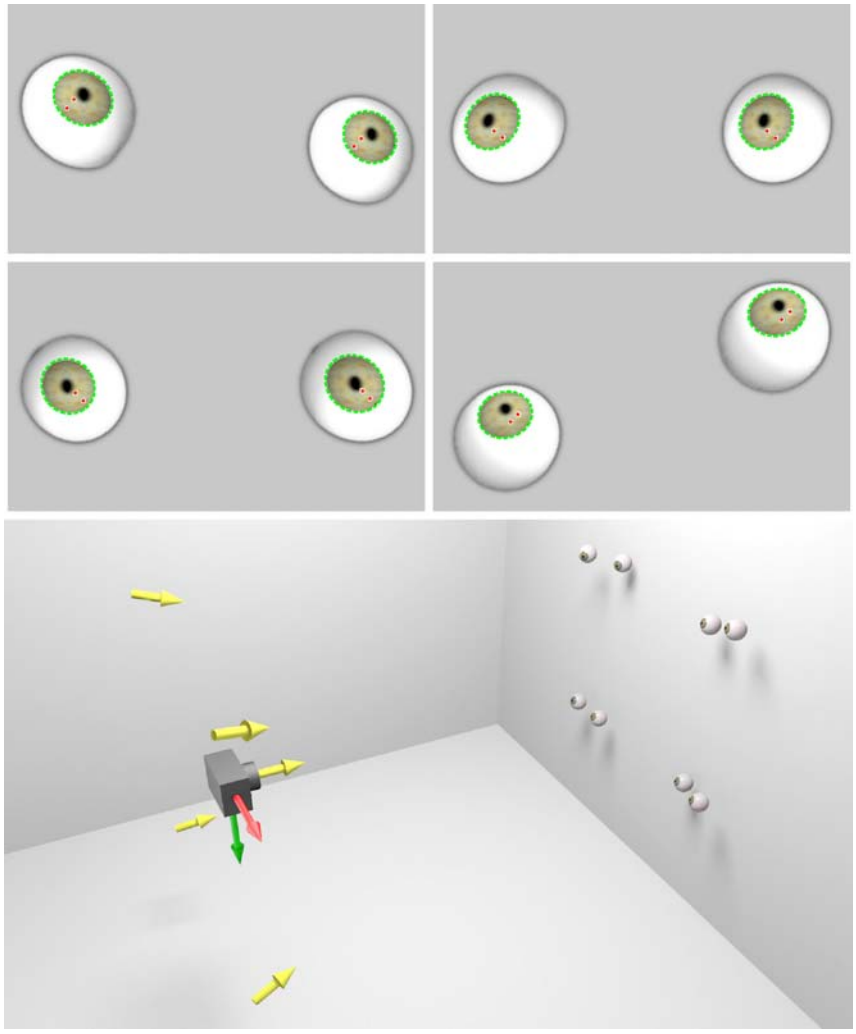

FIG. 5. Synthetically generated eyes. Each of the upper panels corresponds to different positions and orientations of the eyes and locations of the light sources. The ellipse fit to each limbus is shown with a dashed line, and the small dots denote the positions of the specular highlights. Shown below is a schematic of the imaging geometry: the position of the lights, camera, and a subset of the eye positions.

$\vec{l}_0$ light directions is computed as $\phi = \cos^{-1}\left(\vec{l}^{\,\mathrm{T}} \cdot \vec{l}_0\right)$, where the vectors are normalized to be unit length. With a known focal length, the average angular error in estimating the light source direction was 2.8° with a standard deviation of 1.3° and a maximum error of 6.8°. With an unknown focal length, the average error was 2.8° with a standard deviation of 1.3° and a maximum error of 6.3°.

To further test the efficacy of our technique, we photographed a subject under controlled lighting. A camera and two lights were arranged along a wall, and the subject was positioned 250 cm in front of the camera and at the same elevation. The first light L_1 was positioned 130 cm to the left of and 60 cm above the camera. The second light L_2 was positioned 260 cm to the right and 80 cm above the camera. The subject was placed in five different locations and orientations relative to the camera and lights (Fig. 6). A 6-megapixel Nikon D100 camera with a 35 mm lens was set to capture in the highest quality JPEG format.

For each image, an ellipse was manually fit to the limbus of each eye. In these images, the limbus did not form a sharp boundary—the boundary spanned roughly 3 pixels. As such, we fit the ellipses to the better defined inner outline [21] (Fig. 6).

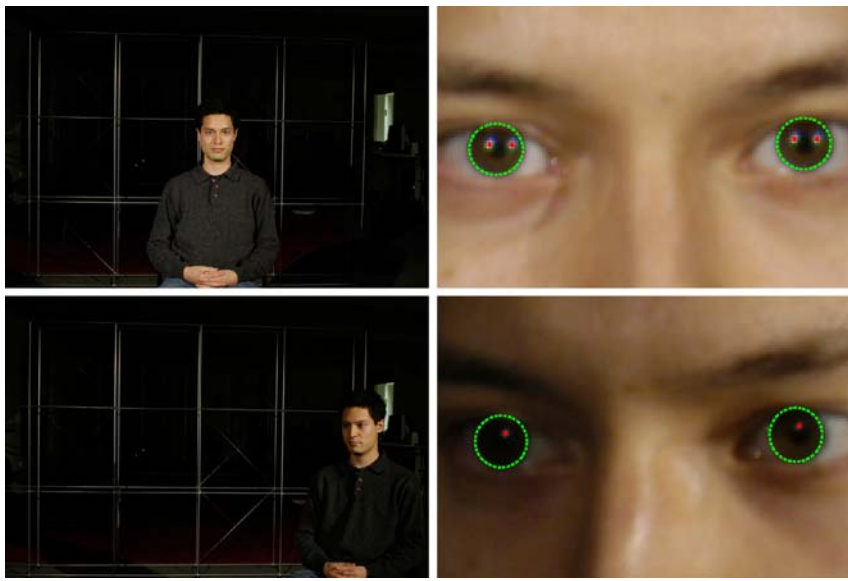

FIG. 6. A subject at different locations and orientations relative to the camera and two light sources. Shown to the right are magnified views of the eyes. The ellipse fit to each limbus is shown with a dashed line and the small dots denote the positions of the specular highlights. See also Table I.

The radius of each limbus was approximately 9 pixels, and the cornea occupied 0.004% of the entire image. Each specular highlight was localized by specifying a bounding rectangular area around each highlight and computing the centroid of the selection. The weighting function for the centroid computation was chosen to be the squared (normalized) pixel intensity. The location to the light source(s) was estimated for each pair of eyes assuming a known and unknown focal length. The angular errors for each image are given in Table I. Note that in some cases an estimate for one of the light sources was not possible when the highlight was not visible on the cornea. With a known focal length, the average angular error was 8.6°, and with an unknown focal length, the average angular error was 10.5°.

When creating a composite of two or more people, it is often difficult to match the lighting conditions under which each person was originally photographed. Specular highlights that appear on the eye are a powerful cue as to the shape, color, and location of the light source(s). Inconsistencies in these properties of the light can be used as evidence of tampering. We can measure the 3D direction to a light source from the position of the highlight on the eye. While we have not specifically focused on it, the shape and color of a highlight are relatively easy to quantify and measure and should also prove helpful in exposing digital forgeries. Since specular highlights tend to be relatively small on the eye, it is possible to manipulate them to conceal traces of tampering. To do so, the shape, color, and location of the highlight would have to be constructed so as to be globally consistent with the lighting in other parts of the image. Inconsistencies in this lighting may be detectable using the technique described in the previous section.

TABLE I
ANGULAR ERRORS (°) IN ESTIMATING THE LIGHT DIRECTION FOR THE IMAGES SHOWN IN FIG. 6

	Left eye		Right eye		Left eye		Right eye	
Image	L_1	L_2	L_1	L_2	L_1	L_2	L_1	L_2
1	5.8	7.6	3.8	1.6	5.8	7.7	3.9	1.7
2	–	8.7	–	0.8	–	10.4	–	18.1
3	9.3	–	11.0	–	17.6	–	10.1	–
4	12.5	16.4	7.5	7.3	10.4	13.6	7.4	5.6
5	14.0	–	13.8	–	17.4	–	16.5	–

On the left are the errors for a known focal length, and on the right are the errors for an unknown focal length. A ‘‘–’’ indicates that the specular highlight for that light was not visible on the cornea.

2.3 Lighting Environment

In the previous two sections, we have shown how to estimate the direction to a light source, and how inconsistencies in the illuminant direction can be used to detect tampering. This approach is appropriate when the lighting is dominated by a single light source, but is less appropriate in more complex lighting environments containing multiple light sources or nondirectional lighting. Here, we describe how to quantify such complex lighting environments and how to use inconsistencies in lighting to detect tampering.

The lighting of a scene can be complex—any number of lights can be placed in any number of positions, creating different lighting environments. To model such complex lighting, we assume that the lighting is distant and that surfaces in the scene are convex and Lambertian. To use this model in a forensic setting, we also assume that the surface reflectance is constant and that the camera response is linear.

Under the assumption of distant lighting, an arbitrary lighting environment can be expressed as a nonnegative function on the sphere, $L(\vec{V})$, where $\vec{V}$ is a unit vector in Cartesian coordinates and the value of $L(\vec{V})$ is the intensity of the incident light along direction $\vec{V}$ (Fig. 7). If the object being illuminated is convex, the irradiance (light received) at any point on the surface is due to only the lighting environment; that is, there are no cast shadows or interreflections [22]. As a result, the irradiance,

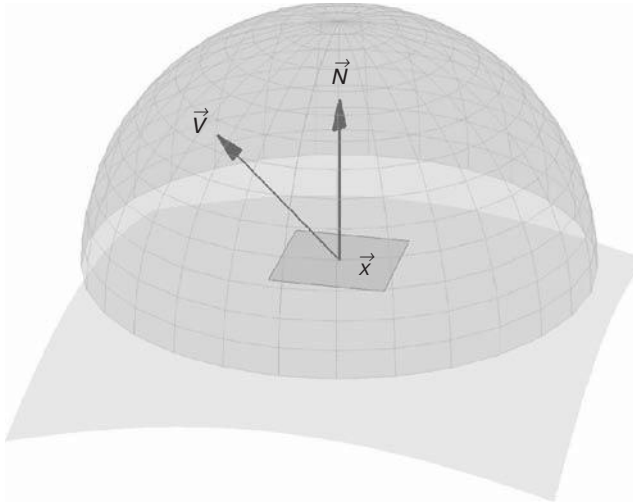

FIG. 7. The irradiance (light received) at a point $\vec{x}$ is determined by integrating the amount of incoming light from all directions $\vec{V}$ in the hemisphere about the surface normal $\vec{N}$.

$E(\vec{N})$, can be parametrized by the unit length surface normal $\vec{N}$ and written as a convolution of the reflectance function of the surface, $R(\vec{V}, \vec{N})$, with the lighting environment $L(\vec{V})$:

$$E(\vec{N}) = \int_{\Omega} L(\vec{V}) R(\vec{V}, \vec{N}) \mathrm{d}\Omega, \tag{32}$$

where Ω represents the surface of the sphere and $\mathrm{d}\Omega$ is an area differential on the sphere. For a Lambertian surface, the reflectance function is a clamped cosine:

$$R(\vec{V}, \vec{N}) = \max(\vec{V} \cdot \vec{N}, 0), \tag{33}$$

which is either the cosine of the angle between vectors $\vec{V}$ and $\vec{N}$, or zero when the angle is greater than 90°. This reflectance function effectively limits the integration in Equation 32 to the hemisphere about the surface normal $\vec{N}$ (Fig. 7). In addition, while we have assumed no cast shadows, Equation 33 explicitly models attached shadows, that is, shadows due to surface normals facing away from the direction $\vec{V}$.

The convolution in Equation 32 can be simplified by expressing both the lighting environment and the reflectance function in terms of spherical harmonics. Spherical harmonics form an orthonormal basis for piecewise continuous functions on the sphere and are analogous to the Fourier basis on the line or plane. The first three orders of spherical harmonics are shown in Fig. 8 and defined as

$$Y_{0,0}(\vec{N}) = \frac{1}{\sqrt{4\pi}}, \quad Y_{1,-1}(\vec{N}) = \sqrt{\frac{3}{4\pi}}y, \quad Y_{1,0}(\vec{N}) = \sqrt{\frac{3}{4\pi}}z,$$
$$Y_{1,1}(\vec{N}) = \sqrt{\frac{3}{4\pi}}x, \quad Y_{2,-2}(\vec{N}) = 3\sqrt{\frac{5}{12\pi}}xy, \quad Y_{2,-1}(\vec{N}) = 3\sqrt{\frac{5}{12\pi}}yz,$$
$$Y_{2,0}(\vec{N}) = \frac{1}{2}\sqrt{\frac{5}{4\pi}}(3z^2 - 1), \quad Y_{2,1}(\vec{N}) = 3\sqrt{\frac{5}{12\pi}}xz, \quad Y_{2,2}(\vec{N}) = \frac{3}{2}\sqrt{\frac{5}{12\pi}}(x^2 - y^2),$$

where $\vec{N} = (x\ y\ z)$ in Cartesian coordinates.

The lighting environment expanded in terms of these spherical harmonics is

$$L(\vec{V}) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} l_{n,m} Y_{n,m}(\vec{V}), \tag{34}$$

where $Y_{n,m}(\cdot)$ is the mth spherical harmonic of order n, and $l_{n,m}$ is the corresponding coefficient of the lighting environment. Similarly, the reflectance function for Lambertian surfaces, $R(\vec{V}, \vec{N})$, can be expanded in terms of spherical harmonics,

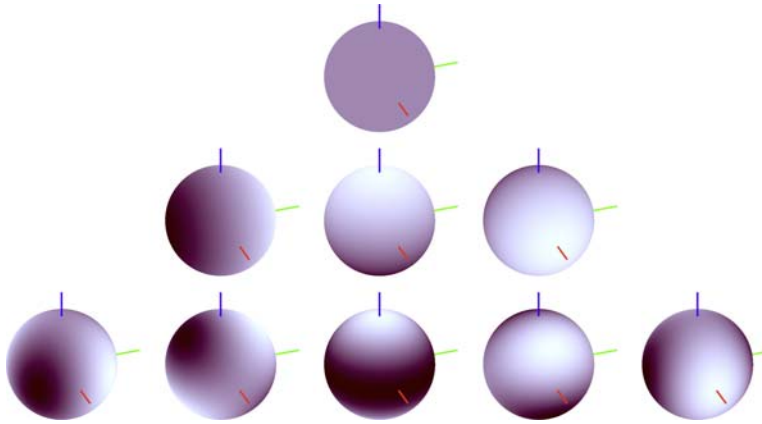

FIG. 8. The first three orders of spherical harmonics as functions on the sphere. Shown from top to bottom are the order zero spherical harmonic, $Y_{0,0}(\cdot)$; the three order one spherical harmonics, $Y_{1,m}(\cdot)$; and the five order two spherical harmonics, $Y_{2,m}(\cdot)$.

and due to its symmetry about the surface normal, only harmonics with $m = 0$ appear in the expansion

$$R\left(\vec{V},\ \vec{N}\right) = \sum_{n=0}^{\infty} r_n Y_{n,0}\left(\left(0\ 0\ \vec{V}\cdot\vec{N}\right)^{\mathrm{T}}\right). \tag{35}$$

Note that for $m = 0$, the spherical harmonic $Y_{n,0}(\cdot)$ depends only on the z-component of its argument.

Convolutions of functions on the sphere become products when represented in terms of spherical harmonics [22, 23]. As a result, the irradiance (Equation 32) takes the form

$$E\left(\vec{N}\right) = \sum_{n=0}^{\infty}\sum_{m=-n}^{n} \hat{r}_n l_{n,m} Y_{n,m}\left(\vec{N}\right), \tag{36}$$

where

$$\hat{r}_n = \sqrt{\frac{4\pi}{2n+1}} r_n. \tag{37}$$

The key observation in [22, 23] was that the coefficients $\hat{r}_n$ for a Lambertian reflectance function decay rapidly, and thus the infinite sum in Equation 36 can be well approximated by the first nine terms:

$$E\left(\vec{N}\right) \approx \sum_{n=0}^{2}\sum_{m=-n}^{n} \hat{r}_n l_{n,m} Y_{n,m}\left(\vec{N}\right). \tag{38}$$

Since the constants $\hat{r}_n$ are known for a Lambertian reflectance function, the irradiance of a convex Lambertian surface under arbitrary distant lighting can be well modeled by the nine lighting environment coefficients $l_{n,m}$ up to order 2.

Irradiance describes the total amount of light reaching a point on a surface. For a Lambertian surface, the reflected light, or radiosity, is proportional to the irradiance by a reflectance term ρ. In addition, Lambertian surfaces emit light uniformly in all directions, so the amount of light received by a viewer (i.e., camera) is independent of the view direction.

A camera maps its received light to intensity through a camera response function $f(\cdot)$. Assuming the reflectance term ρ is constant across the surface, the measured intensity at a point $\vec{x}$ in the image is given by [24]

$$I\left(\vec{x}\right) = f\left(\rho t E\left(\vec{N}\left(\vec{x}\right)\right)\right), \tag{39}$$

where $E(\cdot)$ is the irradiance, $\vec{N}\left(\vec{x}\right)$ is the surface normal at point $\vec{x}$, and t is the exposure time. For simplicity, we assume a linear camera response, and thus the intensity is related to the irradiance by an unknown multiplicative factor, which is assumed to have unit value—this assumption implies that the lighting coefficients can only be estimated to within an unknown scale factor. Under these assumptions, the relationship between image intensity and irradiance is simply

$$I\left(\vec{x}\right) = E\left(\vec{N}\left(\vec{x}\right)\right). \tag{40}$$

Since, under our assumptions, the intensity is equal to irradiance, Equation 40 can be written in terms of spherical harmonics by expanding Equation 38:

$$\begin{aligned} I\left(\vec{x}\right) = {} & l_{0,0}\pi Y_{0,0}(\vec{N}) + l_{1,-1}\frac{2\pi}{3}Y_{1,-1}(\vec{N}) + l_{1,0}\frac{2\pi}{3}Y_{1,0}(\vec{N}) + l_{1,1}\frac{2\pi}{3}Y_{1,1}(\vec{N}) \\ & + l_{2,-2}\frac{\pi}{4}Y_{2,-2}(\vec{N}) + l_{2,-1}\frac{\pi}{4}Y_{2,-1}(\vec{N}) + l_{2,0}\frac{\pi}{4}Y_{2,0}(\vec{N}) \\ & + l_{2,1}\frac{\pi}{4}Y_{2,1}(\vec{N}) + l_{2,2}\frac{\pi}{4}Y_{2,2}(\vec{N}). \end{aligned} \tag{41}$$

Note that this expression is linear in the nine lighting environment coefficients, $l_{0,0}$ to $l_{2,2}$. As such, given 3D surface normals at $p \geq 9$ points on the surface of an object, the lighting environment coefficients can be estimated as the least-squares solution to the following system of linear equations:

$$\begin{pmatrix} \pi Y_{0,0}\left(\vec{N}\left(\vec{x}_1\right)\right) & \frac{2\pi}{3} Y_{1,-1}\left(\vec{N}\left(\vec{x}_1\right)\right) & \dots & \frac{\pi}{4} Y_{2,2}\left(\vec{N}\left(\vec{x}_1\right)\right) \\ \pi Y_{0,0}\left(\vec{N}\left(\vec{x}_2\right)\right) & \frac{2\pi}{3} Y_{1,-1}\left(\vec{N}\left(\vec{x}_2\right)\right) & \dots & \frac{\pi}{4} Y_{2,2}\left(\vec{N}\left(\vec{x}_2\right)\right) \\ \vdots & \vdots & & \vdots \\ \pi Y_{0,0}\left(\vec{N}\left(\vec{x}_p\right)\right) & \frac{2\pi}{3} Y_{1,-1}\left(\vec{N}\left(\vec{x}_p\right)\right) & \dots & \frac{\pi}{4} Y_{2,2}\left(\vec{N}\left(\vec{x}_p\right)\right) \end{pmatrix} \begin{pmatrix} l_{0,0} \\ l_{1,-1} \\ \vdots \\ l_{2,2} \end{pmatrix} = \begin{pmatrix} I\left(\vec{x}_1\right) \\ I\left(\vec{x}_2\right) \\ \vdots \\ I\left(\vec{x}_p\right) \end{pmatrix}, \tag{42}$$

$$M\,\vec{v} = \vec{b},$$

where M is the matrix containing the sampled spherical harmonics, $\vec{v}$ is the vector of unknown lighting environment coefficients, and $\vec{b}$ is the vector of intensities at p points. The least-squares solution to this system is

$$\vec{v} = \left(M^{\mathrm{T}} M\right)^{-1} M^{\mathrm{T}}\,\vec{b}\,. \tag{43}$$

This solution requires 3D surface normals from at least nine points on the surface of an object. Without multiple images or known geometry, however, this requirement may be difficult to satisfy from an arbitrary image.

As in [5, 12], we observe that under an assumption of orthographic projection, the z-component of the surface normal is zero along the occluding contour of an object. Therefore, the intensity profile along an occluding contour simplifies to

$$\begin{aligned} I\left(\vec{x}\right) = & A + l_{1,-1}\frac{2\pi}{3} Y_{1,-1}\left(\vec{N}\right) + l_{1,1}\frac{2\pi}{3} Y_{1,1}\left(\vec{N}\right) + l_{2,-2}\frac{\pi}{4} Y_{2,-2}\left(\vec{N}\right) \\ & + l_{2,2}\frac{\pi}{4} Y_{2,2}\left(\vec{N}\right), \end{aligned} \tag{44}$$

where

$$A = l_{0,0}\frac{\pi}{2\sqrt{\pi}} - l_{2,0}\frac{\pi}{16}\sqrt{\frac{5}{\pi}}. \tag{45}$$

Note that the functions $Y_{i,j}(\cdot)$ depend only on the x- and y-components of the surface normal $\vec{N}$. That is, the five lighting coefficients can be estimated from only 2D surface normals, which are relatively simple to estimate from a single image.[2]

[2] The 2D surface normal is the gradient vector of an implicit curve fit to the edge of an object.

In addition, Equation 44 is still linear in its now five lighting environment coefficients, which can be estimated as the least-squares solution to

$$\begin{pmatrix} 1 & \frac{2\pi}{3}Y_{1,-1}\left(\vec{N}\left(\vec{x}_1\right)\right) & \frac{2\pi}{3}Y_{1,1}\left(\vec{N}\left(\vec{x}_1\right)\right) & \frac{\pi}{4}Y_{2,-2}\left(\vec{N}\left(\vec{x}_1\right)\right) & \frac{\pi}{4}Y_{2,2}\left(\vec{N}\left(\vec{x}_1\right)\right) \\ 1 & \frac{2\pi}{3}Y_{1,-1}\left(\vec{N}\left(\vec{x}_2\right)\right) & \frac{2\pi}{3}Y_{1,1}\left(\vec{N}\left(\vec{x}_2\right)\right) & \frac{\pi}{4}Y_{2,-2}\left(\vec{N}\left(\vec{x}_2\right)\right) & \frac{\pi}{4}Y_{2,2}\left(\vec{N}\left(\vec{x}_2\right)\right) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \frac{2\pi}{3}Y_{1,-1}\left(\vec{N}\left(\vec{x}_p\right)\right) & \frac{2\pi}{3}Y_{1,1}\left(\vec{N}\left(\vec{x}_p\right)\right) & \frac{\pi}{4}Y_{2,-2}\left(\vec{N}\left(\vec{x}_p\right)\right) & \frac{\pi}{4}Y_{2,2}\left(\vec{N}\left(\vec{x}_p\right)\right) \end{pmatrix} \begin{pmatrix} A \\ l_{1,-1} \\ l_{1,1} \\ l_{2,-2} \\ l_{2,2} \end{pmatrix} = \begin{pmatrix} I\left(\vec{x}_1\right) \\ I\left(\vec{x}_2\right) \\ \vdots \\ I\left(\vec{x}_p\right) \end{pmatrix}, \tag{46}$$

$$M\,\vec{v} = \vec{b}, \tag{47}$$

which has the same least-squares solution as before:

$$\vec{v} = \left(M^{\mathrm{T}}M\right)^{-1}M^{\mathrm{T}}\,\vec{b}\,. \tag{48}$$

Note that this solution only provides five of the nine lighting environment coefficients. We will show, however, that this subset of coefficients is still sufficiently descriptive for forensic analysis.

When analyzing the occluding contours of objects in real images, it is often the case that the range of surface normals is limited, leading to an ill-conditioned matrix M. This limitation can arise from many sources, including occlusion or object geometry. As a result, small amounts of noise in either the surface normals or the measured intensities can cause large variations in the estimate of the lighting environment vector $\vec{v}$. To better condition the estimate, an error function $E\left(\vec{v}\right)$ is defined that combines the least-squares error of the original linear system with a regularization term:

$$E\left(\vec{v}\right) = \left|\left|M\,\vec{v} - \vec{b}\right|\right|^2 + \lambda\left|\left|C\,\vec{v}\right|\right|^2, \tag{49}$$

where λ is a scalar and the matrix C is diagonal with $(1\ 2\ 2\ 3\ 3)$ on the diagonal. The matrix C is designed to dampen the effects of higher order harmonics and is motivated by the observation that the average power of spherical harmonic coefficients for natural lighting environments decreases with increasing harmonic order [25].

For the full lighting model when 3D surface normals are available (Equation 49), the matrix C has $(1\ 2\ 2\ 2\ 3\ 3\ 3\ 3\ 3)$ on the diagonal.

The error function to be minimized (Equation 49) is a least-squares problem with a Tikhonov regularization [26]. The analytic minimum is found by differentiating with respect to $\vec{v}$:

$$\begin{aligned}\frac{\mathrm{d}E'(\vec{v})}{\mathrm{d}\,\vec{v}} &= 2M^{\mathrm{T}}M\vec{v} - 2M^{\mathrm{T}}\,\vec{b} + 2\lambda C^{\mathrm{T}}C\vec{v} \\ &= 2(M^{\mathrm{T}}M + \lambda C^{\mathrm{T}}C)\,\vec{v} - 2M^{\mathrm{T}}\vec{b},\end{aligned} \tag{50}$$

setting the result equal to zero, and solving for $\vec{v}$:

$$\vec{v} = \left(M^{\mathrm{T}}M + \lambda C^{\mathrm{T}}C\right)^{+} M^{\mathrm{T}}\vec{b}\,. \tag{51}$$

In practice, we have found that the conditioned estimate in Equation 51 is appropriate if less than 180° of surface normals are available along the occluding contour. If more than 180° of surface normals are available, the least-squares estimate (Equation 48) can be used, though both estimates will give similar results for small values of λ.

The estimated coefficient vector $\vec{v}$ (Equation 51) is a low-order approximation of the lighting environment. For forensics purposes, we would like to differentiate between lighting environments based on these coefficients. Intuitively, coefficients from objects in different lighting environments should be distinguishable, while coefficients from objects in the same lighting environment should be similar. In addition, measurable differences in sets of coefficients should be mostly due to differences in the lighting environment and not to other factors such as object color or image exposure. Taking these issues into consideration, we propose an error measure between two estimated lighting environments. Let $\vec{v}_1$ and $\vec{v}_2$ be two vectors of lighting environment coefficients. From these coefficients, the irradiance profile along a circle (2D) or a sphere (3D) is synthesized, from which the error is computed. The irradiance profiles corresponding to $\vec{v}_1$ and $\vec{v}_2$ are given by

$$\vec{x}_1 = M\vec{v}_1, \tag{52}$$

$$\vec{x}_2 = M\vec{v}_2, \tag{53}$$

where the matrix M is of the form in Equation 42 (for 3D normals) or Equation 46 (for 2D normals). After subtracting the mean, the correlation between these zero-mean profiles is

$$\mathrm{corr}\left(\vec{x}_1, \vec{x}_2\right) = \frac{\vec{x}_1^{\,\mathrm{T}}\vec{x}_2}{\|\vec{x}_1\|\|\vec{x}_2\|}. \tag{54}$$

In practice, this correlation can be computed directly from the lighting environment coefficients:

$$\mathrm{corr}\left(\vec{v}_1, \vec{v}_2\right) = \frac{\vec{v}_1^{\mathrm{T}} Q \vec{v}_2}{\sqrt{\vec{v}_1^{\mathrm{T}} Q \vec{v}_1}\sqrt{\vec{v}_2^{\mathrm{T}} Q \vec{v}_2}}, \tag{55}$$

where the matrix Q for both the 2D and 3D cases is derived in [7]. By design, this correlation is invariant to both additive and multiplicative factors on the irradiance profiles $\vec{x}_1$ and $\vec{x}_2$. Recall that our coefficient vectors $\vec{v}_1$ and $\vec{v}_2$ are estimated to within an unknown multiplicative factor. In addition, different exposure times under a nonlinear camera response function can introduce an additive bias. The correlation is, therefore, invariant to these factors and produces values in the interval $[-1, 1]$. The final error is then given by

$$D\left(\vec{v}_1, \vec{v}_2\right) = \frac{1}{2}\left(1 - \mathrm{corr}\left(\vec{v}_1, \vec{v}_2\right)\right), \tag{56}$$

with values in the range [0, 1].

To test our ability to discriminate between lighting environments we photographed a diffuse sphere in 28 different locations with a 6.3-megapixel Nikon D-100 digital camera set to capture in high-quality JPEG mode. The focal length was set to 70 mm, the *f*-stop was fixed at *f*/8, and the shutter speed was varied to capture two or three exposures per location. In total, there were 68 images. For each image, the Adobe Photoshop ''Quick Selection Tool'' was used to locate the occluding contour of the sphere from which both 2D and 3D surface normals could be estimated. The 3D surface normals were used to estimate the full set of nine lighting environment coefficients and the 2D surface normals along the occluding contour were used to estimate five coefficients. For both cases, the regularization term λ in Equation 51 was set to 0.01. For each pair of images, the error (Equation 56) between the estimated coefficients was computed. In total, there were 2278 image pairs: 52 pairs were different exposures from the same location and 2226 pairs were captured in different locations. The errors for all pairs for both models (3D and 2D) are shown in Fig. 9. In both plots, the 52 image pairs from the same location are plotted first (''+''), sorted by error. The 2226 pairs from different locations are plotted next (''•''). Note that the axes are scaled logarithmically. For the 3D case, the minimum error between an image pair from different locations is 0.0027 and the maximum error between an image pair from the same location is 0.0023. Therefore, the two sets of data, same location versus different location, are separated by a threshold of 0.0025. For the 2D case, 13 image pairs (0.6%) fell below 0.0025. These image pairs correspond to lighting environments that are indistinguishable based on the five-coefficient model.

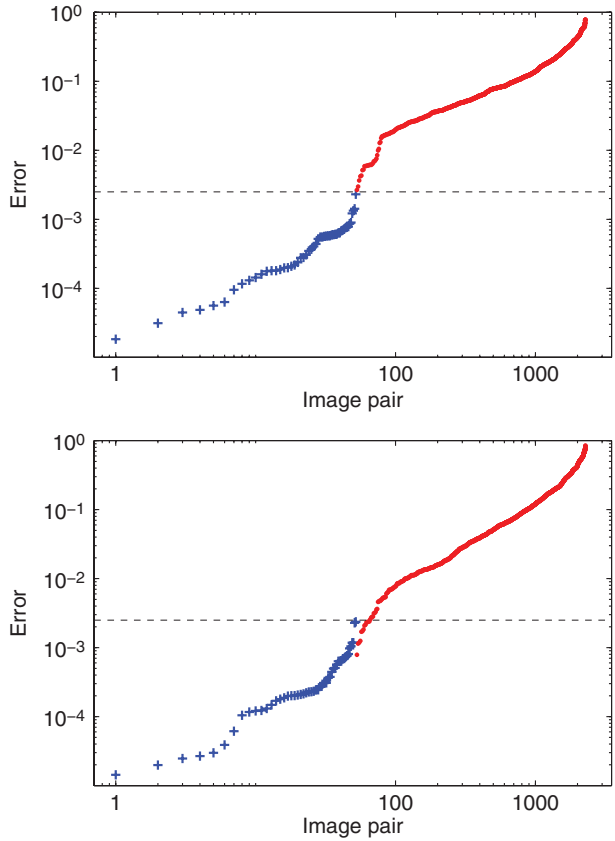

FIG. 9. Errors between image pairs corresponding to the same (''+'') and different (''•'') locations using the full nine-parameter model with 3D surface normals (top) and using the five-parameter model with 2D surface normals (bottom). Both the horizontal and vertical axes are scaled logarithmically.

To be useful in a forensic setting, lighting estimates from objects in the same lighting environment should be robust to differences in color and material type, as well as to geometric differences, since arbitrary objects may not have the full range of surface normals available. To test our algorithm under these conditions, we downloaded 20 images of multiple objects in natural lighting environments from Flickr[3] (Fig. 10). In each image, occluding contours of two to four objects were

[3] http://www.flickr.com

FIG. 10. Superimposed on each image are the contours from which the surface normals and intensity values are extracted to form the matrix M and the corresponding vector $\vec{b}$ (Equation 47).

specified using a semiautomated approach. A coarse contour was defined by painting along the edge of the object using Adobe Photoshop. Each stroke was then automatically divided into quadratic segments, or regions, which were fit to nearby points with large gradients. The analyzed regions for all images are shown in Fig. 10. Analytic surface normals and intensities along the occluding contour were measured from the regions. With the 2D surface normals and intensities, the five lighting environment coefficients were estimated (Equation 51). The regularization term λ in Equation 51 was increased to 0.1, which is larger than in the simulation due to an increasing sensitivity to noise.

Across all 20 images, there were 49 pairs of objects from the same image and 1329 pairs of objects from different images. For each pair of objects, the error between the estimated coefficients was computed. For objects in the same image, the average error was 0.009 with a standard deviation of 0.007 and a maximum error of 0.027. For comparison, between objects in different images the average error was 0.295 with a standard deviation of 0.273. There were, however, 196 pairs of objects (15%) from different images that fell below 0.027. The lighting environments in these images (e.g., the two police images, the trees and skiers images, etc.) were indistinguishable using the five coefficient model. For objects from the same image, the pair with the maximum error of 0.027 is the basketball and basketball player. The sweaty skin of the basketball player is somewhat shiny, a violation of the Lambertian assumption. In addition, the shoulders and arms of the basketball player provide only a limited extent of surface normals, making the linear system somewhat ill conditioned. In contrast, the objects from the same image with the minimum error of 0.0001 are the left and right pumpkins on the bench. Both pumpkins provide a large extent of surface normals, over 200°, and the surfaces are fairly diffuse. Since the surfaces fit the assumptions and the linear systems are well conditioned, the error between the estimated coefficients is small.

We created three forgeries by mixing and matching several of the images in Fig. 10. These forgeries are shown in Fig. 11. Regions along the occluding contour of two to four objects in each image were selected for analysis. These regions are superimposed on the images in the right column of Fig. 11. Surface normals and intensities along these occluding contour were extracted, from which the five lighting environment coefficients were estimated (Equation 51) with the regularization term $\lambda = 0.1$. Shown in each panel is a sphere rendered with the estimated coefficients. These spheres qualitatively show discrepancies between the lighting. For all pairs of objects originally in the same lighting environment, the average error is 0.005 with maximum error of 0.01. For pairs of objects from different lighting environments, the average error is 0.15 with a minimum error of 0.03.

The ability to estimate complex lighting environments was motivated by our earlier work in which we showed how to detect inconsistencies in the direction to

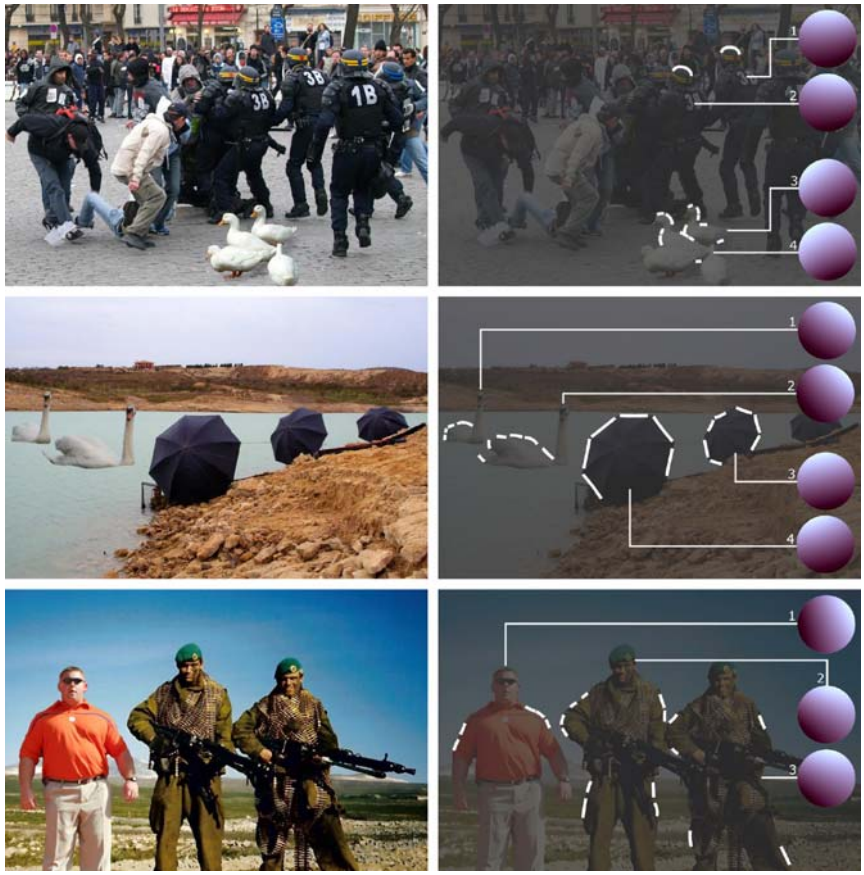

FIG. 11. Shown on the left are three forgeries: the ducks, swans, and football coach were each added into their respective images. Shown on the right are the analyzed regions superimposed in white, and spheres rendered from the estimated lighting coefficients.

an illuminating light source (Section 2.1). The work described here generalizes this approach by allowing us to estimate more complex models of lighting and in fact can be adapted to estimate the direction to a single light source. Specifically, by considering only the two first-order spherical harmonics, $Y_{1,-1}(\cdot)$ and $Y_{1,1}(\cdot)$, the direction to a light source can be estimated as $\tan^{-1}\left(l_{1,-1}/l_{1,1}\right)$.

When creating a composite of two or more people, it is often difficult to exactly match the lighting, even if the lighting seems perceptually consistent. The reason for this is that complex lighting environments (multiple light sources, diffuse lighting,

directional lighting) give rise to complex and subtle lighting gradients and shading effects in the image. Under certain simplifying assumptions (distant light sources and diffuse surfaces), arbitrary lighting environments can be modeled with a nine-dimensional model. This model approximates the lighting with a linear combination of spherical harmonics. We have shown how to approximate a simplified five-dimensional version of this model from a single image, and how to stabilize the model estimation in the presence of noise. Inconsistencies in the lighting model across an image are then used as evidence of tampering.

2.4 Color Filter Array

The previous three sections described forensic analysis based on detecting inconsistencies in lighting. These tools are particularly useful in determining if a photograph was created by compositing several photographs together. Next, we describe a complementary forensic analysis that can determine if any part of an image was manipulated or changed (e.g., airbrushing) from the time of its original recording.

Most digital cameras capture color images using a single sensor in conjunction with an array of color filters. As a result, only one third of the samples in a color image are captured by the camera, the other two thirds being interpolated. This interpolation introduces specific correlations between the samples of a color image. When creating a digital forgery these correlations may be destroyed or altered. We describe the form of these correlations, and propose a method that quantifies and detects them in any portion of an image.

A digital color image consists of three channels containing samples from different bands of the color spectrum, for example, red, green, and blue. Most digital cameras, however, are equipped with a single CCD or CMOS sensor, and capture color images using a color filter array (CFA). The most frequently used CFA, the Bayer array [27], employs three color filters: red, green, and blue. The red and blue pixels are sampled on rectilinear lattices, while the green pixels are sampled on a quincunx lattice (Fig. 12). Since only a single color sample is recorded at each pixel location, the other two color samples must be estimated from the neighboring samples to obtain a three-channel color image. Let $S(x, y)$ denote the CFA image in Fig. 12, and $\tilde{R}(x, y)$, $\tilde{G}(x, y)$, and $\tilde{B}(x, y)$ denote the red, green, and blue channels constructed from $S(x, y)$ as follows:

$$\tilde{R}(x,y) = \begin{cases} S(x,y) & \text{if } S(x,y) = r_{x,y} \\ 0 & \text{otherwise,} \end{cases} \tag{57}$$

$$\tilde{G}(x,y) = \begin{cases} S(x,y) & \text{if } S(x,y) = g_{x,y} \\ 0 & \text{otherwise,} \end{cases} \tag{58}$$

$$\tilde{B}(x,y) = \begin{cases} S(x,y) & \text{if } B(x,y) = b_{x,y} \\ 0 & \text{otherwise,} \end{cases} \tag{59}$$

$r_{1,1}$	$g_{1,2}$	$r_{1,3}$	$g_{1,4}$	$r_{1,5}$	$g_{1,6}$	
$g_{2,1}$	$b_{2,2}$	$g_{2,3}$	$b_{2,4}$	$g_{2,5}$	$b_{2,6}$	
$r_{3,1}$	$g_{3,2}$	$r_{3,3}$	$g_{3,4}$	$r_{3,5}$	$g_{3,6}$	
$g_{4,1}$	$b_{4,2}$	$g_{4,3}$	$b_{4,4}$	$g_{4,5}$	$b_{4,6}$	$\cdots$
$r_{5,1}$	$g_{5,2}$	$r_{5,3}$	$g_{5,4}$	$r_{5,5}$	$g_{5,6}$	
$g_{6,1}$	$b_{6,2}$	$g_{6,3}$	$b_{6,4}$	$g_{6,5}$	$b_{6,6}$	
			$\vdots$			$\ddots$

FIG. 12. The top left portion of a CFA image obtained from a Bayer array. The red, $r_{2i+1,2j+1}$, and blue, $b_{2i,2j}$, pixels are sampled on rectilinear lattices, while the green, $g_{2i+1,2j}$ and $g_{2i,2j+1}$, pixels are sampled twice as often on a quincunx lattice. Notice that at each pixel location only a single color sample is recorded.

where (x, y) span an integer lattice. A complete color image, with channels $R(x, y)$, $G(x, y)$, and $B(x, y)$ needs to be estimated. These channels take on the nonzero values of $\tilde{R}(x, y)$, $\tilde{G}(x, y)$, and $\tilde{B}(x, y)$, and replace the zeros with estimates from neighboring samples.

The estimation of the missing color samples is referred to as CFA interpolation or demosaicking. CFA interpolation has been extensively studied and many methods have been proposed (see, e.g., [28] for a survey and [29–31] for more recent methods).

The simplest methods for demosaicking are kernel-based interpolation methods that act on each channel independently. These methods can be efficiently implemented as linear filtering operations on each color channel:

$$R(x,y) = \sum_{u,v=-N}^{N} h_{\mathrm{r}}(u,v)\tilde{R}(x-u,y-v), \tag{60}$$

$$G(x,y) = \sum_{u,v=-N}^{N} h_{\mathrm{g}}(u,v)\tilde{G}(x-u,y-v), \tag{61}$$

$$B(x,y) = \sum_{u,v=-N}^{N} h_{\mathrm{b}}(u,v)\tilde{B}(x-u,y-v), \tag{62}$$

where $\tilde{R}(\cdot)$, $\tilde{G}(\cdot)$, $\tilde{B}(\cdot)$ are defined in Equations 57–59, and $h_{\mathrm{r}}(\cdot)$, $h_{\mathrm{g}}(\cdot)$, $h_{\mathrm{b}}(\cdot)$ are linear filters of size $(2N+1) \times (2N+1)$. Different forms of interpolation (nearest neighbor, bilinear, bicubic [32], etc.) differ in the form of the interpolation filter used. For the Bayer array, the bilinear and bicubic filters for the red and blue channels are separable. The 1D filters are given by

$$h_{\mathrm{l}} = [1/2 \quad 1 \quad 1/2],$$
$$h_{\mathrm{c}} = [-1/16 \quad 0 \quad 9/16 \quad 1 \quad 9/16 \quad 0 \quad -1/16].$$

There are many other CFA interpolation algorithms including smooth hue transition [33], median filter [34], gradient-based [35], adaptive color plane [36], and threshold-based variable number of gradients [37]. Regardless of their specific implementations, each CFA interpolation algorithm introduces specific statistical correlations between a subset of pixels in each color channel. Since the color filters in a CFA are typically arranged in a periodic pattern, these correlations are periodic. Consider, for example, the red channel, $R(x, y)$, that has been sampled on a Bayer array (Fig. 12), then CFA interpolated using bilinear interpolation. In this case, the red samples in the odd rows and even columns are the average of their closest horizontal neighbors, the red samples in the even rows and odd columns are the average of their closest vertical neighbors, and the red samples in the even rows and columns are the average of their closest diagonal neighbors:

$$R(2x+1, 2y) = \frac{R(2x+1, 2y-1)}{2} + \frac{R(2x+1, 2y+1)}{2},$$
$$R(2x, 2y+1) = \frac{R(2x-1, 2y+1)}{2} + \frac{R(2x+1, 2y+1)}{2},$$
$$R(2x, 2y) = \frac{R(2x-1, 2y-1)}{4} + \frac{R(2x-1, 2y+1)}{4} + \frac{R(2x+1, 2y-1)}{4} + \frac{R(2x+1, 2y+1)}{4}.$$

Note that in this simple case, the estimated samples are perfectly correlated to their neighbors. As such, a CFA-interpolated image can be detected (in the absence of noise) by noticing, for example, that every other sample in every other row or column is perfectly correlated to its neighbors. At the same time, the noninterpolated samples are less likely to be correlated in precisely the same manner. Furthermore, it is likely that tampering will destroy these correlations, or that the splicing together of two images from different cameras will create inconsistent correlations across the composite image. As such, the presence or lack of correlations produced by CFA interpolation can be used to authenticate an image, or expose it as a forgery.

We begin by assuming a simple linear model for the periodic correlations introduced by CFA interpolation. That is, each interpolated pixel is correlated to a weighted sum of pixels in a small neighborhood centered about itself. While perhaps overly simplistic when compared to the highly nonlinear nature of most CFA interpolation algorithms, this simple model is both easy to parametrize and can

reasonably approximate each of the CFA interpolation algorithms described above. Note that most CFA algorithms estimate a missing color sample from neighboring samples in all three color channels. For simplicity, however, we ignore these interchannel correlations and treat each color channel independently.

If the specific form of the correlations is known (i.e., the parameters of the linear model), then it would be straightforward to determine which samples are correlated to their neighbors. On the other hand, if it was known which samples are correlated to their neighbors, the specific form of the correlations could be easily determined. In practice, of course, neither are known. To simultaneously estimate both we employ the expectation/maximization (EM) algorithm [38], as described below.

Let $f(x, y)$ denote a color channel (red, green, or blue) of a CFA-interpolated image. We begin by assuming that each sample in $f(x, y)$ belongs to one of two models (1) M_1 if the sample is linearly correlated to its neighbors, satisfying:

$$f(x,y) = \sum_{u,v=-N}^{N} \alpha_{u,v} f(x+u, y+v) + n(x,y), \tag{63}$$

where the model parameters are given by the linear coefficients $\vec{\alpha} = \{\alpha_{u,v} | -N \le u, v \le N\}$ (N is an integer and $\alpha_{0,0} = 0$), and $n(x, y)$ denotes independent and identically distributed samples drawn from a Gaussian distribution with zero mean and unknown variance σ^2; or (2) M_2 if the sample is not correlated to its neighbors, that is, is generated by an ''outlier process.''

The expectation–maximization algorithm (EM) is a two-step iterative algorithm (1) in the E-step the probability of each sample belonging to each model is estimated and (2) in the M-step the specific form of the correlations between samples is estimated. More specifically, in the E-step, the probability of each sample of $f(x, y)$ belonging to model M_1 is estimated using Bayes' rule:

$$\Pr\{f(x,y) \in M_1 | f(x,y)\} = \frac{\Pr\{f(x,y) | f(x,y) \in M_1\} \Pr\{f(x,y) \in M_1\}}{\sum_{i=1}^{2} \Pr\{f(x,y) | f(x,y) \in M_i\} \Pr\{f(x,y) \in M_i\}}, \tag{64}$$

where the prior probabilities $\Pr\{f(x,y) \in M_1\}$ and $\Pr\{f(x,y) \in M_2\}$ are assumed to be equal to 1/2. The probability of observing a sample $f(x, y)$ knowing it was generated from model M_1 is given by

$$\Pr\{f(x,y) | f(x,y) \in M_1\} = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[- \frac{\left(f(x,y) - \sum_{u,v=-N}^{N} \alpha_{u,v} f(x+u, y+v) \right)^2}{2\sigma^2} \right]. \tag{65}$$

The variance, σ^2, of this Gaussian distribution is estimated in the M-step. A uniform distribution is assumed for the probability of observing a sample generated by the outlier model, M_2, that is, $\Pr\{f(x,y)|f(x,y) \in M_2\}$ is equal to the inverse of the range of possible values of $f(x, y)$. Note that the E-step requires an estimate of the coefficients $\vec{\alpha}$, which on the first iteration is chosen randomly. In the M-step, a new estimate of $\vec{\alpha}$ is computed using weighted least squares, by minimizing the following quadratic error function:

$$E\left(\vec{\alpha}\right) = \sum_{x,y} w(x,y) \cdot \left(f(x,y) - \sum_{u,v=-N}^{N} \alpha_{u,v} f(x+u, y+v) \right)^2, \tag{66}$$

where the weights $w(x,y) \equiv \Pr\{f(x,y) \in M_1|f(x,y)\}$ (Equation 64). This error function is minimized by computing the gradient with respect to $\vec{\alpha}$, setting this gradient equal to zero, and solving the resulting linear system of equations. Setting equal to zero the partial derivative with respect to one of the coefficients, $\alpha_{s,t}$, yields

$$\frac{\partial E}{\partial \alpha_{s,t}} = 0, \tag{67}$$

$$\sum_{x,y} w(x,y) f(x+s, y+t) \sum_{u,v=-N}^{N} \alpha_{u,v} f(x+u, y+v) = \sum_{x,y} w(x,y) f(x+s, y+t) f(x,y). \tag{68}$$

Reordering the terms on the left-hand side yields:

$$\sum_{u,v=-N}^{N} \alpha_{u,v} \left(\sum_{x,y} w(x,y) f(x+s, y+t) f(x+u, y+v) \right) = \sum_{x,y} w(x,y) f(x+s, y+t) f(x,y). \tag{69}$$

This process is repeated for each component, $\alpha_{s,t}$, of $\vec{\alpha}$, to yield a system of linear equations that can be solved using standard techniques. The E-step and the M-step are iteratively executed until a stable estimate of $\vec{\alpha}$ is achieved. The final $\vec{\alpha}$ has the property that it maximizes the likelihood of the observed samples.

We collected 100 images, 50 of resolution 512 × 512, and 50 of resolution 1024 × 1024. Each of these images were cropped from a smaller set of twenty 1600 × 1200 images taken with a Nikon Coolpix 950 camera, and twenty 3034 × 2024 images taken with a Nikon D100 camera. The Nikon Coolpix 950 employs a four-filter (yellow, cyan, magenta, green) CFA, and was set to store the images in

uncompressed TIFF format. The Nikon D100 camera employs a Bayer array, and was set to store the images in RAW format. To avoid interference with the CFA interpolation of the cameras, each color channel of these images was independently blurred with a 3×3 binomial filter, and downsampled by a factor of 2 in each direction. These downsampled color images, of size 256×256 or 512×512, were then resampled onto a Bayer array, and CFA interpolated using the algorithms described above.

Shown in Figs. 13 and 14 are the results of running the EM algorithm on eight 256×256 color images that were CFA interpolated. The parameters of the EM algorithm were: $N = 1$, $\sigma_0 = 0.0075$, and $p_0 = 1/256$. Shown in the left column are the images, in the middle column the estimated probability maps from the green channel (the red and blue channels yield similar results), and in the right column the magnitude of the Fourier transforms of the probability maps.[4] Note that, although the periodic patterns in the probability maps are not visible at this reduced scale, they are particularly salient in the Fourier domain in the form of localized peaks. Note also that these peaks do not appear in the images that are not CFA interpolated (last row of Fig. 13 and 14).

Since it is likely that tampering will destroy the periodicity of the CFA correlations, it may be possible to detect and localize tampering in any portion of an image. To illustrate this, consider the leftmost image in Fig. 15, taken with a Nikon D100 digital camera and saved in RAW format. Each color channel of this image (initially interpolated using the adaptive color plane technique) was blurred with a 3×3 binomial filter and downsampled by a factor of 2 in order to destroy the CFA periodic correlations. The 512×512 downsampled image was then resampled on a Bayer array and CFA interpolated. Next, composite images, 512×512 pixels in size, were created by splicing, in varying proportions (1/4, 1/2, and 3/4), the non-CFA-interpolated image and the same image CFA interpolated with the bicubic algorithm. Shown in Fig. 15 are the probability maps obtained from running EM on the red channel of the composite images. Notice that these probability maps clearly reveal the presence of two distinct regions. Shown below each probability map are the magnitudes of the Fourier transforms of two windows, one from the non-CFA-interpolated portion (right), and one from the CFA-interpolated portion (left).

[4] For display purposes, the probability maps were upsampled by a factor of 2 before Fourier transforming. The periodic patterns introduced by CFA interpolation have energy in the highest horizontal, vertical, and diagonal frequencies, which corresponds to localized frequency peaks adjacent to the image edges. Upsampling by a factor of 2 shrinks the support of a probability map's spectrum, and shifts these peaks into the midfrequencies, where they are easier to see. Also for display purposes, the Fourier transforms of the probability maps were high-pass filtered, blurred, scaled to fill the range [0, 1], and gamma corrected with an exponent of 2.0.

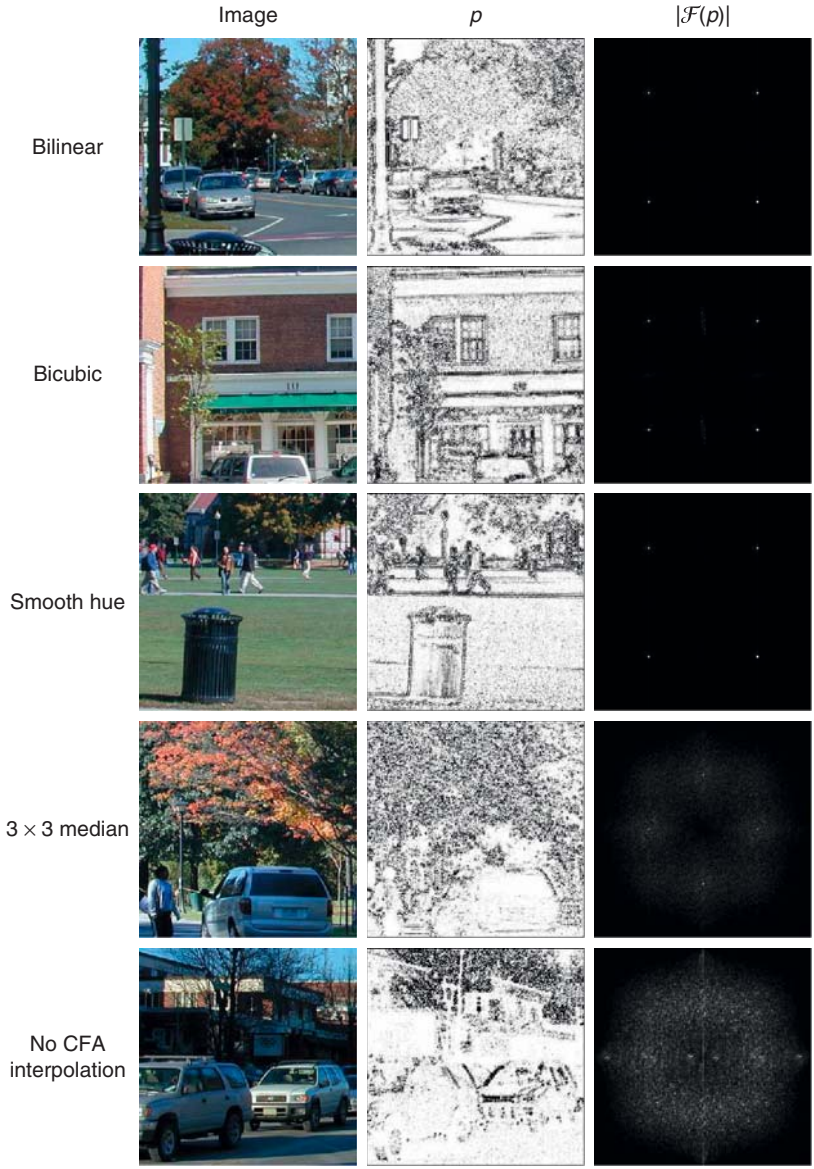

FIG. 13. Shown in each row is an image interpolated with the specified algorithm, the probability map of the green channel as output by the EM algorithm, and the magnitude of the Fourier transform of the probability map. Note the peaks in $|\mathcal{F}(p)|$ corresponding to periodic correlations in the CFA-interpolated images, and the lack of such peaks in the non-CFA-interpolated image (last row).

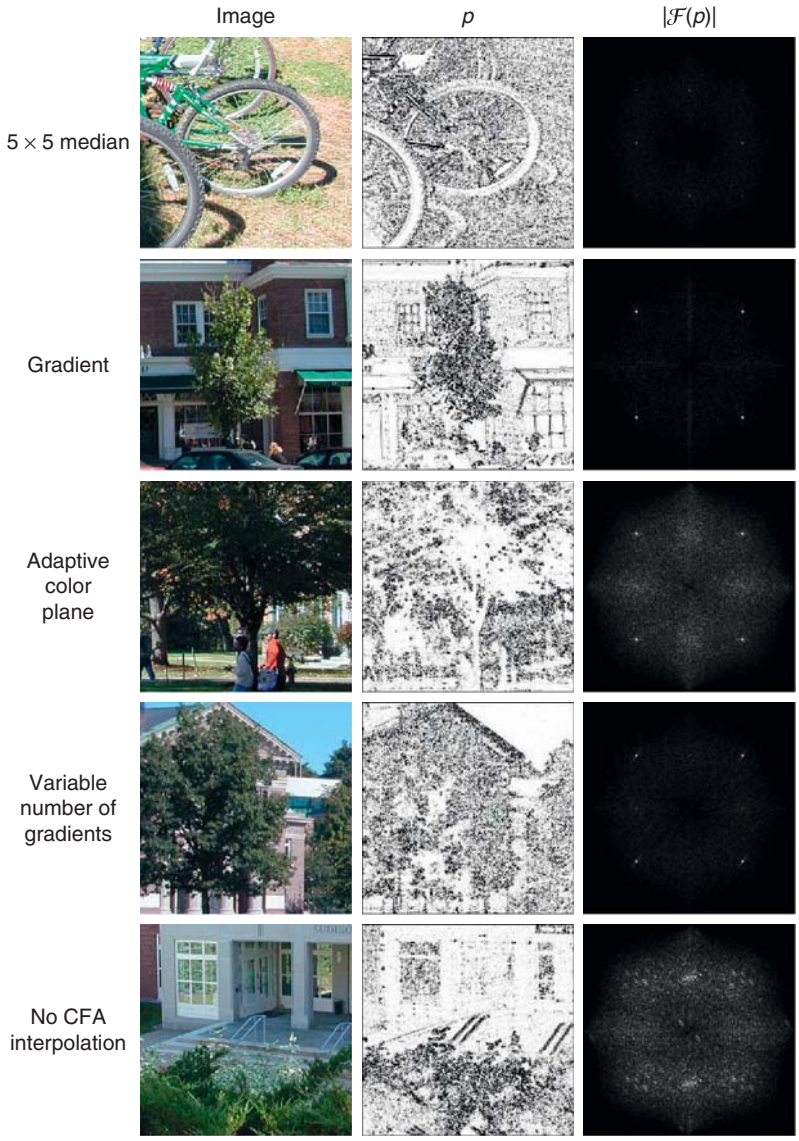

FIG. 14. Shown in each row is an image interpolated with the specified algorithm, the probability map of the green channel as output by the EM algorithm, and the magnitude of the Fourier transform of the probability map. Note the peaks in $|\mathcal{F}(p)|$ corresponding to periodic correlations in the CFA-interpolated images, and the lack of such peaks in the non-CFA-interpolated image (last row).

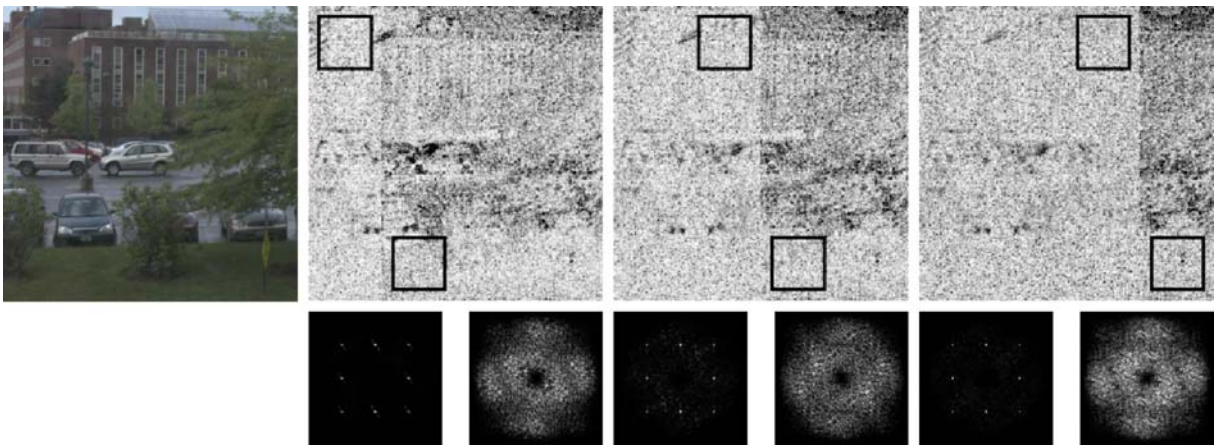

FIG. 15. Shown are an image, and the probability maps of composite images obtained by splicing together the CFA-interpolated image (left portion of each image) and the same image without CFA interpolation (right portion). Also shown are the magnitudes of the Fourier transforms of windows from the two regions. Note that windows from the CFA-interpolated regions (left) have localized peaks in the Fourier domain, while the windows from the ''tampered'' regions (right) do not.

Notice the presence of localized frequency peaks in the CFA-interpolated portion, and the lack of such peaks in the non-CFA-interpolated portion.

Shown in Fig. 16 is a perceptually plausible forgery created using Adobe Photoshop (top row, right). The tampering consisted of hiding the damage on the car hood using airbrushing, smudging, blurring, and duplication. Also shown is the original image CFA interpolated using bicubic interpolation (top row, left), the estimated probability map of the tampered image (middle row), and the magnitude of the Fourier transforms of two windows one from a tampered portion (bottom row, left), and one from an unadulterated portion (bottom row, right). Note that even though the periodic pattern is not visible in the probability map, localized frequency peaks reveal its presence. Note also that the window from the tampered region does not contain frequency peaks.

The benefit of this approach is that nearly any manipulation to the image can be detected. The drawback is that an original resolution version of the image is required for authentication. And, as with any authentication scheme, our forensic analysis is vulnerable to counterattack. A tampered image could, for example, be resampled onto a CFA, and then reinterpolated. This attack, however, requires knowledge of the camera's CFA pattern and interpolation algorithm, and may be beyond the reaches of a novice forger.

2.5 JPEG Ghosts

Although highly effective in some situations, the techniques described above are not applicable when analyzing low-quality images. Detecting tampering in low-quality images is particularly challenging since low-quality images often destroy many artifacts that could be used to detect tampering. One such technique is described next that detects tampering which results when part of a JPEG image is inserted into another higher quality JPEG image. For example, when one person's head is spliced onto another person's body, or when two separately photographed people are combined into a single composite. This approach works by explicitly determining if part of an image was originally compressed at a lower quality relative to the rest of the image.

In the standard JPEG compression scheme [39, 40], a color image (RGB) is first converted into luminance/chrominance space (YCbCr). The two chrominance channels (CbCr) are typically subsampled by a factor of 2 relative to the luminance channel (Y). Each channel is then partitioned into 8×8 pixel blocks. These values are converted from unsigned to signed integers (e.g., from [0, 255] to [−128, 127]). Each block is converted to frequency space using a 2D discrete cosine transform (DCT). Each DCT coefficient, c, is then quantized by an amount q:

$$\hat{c} = \text{round}(c/q), \tag{70}$$

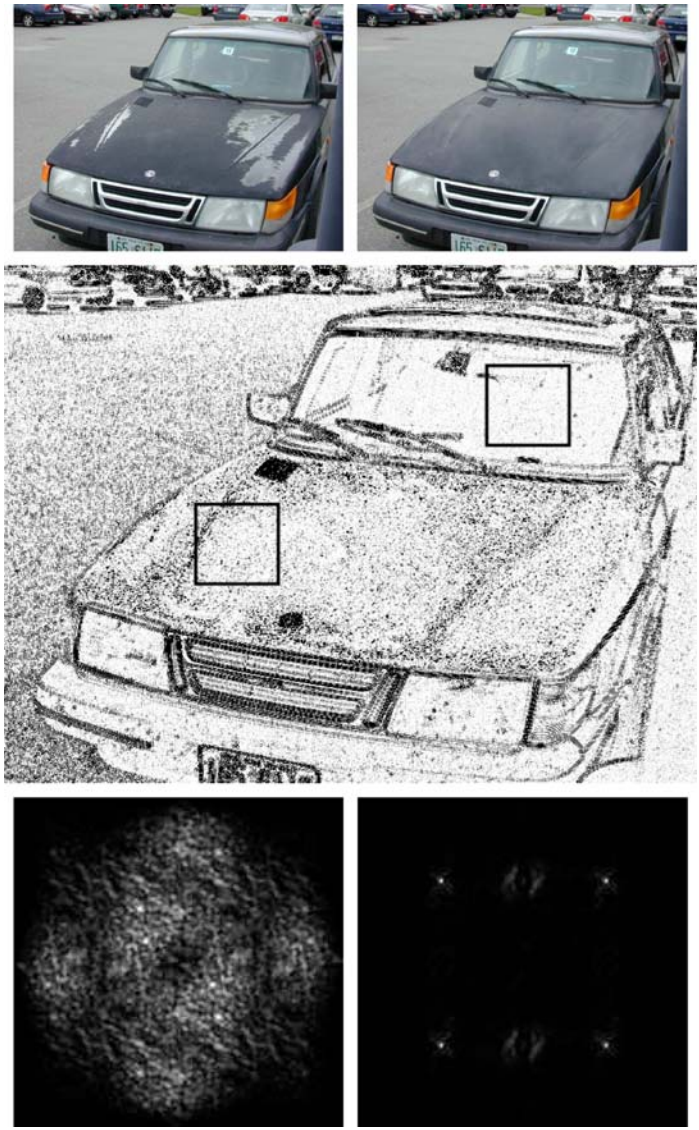

FIG. 16. Shown are: (top) an original and a tampered image, (middle) the probability map of the tampered image's green channel, and (bottom) the magnitude of the Fourier transform of two windows from the probability map. The windows correspond to a tampered and an unadulterated portion of the forgery. Note the lack of peaks in the tampered region signifying the absence of CFA interpolation.

where the quantization q depends on the spatial frequency and channel. Larger quantization values q yield better compression at the cost of image degradation. Quantization values are typically larger in the chrominance channels, and in the higher spatial frequencies, roughly modeling the sensitivity of the human visual system.

Consider now a set of coefficients c_1 quantized by an amount q_1, which are subsequently quantized a second time by an amount q_2 to yield coefficients c_2. With the exception of $q_2 = 1$ (i.e., no quantization), the difference between c_1 and c_2 will be minimal when $q_2 = q_1$. It is obvious that the difference between c_1 and c_2 increases for quantization value $q_2 > q_1$ since the coefficients become increasingly more sparse as q_2 increases. For values of $q_2 < q_1$, the difference between c_1 and c_2 also increases because although the second quantization does not affect the granularity of the coefficients, it does cause a shift in their values. Shown in Fig. 17A, for example, is the sum of squared differences between c_1 and c_2 as a function of the second quantization q_2, where $q_1 = 17$, and where the coefficients c_1 are drawn from a normal zero-mean distribution. Note that this difference increases as a function of increasing q_2, with the exception of $q_2 = q_1$, where the difference is minimal. If q_1 is not prime, unlike the above example, then multiple minima may appear at quality values q_2 that are integer multiples of q_1. As will be seen below, this issue can be circumvented by averaging over all of the JPEG DCT coefficients.

Consider next a set of coefficients c_0 quantized by an amount q_0, followed by quantization by an amount $q_1 < q_0$ to yield c_1. Further quantizing c_1 by q_2 yields the coefficients c_2. As before, the difference between c_1 and c_2 will be minimal when $q_2 = q_1$. But, since the coefficients were initially quantized by q_0, where $q_0 > q_1$, we expect to find a second minimum when $q_2 = q_0$. Shown in Fig. 17B is the sum of squared differences between c_1 and c_2, as a function of q_2, where $q_0 = 23$ and $q_1 = 17$. As before, this difference increases as a function of increasing q_2, reaches a minimum at $q_2 = q_1 = 17$, and most interestingly has a second local minimum at $q_2 = q_0 = 23$. We refer to this second minimum as a JPEG ghost, as it reveals that the coefficients were previously quantized (compressed) with a larger quantization (lower quality).

Recall that the JPEG compression scheme separately quantizes each spatial frequency within a 8×8 pixel block. One approach to detecting JPEG ghosts would be to separately consider each spatial frequency in each of the three luminance/color channels. However, recall that multiple minima are possible when comparing integer multiple quantization values. If, on the other hand, we consider the cumulative effect of quantization on the underlying pixel values, then this issue is far less likely to arise (unless all 192 quantization values at different JPEG qualities are integer multiples of one another—an unlikely scenario). Therefore,

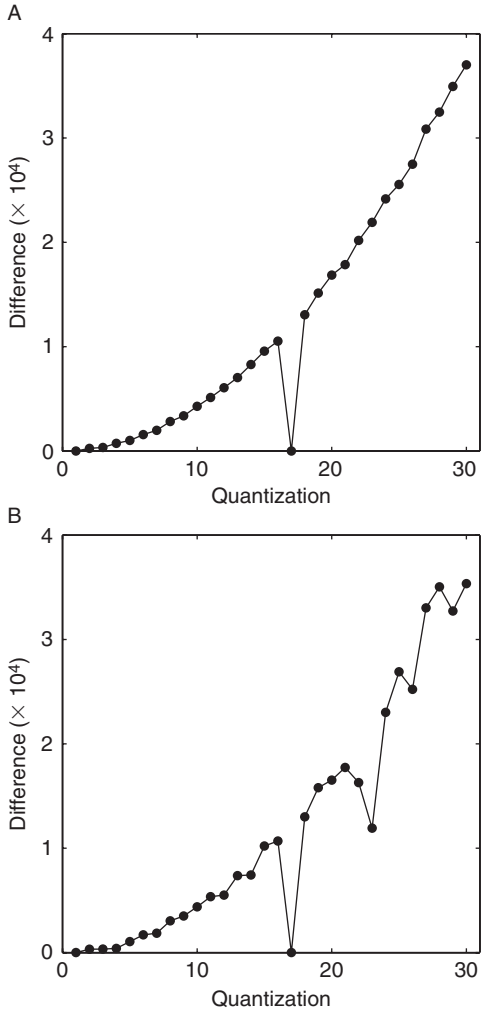

FIG. 17. Shown in panel (A) is the sum of squared differences between coefficients quantized by an amount $q_1 = 17$, followed by a second quantization in the range $q_2 \in [1, 30]$ (horizontal axis)—this difference reaches a minimum at $q_2 = q_1 = 17$. Shown in panel (B) is the sum of squared differences between coefficients quantized initially by an amount $q_0 = 23$ followed by $q_1 = 17$, followed by quantization in the range $q_2 \in [1, 30]$ (horizontal axis)—this difference reaches a minimum at $q_2 = q_1 = 17$ and a local minimum at $q_2 = q_0 = 23$, revealing the original quantization.

instead of computing the difference between the quantized DCT coefficients, we consider the difference computed directly from the pixel values, as follows:

$$d(x,y,q) = \frac{1}{3}\sum_{i=1}^{3}\left[f(x,y,i) - f_q(x,y,i)\right]^2, \tag{71}$$

where $f(x, y, i)$, $i = 1, 2, 3$, denotes each of three RGB color channels,[5] and $f_q(\cdot)$ is the result of compressing $f(\cdot)$ at quality q.

Shown in the top left panel of Fig. 18 is an image whose central 200 × 200 pixel region was extracted, compressed at a JPEG quality of 65/100, and reinserted into the image whose original quality was 85. Shown in each subsequent panel is the sum of squared differences (Equation 71) between this manipulated image, and a resaved version compressed at different JPEG qualities. Note that the central region is clearly visible when the image is resaved at the quality of the tampered region (65). Also note that the overall error reaches a minimum at the saved quality of 85. There are some variations in the difference images within and outside of the tampered region which could possibly confound a forensic analysis. These fluctuations are due to the underlying image content. Specifically, because the image difference is computed across all spatial frequencies, a region with small amounts of high spatial frequency content (e.g., a mostly uniform sky) will have a lower difference as compared to a highly textured region (e.g., grass). To compensate for these differences, we consider a spatially averaged and normalized difference measure. The difference image is first averaged across a $b \times b$ pixel region:

$$\delta(x,y,q) = \frac{1}{3}\sum_{i=1}^{3}\frac{1}{b^2}\sum_{b_x=0}^{b-1}\sum_{b_y=0}^{b-1}\left[f\left(x+b_x, y+b_y, i\right) - f_q\left(x+b_x, y+b_y, i\right)\right]^2, \tag{72}$$

and then normalized so that the averaged difference at each location (x, y) is scaled into the range [0, 1]:

$$d(x,y,q) = \delta(x,y,q) - \frac{\min_q[\delta(x,y,q)]}{\max_q[\delta(x,y,q)] - \min_q[\delta(x,y,q)]}. \tag{73}$$

Although the JPEG ghosts are often visually highly salient, it is still useful to quantify if a specified region is statistically distinct from the rest of the image. To this end, the two-sample Kolmogorov–Smirnov (K–S) statistic [41] is employed

[5] The detection of JPEG ghosts is easily adapted to grayscale images by simply computing $d(x, y, q)$ (Equation 71) over a single image channel.

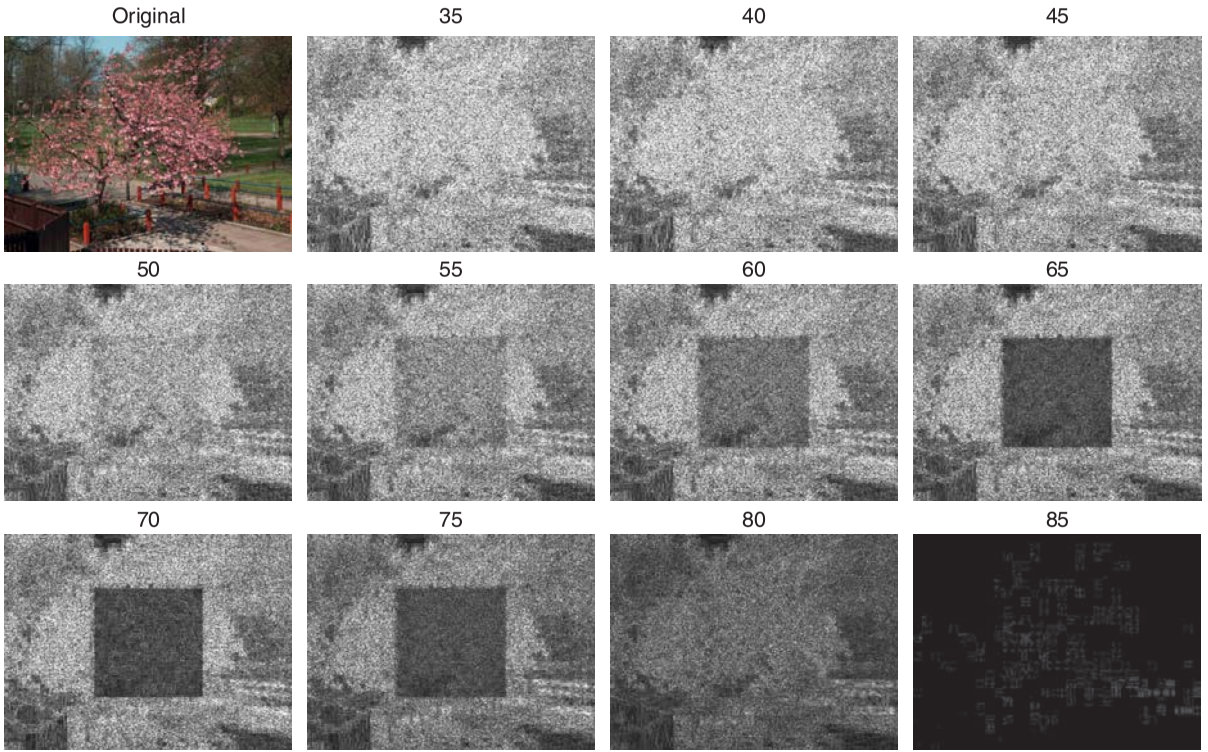

FIG. 18. Shown in the top left panel is the original image from which a central 200 × 200 region was extracted, saved at JPEG quality 65, and reinserted into the image whose original quality was 85. Shown in each subsequent panel is the different between this image and a resaved version compressed at different JPEG qualities in the range [35, 85]. At the originally saved quality of 65, the central region has a lower difference than the remaining image.

to determine if the distribution of differences (Equation 73) in two regions is similar or distinct. The K–S statistic is defined as

$$k = \max_u |C_1(u) - C_2(u)|, \tag{74}$$

where $C_1(u)$ and $C_2(u)$ are the cumulative probability distributions of two specified regions in the computed difference $d(x, y, q)$, where each value of q is considered separately.

There are two potential complicating factors that arise when detecting JPEG ghosts in a general forensic setting. First, it is likely that different cameras and photo-editing software packages will employ different JPEG quality scales and hence quantization tables [42]. When iterating through different qualities it would be ideal to match these qualities and tables, but this may not always be possible. Working to our advantage, however, is that the difference images are computed by averaging across all spatial frequencies. As a result small differences in the original and subsequent quantization tables will likely not have a significant impact. The second practical issue is that in the above examples we have assumed that the tampered region remains on its original 8×8 JPEG lattice after being inserted and saved. If this is not the case, then the misalignment may destroy the JPEG ghost since new spatial frequencies will be introduced by saving on a new JPEG block lattice. This problem can be alleviated by sampling all 64 possible alignments (a 0–7 pixel shift in the horizontal and vertical directions). Specifically, an image is shifted to each of these 64 locations prior to saving at each JPEG quality. Although this increases the complexity of the analysis, each comparison is efficient, leading to a minimal impact in overall run-time complexity.

To test the efficacy of detecting JPEG ghosts, 1000 uncompressed TIFF images were obtained from the Uncompressed Color Image Database (UCID) [43]. These color images are each of size 512×384 and span a wide range of indoor and outdoor scenes. A central portion from each image was removed, saved at a specified JPEG quality of Q_0, reinserted into the image, and then the entire image was saved at the same or different JPEG quality of Q_1. The MatLab function imwrite was used to save images in the JPEG format. This function allows for JPEG qualities to be specified in the range of 1–100. The size of the central region ranged from 50×50 to 200×200 pixels. The JPEG quality Q_1 was selected randomly in the range 40–90, and the difference between JPEG qualities Q_0 and Q_1 ranged from 0 to 25, where $Q_0 \leq Q_1$ (i.e., the quality of the central region is less than the rest of the image, yielding quantization levels for the central region that are larger than for the rest of the image). Note that this manipulation is visually seamless, and does not disturb any JPEG blocking statistics.

Note that is assumed here that the same JPEG qualities/tables were used in the creation and testing of an image, and that there is no shift in the tampered region from its original JPEG block lattice. The impact of these assumptions will be explored below, where it is shown that they are not critical to the efficacy of the detection of JPEG ghosts.

After saving an image at quality Q_1, it was resaved at qualities Q_2 ranging from 30 to 90 in increments of 1. The difference between the image saved at quality Q_1 and each image saved at quality Q_2 was computed as specified by Equation 73, with $b = 16$. The K–S statistic (Equation 74) was used to compute the statistical difference between the image's central region and the rest of the image. If the K–S statistic for any quality Q_2 exceeded a specified threshold, the image was classified as manipulated. This threshold was selected to yield a less than 1% false-positive rate (an authentic image incorrectly classified as manipulated).

Many of the images in the UCID database have significant regions of either saturated pixels, or largely uniform intensity patches. These regions are largely unaffected by varying JPEG compression qualities, and therefore exhibit little variation in the computed difference images (Equation 73). As such, these regions provide unreliable statistics and were ignored when computing the K–S statistic (Equation 74). Specifically, regions of size $b \times b$ with an average intensity variance less than 2.5 gray values were simply not included in the computation of the K–S statistic.

Shown in Table II are the estimation results as a function of the size of the manipulated region (ranging from 200×200 to 50×50) and the difference in JPEG qualities between the originally saved central region, Q_0, and the final saved quality, Q_1 (ranging from 0 to 25—a value of $Q_1 - Q_0 = 0$ denotes no tampering). Note that accuracy for images with no tampering (first column) is greater than 99% (i.e., a less than 1% false-positive rate). Also note that the detection accuracy is above 90% for quality differences larger than 20 and for tampered regions larger than 100×100 pixels. The detection accuracy degrades with smaller quality differences and smaller tampered regions. Shown in Fig. 19A are ROC curves for

TABLE II
JPEG GHOST DETECTION ACCURACY (%)

	$Q_1 - Q_0$					
Size	0	5	10	15	20	25
200×200	99.2	14.8	52.6	88.1	93.8	99.9
150×150	99.2	14.1	48.5	83.9	91.9	99.8
100×100	99.1	12.6	44.1	79.5	91.1	99.8
50×50	99.3	5.4	27.9	58.8	77.8	97.7

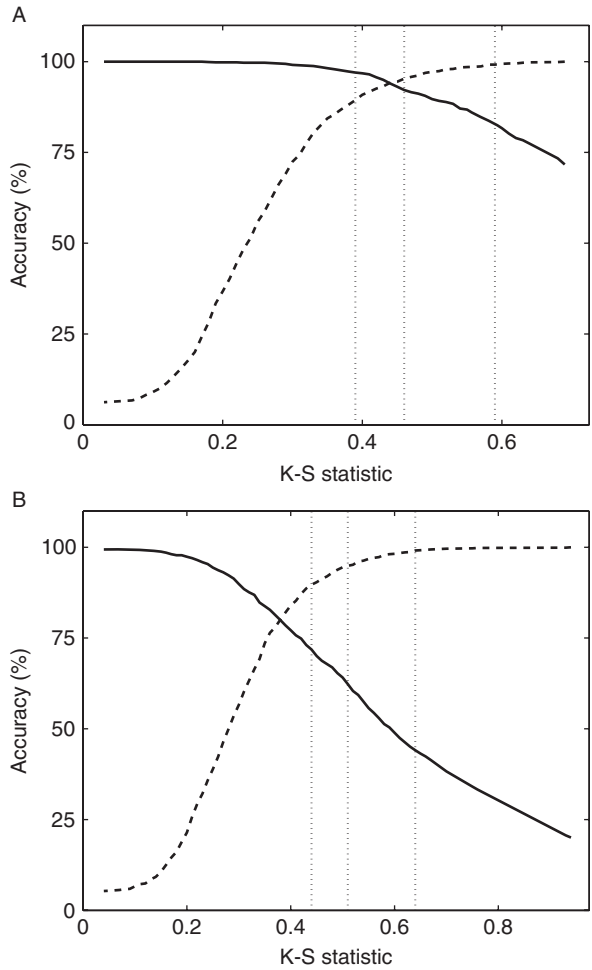

FIG. 19. Shown are ROC curves for (A): a tampered region of size 150 × 150 and a quality difference of 15; and (B) a tampered region of size 100 × 100 and a quality difference of 10. The solid curve corresponds to the accuracy of detecting the tampered region, and the dashed curve corresponds to the accuracy of correctly classifying an authentic image. The vertical dotted lines denote (from left to right) false-positive rates of 10%, 5%, and 1%. See also Table II.

a tampered region of size 150 × 150 and a quality difference of 15. Shown in Fig. 19B are ROC curves for a tampered region of size 100 × 100 and a quality difference of 10. In each panel, the solid curve corresponds to the accuracy of

detecting the tampered region, and the dashed curve corresponds to the accuracy of correctly classifying an authentic image. The vertical dotted lines denote false-positive rates of 10%, 5%, and 1%. As expected, there is a natural tradeoff between the detection accuracy and the false positives which can be controlled with the threshold on the K–S statistic.

The next few examples illustrate the efficacy of detecting JPEG ghosts in visually plausible forgeries. In each example, the forgery was created and saved using Adobe Photoshop CS3 which employs a 12-point JPEG quality scale. The MatLab function imwrite was then used to recompress each image on a 100-point scale. To align the original JPEG block lattice with the resaved lattice, the image was translated to each of 64 possible spatial locations (between 0 and 7 pixels in the horizontal and vertical directions). The shift that yielded the largest K–S statistic was then selected.

Shown in Fig. 20 are an original and doctored image. The inserted flying car was originally of JPEG quality 4/12 and the final image was saved at quality 10/12. Shown in the bottom portion of Fig. 20 are the difference images between the tampered image saved at JPEG qualities 60–98 in steps of 2. The maximal K–S statistic for the car was 0.92. Regions of low variance are coded with midlevel gray in each panel. A second example is shown in Fig. 21. The inserted dolphin was originally of JPEG quality 5/12 and the final image was saved at quality 8/12. Shown in the bottom portion of Fig. 21 are the difference images between the tampered image saved at JPEG qualities 60–100 in steps of 2. The maximal K–S statistic for the dolphin was 0.84. In both examples, the JPEG ghosts of the inserted car and dolphin are visually salient and statistically distinct from the rest of the image.

The advantage of this approach is that it is effective on low-quality images and can detect relatively small regions that have been altered. The disadvantage of this approach is that it is only effective when the tampered region is of lower quality than the image into which it was inserted.

3. Discussion

Today's technology allows digital media to be altered and manipulated in ways that were simply impossible 20 years ago. Tomorrow's technology will almost certainly allow for us to manipulate digital media in ways that today seem unimaginable. And as this technology continues to evolve it will become increasingly more important for the science of digital forensics to try to keep pace.

There is little doubt that as we continue to develop techniques for exposing photographic frauds, new techniques will be developed to make better and harder to detect fakes. And while some of the forensic tools may be easier to fool than

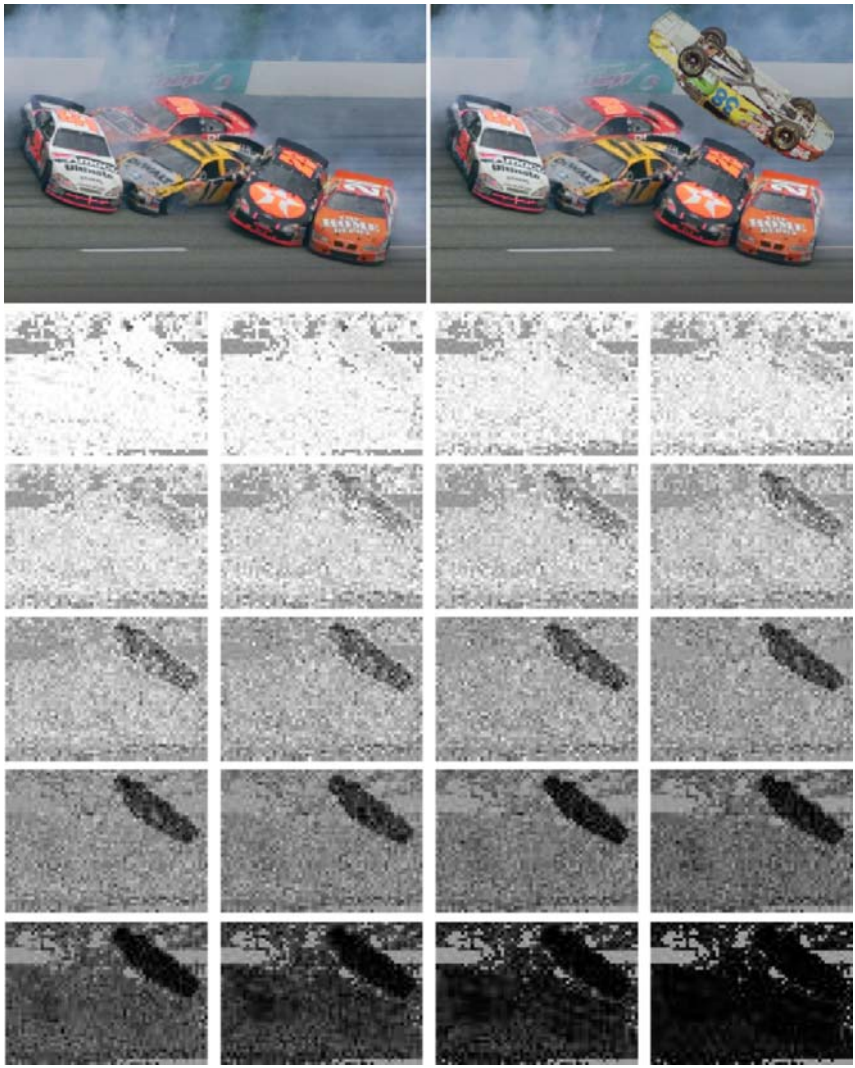

FIG. 20. Shown are the original (left) and doctored (right) image. Shown below are the different images at qualities 60–98 in steps of 2.

others, some tools will be difficult for the average user to circumvent. For example, three of the techniques described here leverage complex and subtle lighting and geometric properties of the image formation process that are nontrivial to correct in

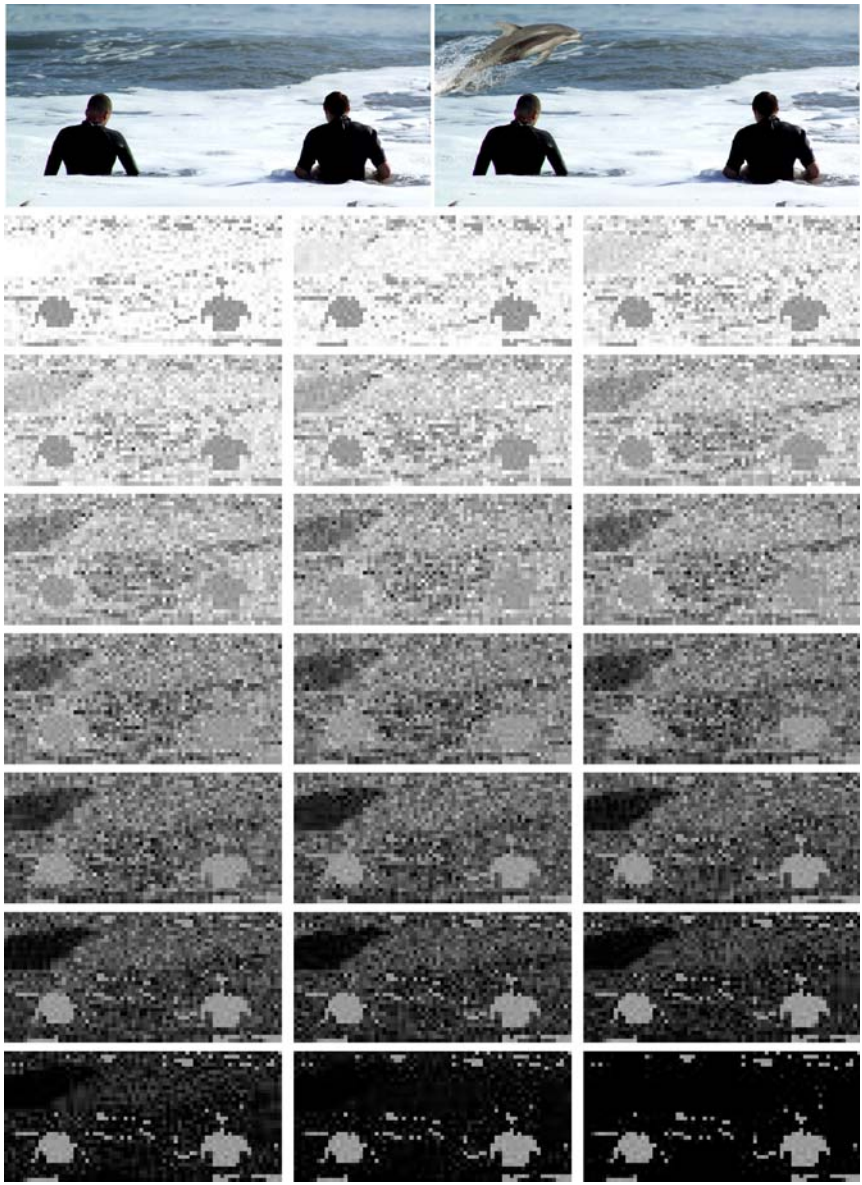

FIG. 21. Shown are the original (left) and doctored (right) image. Shown below are the different images at qualities 60–100 in steps of 2.

a standard photo-editing software. As with the spam/antispam and virus/antivirus game, an arms race between the forger and forensic analyst is somewhat inevitable. The field of image forensics however has and will continue to make it harder and more time consuming (but never impossible) to create a forgery that cannot be detected. It is my hope that this new technology, along with awareness and sensible policy and law, will help the media, the courts, and our society contend with this exciting and at times puzzling digital age.

ACKNOWLEDGMENTS

Thanks to my students and colleagues with whom I have worked over the years to develop these and other digital forensic methods, in particular Micah K. Johnson, Eric Kee, Siwei Lyu, Alin Popescu, Weihong Wang, and Jeffrey Woodward. This work was supported by a gift from Adobe Systems, Inc., a gift from Microsoft, Inc., a grant from the National Science Foundation (CNS-0708209), a grant from the U.S. Air Force (FA8750-06-C-0011), and by the Institute for Security Technology Studies at Dartmouth College under grants from the Bureau of Justice Assistance (2005-DD-BX-1091) and the U.S. Department of Homeland Security (2006-CS-001-000001). Points of view or opinions in this document are those of the author and do not represent the official position or policies of the U.S. Department of Justice, the U.S. Department of Homeland Security, or any other sponsor.

REFERENCES

[1] H. Pearson, Image manipulation: CSI: Cell biology, Nature 434 (2005) 952–953.
[2] S. Katzenbeisser, F.A.P. Petitcolas, Information Techniques for Steganography and Digital Watermarking, Artec House, Northwood, MA, 2000.
[3] I.J. Cox, M.L. Miller, J.A. Bloom, Digital Watermarking, Morgan Kaufmann Publishers, San Francisco, CA, 2002.
[4] S.A. Craver, M. Wu, B. Liu, A. Stubblefield, B. Swartzlander, D.S. Wallach, Reading between the lines: Lessons from the SDMI challenge, in: Proc. 10th USENIX Security Symposium, Washington, DC, 2001.
[5] M.K. Johnson, H. Farid, Exposing digital forgeries by detecting inconsistencies in lighting, in: ACM Multimedia and Security Workshop, New York, NY, 2005.
[6] M.K. Johnson, H. Farid, Exposing digital forgeries through specular highlights on the eye, in: Proc. 9th International Workshop on Information Hiding, Saint Malo, France, 2007.
[7] M.K. Johnson, H. Farid, Exposing digital forgeries in complex lighting environments, IEEE Trans. Inf. Forensics Security 3 (2) (2007) 450–461.
[8] A.C. Popescu, H. Farid, Exposing digital forgeries in color filter array interpolated images, IEEE Trans. Signal Process. 53 (10) (2005) 3948–3959.
[9] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, Computer Graphics: Principles and Practice, second Ed., Addison-Wesley Publishing Company, Inc., Boston, MA, 1993.
[10] J.M. Pinel, H. Nicolas, C. Le Bris, Estimation of 2D illuminant direction and shadow segmentation in natural video sequences, in: Proc. VLBV, 2001, pp. 197–202.
[11] R. Brunelli, Estimation of pose and illuminant direction for face processing, Image Vision Comput. 15 (10) (1997) 741–748.

[12] P. Nillius, J.O. Eklundh, Automatic estimation of the projected light source direction, in: Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2001.
[13] P. Sinha, Perceiving illumination inconsistencies, Invest. Ophthalmol. Vis. Sci. 41 (4) (2000) 1192.
[14] Y. Ostrovsky, P. Cavanagh, P. Sinha, Perceiving illumination inconsistencies in scenes, Technical Report AI Memo 2001-029, Massachusetts Institute of Technology, 2001.
[15] K. Nishino, S.K. Nayar, Eyes for relighting, ACM Trans. Graphics 23 (3) (2004) 704–711.
[16] R. Hartley, A. Zisserman, Multiple View Geometry in Computer Vision, Cambridge University Press, Cambridge, 2004.
[17] A. Ruszczyński, Nonlinear Optimization, Princeton University Press, Princeton, NJ, 2006.
[18] A. Lefohn, R. Caruso, E. Reinhard, B. Budge, P. Shirley, An ocularist's approach to human iris synthesis, IEEE Comput. Graphics Appl. 23 (6) (2003) 70–75.
[19] M.J. Hogan, J.A. Alvarado, J.E. Weddell, Histology of the Human Eye, W.B. Saunders Company, Philadelphia, PA, 1971.
[20] M. Pharr, G. Humphreys, Physically Based Rendering: From Theory to Implementation, Morgan Kaufmann Publishers, San Francisco, CA, 2004.
[21] D.R. Iskander, A parametric approach to measuring limbus corneae from digital images, IEEE Trans. Biomed. Eng. 53 (6) (2006) 1134–1140.
[22] R. Ramamoorthi, P. Hanrahan, On the relationship between radiance and irradiance: Determining the illumination from images of a convex lambertian object, J. Opt. Soc. Am. A 18 (2001) 2448–2559.
[23] R. Basri, D.W. Jacobs, Lambertian reflectance and linear subspaces, IEEE Trans. Pattern Mach. Intell. 25 (2) (2003) 218–233.
[24] P. Debevec, J. Malik, Recovering high dynamic range radiance maps from photographs, in: SIGGRAPH'97: Proc. 24th Annual Conference on Computer Graphics and Interactive Techniques, 1997, pp. 369–378.
[25] R.O. Dror, A.S. Willsky, E.H. Adelson, Statistical characterization of real-world illumination, J. Vis. 4 (9) (2004) 821–837.
[26] G.H. Golub, P.C. Hansen, D.P. O'Leary, Tikhonov regularization and total least squares, SIAM J. Matrix Anal. Appl. 21 (1) (1999) 185–194.
[27] B.E. Bayer, Color imaging array, US Patent 3971065, 1976.
[28] R. Ramanath, W.E. Snyder, G.L. Bilbro, W.A. Sander, III, Demosaicking methods for Bayer color arrays, J. Electron. Imaging 11 (3) (2002) 306–315.
[29] B.K. Gunturk, Y. Altunbasak, R.M. Mersereau, Color plane interpolation using alternating projections, IEEE Trans. Image Process. 11 (9) (2002) 997–1013.
[30] K. Hirakawa, T.W. Parks, Adaptive homogeneity-directed demosaicing algorithm, in: Proc. IEEE International Conference on Image Processing, vol. 3, September 2003, pp. 669–672.
[31] D.D. Muresan, T.W. Parks, Adaptively quadratic (AQua) image interpolation, IEEE Trans. Image Process. 13 (5) (2004) 690–698.
[32] R.G. Keys, Cubic convolution interpolation for digital image processing, IEEE Trans. Acoustics Speech Signal Process. 29 (6) (1981) 1153–1160.
[33] D.R. Cok, Signal processing method and apparatus for producing interpolated chrominance values in a sampled color image signal, US Patent 4642678, 1987.
[34] W.T. Freeman, Median filter for reconstructing missing color samples, US Patent 4724395, 1988.
[35] C.A. Laroche, M.A. Prescott, Apparatus and method for adaptively interpolating a full color image utilizing chrominance gradients, US Patent 5373322, 1994.
[36] J.F. Hamilton, J.E. Adams, Adaptive color plan interpolation in single sensor color electronic camera, US Patent 5629734, 1997.

[37] E. Chang, S. Cheung, D.Y. Pan, Color filter array recovery using a threshold-based variable number of gradients, in: N. Sampat and T. Yeh, (Eds.), Sensors, Cameras, and Applications for Digital Photography, Proceedings of the SPIE, vol. 3650, 1999, pp. 36–43.
[38] A.P. Dempster, N.M. Laird, D.B. Rubin, Maximum likelihood from incomplete data via the EM algorithm, J. R. Stat. Soc. B 99 (1) (1977) 1–38.
[39] Digital compression and coding of continuous-tone still images. Part 1. Requirements and guidelines, ISO/IEC JTC1 Draft International Standard 10918-1, 1991
[40] G.K. Wallace, The JPEG still picture compression standard, IEEE Trans. Consum. Electron. 34 (4) (1991) 30–44.
[41] W.J. Conover, Practical Nonparametric Statistics, John Wiley & Sons, New York, NY, 1980.
[42] H. Farid, Digital image ballistics from JPEG quantization, Technical Report TR2006-583, Department of Computer Science, Dartmouth College, 2006.
[43] G. Schaefer, M. Stich, UCID—An uncompressed colour image database, Technical Report, School of Computing and Mathematics, Nottingham Trent University, UK, 2003.

Advances in Computer Displays

JASON LEIGH

Computer Science Department, University of Illinois at Chicago, Chicago, Illinois 60607, USA

ANDREW JOHNSON

Computer Science Department, University of Illinois at Chicago, Chicago, Illinois 60607, USA

LUC RENAMBOT

Computer Science Department, University of Illinois at Chicago, Chicago, Illinois 60607, USA

Abstract

While the traditional role of displays has been for the viewing of television programming, over the past 10 years displays have increasingly become portals into virtual environments. It is therefore important to understand the current state of the art in image display technology so that one can begin to anticipate how this technology in the future will evolve, and how best to take advantage of that evolution once it matures. This chapter focuses on the issues of image display and how new advanced displays that are being used mainly in research and higher education today may become commonplace in the next 5–10 years.

ADVANCES IN COMPUTERS, VOL. 77
ISSN: 0065-2458/DOI: 10.1016/S0065-2458(09)01202-9

1. Introduction

Everything from image capture, modification, creation, and output, either on displays or on printers has been rapidly migrating from traditional photographic and print techniques to computer-based techniques. It is inevitable, except in perhaps artistic circumstances, that within 10 years all imaging will have switched over to using computing rather than traditional film and print-based media. While the traditional role of displays has been for the viewing of television programming, over the past 10 years displays have increasingly become portals into virtual environments. It is therefore important to understand the current state of the art in image display technology so that one can begin to anticipate how this technology in the future will evolve, and how best to take advantage of that evolution once it matures. This chapter focuses on the issues of image display and how new advanced displays that are being used mainly in research and higher education today may become commonplace in the next 5–10 years.

2. Advances in Desktop Displays

2.1 Cathode Ray Tube Displays

Cathode ray tube (CRT) displays, first invented in 1847 by German physicist Ferdinand Braun, became commercially available in 1922. CRTs work by shining an electron beam against a phosphor surface which causes the phosphor to emit visible light. Deflecting the beam with a magnetic field enables the beam to be precisely controlled to sweep across the CRT screen to "paint" a complete picture line by line. The chief advantage of CRTs is that it can display higher resolutions on a smaller area and has better color rendition than technologies such as LCD and plasma. The chief disadvantages of CRTs, however, are that image generation by the deflection of an electron beam requires the use of a large and deep evacuated glass tube which is heavy and relatively fragile. Despite CRT's superior image display characteristics, with the advent of plasma and LCD displays, which do not suffer from these disadvantages, the economies of scale driven by consumer demand for low-cost large flat-screen televisions has resulted, for the most part, in the extinction of these displays except in niche applications.

2.2 Plasma Displays

Plasma displays work by exciting xenon and neon gas contained in tiny cells that are sandwiched between two layers of glass and electrodes. The application of a charge to an individual cell causes the gas to ionize into plasma which then excites phosphors to emit light. Color is achieved as a triad of subpixels (three cells). Each cell is coated with a particular phosphor that results in the emission of either red, green, or blue light. The brightness of the color subpixel is achieved by varying the pulses of current flowing through the cell. The blending of the emitted light from the triad results in the overall color of the pixel. As plasma displays use the same phosphor coating as CRTs, they tend to have color reproduction comparable to CRTs. They also consume approximately the same amount of power as CRTs.

2.3 Active Matrix Liquid Crystal Displays

LCDs work by passing light from a cold cathode fluorescent tube through a vertical polarizing filter which is then rotated by the individual liquid crystal cells that make up the display. The resulting light is passed through a horizontal polarizer and then finally a color filter. The degree of alignment of the light with the horizontal polarizer determines how much light passes through. A matrix of thin-film transistors

holds a varying degree of charge in each cell which determines the degree of rotation of the light. Color is achieved by individual red, green, and blue filters covering each liquid crystal cell. Each cell makes up one of three color subpixels much like in a plasma display. LCDs have been a major competitor for plasma and while plasmas provide better color gamut, LCDs have surpassed plasmas in their ability to display very high-resolution images.

For example, introduced in 2001, the IBM T220 and T221 (nicknamed "Big Bertha") provide over 9 megapixels of resolution (3840 $\times$ 2400) in a 22-in. form factor. Recent LCDs from Sharp and Toshiba have 8 megapixels in a 52–62-in. form factor. These latter screens are targeted as the ultimate replacements for CRTs in digital film production.

3. Advances in Wall Displays

Until the mid-1990s, image creation for large wall displays (often called Power Walls) primarily used CRT-based projectors. The main advantage of CRT projectors is that they maintain good brightness to 10,000 h, are capable of generating very high-resolution images (up to 1920 $\times$ 1200), and are able to refresh at high rates. When compared to LCD and DLP (Digital Light Processing) projectors, CRT projectors also deliver superior black levels. Because of their high refresh rates, CRT tubes were the basis for many projected stereoscopic displays whereby the viewer donned 3D LCD shutter glasses that were synchronized with the update of the projection screen. Virtual reality (VR) systems such as the CAVE (see below) employed CRT projection walls to create a room capable of showing stereoscopic 3D images whereby objects would appear to float in the room as if real. The disadvantage of CRTs is their size and weight relative to LCD and DLP projectors. CRT projectors require three tubes (for each of the primary colors), whereas LCD/DLP projectors require a light source consisting of a single lamp. Another disadvantage of CRT projectors is that they need to be both geometry and color aligned as part of regular maintenance. The need to align the three tubes separately has historically been a complex and time-consuming task but is now achieved automatically in many commercial products.

In the mid- to late 1990s, LCD and DLP projectors were rapidly becoming cheaper and also smaller. This was driven largely by the business presentation market which required low-cost projectors for projecting presentation slides in boardroom meetings. Tilings of LCD or DLP projectors were used to create larger walls [1]. While these projectors were relatively inexpensive, it was difficult to align the geometry and color of the screens. Techniques have been developed for automatic alignment through the use of computer vision. As a result, commercial turnkey

FIG. 1. The 100-Megapixel LambdaVision LCD wall.

solutions are now available—but at a relatively high cost. Today, DLP projectors such as Sony's SXRD are capable of 8-megapixel resolution (4096 × 2160) and are largely used in movie theaters. In prototype form is NHK's Super Hi-Vision project [2] which is capable of 7680 × 4320 (Fig. 1).

Currently, the most economical way to build scalable wall displays is by using LCD panels rather than projectors due to the panels' long life, high resolution, and low cost. Other benefits include the fact that LCD panels are quite well color calibrated and apart from the physical alignment of the panels during tiling, require no further alignment, unlike projectors. Tilings of these have been used to create ever greater resolution walls such as the 100 Megapixel LambdaVision display [3]. The main drawback of LCD panels is that they have mullions that prevent them from producing truly seamless displays. Mullions do not adversely impact the viewing of an image if they are taken into account in the rendering of the image. However, mullions do make the reading of text difficult when it occludes either an entire sentence or words. Driving these tilings of displays requires a cluster of computers, often each computer drives anywhere between 1 and 4 displays, and a master computer coordinates the entire cluster.

4. Advances in Portable Displays

The earliest portable displays from the 1980s all the way to the mid-1990s were either monochromatic LEDs or LCDs—such as the ones used in digital watches and calculators, and PDAs—such as the Apple Newton or Powerbook. After the mid-1990s saw the advent of color LCDs which initially appeared in laptop

computers and then increasingly on desktops as replacements for traditional CRTs. While low-resolution head-mounted displays (HMDs) used small and therefore low-resolution LCD panels, their high-resolution counterparts still used CRT displays. These HMDs captured the imagination of the public in the early 1990s when the concept of virtual reality was becoming popular. The iconic image of VR is that of a person with a large helmet display covering their face. They were, however, not particularly practical because having their eyes covered made it exceedingly cumbersome for anyone to move around physically. However, one practical application of HMDs is its use in augmented reality applications whereby a viewer is able to see computer graphics superimposed over the real-world environment. In the late 1990s, the use of HMDs had largely faded in favor of using lightweight, and therefore comfortable LCD shutter glasses, such as the ones used in the CAVE, which permitted more than one person to view the 3D imagery at the same time.

New display technologies are on the horizon and will prove to enable entirely new classes of applications. For example, pico projectors[1] are small enough that they can be encased in objects as small as a pack of cigarettes and can be driven by small power cells such as AA batteries. Pico projectors consist of colored laser diodes whose light reflects off a micromirror that scans horizontally and vertically to paint the image—much like a CRT. Manufacturers of mobile devices such as cellular phones are actively developing cellular phones with built-in pico projectors, in the belief that it will be attractive to business travelers who will be able to carry large display screens without incurring the physical burden associated with current LCD laptops and screens.

Electronic paper displays [4] depict images in a manner similar to print on paper (although presently at much lower resolution). The key advantage to e-paper is its extremely low-power consumption. Once a pixel is activated the image persists even when power is turned off. This is highly favorable in e-paper's most popular application, electronic book readers. In the simplest implementation, e-paper displays are built from a grid of picture elements containing a dark-colored dyed hydrocarbon oil and particles of titanium dioxide and a charging agent. These cells are sandwiched between two transparent layers of electrodes. When a voltage is applied to a cell, the charged particles are drawn to the layer with the opposite charge. When the particles appear on the top side of the display, the display cell appears white, and when the particles appear on the bottom side of the display the display cell appears dark. By manufacturing the display with a plastic substrate, e-paper displays can be made flexible therefore beginning to resemble the physical properties of paper.

[1] http://www.microvision.com

Newer generation of e-paper displays also provides the ability to show color images by including a separate red, green, and blue filter under each element of the display, which reduces the overall effective resolution of the display by a third. Furthermore, because e-paper displays have no backlight, and therefore depend on reflected light, color displays tend to appear murky since as much as 50% of reflected light is lost in the image. The chief disadvantage of current e-paper displays compared to other displays is that they have extremely slow refresh rates (on the order of a second) making them unsuitable for depicting real-time interactive computer graphics. Another anticipated application for these displays is for digital signage. Researchers have proposed digital sign designs whereby an e-paper poster would be refreshed by a computer wirelessly. The sign and computer itself would be powered by solar power.

Flexible displays [5] are driven by the interest to apply displays to surfaces that are not necessarily flat or to enable displays that can be folded away to enhance portability. This technology has the potential to allow users to carry much larger and higher resolution displays without the bulk of a desktop display. One can envision that in future laptops, very high-resolution displays can be opened like sheet music between two poles. At a larger scale, flexible displays have the potential to replace the front-projected pull-down screens often found in boardrooms. However, presently flexible display technologies are still very much under research and development rather than fully commercial products.

5. Touch Interfaces

The most widespread use of touch screens has been in automated teller machines. They are popular in public displays because they lack any form of mechanical part that could potentially suffer wear and tear over the course of millions of uses. More importantly, they allow custom user interfaces to be created depending on the application. Early touch interfaces tended to use beams of infrared light arranged along two of the four sides of a display that shone upon either photoresistors or diodes arranged on the opposite sides of the display. A finger occluding the light beam would trigger a response in the software that could be equated with the press of a button.

Resistive touch screen is another example of an older touch technology. These are prevalent in cell phones, PDAs, tablet computers, and game systems such as the Nintendo DS. A resistive touch screen is composed of a layer of glass (at the bottom) followed by a conductive and a resistive layer on top. On top of this resides an additional layer which the user depresses. Each layer is separated by a very small space. When the screen is pressed, the conductive and resistive layers touch each

other closing the circuit. The resulting level of the current determines the position of the touch. Since resistive touch screens work by applying pressure to the surface, they can be operated with a finger or a stylus. The chief advantage of resistive displays is that they are cheap to manufacture and robust to outside elements such as water and dust. The disadvantages are that the screens offer only 75% clarity, and it is difficult to create a touch screen that can register multiple simultaneous touches—often described as multitouch screens.

The once more expensive capacitive touch screen [6] is fast replacing many applications that have formerly used resistive touch screens—especially in popular consumer items such as the iPhone. Two types of capacitive screens exist: surface capacitive and projected capacitive. Surface capacitive relies on the screen and the object that touches the screen to hold some charge (hence capacitance). When the object is brought close to the screen, the resultant change in the reference charge on the screen is detected by sensors at the corners of the screen which determine the *X* and *Y* location of the touch point. Projected-capacitive touch screens, on the other hand, work by projecting electrostatic field lines from the sensors—the object and the surface point on the touch screen become two plates of a capacitor. The sensors determine the position of the touch by detecting the resulting capacitive coupling when the finger touches the screen.

All touch-screen displays prior to 2006 would only register one touch at a time. SmartTech[2] was one of the earliest developers of screens that could register more than one touch—in their case two simultaneous touches. SmartTech's approach works by using cameras at the edges to produce 2D tracking of at most two targets. Users can point with their fingers as well as use a special stylus which has a unique identifying pattern at the tip that can be detected by the cameras. As a result, multicolored pens can be accommodated by uniquely identifying the styli. Also hovering is possible by detecting partial coverage of the stylus by the camera.

While research in touch-screen technology, user interfaces, and applications has been active for over a decade, touch technologies saw a dramatic increase in popularity with the advent of Apple's iPhone (which uses a capacitive touch screen) and New York University's (NYU) multitouch table [7].

NYU's multitouch table works on the principle of frustrated total internal reflection whereby infrared LEDs illuminate an acrylic touch surface by shining into and along the edges of the display. Pressing the acrylic screen causes the infrared to pass through, reflect off one's finger, and then pass back through the acrylic where the depression is sensed by an infrared camera. Image-tracking algorithms are used to find "blobs" representing touches, and the movement of the blobs is correlated with

[2] http://www.smarttech.com

FIG. 2. TacTile: a 52-in. high-definition LCD-based multitouch table.

actions that are specific to the application such as panning and zooming. A projector is placed beneath the screen to create the computer-generated image. This technique also works effectively for LCD screens, which in general result in sharper looking displays that are capable of greater resolution and brightness at a much lower cost. Researchers at the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago have successfully built a 30-in. LCD multitouch screen, called the MicroTable, with 4 megapixels of resolution. Pictured is EVL's TacTile, a 52-in. high-definition-resolution LCD-based multitouch table. Underway is the development of the OmegaTable, which combines autostereoscopic techniques (described below) with this type of capability to provide a multitouch display surface that can show both 2D and 3D content without the need to wear specialized glasses (Fig. 2).

More advanced touch displays are in prototype form and the earliest products will likely emerge in 2009. For example, Sharp's touch screen places an optical sensor at every LCD pixel. This allows it to be used as both a display and a touch screen simultaneously rather than requiring a separate display and touch sensor layer as in other approaches. The end result is a much thinner touch screen that can also be used as a scanner.

6. Advances in Stereoscopic Displays

In the early 1990s, computer-generated stereoscopic displays consisted of either CRT-based monitor and projectors that were mediated by LCD shutter glasses, or HMDs that used pairs of miniature CRTs. The former was primarily intended to

facilitate stereoscopic viewing. The latter combined stereoscopic viewing and a viewer-centered perspective to achieve virtual reality.

6.1 Head-Mounted Displays

Invented in the mid-1960s by Ivan Sutherland, the Sword of Damocles is widely considered the first VR and augmented reality HMD system. In a HMD, a pair of miniature display elements is worn over the head, covering the users' eyes. The early HMDs used CRTs whereas more modern HMDs use either LCDs or organic light-emitting diode (OLED) microdisplays. Some HMDs also include headphones as well as six-degree-of-freedom tracking system that allowed the computer to track the position and orientation of the viewer's head (and hence eyes) in real time. The head-tracking information is used to create a view of the world that is based on the users' viewpoint. This is generally called viewer-centered perspective, and when combined with stereoscopic computer graphics results in true VR [8].

The chief advantage of HMDs was that it isolated the user from the real world to provide full immersion in a virtual world. The chief disadvantage was that these HMDs were heavy and had high lag between tracking head movements and updating the visuals. As a result, users tended to become nauseated after only short periods of use. Since the viewer's eyes are covered with the displays, they are only able to see what the screens display. Therefore, in many HMD configurations, a guardrail surrounds the user to prevent them from accidentally tripping over obstacles.

6.2 The CAVE

An alternative method to achieving immersive virtual reality appeared in late 1991. Initially called the "Closet Cathedral," by its inventors, the system was later given its more widely known name, the CAVE [9]. The CAVE (a recursive acronym for CAVE audio–visual experience automatic environment) consists of a 10 ft^3 of a room whose walls consist of rear projection screens. The original CAVE used large CRT projectors capable of refreshing at 120 Hz. Stereoscopic viewing was achieved using lightweight LCD shutter glasses that were synchronized with the projectors. An Ascension Flock of Birds electromagnetic six-degree-of-freedom tracker was used to track the main viewer and also to support three-space interaction with absolute position and orientation tracking using a device called the Wand. The CAVE allowed the user to wear lighter weight glasses and was still able to provide a sense of immersion by its surround-screen walls. As a result, users were able to

FIG. 3. The CAVE.

comfortably work in the CAVE for hours at a time. Since a user can see his/her physical body in relation to the virtual world, it allowed them to roam more freely without fear of crashing into walls (Fig. 3).

6.3 Stereoscopic Wall Displays

Since the invention of the CAVE, many variations emerged—such as the RAVE, a CAVE that has the ability to open its two side walls; and the CABIN, a six-sided CAVE developed by Tokyo University [10]. These systems have been commercially available since the mid-1990s. After the mid-1990s, the emergence of DLP projectors that could be synchronized with 3D graphics at 100 Hz emerged and began to replace CRT projectors in large CAVE environments. Today, these types of projectors are the primary means to drive CAVE walls.

6.4 The GeoWall

Up until 2000, expensive computer systems (such as Silicon Graphics) and projectors were necessary to drive these stereoscopic systems. However, these were soon replaced with high-performance graphics cards intended for the PC gaming market. In 2001, the Electronic Visualization Lab developed a low-cost stereoscopic graphics wall driven by commodity DLP projectors, polarizing filters, and a PC equipped with a dual-headed graphics card. Stereoscopic capability is usually achieved by using a pair of DLP projectors that are fitted with linear or

circular polarizers that are oriented in opposing directions. Audiences see the stereo image by wearing low-cost polarized glasses that are oriented in the opposite direction. Within 5 years over 500 were deployed in Geoscience Research Laboratories and Classrooms which ultimately resulted in the adoption of the name GeoWall [11]. The popularity within the geosciences was driven by several factors: firstly, the cost of the entire system was just under $10,000 rather than $1,000,000 (which was the typical cost of the CAVE); secondly, Geoscience data are highly three dimensional; thirdly, the device could be easily constructed and maintained; and lastly, a consortium was formed to support the development of software and nurture collaborations with commercial Geoscience software companies.

6.5 Autostereo Displays

The holy grail of stereoscopic computer displays is autostereoscopy—the ability to show stereoscopic images without requiring the user to wear any form of mediating glasses. Autostereo displays work by presenting the viewer with a vertically interleaved set of left and right eye stereoscopic images, and then using either vertical strips of plastic lenses or a physical line screen barrier separated by a small gap [11]. This allows separate left and right eye images to be presented to the viewer without requiring them to wear any form of eyewear such as those used in HMDs or the CAVE. One drawback of this technique is that when a viewer moves his/her head, an opposite pair of images can potentially be presented to the viewer creating reversed stereo (also known as pseudostereo). To compensate for this, head-tracking systems such as the ones used in the CAVE, or camera-based tracking systems, were used to track the viewers head to ensure the correct set of left/right eye images were presented to the viewer. To support multiple viewers, multiple views must be simultaneously projected from the display. Approaches include spatial multiplexing—where the total resolution of the display is split between multiple views; multiprojector—where an array of projectors simultaneously project multiple views onto a special transmissive or reflective screen; or time sequential—where a single very fast display device creates multiple views that are synchronized with a secondary optical component (such as a ferroelectric liquid crystal shutter) that directs the images to the appropriate zones in space. At the present time, the only practical approach is spatial multiplexing as the approach can take advantage of new and higher density LCD displays.

6.6 Autostereo Walls

One significant drawback of autostereo displays is that at least half the horizontal resolution of a display is lost to present both left and right eye images. This has led some researchers to develop high-resolution autostereo displays using either a tiling of

LCD panels or an array of horizontally placed projectors [12]. In the former case, a barrier screen-based system called Varrier used a 7 × 5 tiling of 21-in. LCD panels providing an autostereo resolution of 3000 × 6000 [13]. Shortly after the invention of the Varrier, a major breakthrough was achieved whereby a screen could display both autostereoscopic content and monoscopic content within subwindows on a single screen. Known as the Dynallax (for dynamic parallax barrier), this particular approach replaced the static barrier screen with an LCD panel that could be dynamically controlled to draw lines of any width and pitch as well as regions of with and without the line screens (and hence regions with and without the stereo effect) [14].

Whereas Varrier was designed to provide autostereo for a single viewpoint, the Dynallax technique has been shown to be able to potentially support multiple views. The ability to present multiple views simultaneously allows more than one viewer to see autostereoscopic images at the same time. This is important if autostereo is to ever become a practical mainstream display device (Fig. 4).

Commercial products are also beginning to emerge albeit still at a very low resolution. For example, Philips' WOWzone system consists of a tiling of 3 × 3 42-in. 1920 × 1080 LCDs that provides nine views at an effective autostereo resolution of 640 × 3240 pixels.

FIG. 4. A Varrier display built with an array of 10 × 5 LCD displays.

7. Display Environments of the Future

Computers and displays are embedding themselves into ever device and surface, and will continue to do so as technology becomes cheaper and more powerful so as to enable wholly new classes of applications. Within a decade, one can envision that flat-screen displays will not only become ubiquitous, but will be embedded into every surface of our homes, offices, and academic and research laboratories.

7.1 In the Home

US households on average have 2.24 televisions, and as many as 66% have three or more televisions (not counting computer displays) [15]. In the next 5 years, we imagine that all of these TVs will be replaced with large high-definition (HD) flat screens. These screens will be used for entertainment/gaming, video conferencing with friends and relatives, accessing the Internet, etc. The living room or dining room might have a large screen that is used as a digital picture frame. Coffee tables with intuitive touch-screen interfaces will also become attractive, enabling guests to thumb through digital family albums. Even wall-mounted screens will have touch-screen capabilities so that they can become digital whiteboards on which children doodle or family members leave notes for one another.

Looking further out, the walls themselves might be covered from top to bottom with display-capable materials with astounding dynamic range so that, for example, the room could be bathed with lifelike vistas of outdoor scenery. The walls might even become a major light source in the home. Digital wallpaper could be changed on a whim, and live media could be placed anywhere on the walls, artwork, weather information, stock quotes, or world news. For this to become a reality, the concept of television has to evolve from a traditional passive viewing appliance into an interactive appliance ubiquitously and seamlessly integrated into household structures like walls and furniture.

7.2 In the Workplace

In the workplace, displays will pervade meeting rooms and individual offices/cubicles far beyond what is seen in the classical boardroom. In meeting rooms, ultra-large flat-panel displays (either rigid or flexible) will replace projectors. Apart from the "wow" factor of having such a display, it makes practical sense when the cost of maintaining a projection-based system exceeds that of maintaining a flat-panel display system. In approximately 2 years, it will become economical to replace boardroom projectors with massive (80+ in.) flat-panel displays. Flat panels also

have the added benefit that the content is clearly visible without dimming the room lights, and they consume less power than projectors. As more flat-panel displays are deployed in meeting rooms, the desire to use them for more than just presentations will grow. They will be used for multisite video conferencing (such as in Cisco, Polycom, and HP's recent telepresence products) as well as for poster boards on which digital information can be posted. The traditional paper-based War Room/Project Room will be forever transformed. In traditional Project Rooms, the walls are usually covered with notes and drawings from intense brainstorming sessions. Teasley et al. [16] at the University of Michigan's School of Information found that engineering teams who worked in these environments enjoyed considerable performance improvements over teams that worked in more traditional work environments. A person's working memory can hold approximately 6–7 pieces of information at a time. Externalizing one's thoughts on a wall expands one's working memory and enables a team to collectively organize hundreds of thoughts at a time.

7.2.1 Cubicles

One would expect that, in the future, cubicle walls will have lightweight, low-power, low-heat, displays imbedded in them that have both near-print-quality resolution (at least 100 dpi) and can be controlled with touch. Higher resolution is necessary in a cubicle because of the close proximity a person is to the walls/screens. One can imagine that the display-enabled cubicle walls (which we call Trans-Walls—for Transformable Walls) are modular, lightweight, and can snap together to provide daisy-chained power and networking as well as physical configuration information, so that an intelligent software controller can manage them as one continuous surface. The computing to support the intelligence could be built directly into the walls. Cameras could also be embedded in the walls, to be used for seamless video conferencing, just as in Project Rooms. Novelist Will Shelf covers almost all the wall surfaces of his office with Post-It notes to organize the little pieces of detailed information that will eventually be woven into a coherent story. Many knowledge workers, such as analysts, can relate to this; it is essentially a mini-Project Room within one's office where one can externalize all one's thoughts on the walls.

7.2.2 The Tradeshow Conferences

It is now quite common to see large panel displays at tradeshows, especially at technology-related venues. Panel displays can be found in both commercial booths and university research booths. In the future, one can imagine that conferences will replace traditional pinup boards in poster sessions with digital poster boards

constructed from TransWalls that can be rented from tradeshow audio–visual companies. A presenter could simply walk up to a wall, enter a login ID or plug in a thumbdrive, and immediately give a multimedia presentation (Fig. 5).

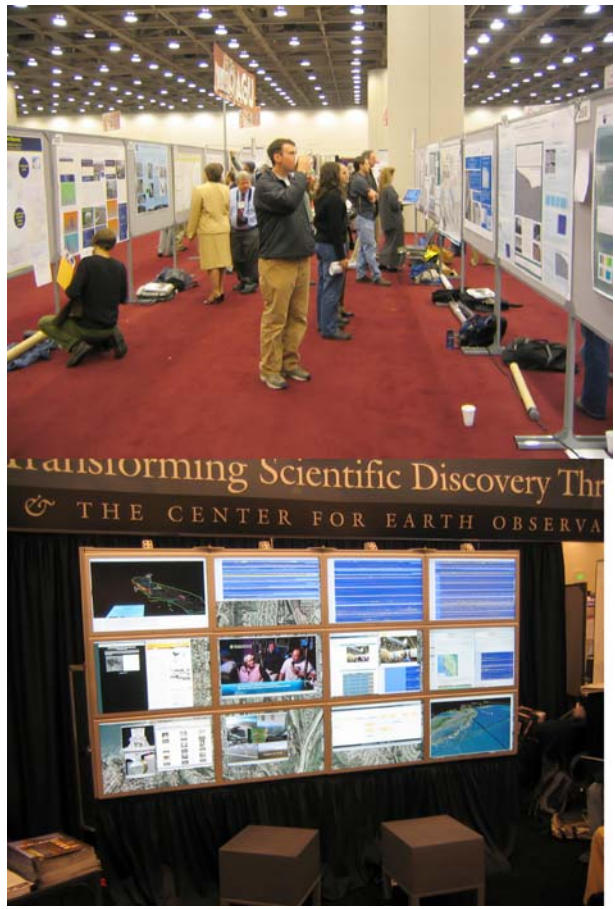

FIG. 5. A traditional paper-based conference poster session (left). University research booths are making use of tiled display walls as digital poster boards. The poster on the right, consisting of twelve 30-in. Apple screens, is from the Scripps Institution's booth at the American Geophysical Union meeting in San Francisco.

7.2.3 Research Labs

High-resolution displays that are greater than 4 megapixels are becoming a standard part of scientific research. For scientific disciplines, large displays are the only means by which they can see data from their instruments. With the advent of low-cost LCDs, research labs are now using tiled display walls as "mash-up" environments where they can juxtapose a variety of data so that they can look at them as a whole [17]. While similar to the notion of Project Rooms, described earlier, a key difference is that for large-scale scientific research, there is no other way to look at the data. These projects routinely deal with time-varying data on the order of terabytes to petabytes. It is impossible to manage this information by printing out static diagrams on sheets of paper and pinning them to a wall. The microscopes and telescopes used by scientists today are no longer simple optical instruments, but are integrated with complex computing systems that perform noise filtering, mosaicing, and feature detection. High-resolution displays are the lenses into those instruments. TransWalls would be attractive in research laboratories so that individual office tiled displays, or larger tiled walls, could be quickly built. Scientists expect this degree of simplicity, as they typically have neither the patience nor the expertise to work with esoteric computer gadgets (Fig. 6).

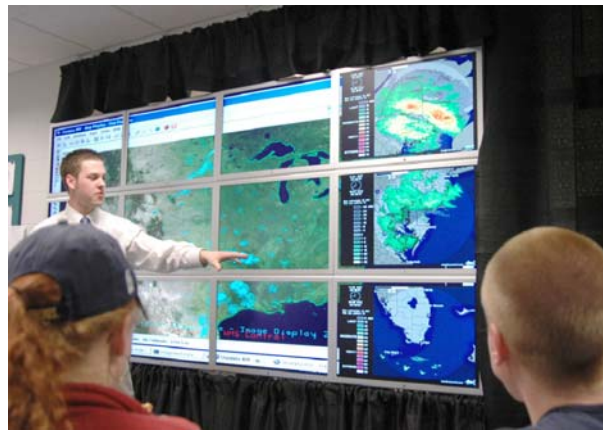

FIG. 6. A tiled display at the University of Michigan's Atmospheric, Oceanic, and Space Sciences Department is on the right. Undergraduate students use it to give presentations on class projects. The wall is used as a storytelling environment, much like a traditional poster board.

8. Content Generation and Delivery to Displays

Thus far, we have discussed fundamental display technologies and their uses. Two crucial components for enabling these displays to be useful are intelligent middleware and high-speed networking.

8.1 Middleware for Scalable Content Rendering

Content generation typically occurs in a number of ways: Localized content rendering is where the work of doing the rendering of the content is conducted near the displays. Interactive video games use this model because it minimizes latency. Streamed content rendering involves the delivery of prerendered content such as a movie that is streamed over a network link or over the air such as TV broadcast and satellite.

High-resolution tiled displays have been found to be ideal for scientific research because of the shear amount of data that need to be visualized. Increased screen real estate and resolution enable users to display and juxtapose more data simultaneously and thereby enhancing the users' ability to derive insight from the data. The major challenge in these scalable display environments is how to scale graphics rendering algorithms to be able to handle the exponentially growing data volumes that are accumulated in scientific research.

The traditional model for scalable rendering has assumed that faster graphics cards will be more than capable of supporting rendering to keep pace with the exponential growth of data size. Middleware such as WireGL, and its enhanced version, chromium managed a high-resolution scene by distributing polygons only to the computers responsible for rendering the particular viewport into the overall scene [18]. Termed "sort-first rendering," the advantage of this scheme was that if an image filled a significant portion of the tiled display much of the geometry would be evenly distributed across all the computers. However, if the image fell on only a small portion of the wall, a load imbalance would result and overall rendering rate would decrease dramatically.

An alternative approach replicates all the data across all the computers and simply uses the raw graphics power to process the data to generate the images for the individual viewports of the display tiles. Used in middleware such as CGLX, the advantage of this approach is that many existing OpenGL applications can be easily ported without modification of the code. The primary disadvantage is that the amount of data that can be rendered is limited by the capabilities of the individual graphics card.

A third approach delegates the rendering to a remote cluster of computers and instead treats the tiled display simply as a large frame buffer connected via a high-speed network. This approach was first pioneered in SAGE (the Scalable Adaptive Graphics Environment) [19]. The scheme has greater scalability than previous approaches because it allows multiple cluster computers that are potentially rendering different and very large data sets, to work concurrently and stream visualizations to be displayed on the wall as individual windows as if on an enormous desktop. This allows users to arbitrarily position and resize these windows on the wall and therefore enabling them to work with multiple visualizations simultaneously. By contrast, the techniques used in WireGL/Chromium and CGLX require that the entire wall be used to display only one visualization application at a time. This is acceptable for small tiled displays but becomes impractical for very large display walls, especially those that can in the future potentially cover all the walls of a room. The SAGE model also has the advantage that far less powerful graphics cards and computers can be used to drive the display walls thereby making it more cost effective for scientists to acquire them. These display walls can then connect into global high-performance networks to take advantage of large-scale computing resources that are deployed at supercomputing centers around the world.

8.2 High-Speed Networking for Scalable Content Display

Traditional broadcast transmission (such as radio and television) has the advantage that it can reach wide populations with low infrastructure cost. The disadvantage is that limited bandwidth will constrict the number of channels that can be served at any given time. In a networked streaming model, distributed servers provide users with the desired content in an on-demand basis. However, these systems can become overloaded if they are oversubscribed and therefore careful replication and distribution of the content is needed in anticipation of access demands. Furthermore, because of the lack of network service guarantees streamed content can stutter when there is network congestion. To mitigate this most streaming solutions buffer the content or allow for the full download of the content so that it can be played off local storage. While buffering is tolerable for viewing movies, it is unacceptable for applications such as video conferencing.

It is clear that the networks will be as important as the displays in the future. Next-generation networks that enable users to dynamically provision light paths of 1–10 Gb/s are becoming a routine part of the world's research infrastructure. The Global Lambda Integrated Facility [20] is a consortium of network researchers around the world that have created a vast network of 10 Gb network links for the

purposes of providing scientific applications the bandwidth to stream data without suffering the congestion experienced over the regular Internet. Researchers as part of the Global Lambda Visualization Facility have been using these networks to connect the aforementioned ultra-high-resolution display systems together for the purposes of developing technology to enable routine collaborative science involving participants around the world and large-scale data and visualization. It is anticipated that this model of a hybrid network whereby some of the data are routed via traditional network routers, and other data are switched using all-optical switches will proliferate initially to large global businesses, and then eventually to the homes so that optical fibers carrying gigabits of bandwidth can bring all content to the homes. In the US, Verizon has already begun trials to bring fiber to the home. In Japan, fiber to the home is already a service offered by Nippon Telephone and Telegraph (NTT)—providing 100 Mb/s to subscribers. All these services, however, are offered over the traditional Internet routed infrastructure and as of yet none provide the kind of hybrid network capability currently used by research scientists.

9. Conclusion

In this chapter, we have discussed the evolution of display technologies, how content is generated on them and how they will likely impact us in the future. Displays will no doubt become more ubiquitous, possessing higher resolution and capable of both 2D and autostereoscopic 3D. They will become more portable and configurable, capable of touch interaction and will contain imbedded sensors, such as cameras so that they are both an input and output device. While the current generation of tileable displays is driven by separate clusters of computers, future displays will have computing directly imbedded in them. All that is needed to operate the display is a high-speed multigigabit network connection and electrical power.

In the homes, displays will be used for everything from table surfaces to wallpaper. In work environments, displays will become the lenses that enable users to control, filter, and examine the exponentially growing tsunami of data from the Internet and from sensor arrays that are rapidly being deployed around the world.

References

[1] T. Funkhouser, K. Li, Large format displays, IEEE Comput. Graph. Appl. 25 (4) (2000) 20–21.
[2] M. Kanazawa, et al., Ultrahigh-definition video system with 4000 scanning lines, NHK Technical Report, 2003.

[3] L. Renambot, A. Johnson, J. Leigh, Techniques for building cost-effective ultra-high-resolution visualization instruments, in: 2005 NSF CISE/CNS Pervasive Computing Infrastructure Experience Workshop, Urbana, IL, 27 July 2005.
[4] B. Comiskey, J.D. Albert, H. Yoshizawa, J. Jacobson, An electrophoretic ink for all-printed reflective electronic displays, Nature 394 (6690) (1998) 253–255.
[5] G.H. Gelinck, et al., Flexible active-matrix displays and shift registers based on solution-processed organic transistors, Nat. Mater. 3 (2) (2004) 106–110.
[6] R.G. Kable, Electrographic apparatus, United States Patent 4,600,80, 71986.
[7] J. Han, Low-cost multi-touch sensing through frustrated total internal reflection, in: Proc. 18th Annual ACM Symposium on User Interface Software and Technology, 2005.
[8] B. Sherman, A.B. Craig, Understanding Virtual Reality, Morgan Kaufmann Publishers, San Francisco, CA, 2003.
[9] C. Cruz-Neira, D.J. Sandin, T.A. DeFanti, Surround-screen projection-based virtual reality: the design and implementation of the CAVE, in: Proc. SIGGRAPH'93 Computer Graphics Conference, ACM SIGGRAPH, August 1993, pp. 135–142.
[10] M. Hirose, CABIN-A multiscreen display for computer experiments, in: Proc. International Conference on Virtual Systems and MultiMedia, 1997, pp. 78–83.
[11] A. Johnson, J. Leigh, P. Morin, P. Van Keken, GeoWall: Stereoscopic Visualization for Geoscience Research and Education, IEEE Comput. Graph. Appl. 25 (6) (2006) 10–14.
[12] G.J. Martin, A.L. Smeyne, J.R. Moore, S.R. Lang, N.A. Dodgson, Three-dimensional visualization without glasses: a large-screen autostereoscopic display, in: Proc. SPIE 4022, Cockpit Displays VII: Displays for Defense Applications, Orlando, Florida, 26–28 April 2000.
[13] D. Sandin, T. Margolis, J. Ge, J. Girado, T. Peterka, T. DeFanti, The Varrier autostereoscopic virtual reality display, in: Proc. ACM SIGGRAPH 2005, ACM Transactions on Graphics, 30 July 2005–4 August 2005.
[14] T. Peterka, R.L. Kooima, J.I. Girado, J. Ge, D.J. Sandin, A. Johnson, J. Leigh, J. Schulze, T. A. DeFanti, Dynallax: solid state dynamic parallax barrier autostereoscopic VR display, in: Proc. IEEE Virtual Reality Conference 2007 (VR'07), Charlotte, NC, 10–14 March 2007.
[15] N. Herr, Television & health, Sourcebook for Teaching Science, 20 May 2007.
[16] S.D. Teasley, A. Covi Lisa, M.S. Krishnan, J.S. Olson, Rapid software development through team collocation, IEEE Trans. Software Eng. 28 (7) (2002) 671–683.
[17] J. Leigh, M. Brown, Cyber-commons: merging real and virtual worlds. Commun. ACM, 51 (1) (2007) 82–85.
[18] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, P. Hanrahan, WireGL: a scalable graphics system for clusters, in: International Conference on Computer Graphics and Interactive Techniques, Proc. 28th Annual Conference on Computer Graphics and Interactive Techniques, 2001, pp. 129–140.
[19] B. Jeong, L. Renambot, R. Jagodic, R. Singh, J. Aguilera, A. Johnson, J. Leigh, High-performance dynamic graphics streaming for scalable adaptive graphics environment, in: SC'06: Proc. 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL, 11–17 November 2006, p. 108.
[20] M. Brown (Ed.), Blueprint for the Future of High-Performance Networking. Commun. ACM (CACM) 46 (11) (2003) 30–77.

Playing with All Senses: Human–Computer Interface Devices for Games

JÖRN LOVISCACH

Fachhochschule Bielefeld, University of Applied Sciences, Fachbereich Ingenieurwissenschaften und Mathematik, Wilhelm-Bertelsmann-Straße 10, 33602 Bielefeld, Germany

Abstract

For a long time, computer games were limited to input and output devices such as mouse, joystick, typewriter keyboard, and TV screen. This has changed dramatically with the advent of inexpensive and versatile sensors, actuators, and visual and acoustic output devices. Modern games employ a wide variety of interface technology, which is bound to broaden even further. This creates a new task for game designers. They have to choose the right option, possibly combining several technologies to let one technology compensate for the deficiencies of the other or to achieve more immersion through new modes of interaction. To facilitate this endeavor, this chapter gives an overview on current and upcoming human–computer interface technologies, describes their inner workings, highlights applications in commercial games and game research, and points out promising new directions.

ADVANCES IN COMPUTERS, VOL. 77
ISSN: 0065-2458/DOI: 10.1016/S0065-2458(09)01203-0

1. Introduction

The quick success of Nintendo's remote controller for the Wii gaming console, see Fig. 1, after its launch in 2006 did not lead to the demise of the mouse and the joystick as game input devices. It exposed, however, a dire need to explore other options for user interface devices. Even though its technology as such cannot be considered novel, it gained breakthrough popularity through its ingenious combination of different sensors in one wireless box, the availability of many games that employ the new options, its very affordable price point (about US $40)—and its ''hackability.'' Quickly after its release, amateur software developers were able to reverse engineer the device's protocols to use it with standard desktop and notebook computers.

In the wake of this success, more and more novel input devices for games become available that turn technology that before was fragile and expensive into robust and inexpensive packages. For instance, haptic input/output became affordable with the Novint Falcon. And at press time of this book, Emotiv is expected to sell the EPOC, a brain-to-computer interface based on electroencephalography (EEG), for about US $300.

Several factors are key to this development. First, mass production can leverage the cheapness of the bare hardware, even though its development may be expensive; this effect also allowed consumer-level graphics cards to take the lead over highly specialized professional 3D rendering subsystems. Second, embedded processors within the devices as well as the computer's central processing units have gained

FIG. 1. The Nintendo Wii Remote—depicted here twice with the gyroscope add-on called Motion Plus—has triggered a new wave in tangible interaction. Image © 2008 Nintendo of Europe GmbH. Used with permission.

considerably in power, which facilitates tasks such as image processing. Third, accompanying technologies such as highly integrated chips (custom or configurable) and wireless connections have matured along with the processors and sensors.

Another driver of the trend to novel interfaces for games are mobile games as offered for cellular phones and for Personal digital assistants (PDAs). Even though these devices comprise keyboards (possibly on-screen), displays, and joysticks, their small form factor severely hampers usability. Additional ways of input and output are sought to enhance interaction.

This all has created a new playing field (pun intended) for tangible interaction [1] and multimodal interfaces [2], with a particular focus on pervasive games [3], that is, computer games that no longer detach the player from the physicality and the social relations of the real world. In the long run, the devices that now become available for gaming will also influence other strands of computer applications, possibly starting with music creation (see Bierbaum et al. [4] for an example and Blaine [5] for a survey), audio and video playback (see e.g., Tokuhisa et al. [6]), and pedestrial navigation (see e.g., Stahl [7]). This effect, too, has been visible with 3D graphics accelerators. Now that they are a standard facility in almost every computer, they are increasingly employed for nongame purposes as well, ranging from visually sophisticated windowing interfaces such as Apple OS X's Aqua and Windows Vista's Aero Glass to complex scientific applications based on frameworks such as Nvidia CUDA.

2. Typology

There is a number of ways to structure the vast field of user interface devices. A fundamental characteristic is the physical quantity upon which a device operates; this quantity may be measured (input) and/or changed (output) by the device, see Table I. Note, however, that the assignment of a given device to a physical quantity may be ambiguous: Does not an optical computer mouse rather sense the motion of the light pattern reflected from the underground than the position of the user's hand?

Another basic attribute to look into may be whether the control is discrete (a standard key can be either released or pressed) or continuous (a slider control can occupy virtually infinitely many positions). One may even want to introduce classes between these extremes. For instance, the shutter release button of most digital cameras has three states (idle; pressed half for focusing; pressed fully for releasing the shutter). A rotary knob may have 11 detents numbered from 0 to 10; a slider may possess an electronically limited resolution of 256 steps.

Game adaptation through emotion [8] raises another distinction: some input measurements may be controlled voluntarily by the user; others may not be subject to immediate voluntary control such as the heart beat rate or the electrical conductivity of the skin (''galvanic skin response''). It can even be the purpose of the game to use variables that the user cannot influence by will; or the objective may be to train the user in doing so through biofeedback.

A vital issue with a typology based on physical or other fundamental aspects of the interfaces is that a technique and its application in a game are only loosely coupled. It may easily be possible to substitute one technique with another to

TABLE I
INTERFACE DEVICES AND THE PHYSICAL QUANTITIES THEY ARE BASED ON

Physical quantity	Input device	Output device
Position and/or orientation	Keyboard, mouse, slider, joystick	Motorized slider
Radiance and/or reflectance	Camera input	Visual display
Sound pressure	Microphone	Loudspeaker
Force and/or acceleration	Force sensor, accelerometer, gyroscope	Force-feedback device, hydraulic chair, haptic device
Voltage and/or electrical current	ECG, EEG, EMG, galvanic skin response	Electrotactile display

This selection leaves out more exotic means such as air draft produced by fans.

achieve the same effect in gameplay. For instance, gesture input may be accomplished by camera, accelerometer, or mouse. The intelligent use of (simple?) hardware together with potentially sophisticated signal processing and imaginative game design renders many options exchangeable.

This chapter employs a pragmatic organization of the plethora of input devices according to categories of interface hardware:

- Buttons, keys, and keyboards
- Mice, joysticks, faders, and similar
- Pen and touch input
- Inertial sensors
- Cameras
- Specific position and orientation sensors
- Displays
- Audio input
- Audio output
- Tactile, haptic and related interfaces
- Kinetic devices and robots
- Biosignal sensors

This structure may help in figuring out alternative, possibly unorthodox uses of devices to fully leverage their potential. Every section outlines the working principle of the specific class of devices, points out historic, current, and upcoming implementations as well as existing and potential applications of those devices in games. This chapter mostly covers technology that is available to end users or close to that point. For further developments such as olfactory displays and for detailed usage guidelines see the survey on ''unconventional human–computer interfaces'' [9] and the collection on ''nontraditional interfaces'' [10].

3. Buttons, Keys, and Keyboards

Even something as simple as a pushbutton is employed by game designers in a large number of ways. At one extreme, a user may hold single buttons in his or her hand; at another extreme, he or she may have to step on a floor panel containing a dozen of buttons such as in the game ''Dance Dance Revolution'' (Konami, 1998). Even the mockup musical instrument for ''Guitar Hero'' (Red Octane, 2005) can be considered a number of buttons mounted in a sophisticated body. The game's

objective is to press combinations of five buttons with four fingers on the ‘‘guitar’s’’ fretboard and to move the ‘‘strum bar’’—technically a combination of two buttons—up or down with the other hand.

Technically, a standard typewriter keyboard is nothing more than a collection of pushbuttons. Keyboards have been a (if not *the*) standard input device for computer games, so that for instance using the keys W, A, S, D has become a *de facto* standard for moving about in space. Despite the long history, there is still innovation at least in three different fields. First, keyboards may be reduced to only contain the most important keys and do so in an ergonomic fashion. Sharkoon Rush Pad, for instance, comprises only 62 keys in an almost square layout, with the W, A, S, D keys being specially coated for better grip, see Fig. 2. Second, the keycaps may contain small displays to facilitate assigning special functions to them. This is exemplified by the Art Lebedev Studio’s Optimus Maximus keyboard (prohibitively priced at nearly US $1900). Third, the user may place the keys freely to construct an individual keyboard adapted to his or her playing style and/or to a specific game. In its product DX1, the manufacturer Ergodex employs passive radio frequency identification to allow the user to place solitary keys anywhere on a base panel, see Fig. 3. A different approach to free placement is to use a base panel with two conductive layers; buttons, sliders, and joysticks are equipped with embedded computers that connect with pins to those layers to receive power and to transmit encoded data [11].

FIG. 2. Sharkoon Rush Pad—a reduced keyboard for gaming—features enhanced keys for steering. Image © 2008 SHARKOON Technologies GmbH. Used with permission.

FIG. 3. The Ergodex DX1 system allows placing keys anywhere on its surface. Image © 2008 Ergodex. Used with permission.

In the realm of music, many more sophisticated pushbutton-type interfaces can be found, such as musical instrument digital interface (MIDI) piano keyboards or input devices that resemble clarinets or saxophones, the tone hole pads of which act like buttons. Some of these may make their way into games—if appropriately priced. For instance, Ion Audio manufactures the electronic drum kit ''Drum Rocker'' for about US $300 to be used with the game ''Rock Band'' (MTV Games, 2007)—or to be used for regular music making with appropriate additional electronics.

4. Mice, Joysticks, Faders, and Similar

Since its invention in 1967 by Douglas Engelbart, the operating principle of the computer mouse has not changed much, with the biggest addition to the original design possibly being the scrollwheel. Computer mice sense a position set by the user's hand. Thus, they can be grouped together with:

- Trackballs (inverted mice; much easier to handle in mobile settings)
- Joysticks (borrowed from an aircraft's cockpit and suited well for corresponding steering tasks)
- Rotary knobs (as in ball-and-paddle games like Atari's ''Pong'' from the 1970s)
- Steering wheels and foot pedals (as used for car simulations)

Some variants of these devices can be seen: A mouse can come in the shape of a gun handle and trigger such as the Zalman FPSGun. The Gyroxus Full-Motion Game Chair

priced at nearly US $600 is a full-body joystick controlled by the user's leaning in the intended direction. Mice, joysticks, and steering wheels are also available equipped with force feedback, see Section 12. Special gaming mice are marketed for their higher resolution (e.g., 1600 rather than 300 steps per inch) and alleged faster gliding. In cheap or miniature versions such as in mobile phones, joysticks may be implemented through a set of four pushbutton switches placed in up/down/left/right directions and operated by a small handle or a cross-shaped rocker. Pushbuttons, however, cannot directly control a continuous position, but only steer the direction of a cursor, which slows down the operation. A different option to create a miniature version of a joystick is to measure force instead of position. This is the operating principle behind IBM's TrackPoint input device mounted in the center of a notebook computer's keyboard. This device still cannot directly control a position, but at least allows the user to control a motion's speed in addition to its direction.

Mouse-like force input devices can also be applied to control 3D motion and rotation. The user clasps a handle which he or she can push or pull in every direction and twist along every axis. The handle only moves by fractions of an inch; only the applied force is important. Probably the most inexpensive of these devices is 3DConnexion's SpaceNavigator, priced below US $100. It comes with driver software that for instance supports Google Earth as well as many 3D design applications.

Again, inexpensive controllers used for computer music could be beneficial for games: ''Fader boxes'' offer up to dozens of sliders and/or rotary knobs. In more advanced versions, each rotary knob is implemented as a rotary encoder encircled by a ring of light-emitting diodes (LEDs). Similar to a computer mouse, the rotary encoder does not register absolute position but only the amount of motion that occurred. The computer uses the ring of LEDs to indicate the current setting. This design has a decisive advantage over traditional, analog rotary knobs. The software itself can change the apparent position of the controls simply by switching on a different LED. This is indispensable to recall presets.

Modern connectors for computer periphery such as USB (cable-bound) and BlueTooth (wireless) allow connecting different devices and/or several copies of the same device at the same time. Technically, every player in a group in front of a screen can hold his or her own game controller. Another option is to use one mouse for the left hand and another for the right hand. Even though operating systems such as Microsoft Windows will mix the data received from all mice to control the single standard cursor on the screen, the position data of the individual mice are available to software developers (through Windows' `RegisterRawInputDevices` function). Thus, proprietary software that switches off the standard cursor can make full use of several mice.

5. Pen and Touch Input

Pen-based interaction with a screen dates back to the 1950s, which is even further back than the invention of the computer mouse. Pen-based input never became popular with games on standard-sized consoles and PCs, even though graphics tablets are readily available—as *the* standard computer peripheral for graphic design. The reluctant use in games was not even changed by the advent of TabletPCs in which display screen and graphics tablet are united. ''Crayon Physics Deluxe'' by Petri Purho hints at the opportunities for game design, see Fig. 4. In the 2D world of the game, the player has to draw shapes that are subject to collisions and physical forces; the objective is to bring a given object to a predefined target position.

The situation is different with handheld computers (PDAs): Here, the pen is the premium input device for any application—including games. The Nintendo DS handheld gaming console possesses two screens, the lower of which also serves as pen input device, see Fig. 5. Note, however, that a simple, PDA-like stylus neither possesses buttons nor differentiates between different amounts of pressure; thus, the interaction repertoire is reduced in comparison to a mouse.

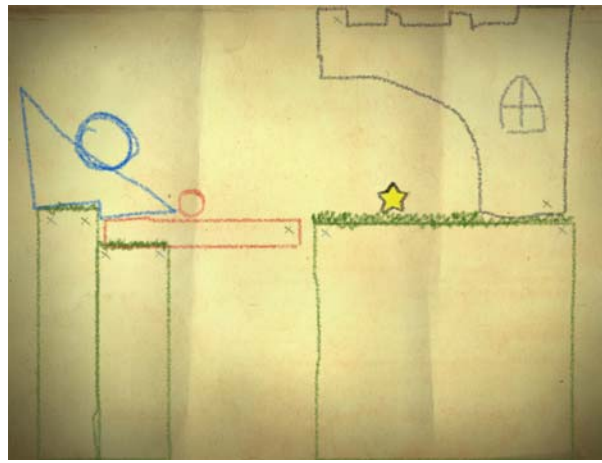

FIG. 4. In Crayon Physics Deluxe, the player can use a TabletPC to draw shapes that are subject to gravitational force and collisions with other shapes. Image © 2008 Petri Purho. Used with permission.

FIG. 5. The Nintendo DS mobile gaming console offers one standard screen and one touch-sensitive screen. Its updated version DSi—which is depicted here—also contains two cameras, one of which faces the user. Image © 2008 Nintendo of Europe GmbH. Used with permission.

Advanced pen input devices such as graphics tablets and the screens of most TabletPCs operate through radio frequency position detection and even employ active electronics in the stylus. In contrast, simple ''resistive'' touch panels use two slightly separated layers of transparent conductive but resistive plastic on top of the display screen. The touch with a stylus brings the two layers into contact below the stylus' tip. Voltage measurements can determine the position where the contact took place. These panels can also be operated by a bare finger, which is more natural, but less precise. A third touch panel technology is intended to be exclusively operated by a finger: capacitive sensing. Here, the input device senses at which locations the environment (i.e., the fingertip) can be electrically charged. This technology is standard for touch pads on notebook computers, but can also be found in more expensive touch panels.

Panels that can sense several fingers at the same time have been around at least since the 1980s. Since 2004, Jazzmutant sells the Lemur multitouch display intended for computer music applications. Its price of about US $2000 restricts its

use to the professional realm, however. In addition, it can only employ user interface elements that are predefined by the manufacturer. With its name changed from Jazzmutant to Stantum, this company, however, has begun to sell more versatile multitouch kits to developers.

In 2006, the demonstrations of Jefferson Y. Han and his group directed the public's attention toward multitouch technology. The first success of a multitouch device in the general market was the Apple iPhone, which can track four fingertips at the same time. The first games that make use of this functionality do not use multifinger gestures but rather track the two thumbs of the user while he or she holds the iPhone in landscape mode with both hands. Another broad application area may open with the successor of Windows Vista, as Microsoft has announced to support multitouch devices in the company's upcoming operating system version. Microsoft already manufactures the multitouch device Surface, which, however, is not intended for home gaming but for installation for instance in restaurants and shops. Hardly any of the Surface demos known at the time of the writing of this chapter address games.

Standard implementations of table-sized multitouch displays employ a horizontal projection surface below which the projector is mounted. In addition, the positions of the finger tips of (potentially many) users are tracked from below through infrared (IR) lighting and cameras. In addition to detecting the locations of fingertips, one can also recognize graphical identification patterns on the bottom of objects placed on a semitransparent screen. This allows distinguishing between several objects and determining their location and orientation. This is the basic sensing technology behind the ReacTable [12], originally intended for music making. However, the ReacTable sensing technology called ReacTIVision is freely available, so that many student projects have created games with it.

Multitouch screen technology is a highly active field of research. In 2008, Microsoft Research demonstrated an inexpensive solution called LaserTouch that employs an invisible sheet of IR light above a standard display screen; fingertips placed on the screen become visible to an IR-sensitive camera. In a joint venture, LG and Philips showed a 52-in. dual-touch display on CES fair 2008. In addition, LG, Planar Systems, and Toshiba Matsushita Displays develop standard-sized LC screens that possess a dense array of optical sensors within into the screen.

A highly different kind of multitouch screen can be applied to create an ''exertion interface.'' In the game ''Remote Impact'' [13], users punch or kick a wall-sized stuffed textile sheet, aiming to hit the shadow projection of a remote user. The hits are to be delivered with brute force in a boxing-game style. The elastic bands that connect to a stationary frame and hold the fabrics in place are equipped with stretch sensors. The data thus gained are used to compute a 2D force field.

6. Inertial Sensors

Inertial sensors respond to changes in linear motion. This category of input devices comprises two major classes: accelerometers, which sense linear acceleration, and gyroscopes, which sense angular velocity.

Accelerometers measure how quickly and in which direction the velocity changes at the current point of time. By the laws of physics, gravitation (the attraction of all objects to the earth's center) is invariably mingled with mechanical force: By physical experiments, one can not tell whether one rests on the earth's surface or whether one flies with a space rocket that continually accelerates at the right rate (standard gravity, $g = 9.81\ \mathrm{m/s^2}$). This phenomenon makes it hard to tell actual motion from gravity: A measurement of $1g$ can mean that the accelerometer is held still on the earth's surface; however, it could also mean that it is pushed with a speed growing in the right fashion.

If there is little user-generated acceleration, the sensitivity of accelerometers to gravity allows using them as tilt sensors that measure for instance by how many angular degrees and in which direction an object to which the accelerometer is affixed is off the vertical. Tilting the device may be more enjoyable than using a joystick, even though it is also more challenging [14]. A very intuitive gaming application of tilt sensing is to simulate water drops that flow downhill, depending on the amount and the direction of tilt. However, tilt can also replace a joystick control. Newer versions of the Guitar Hero controller (see Section 4) contain a three-axis accelerometer to determine for instance if the ''guitar'' is held upright to play a solo.

Today, accelerometers are relatively easy to build as part of integrated circuits. To this end, one etches a silicon chip in such a way that microscopic levers remain, forming a microelectromechanical system (MEMS). The levers are flexed by gravity or by kinetic acceleration. This way of production allows a price point of US $20 when bought in single units. Thanks to the efficient production, accelerometer chips typically contain three basic units, one oriented along each of the x, y, and the z axes (three-axis accelerometer). This allows determining a 3D acceleration vector.

Accelerometer chips are commonplace in mobile phones. Some models of notebook computers employ accelerometers to bring the hard disk's heads in a safe position should the computer be dropped. Mostly, the data of such accelerometers are available for other purposes such as tilt measurement in games. The most prominent device containing an accelerometer (among other sensors) may be Nintendo's Wii Remote controller, see Fig. 1. Current research deals for instance with the recognition of gestures based on its acceleration measurements [15]. Nintendo also sells a holder for this controller in the shape of a steering wheel. Here, the accelerometer is applied as a tilt sensor to measure the angle of rotation of the wheel.

In the basic game collection ''Wii Sports,'' the controller acts for instance as a tennis racket (with an optional racket-shaped holder) that distinguishes between different ways of hitting the ball, as bowling ball, or the virtual character's fist in a boxing game. The Wii Remote Controller can be extended by a second unit called Nunchuk. This device is to be used with the other hand. It is attached via wire and contains a joystick and another accelerometer chip. At press time of this chapter, Asus has announced to manufacture a similar tandem of input devices called Eee Stick. Both parts of the Eee Stick, however, are planned to communicate wirelessly.

Gyroscopes, the second major kind of inertial sensors, are based on miniature gyros: spinning objects. As one changes the orientation of a gyro's axis, it produces a counterforce, which can be measured to determine the angular velocity (technically for: rotation speed) of the orientation change. From the angular speed and the initial orientation one can, in principle, derive the current orientation. Rotating elements are not yet easily etched from silicon. Thus, gyroscopes produced as MEMS are typically based on microscopic units that are forced to vibrate in one direction. If the unit is subjected to rotation, the vibratory motion will slightly change the distance of the tiny moving mass to the center of the rotation. This induces a measurable lateral shift (Coriolis force).

Given the more complex technology, even single-axis gyroscopes easily cost US $100 when bought in single units, so that they are hardly found in consumer devices. This is bound to change with the Wii MotionPlus extension that Nintendo has announced for early 2009, see Fig. 1. At a price of approximately US $50, it contains a two-axis gyroscope and comes with a number of games that employ rotational motion for instance to start frisbee disks. Since 2006, Sony's SIXAXIS controller for its PlayStation 3 gaming console offers a three-axis accelerometer and a three-axis gyroscope. Even though instructions can be found on the Web how to use this device with PCs, it did not gain widespread acceptance in the PC world. This may be attributed to its form factor: It is intended to be held with both hands, which prohibits many of the activities the Wii Remote is used for.

To compute a device's orientation from the angular velocity output of a gyroscope is a complex process. First, the mathematics is sophisticated, as one cannot look at each axis on its own, which is a consequence of incommutativity. Typically, the result of two rotations applied in sequence will depend on which one of the two is applied first. Second and much harder, even small errors in the measurement of the angular velocity will quickly pile up and render the computed orientation meaningless. Such errors stem from imprecise calibration, poor resolution of the value (such as 256 steps), limited temporal resolution (such as 60 measurements per second), noise, nonlinearity, and momentary overloads due to impacts or quick motion.

Attempting to compute an object's position from accelerometer data runs into similar or even worse problems. In this case, one has the acceleration data, that is,

the rate of change of the velocity, not even the velocity itself. The standard solution for a reasonably stable determination of position and orientation through inertial sensors is to fuse the data of different sensors, typically through a Kalman filter, which weighs each information source by its reliability [16].

7. Cameras

Cheap webcams have become commonplace for video telephony and conferencing applications. Increasingly more notebook computers are sold with integrated cameras that capture the user's face. Many mobile phones and Nintendo's mobile console DSi, see Fig. 5, contain even two cameras: one as a standard digital still camera, the other directed at the user when he or she looks at the display.

The most lightweight use of a camera in a game may be to photograph bar codes or more advanced graphical codes [17] such as semacodes. Even a basic mobile phone can take the photos and decipher the codes. Such codes can for instance serve as proof that one player has found another player or a certain item, very similar to RFID chips, see Section 8. For instance, in the ''Assassin'' mode of the game ''Gridlockd'' (http://gridlockd.net/, 2005), all players carry an individual semacode pattern on their T-shirts and aim to eliminate their opponents from the game by taking photographs of the semacodes of these players.

Even in games, a camera may be used for video conferencing. The computer may also subject the camera's signals to image processing and higher level computer vision techniques such as pattern recognition. A popular input device for this purpose is Sony's EyeToy camera sold as an extension to the PlayStation 2 console; games employ this camera to show the user's mirror image immersed in game graphics and to recognize when the player hits on-screen targets. One may borrow notations developed in choreography to describe and analyze the user's movements [18].

Basic image processing can track real puppets so that their motion can be duplicated in a game engine [19]. A camera mounted below the ceiling, facing downward, makes it relatively easy to track the user's location and recognize arm gestures [20]. In a sports game with an actual ball, one can use two cameras mounted in different corners of the room to determine the heading and the velocity of the ball [21]. With two cameras mounted close together, one can apply stereoscopic analysis to recognize gestures for gaming [22]. Note that the use of connector ports such as USB and FireWire facilitates attaching several cameras at the same time to a single computer. However, if one software driver is used for both cameras—for instance because both stem from the same manufacturer—the driver has to support this mode of operation.

Cameras that capture the user's face may be employed to track the position of the head or—mode precisely—the two eyes, for instance to steer autostereoscopic displays (see Section 9.3). Current research work is directed at visually recognizing the user's emotional state [23] and capturing the face's motion [24], for instance to transfer the mimics to a avatar.

Handheld computers and potentially also future mobile phones with appropriate computational power offer another use of the integrated cameras. The motion of the device relative to its environment or specific objects or image patterns in the environment can be tracked. This supports augmented reality applications, such as the ''Invisible Train'' that drives around a real wooden miniature railroad track—if watched through a PDA's display held like a lense [25]. This is based on a port of the augmented reality framework ARToolKit to the PocketPC platform [26]. Video-based tracking of graphical markers also allows turning a handheld device into a paddle for a virtual table tennis game [27]. The cameras of handheld devices can capture regular street maps with a small number of markers to input locations for quests in a mobile game [28]. Without any markers, the camera of the handheld device can detect the motion of the foot of the user to let him or her kick a virtual ball [29].

With today's image processing on standard computers, the visible light reflected off the environment does not ensure a highly robust and precise tracking. The standard solution to this issue is to use IR light that either is emitted by LEDs mounted on the object to be tracked or is emitted by LEDs mounted around the camera and mirrored back by special retroreflective markers (spheres or flat dots) placed on the object.

The IR sensor that is part of Nintendo's Wii Remote controller is based on the former option. A ''sensor bar'' houses two IR beacons, each formed from several LEDs; an IR camera on the front of the Wii Remote locates these bright spots that are invisible to humans. Actually, this device does not transmit the camera image to the host computer, but only tracks and reports the locations of the brightest positions. These data can be used by the host computer to compute the position and orientation of the Wii Remote with respect to the sensor bar, up to some ambiguities, since two *xy* image points do not suffice to fully determine all six degrees of freedom, that is, *xyz* position plus three rotation angles. A more complex system was employed in the P5 data glove, which was introduced in 2002 by Essential Reality for game use and sold at about US $100. Eight IR LEDs were mounted all across the body of this data glove; a separated IR receiver comprised two cameras.

The use of IR light reflected off markers on the object to be traced is well established in professional motion capture systems, which are for instance employed to animate synthetic characters for movie production. NaturalPoint sells a relatively simple game input device that is based on this technology. The TrackIR 4:PRO,

FIG. 6. TrackIR 4 employs reflectors affixed to a cap or LEDs attached to a headset to determine the position and orientation of the user's head. Image © 2008 NaturalPoint. Used with permission.

priced at about US $180, see Fig. 6, tracks reflective markers on the user's head to determine its 3D position and orientation; these data may be used to adapt the computer screen's perspective to the position of the user's eyes. Professional-style IR motion capture cameras such as the NaturalPoint OptiTrack FLEX:V100 are now available for little more than US $500; this price tag, however, still seems to be too high for gaming applications. A little-used fact advertised by some tutorials on the Internet is that many inexpensive webcams can be turned into IR cameras by replacing their IR blocking filter (if present at all) by a filter that blocks visible light.

Instead of tracking body gestures or a face's mimics, one can even try to capture something as minute as the position of the eye's pupil. Such eye-tracking is a standard technique in user interface research to learn what the user is looking at. Most systems employ the reflection of IR light off the surface (cornea) of the eye's lens and from the light-sensing tissue (retina) in its interior. This is relatively straightforward in terms of technology and could in principle be sold at consumer-level prices. First tests on shooting games modified to work with an eye-tracker show promising results [30].

3DV Systems employs a completely different use of IR light in their DeepC chipset. After emitting a strobe of IR light, the camera measures how long the light takes to arrive on the sensor, thus effectively turning the camera into a 3D sensor that measures the depth of the scene for every pixel. Since light requires only 0.09 ns (a ten billionth fraction of a second) to travel by 1 in., such a measurement seems a

highly complex task. The trick is to use a high-speed shutter and measure the percentage of light energy that has arrived before the shutter has closed. 3DV Systems has announced to deliver a product called ZCam [31] intended to be used with games in late 2009. This camera is said to take regular images at a resolution of 1.3 megapixels and depth images at 320×240 pixels with a depth resolution of 1–2 cm. In games, this device may work particularly well for gesture recognition and to track a user's motion to animate an avatar [32]. Similar cameras are developed by Canesta and by Primesense. The z-Snapper system of the German manufacturer Vialux employs fringe patterns projected by a fast digital mirror device.

8. Specific Position and Orientation Sensors

Position and orientation or direction sensing is so vital to human–computer interfaces in general and to gaming interfaces in special that many more approaches than mouse, joystick, and their ilk as well as camera-based solutions have been invented. One of the interfaces popular with video games in the 1980s does no longer work with current screens: the light gun. It sensed a bright mark on the screen or—more precisely—the dot of the scanning beam inside a television's tube; this dot is too fast to be noticed by the human eye. Another technology that has mostly vanished from the market of interfaces devices is distance sensing through ultrasound.

Today's interfaces tend to be much more advanced, the simplest one possibly being an electronic compass, that is, a highly sensitive magnetometer that detects the direction of the earth's magnetic field. The basic, two-axis version of such a device suffices if one can guarantee that it is held horizontally flat. Otherwise, a three-axis magnetometer is required that can determine the full 3D field vector. Even though these devices can be built as integrated circuits with relatively simple external coils, their prices start at about US $60 when bought in single units.

Handheld devices allow the player to leave his or her seat and make real places part of the game activity. A major technical issue here is to track the players' positions as they may be scattered over several blocks or even all of a town. The satellite-based Global Positioning System (GPS) installed and maintained by the US Department of Defense has become the standard means to this end. GPS also forms the basis for car navigation systems, so that more and more handheld computers and mobile phones come equipped with a GPS receiver. Such receivers are also available as inexpensive add-ons that can be used with almost every computing device through a serial connection or a BlueTooth wireless connection.

GPS operates through precisely timed radio signals emitted by satellites; depending on the distance, the signals of different satellites will reach a receiver with

different delay. Knowing the timing of at least four satellites, the receiver can determine its 3D location. A similar principle underlies the Russian GLONASS system that is currently being (reinstalled and the planned Galileo system of the European Union.

Under best conditions, bare-bones GPS localization currently has an error of approximately 10 m. With helper signals, the error can be reduced dramatically [33]. Many GPS receivers employ the US-based Wide Area Augmentation System (WAAS) or the equivalent European Geostationary Navigation Overlay Service (EGNOS). These systems measure the GPS position errors at a number of base stations with known location and transmit region-specific correction data via satellites (hence ''Satellite-Based Augmentation System,'' SBAS). This correction brings the typical error down to a level of several meters.

Most GPS receivers cannot only output position data, but can also determine their velocity. These measurements are based on the Doppler effect, that is, the frequency shift introduced into the received signal through motion. This measurement essentially comes free of charge since the GPS receiver has to adjust its tuning to the Doppler shift, the major part of which is due to the motion of the satellite and the rotation of the earth. The data thus gained tend to be very accurate, the typical error being in the order of velocities of several inches per second.

GPS operates in a radio frequency band that is attenuated or reflected by walls; hence, GPS hardly works in buildings and may not work near high buildings. In case the reception of the satellites' signals breaks down, navigation systems can resort to ''dead reckoning'' based on last known location and direction and the current speed and/or inertial sensors (see Section 6 and Lachapelle [34]).

A location-sensing method that is less precise but more robust and works inside buildings is to use the cell structure of a mobile phone network (GSM or 3G). The identification code (cell ID) of the cell to which the phone is connected allows determining the location down to a single cell, which may have a diameter of a kilometer in a city's center, with 3G networks employing smaller cells. Time delay measurements either within the network infrastructure or with specially equipped phones can bring the error down to 50 m. Such localization abilities of mobile phones are propelled by national legislations concerning the precise localizability of emergency calls. Similar working principles—however, rather evaluating the differences in signal strength than in delay time—can be used on a smaller scale for wireless local-area networks (WLAN). For a study on results with GSM and WLAN see Varshavsky et al. [35]. A comprehensive survey on games employing cell IDs and other localization methods is given in Rashid et al. [36].

Instead of being able to locate the user at any time, it may be sufficient to know if he or she has reached a certain spatial station of the game. To this end, one can equip the stations with visual markings to be detected by a handheld camera, see Section 7,

or—less intrusively—equip the stations with radio transmitters. If these are few and are protected against theft, mains- or battery-powered embedded computers equipped with BlueTooth transceivers may work well (see for instance Bruns et al. [37]), totaling a cost of about US $100 per station. These can mostly be used with to handheld computers or mobile phones without any additions.

If, however, one has to build many stations in an uncontrolled area, the electronics they contain should be simple and cheap. No special hardware may be needed on the receiver's side if one installs beacons that send out high-power infrared light patterns [38], since many handheld computers are equipped with an IrDA port for communication through infrared light. This type of location sensing can be also achieved through radio frequency identification (RFID) chips [39], which originally were introduced to serve as intelligent replacements for bar codes on product packages. On the downside, virtually no consumer-class device comes with an integrated RFID reader. Even though bare-bones reader modules start at about US $30, commercially sold readers easily cost several hundred US-$.

More along their original purpose, RFID chips can be embedded into physical game objects such as playing cards. This provides a simple means to detect if one player has hunted down another one [40] or if the player possesses a specific item or has played a certain card, compare the Fruit Salad game of Jung et al. [41]. For applications of this type, *passive* RFID tags are ideal due to their lower price (US $1–2 when bought in single units) and due to their small reading range of up to 1 in. They are almost paper-flat and are maintenance-free because they do not include a battery but are powered by the radio field emitted by the reader. The German company PublicSolution has introduced board games with a controller called Yvio that recognizes the positions of the pieces on the board and guides the players through voice output.

Another close relative of position sensors is Nintendo's ''Wii Balance Board,'' a device that looks like a bathroom scale and contains four force sensors. These can be used to measure the weight of a user standing on it or to track the location of the center of mass when the user does physical exercises on the board. The Wii Balance Board is sold with a set of games for muscle building, yoga, and balancing, but may also be used as a steering device in a snowboard simulation.

9. Displays

The form factors of visual displays range from electronic goggles to cinema screens and room lighting. This section first looks at screens and projections and then at goggle-type personal displays. Stereoscopy (colloquially often called ''real 3D'') is treated in Section 9.3.

9.1 Standard Screens and Projected Displays

Whereas the typical computer screen is also the standard output device for games, some specific variants can be noted: Flight simulator aficionados mount a number of displays next to each other, leveraging the capabilities of modern computers to host several graphics cards, each with several video outputs. The same way of using multiple displays can also be found in professional flight simulators and in virtual reality environments. For enhanced immersion or to accommodate more than one user, wall-sized projected screens or completely surrounding projected screens may be used such as in CAVE systems [42]. To enhance the graphics performance, several computers may be networked instead of using one computer that would have to feed lots of video output ports. Thanks to frameworks such as VR Juggler [43], such systems can be built easily.

Possibly the simplest augmentation of a standard screen is to combine it with a computer-controlled room illumination system, as is part of the amBX system manufactured by Philips, see Fig. 7. The LED-based illumination contained in this system can automatically adapt the color of the room to fit to the background color displayed of the current image on the computer screen. Licensed developers may use the programming interface provided by Philips to create other effects. This system can be considered a minimum implementation of a ‘‘foveated display.’’ The human visual perception is only acute in the very center (fovea) of the retina and then quickly deteriorates. This ‘‘foveatisation’’ is not obvious because the brain steps in to create a seemingly sharp perception. Computer displays can make use of this

FIG. 7. Philips’ amBX system comprises speakers with high-power LEDs to illuminate the room, fans, and a rumbling rest for the user’s wrists. Image © 2008 Koninklijke Philips Electronics N.V. Used with permission.

effect to fake high resolution where there is none. Foveated displays provide detailed pictures around the area of interest and then—more or less gracefully—let the resolution deteriorate outside of that range.

9.2 Head-Mounted Displays

In terms of size, electronic goggles are the most efficient option for displays. Small screens based on liquid-crystal or organic-LED technology are placed right in front of each of the user's eyes, with a small intervening lens system letting the displays appear as a large screen in a TV-like distance. The working principle dates back to Ivan Sutherland's constructions in the late 1960s.

The trend to mobile entertainment through notebook computers and portable video players has lead to a simpler and affordable type of head-mounted displays (HMDs), albeit with poor resolution. Some of these offer a stereoscopic mode, see Section 9.3, only few possess the head-tracking sensors needed for believable virtual reality. As the user moves his or her head around, the images displayed to the eyes have to be changed accordingly. The head tracking data can also be employed to control audio processing. The positioning of sound sources in the headphones that typically are integrated with the HMD should correspond to the orientation of the viewer's head.

Even though they are highly portable and often are battery-powered, standard HMDs as such cannot be used in mobile settings because the user is essentially blindfolded. An option to solve this issue is to mount one or even two cameras on the frame of the HMD. The video stream(s) thus created are delivered to the HMD, with some additions or even replacements. Such a configuration supports gaming in Augmented/Mixed Reality (AR/MR). An approach that may be more lightweight in terms of technology is to use see-through displays, which overlay the computer-generated picture optically. Being specialized AR devices, such displays, however, are not yet affordable for end-user applications. AR/MR games may employ HMDs for instance to add fictitious extraterrestrial invaders to the environment [44].

9.3 Stereoscopy

To create a better 3D impression, one needs to display an image to the left eye that is slightly different from the one for the right eye. Such a ''stereoscopic'' presentation still is not perfectly realistic and is potentially disturbing, as the user's eyes focus a single screen depth and do not adjust (''accommodate'') to the virtual depth of the depicted objects. Nonetheless, stereoscopic displays are mostly considered ''real'' 3D. Whereas they are standard in professional virtual reality applications, they are known to home users mostly from well-equipped movie theaters. The broad

availability of digital projectors and the omnipresent use of computer animation and digital postproduction initiated a renaissance of 3D movies, which already had a big time in the 1950s. Many cinema-goers may experience 3D stereoscopy in the near future and ask for similar displays in games.

The most efficient and effective solution for stereoscopy consists in HMDs, see Section 9.2, as one can address the left and the right eye without any crosstalk between them. A less costly route is to use a monitor or projector that displays the images for the left and the right images in quick sequence; the user wears computer-controlled liquid-crystal-based ''shutter glasses'' that alternatingly block each eye. To prevent flicker, the display unit needs to support frame rates in excess of 120 frames per second. Current projectors and monitors—apart from outdated cathode-ray tubes—mostly offer no frame rates that are substantially larger than 60 frames per second; hence, shutter glasses are currently no practical solution for home users.

The video signal required for most HMDs is the same as that for shutter glasses: a stream of images each of which is alternatingly intended for the left and for the right eye. This allows to use a single standard video connector (typically, still of VGA type), but cuts the effective frame rate down to 30 frames per second. This does not lead to flicker, however, as the displays in front of the left and the right eye do not go black—as shutter glasses would do—when it is the other display's turn. Instead, each display keeps its last image.

In the professional realm, stereoscopy via one video connector is done on ''quad-buffer'' OpenGL graphics cards, a term which refers to the graphics card operating with independent double-buffering for each eye. (Double-buffering means that the graphics card outputs one image frame previously stored while the graphics chip computes the next one.) Such cards are also equipped with a three-pin Mini-DIN connector to control shutter glasses. Standard gaming graphics cards only offer double-buffering or triple-buffering for one eye per video output.

Nvidia offers an inexpensive solution for stereoscopy throughout its line of graphics chips. The graphics chips can send a control signal for shutter glasses via the VGA connector, controlled by Nvidia's ''GeForce 3D Stereo Driver,'' which at the time of writing is only available for Windows Vista in an up-to-date version. This driver supports shutter glasses, HMDs, and some types of autostereoscopic displays (see below). Game software running on the computer only sees a standard—so to speak monocular—graphics card; behind the scenes, however, the driver creates stereoscopic image pairs. To this end, it extracts 3D data from the command stream sent to the graphics processor and builds views for the left and for the right eye. These views are rendered and output alternatingly, frame by frame. This transparent solution works with many games, but may fail for some special visual effects that do not supply correct 3D depth data.

The standard solution employed for stereoscopy in movie theaters is often based on polarization filters. Two projectors, each with separate video inputs, run in parallel at the standard frame rate, the first projector for the left-eye images, the second for the right-eye ones. Each projector's light beam is fed through a different polarization filter and directed onto the same, common screen. The silver screen material is chosen so that the light's polarization will not change as the light is reflected toward the audience. The audience wears glasses with different polarization filters in front of their left and right eyes that suppress the light that stems from the opposite projector. The dual projection systems required for this solution are already available in form factors that resemble standard data projectors and direct both beams through one joint lens. Currently priced starting at almost US $4000, stereoscopic projectors are, however, not yet a choice for the home.

''Autostereoscopic'' displays are intended to overcome the use of goggles and bulky head-mounted gear. They create hologram-like images that can be viewed with the naked eye. Half a dozen of such technologies are already in the marketplace, even though none has yet had a commercial breakthrough. Most autostereoscopic displays are based on a thin layer of approximately pixel-sized lenses or blocking stripes on the front of a standard liquid-crystal display. The lenses or blockers direct the light of different pixels into different spatial directions, effectively cutting the screen's resolution at least by half. Some of these devices are supported by Nvidia's 3D Stereo Driver mentioned before.

In most implementations of autostereoscopic displays, either the user has to stand in a particular place or the display has to adjust the lens or blocker geometry automatically with the help of a head tracker. Such displays typically only support one user at a time. This limitation can be overcome for instance through displays that use more sophisticated blocker screens to send different images into, say, eight spatial directions. However, this induces a strong loss in the brightness and in the effective image resolution available in every single of the different directions.

10. Audio Input

The primary use of audio input is player-to-player communication in games. In addition, games such as ''SingStar'' (Sony Computer Entertainment, 2004) have popularized a karaoke-style mixing of the user's singing with accompanying music, awarding scores based on an automatic analysis of the rhythmic and tonal quality as well as the expressivity. In principle, games could also easily make use of speech recognition for hands-free command input. The necessary recognition routines are part of today's leading operating systems. Nonetheless, speech recognition is rarely

used—in games as well as in standard applications. Some notable exceptions from this rule are the games ‘‘SWAT’’ (Vivendi Universal, 2003) and ‘‘Tom Clancy’s Rainbow Six 3’’ (Ubisoft, 2003) that use speech recognition to allow the player to issue commands. OC3 Entertainment employs speech recognition in its product FaceFX Live to realistically animate the face of an avatar that represents the user.

More experimental uses of audio input have been proposed in research but have not yet made their way into games or other applications. The time delay of the sound of finger snapping to a number of microphones can be applied to localize the user [45]. If the user blows onto a notebook computer’s screen, the target location of the blowing is detectable through the built-in microphone [46]. The acoustic analysis of speech can reveal the emotional state of the user [23], which could be employed for instance to adapt the difficulty level of a game.

11. Audio Output

Audio output may be employed for better immersion, like the soundtrack of a movie. It may, however, also serve as the main medium of output. Audio-only games [47] can be used by visually impaired persons or they can be used in places where watching a computer screen is distracting (e.g., when walking across the street) or too clumsy (e.g., when sitting in a bus).

Television sets and computers are used both to play games and to watch movies. Due to their support by DVD-based videos, surround-sound systems are available in many homes and thus can be used for games as well—if the users did not even buy them for gaming in the first place. The standard setup is still ‘‘5.1,’’ which comprises six speakers: one above, below or behind the screen, one left and one right from it, two speakers in the back, and one subwoofer.

The large number of speakers is required to ensure a consistent spatial impression even when for listeners who are not placed in the center spot of the arrangement. Theoretically, two speakers would suffice to provide a full three-dimensional sound image, as a human has only two ears. This idea, however, faces severe practical problems. A technique widely supported in audio programming frameworks such as OpenAL and Microsoft DirectSound is the use of head-related transfer functions (HRTFs) for headphone output. Depending on the 3D position of a virtual sound source relatively to the user, the software computes how the sound would arrive at the user’s ears. This result is output via headphones. The ‘‘how’’ comprises:

- *Delay*: Due to the limited speed of sound, the signal reaches one ear before the other.

- *Attenuation*: The spatial expansion of the acoustic waves and the damping by the travel through the air reduce the level. Most prominently, the listener's head itself often shadows one ear from the sound source.
- *Filtering*: The complex shape of the outer ear (''pinna'') boosts or cuts specific frequencies, depending on the direction of the incoming sound wave. This is described by HRTFs, which may also include the effects of delay and attenuation.

With the inclusion of HRTFs, an immersive 3D sound reproduction can in principle be achieved through stereo headphones. Consumer-level systems are, however, far from this goal. With them it is still not easy to tell if a sound is placed in front of the listener or if it is placed in his or her back; and it is virtually impossible to tell a sound position above the head from one below. The main reason for this is that the pinna shape is highly individual, so that standardized HRTFs as offered by typical software do not work well. One needs to use customized HRTFs, which are very laborious to measure. Another issue is that small involuntary motions of the head are vital to spatial sensing. Headphones, however, glue the sound field to the listener's head. This can be overcome through head tracking. As the listener rotates the head, the presented sound field is rotated in reverse direction. An integrated solution for audio production, the Beyerdynamic Headzone PRO headphone system employs ultrasound to determine the orientation of the user's head and adjusts the sound field accordingly. Some less expensive surround-sound headphones for games are not based on HRTF simulations but actually use several spatially distributed loudspeakers in the earpieces.

Since decades, audio researchers have looked into realistic 3D sound reproduction. The Ambisonics system, for instance, is intended to model the local behavior of the sound field at a given location. Wave field synthesis, a more general approach, attempts to create a holographic sound image in a larger area. Both systems rely on large arrays of loudspeakers. Up to now, such advanced audio solutions have rarely made their way into consumer-level products. One exception, however, is the series of ''digital sound projectors'' built by Yamaha. These contain up to 42 single loudspeakers in a single box to be mounted below the TV screen. The multitude of speakers is employed to create acoustic beams that are reflected off the walls and hence reach the listener from the sides and from the back.

12. Tactile, Haptic, and Locomotion Interfaces

All interfaces that employ touch can be called ''tactile,'' which at its extreme includes devices as simple as a sensor operated by touching. Generally, the term ''haptic'' is reserved for more complex devices that also possess kinesthetic aspects

such as position sensing. Resembling audio output devices, there are two major strands of applications of tactile and haptic devices: First, they can improve immersion (e.g., one can grasp and feel virtual objects); second, they provide a different channel to the user that may be employed when other interactions modes are not feasible due to the situation or due to a handicap. Thus, tactile and haptic interfaces are found in mobile settings as well as in assistive technology such as braille readers for the visually impaired. For a survey see Eid et al. [48].

This group of interface devices addresses both input and output. Mostly, they can be seamlessly combined such as in the case of a steering wheel that requires increasingly more force for larger turning angles and that shakes when a tire of the virtual car plunges into a pothole. The most basic device may be a motor that rotates an unbalanced mass. Such vibrators are used for the silent alert of mobile phones but also in simple force-feedback computer mice, joysticks, and steering wheels. Vibrators can be found in the Wii Remote controller as well as in the wrist rest of Philips' amBX system. Several manufacturers of mobile phones build in Immersion's VibeTonz system that allows a fine-grained design of vibration responses. A momentary vibration is particularly beneficial as feedback for touch panels or pen-based input devices, for instance to signal that a finger hovers above a button on a screen or to inform the user that a key has been pressed [49].

Rather than only presenting vibrations or pushes, many force-feedback joysticks can exert a defined amount of force in a specified *xy* direction. This is typically accomplished through Microsoft's DirectInput programming interface. Even though standard gaming joysticks are rather imprecise, their functionality already hints at the features of professional haptic input devices that allow the user to explore a 3D space while feeling for instance the contact forces of surfaces that are flexible, slippery, sticky, rough, or have a high friction. Such devices are used for virtual reality applications such as surgeon training as well as for computer-aided 3D construction and design.

With the Novint Falcon, precise haptic feedback has arrived in the arena of games. At a price of about US $200, it allows the user to move a grip within a volume with a diameter of approximately 4 in.; three motors can exert directed forces of up to the weight force of 2 pounds. The Falcon comes with software that for instance simulates the recoil of a gun or the weight of a fishing rod with a fish dangling on the line. At the time of writing, only the *xyz* position plus the state of the buttons were input and only a linear 3D force was output by the device. In principle, the modular design allows more parameters to be recorded and/or influenced by replacing the grip with a more sophisticated one. According to the manufacturer, such alternate grips are under development. Expensive professional haptic devices record for instance not only the position but also the orientation of a pen held by the user and may apply translational as well as rotational forces to it.

On the experimental side, researchers have employed motorized faders (i.e., slide controllers) for cheap but precise haptic feedback. These devices stem from sound mixing consoles where they allow the instant recall of the settings made for a specific piece of music. The slider can be used as a position sensor, where—which is used in a mixing console—the motor can drive the slider to a given position or—which can be used for haptics—the motor can exert a force onto the finger with which the user touches the slider's lever. In a game, this can for instance be applied as an interface for a catapult [50].

To create a perception of touch, one can apply mechanics—as described before—or one can try to stimulate the receptor cells in the skin by electrical pulses of several hundred volts strength and lengths in the order of a millisecond. Such ''electrotactile displays'' can easily display tactile patterns over larger areas, much easier than mechanical solutions. They are researched into particularly as a means to support visually impaired persons. Regularly, electrotactile displays do not believably simulate a relief on the surface; the sensation they cause rather resembles ''pins and needles.'' This may explain their rare use.

Haptic devices are not confined to the human hand. More exotic applications of computer-delivered pushes have been devised: TN Games manufactures the 3rd Space Vest, sold for approximately US $140, that allows gamers to feel impacts on their body, see Fig. 8. This is realized through eight pneumatic pads in the vest that are driven by a compressor, controlled through a freely available software development kit.

Many professional installations of Virtual Reality systems try to maintain the illusion of large-scale locomotion. The user believes to walk or run over a seemingly unlimited terrain. To realize this within a spatially confined system, techniques are required to undo the user's motion. For a unidirectional motion, a conveyor belt suffices that is synchronized to the user's gait. For two-dimensional motion, one may for instance enclose the user in a huge sphere that can rotate freely on a mounting base. The base may include motion sensors, such as in the case of VirtuSphere Inc.'s VirtuSphere, which is sold in single units to professional users. Also prohibitively expensive for home gaming, hydraulic seats such as the D-Box GP PRO-200 series push, pull, and tilt the user's seat to simulate the joggling on a rough road or the *g*-forces during a jet plane's U-turn. At the other end of the price scale one finds such systems as the SmartCycle manufactured by Fisher-Price. This home-trainer style stationary bike contains a computer to be connected to a TV set. Riding the bike, children can take tours through ''Math Mountain,'' along ''Letter Creek,'' or engage in other educational games.

The ''Artwork formerly known as PainStation''—renamed due to legal issues—by Volker Morawe and Tilman Reiff whips each of the two players depending on the process of the game. On the fringe of the domain of tactile interfaces lies the output

FIG. 8. The 3rd Space™ vest provides computer-controlled pushes to the user's torso to simulate impacts. Image © 2008 TN Games®. Used with permission.

of an air draft. The computer-controlled fans of Philips' amBX system, see Fig. 7, are intended to provide better immersion for instance in racing games. It has to be noted, though, that already the Videorama system of the 1950 used wind. For more background and a current use example see Moon and Kim [51].

13. Kinetic Devices and Robots

Currently, tangible user interfaces mostly address *input*. The *output* of physical motion [52], however, is only rarely used in common computer applications. It can be accomplished through simple actuators such as servo motors or through devices as complex as a humanoid robot.

Puppets equipped with sensors and actuators may for instance be used in a boxing game, even though the implementation put forward by Shimizu et al. [53] mostly concerns motion input; the output consists of vibration. Extending the idea of tracking the users' actions from below a translucent table (see Section 5), one can employ the projector to control palm-sized robots placed on the surface. The computer produces specific graphical patterns below every robot; the robots detect these patterns through light sensors and move in a way that is prescribed by the respective pattern [54].

As opposed to industry robots, many robots for the home can be considered ''smart toys,'' and hence games, even though the objectives and the rules of the game are not rigidly defined. From 1999 to 2006, Sony manufactured three different generations of the robotic dog AIBO, which sold for approximately US $2000. A related, but simpler product currently sold is the dinosaur-shaped Ugobe Pleo, priced at about US $350, see Fig. 9. Whereas motion output is the primary interaction mode of such robots, they contain lots of sensors to autonomously generate plausible behavior. Both AIBO and Pleo can be programmed with the help of freely available software development kits. This allows such applications as the AIBO turning its head to a talking human and produce idle motion to simulate intense listening [55].

FIG. 9. Smart toys such as Pleo, an autonomous dinosaur-shaped robot, can be considered a special kind of computer game interfaces. Image © 2008 UGOBE. Used with permission.

14. Biosignal Sensors

In the realm of Affective Computing, a number of biosignals has been put forward for automated emotion detection. These comprise the electrical conductivity of the skin (‘‘galvanic skin response’’), the heart beat rate as well as the rate and the volume of breath. For a study of short-term responses of a number of signals see Ravaja et al. [56]. Such signals can be employed to adapt a game to the emotional state of the user [8] or to use that state to control the game [57]. In a biathlon-type game [58], the heart beat rate may increase skiing speed but reduce shooting precision. Similar signals can also be applied for relaxation purposes. The biofeedback game ‘‘The Journey to the Wild Divine: The Passage’’ (Wild Divine Project, 2003) comes with three finger sensors to measure skin conductivity and heart rate.

The electrical potentials of the brain present a much more complex vein of biosignals. Their widespread use was long hampered by the complex pattern analysis required and by the cumbersome and expensive EEG technology needed to measure them. About a dozen up to more than hundred electrodes are planted in a defined geometric pattern on a subject’s head. Typically, gel is applied below the electrodes to improve contact. The electrodes’ output amounts to voltages as small as the millionth fraction of a volt; it is fed into highly sensitive amplifiers that contain strong filters to suppress power line hum.

The waveforms of EEG data easily give away if the user is awake and if his or her eyes are open. Parameters such as the general alertness of the user may be extracted in addition to parameters he or she can easily control voluntarily. Both types of parameters can be exploited in game design [59]. Active triggers comprise the response to visual input or to a rare event, imagined motion, and movement preparation [60]. Since such activities of the brain are not easy to recognize with help of a computer, one typically employs training. First, machine-learning algorithms on the computer can be trained by presenting known data from a user. Second, the user can learn to adapt to the system; similar to biofeedback training, the system informs the user how well he or she performs.

Only adapting the machine to the user and not vice versa is hard: Krepki et al. [61] employ a cap with 128 electrodes plus further measurement channels. A simpler solution may be to train both the user and the system to cooperate. Together with high integration, mass production, wireless and gel-less electrode caps this could pave the way to the use of brain–computer interface techniques in games. This is the objective of the manufacturer Emotiv that plans to sell the EEG-based input device called EPOC in 2009 for a price of less than US $300, see Fig. 10. In the demonstrations given so far, the 16-electrode system employed a game-like scenario to adapt the user to the system and vice versa. It comes with driver software that can signal

FIG. 10. The Emotiv EPOC hardly resembles the bulky EEG caps used in traditional systems. Image © 2008 Emotiv Systems. Used with permission.

cognitive intent such as lifting an object, basic emotions such as excitement and facial expressions such as eye winking and smiling. A basic version of the software development kit needed to build application software that leverages these data is available free of charge. A gyroscope in the cap may detect nod and shake motion.

Also the manufacturer NeuroSky announced a series of EEG-based gaming input devices, called ThinkGear, MindSet, and MindBuilder. According to the little information that is publicly available at press time, these employ a single EEG channel and output percentage levels of brain states called ''attention'' and ''meditation.''

A sibling to EEG, electromyography (EMG) records the signals controlling a muscle. It may for instance be used to create subtle interfaces that can be operated for instance by tensing the biceps without moving the arm, like in an isometric

FIG. 11. OCZ Technology Group's Neural Impulse Actuator comprises a headband with three electrodes. Image © 2008 OCZ Technology Inc. Used with permission.

exercise [62]. OCZ Technology Group's Neural Impulse Actuator, sold for about US $130, employs three sensors in a headband to collect a mixture of EEG and EMG data, see Fig. 11. The software accompanying the device allows mapping the signals to keystrokes, simulating joystick motion and hence controlling almost any software.

Games rarely deal directly with the inner chemistry of the human body. The game Glucoboy (Guidance Interactive Healthcare, 2007) is exceptional in this respect. To be used by young gamers who suffer from diabetes, it comes with a blood glucose meter that grants access to games or rewards the user with game currency depending on the measurement results.

15. Conclusion

Games are not as strictly tied to well-known modes of interaction as standard applications are. They can more easily break with traditional interface devices. And if they do so, the huge market volume allows to demonstrate otherwise expensive technologies at affordable prices. In some cases, this can open up broader application domains outside of gaming, first due to availability, second due to the public's familiarity with the device. Thus, gaming applications can act like both an eye opener and a dam breaker. At the time of writing, however, this route remained hypothetical as even the Wii Remote controller or the use of accelerometers in

general has not made it into standard applications, not even on handheld devices. Even worse, many attempts to create novel game interfaces such as the P5 data glove failed already in the gaming market itself.

One may speculate about the success factors. A device has advantages in the marketplace if it is cheap enough so that prospective users do not mind losing their investment if they find that it does not work for them. Another decisive point is the support by a substantial number of games or other applications. Hoping to address this issue, many manufacturers open up their communication protocols and/or distribute free software development kits. This may, however, only result in a number of homegrown software programs that do not address a larger audience. An often overlooked aspect of a device's usability is the setup required. For casual games [63], even plugging in a cord may be too much, let alone finding a place where to put a secondary box such as with the discontinued P5 data glove. Wireless technology such as BlueTooth or its new siblings ZigBee and Wireless USB may be a key enabler here.

Wireless data transmission is also vital in other respects: First, the physicality introduced into games by some of the new controllers requires wireless connections to the computer. This trend may grow up to the point where a number of interface sensors and possibly also actuators are placed all over the user's body [64]. Second, wireless connections also allow users to connect on the fly, for instance for casual gaming on the commuter train or for group gaming in front of the TV set in the home. Mobile consoles such as Nintendo DS and Sony PlayStation Portable (PSP) are equipped with WLAN transceivers—as are some mobile phones. Almost all mobile phones support BlueTooth.

Game programmers are trained to work with input via joystick and keyboard and output via graphics and audio. Most professional development is based on game engines that wrap around the basic functions and offer a set of high-level routines for tasks such as character animation or on-cue sound playback. The novel interface devices presented in this chapter are, however, only rarely backed by high-level programming interfaces. And even where this is a case, the programming interfaces are not integrated with a standard game engine. For instance, triggering the playback of a sound, starting a vibratory signal, and changing the room's background illumination should all be part of a single game event managed by the engine. This vision calls for middleware solutions that address a broader range of input and output devices to relieve game programmers from two laborious tasks: first, to deal for instance with the mathematics of signal processing and pattern classification; second, to integrate the different input channels and output channels, which may also comprise the fusion of data from different sensors for more robust recognition. The first task is already taken up by companies such as AiLive, who create gesture recognition middleware, and HaptX, who have developed a haptics engine.

On the side of game design, the novel options require a fresh and deep look at what a human being is able to sense and to do, see Fisher et al. [65]. For instance, computer-based workout games were unthinkable in the years of the Pong game, but nowadays game designers have to look into the short- and long-term effects of physical exercises. A second key issue for game designers is to combine different interfaces in an effective and efficient way, which mostly will also be a natural and intuitive way. Where once the mouse was used to select items and invoke a context menu, one can now point with one's finger onto a table-sized screen and speak a command. This is applicable to control troops in a command-and-conquer-style strategy game or to characters in The Sims [66].

REFERENCES

[1] H. Ishii, Tangible bits: Beyond pixels, in: TEI'08: Proceedings of the International Conference on Tangible and Embedded Interaction, ACM, New York, USA, 2008, pp. xv–xxv.
[2] A. Jaimes, N. Sebe, Multimodal human-computer interaction: A survey, Comput. Vis. Image Underst. 108 (1–2) (2007) 116–134.
[3] C. Magerkurth, A.D. Cheok, R.L. Mandryk, T. Nilsen, Pervasive games: Bringing computer entertainment back to the real world, Comput. Entertain. 3 (3) (2005) 4.
[4] G. Essl, M. Rohs, Shamus—A sensor-based integrated mobile phone instrument, in: ICMC'07: Proceedings of the International Computer Music Conference, 2007.
[5] T. Blaine, The convergence of alternate controllers and musical interfaces in interactive entertainment, in: NIME'05: Proceedings of the Conference on New Interfaces for Musical Expression, National University of Singapore, Singapore, 2004, pp. 27–33.
[6] S. Tokuhisa, Y. Iwata, M. Inakage, Rhythmism: A VJ performance system with maracas based devices, in: ACE'07: Proceedings of the International Conference on Advances in Computer Entertainment Technology, ACM, New York, 2007, pp. 204–207.
[7] C. Stahl, The roaring navigator: A group guide for the zoo with shared auditory landmark display, in: MobileHCI'07: Proceedings of the International Conference on Human-Computer Interaction with Mobile Devices and Services, ACM, New York, USA, 2007, pp. 383–386.
[8] T. Saari, N. Ravaja, J. Laarni, M. Turpeinen, Towards emotionally adapted games based on user controlled emotion knobs, in: DiGRA'05: Proceedings of the International Conference of the Digital Games Research Association, 2005.
[9] S. Beckhaus, E. Kruijff, Unconventional human computer interfaces, in: SIGGRAPH'04 Course Notes, 2004.
[10] P. Kortum (Ed.), HCI Beyond the GUI. Design for Haptic, Speech, Olfactory, and other Nontraditional Interfaces, Morgan Kaufmann, Burlington, MA, 2008.
[11] N. Villar, K.M. Gilleade, D. Ramdunyellis, H. Gellersen, The VoodooIO gaming kit: A real-time adaptable gaming controller, Comput. Entertain. 5 (3) (2007) 7.
[12] S. Jordà, G. Geiger, M. Alonso, M. Kaltenbrunner, The reacTable: Exploring the synergy between live music performance and tabletop tangible interfaces, in: TEI'07: Proceedings of the International Conference on Tangible and Embedded Interaction, ACM, New York, 2007, pp. 139–146.
[13] F. Mueller, S. Agamanolis, M.R. Gibbs, F. Vetere, Remote impact: Shadowboxing over a distance, in: CHI'08: Extended Abstracts on Human Factors in Computing Systems, ACM, New York, 2008, pp. 2291–2296.

[14] S. Bucolo, M. Billinghurst, D. Sickinger, User experiences with mobile phone camera game interfaces, in: MUM'05: Proceedings of the International Conference on Mobile and Ubiquitous Multimedia, ACM, New York, 2005, pp. 87–94.
[15] T. Schlömer, B. Poppinga, N. Henze, S. Boll, Gesture recognition with a Wii controller, in: TEI'08: Proceedings of the International Conference on Tangible and Embedded Interaction, ACM, New York, 2008, pp. 11–14.
[16] D. Jurman, M. Jankovec, R. Kamnik, M. Topic, Calibration and data fusion solution for the miniature attitude and heading reference system, Sens. Actuators A Phys. 138 (2) (2007) 411–420.
[17] E. Toye, R. Sharp, A. Madhavapeddy, D. Scott, E. Upton, A. Blackwell, Interacting with mobile services: An evaluation of camera-phones and visual tags, Pers. Ubiquitous Comput. 11 (2) (2007) 97–106.
[18] L. Loke, A.T. Larssen, T. Robertson, Labanotation for design of movement-based interaction, in: IE2005: Proceedings of the Australasian Conference on Interactive Entertainment, Creativity & Cognition Studios Press, Sydney, Australia, 2005, pp. 113–120.
[19] D. Hunt, J. Moore, A. West, M. Nitsche, Puppet show: Intuitive puppet interfaces for expressive character control, in: Mediterra 2006: Proceedings of the International Conference on Gaming Realities: The Challenge of Digital Culture, 2006, pp. 159–167.
[20] S. Laakso, M. Laakso, Design of a body-driven multiplayer game system, Comput. Entertain. 4 (4) (2006) 7.
[21] F.F. Mueller, G. Stevens, A. Thorogood, S. O'Brien, V. Wulf, Sports over a distance, Pers. Ubiquitous Comput. 11 (8) (2007) 633–645.
[22] Y. Wang, T. Yu, L. Shi, Z. Li, Using human body gestures as inputs for gaming via depth analysis, in: ICME 2008: Proceedings of the International Conference on Multimedia and Expo, 2008, pp. 993–996.
[23] Z. Zeng, M. Pantic, G.I. Roisman, T.S. Huang, A survey of affect recognition methods: Audio, visual and spontaneous expressions, in: ICMI'07: Proceedings of the International Conference on Multimodal Interfaces, ACM, New York, 2007, pp. 126–133.
[24] E. Vendrovsky, I. Neulander, Markerless facial motion capture using texture extraction and nonlinear optimization, in: SIGGRAPH'06: ACM SIGGRAPH 2006 Sketches, ACM, New York, 2006, p. 27.
[25] D. Wagner, T. Pintaric, D. Schmalstieg, The invisible train: Collaborative handheld augmented reality demonstrator, in: *SIGGRAPH'04: ACM SIGGRAPH 2004 Emerging Technologies*, ACM, New York, 2004, p. 2.
[26] D. Wagner, D. Schmalstieg, First steps towards handheld augmented reality, in: ISWC'03: Proceedings of the 7th IEEE International Symposium on Wearable Computers, IEEE Computer Society, Washington, DC, 2003, p. 127.
[27] A. Henrysson, M. Billinghurst, M. Ollila, Face to face collaborative AR on mobile phones, in: ISMAR'05: Proceedings of the IEEE/ACM International Symposium on Mixed and Augmented Reality, IEEE Computer Society, Washington, DC, 2005, pp. 80–89.
[28] O. Rath, J. Schöning, M. Rohs, A. Krüger, Sight quest: A mobile game for paper maps, in: Intertain (Ed.), INTETAIN 2008: Adjunct Proceedings of the 2nd International Conference on Intelligent Technologies for interactive entertainmen, 2008.
[29] V. Paelke, C. Reimann, D. Stichling, Foot-based mobile interaction with games, in: ACE'04: Proceedings of the ACM SIGCHI International Conference on Advances in Computer Entertainment Technology, ACM, New York, 2004, pp. 321–324.
[30] E. Jönsson, If looks could kill—An evaluation of eye tracking in computer games, Master's Thesis at the School of Computer Science and Engineering, Royal Institute of Technology, Stockholm, 2005.

[31] G. Yahav, G.J. Iddan, D. Mandelboum, 3D imaging camera for gaming application, in: ICCE'07: Digest of Technical Papers of the International Conference on Consumer Electronics, 2007.
[32] Y. Pekelny, C. Gotsman, Articulated object reconstruction and markerless motion capture from depth video, Comput. Graph. Forum 27 (2008) 399–408.
[33] B. Eissfeller, G. Ameres, V. Kropp, D. Sanroma, Performance of GPS, GLONASS and Galileo, in: Photogrammetrische Woche, 2007, pp. 185–199.
[34] G. Lachapelle, Pedestrian navigation with high sensitivity GPS receivers and mems, Pers. Ubiquitous Comput. 11 (6) (2007) 481–488.
[35] A. Varshavsky, M.Y. Chen, E. de Lara, J. Froehlich, D. Haehnel, J. Hightower, A. LaMarca, F. Potter, T. Sohn, K. Tang, I. Smith, Are GSM phones the solution for localization? in: WMCSA'06: Proceedings of the IEEE Workshop on Mobile Computing Systems & Applications, IEEE Computer Society, Washington, DC, 2006, pp. 20–28.
[36] O. Rashid, I. Mullins, P. Coulton, R. Edwards, Extending cyberspace: Location based games using cellular phones, Comput. Entertain. 4 (1) (2006) 4.
[37] E. Bruns, B. Brombach, T. Zeidler, O. Bimber, Enabling mobile phones to support large-scale museum guidance, IEEE MultiMedia 14 (2) (2007) 16–25.
[38] A. Butz, J. Baus, A. Kruger, Augmenting buildings with infrared information, in: ISAR'00: Proceedings of the International Symposium on Augmented Reality, 2000, pp. 93–96.
[39] L.M. Ni, Y. Liu, Y.C. Lau, A.P. Patil, LANDMARC: Indoor location sensing using active RFID, Wirel. Netw. 10 (6) (2004) 701–710.
[40] O. Rashid, W. Bamford, P. Coulton, R. Edwards, J. Scheible, PAC-LAN: Mixed-reality gaming with RFID-enabled mobile phones, Comput. Entertain. 4 (4) (2006) 4.
[41] B. Jung, A. Schrader, D.V. Carlson, Tangible interfaces for pervasive gaming, in: DiGRA'05: Proceedings of the International Conference of the Digital Games Research Association, 2005.
[42] C. Cruz-Neira, D.J. Sandin, T.A. DeFanti, Surround-screen projection-based virtual reality: The design and implementation of the cave, in: SIGGRAPH'93: Proceedings of the Conference on Computer Graphics and Interactive Techniques, ACM, New York, 1993, pp. 135–142.
[43] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, C. Cruz-Neira, VR Juggler: A virtual platform for virtual reality application development, in: VR'01: Proceedings of the Virtual Reality Conference, IEEE Computer Society, Washington, DC, 2001, pp. 89–96.
[44] B. Avery, W. Piekarski, J. Warren, B.H. Thomas, Evaluation of user satisfaction and learnability for outdoor augmented reality gaming, in: AUIC'06: Proceedings of the Australasian User Interface conference, Australian Computer Society, Inc, Darlinghurst, Australia, 2006, pp. 17–24.
[45] J. Scott, B. Dragovic, Audio location: Accurate low-cost location sensing, in: PERVASIVE'05: Proceedings of the International Conference on Pervasive Computing, Springer LNCS 3468, 2005, pp. 1–18.
[46] S.N. Patel, G.D. Abowd, BLUI: Low-cost localized blowable user interfaces, in: UIST'07: Proceedings of the ACM Symposium on User Interface Software and Technology, ACM, New York, 2007, pp. 217–220.
[47] N. Rober, M. Masuch, Playing audio-only games: A compendium of interaction with virtual, auditory worlds, in: DiGRA'05: Proceedings of the International Conference of the Digital Games Research Association, 2005.
[48] M. Eid, M. Orozco, A.E. Saddik, A guided tour in haptic audio visual environments and applications, Int. J. Adv. Media Commun. 1 (3) (2007) 265–297.
[49] E. Hoggan, S.A. Brewster, J. Johnston, Investigating the effectiveness of tactile feedback for mobile touchscreens, in: CHI'08: Proceeding of the SIGCHI Conference on Human Factors in Computing Systems, ACM, New York, 2008, pp. 1573–1582.

[50] A. Shahrokni, J. Jenaro, T. Gustafsson, A. Vinnberg, J. Sandsj, M. Fjeld, One-dimensional force feedback slider: Going from an analogue to a digital platform, in: NordiCHI'06: Proceedings of the Nordic Conference on Human-Computer Interaction, ACM, New York, 2006, pp. 453–456.
[51] T. Moon, G.J. Kim, Design and evaluation of a wind display for virtual reality, in: VRST'04: Proceedings of the ACM Symposium on Virtual Reality Software and Technology, ACM, New York, 2004, pp. 122–128.
[52] A. Parkes, I. Poupyrev, H. Ishii, Designing kinetic interactions for organic user interfaces, Commun. ACM 51 (6) (2008) 58–65.
[53] N. Shimizu, N. Koizumi, M. Sugimoto, H. Nii, D. Sekiguchi, M. Inami, A teddy-bear-based robotic user interface, Comput. Entertain. 4 (3) (2006) 8.
[54] M. Kojima, M. Sugimoto, A. Nakamura, M. Tomita, M. Inami, H. Nii, Augmented coliseum: An augmented game environment with small vehicles, in: TABLETOP'06: Proceedings of the IEEE International Workshop on Horizontal Interactive Human-Computer Systems, IEEE Computer Society, Washington, DC, 2006, pp. 3–8.
[55] J.D. Decuir, T. Kozuki, V. Matsuda, J. Piazza, A friendly face in robotics: Sony's AIBO entertainment robot as an educational tool, Comput. Entertain. 2 (2) (2004) 14.
[56] N. Ravaja, T. Saari, J. Laarni, K. Kallinen, M. Salminen, The psychophysiology of video gaming: Phasic emotional responses to game events, in: DiGRA'05: Proceedings of the International Conference of the Digital Games Research Association, 2005.
[57] R. Bernhaupt, A. Boldt, T. Mirlacher, D. Wilfinger, M. Tscheligi, Using emotion in games: Emotional flowers, in: ACE'07: Proceedings of the International Conference on Advances in Computer Entertainment Technology, ACM, New York, 2007, pp. 41–48.
[58] V. Nenonen, A. Lindblad, V. Häkkinen, T. Laitinen, M. Jouhtio, P. Hämäläinen, Using heart rate to control an interactive game, in: CHI'07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, New York, 2007, pp. 853–856.
[59] A. Nijholt, D. Tan, Playing with your brain: Brain-computer interfaces and games, in: ACE'07: Proceedings of the International Conference on Advances in Computer Entertainment Technology, ACM, New York, 2007, pp. 305–306.
[60] R. Krepki, G. Curio, B. Blankertz, K.R. Müller, Berlin Brain-computer interface—The HCI communication channel for discovery, Int. J. Hum. Comput. Stud. 65 (5) (2007) 460–477.
[61] R. Krepki, B. Blankertz, G. Curio, K.R. Müller, The Berlin brain-computer interface (BBCI)—Towards a new communication channel for online control in gaming applications, Multimedia Tools Appl. 33 (1) (2007) 73–90.
[62] E. Costanza, S.A. Inverso, R. Allen, Toward subtle intimate interfaces for mobile devices using an EMG controller, in: CHI'05: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, New York, 2005, pp. 481–489.
[63] IGDA Casual Games SIG, Casual games white paper, 2006, http://www.igda.org/wiki/index.php/Casual_Games_SIG/Whitepaper.
[64] E. Farella, A. Pieracci, L. Benini, L. Rocchi, A. Acquaviva, Interfacing human and computer with wireless body area sensor networks: The WiMoCA solution, Multimedia Tools Appl. 38 (3) (2008) 337–363.
[65] B. Fisher, S. Fels, K. MacLean, T. Munzner, R. Rensink, Seeing, hearing, and touching: Putting it all together, in: SIGGRAPH'04 Course Notes, 2004.
[66] E. Tse, S. Greenberg, C. Shen, C. Forlines, Multimodal multiplayer tabletop gaming, Comput. Entertain. 5 (2) (2007) 12.

A Status Report on the P Versus NP Question

ERIC ALLENDER

Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08855, USA

Abstract

We survey some of the history of the most famous open question in computing: the P versus NP question. We summarize some of the progress that has been made to date, and assess the current situation.

ADVANCES IN COMPUTERS, VOL. 77
ISSN: 0065-2458/DOI: 10.1016/S0065-2458(09)01204-2

1. Prologue

Does the world really need another survey of the P versus NP question? There are excellent textbooks that deal with this topic at length (here is a partial list [1–4]) and there have been a number of short survey articles by eminent authors [5–9]—not to mention the excellent series of NP-completeness columns written by David Johnson [10] that serve as an on-going ''status report'' on the P versus NP question. What reasons can be given, to justify spending time and effort in creating yet another overview of this topic?

First, I must confess to a selfish motive. I love computational complexity theory, and I think that the world is a better place if more people have the opportunity to learn something about the topic. The *Advances in Computer Science* series has a venerable pedigree and has published wonderful papers about progress on many exciting topics, but there has *not* been an update on the P versus NP problem in this series. I see this is an opportunity to claim some prime real estate and use it for a noble purpose: to trumpet the news that there has been thrilling and spectacular progress on several fronts in complexity theory. We have learned that the world of complexity is, in some ways, a much stranger place than anyone could have suspected back in the early days of the field. (In particular, the theory of probabilistically checkable proofs has led to very counterintuitive conclusions.) Simultaneously, the overarching lesson is that the computational universe exhibits much more structure than we should have had any right to hope for; a surprisingly small collection of tools allows us to categorize the overwhelming majority of computational problems that we really want to understand. How could I toss aside an opportunity to spread the good news about complexity theory?

Second, the editor's plea was hard to refuse:

> I am interested in a chapter on complexity theory and the chronology of the ''P = NP?'' problem—What is it? Why is it important? What has been done over the past 30 years? And what is its current status? Has any progress been made since I was a graduate

student? What is the impact if we do solve the question? What would it mean in the long run if P = NP? if P is not equal to NP? if it is proven undecidable if P = NP?

This is an excellent list of questions. In fact, it provides me with a ready-made outline for this chapter. Surely there are many readers of the *Advances in Computer Science* series that share his feelings and questions. How could I turn my back on such a need?

But the third and final reason is this: The P versus NP problem deals with the central mystery of computation. The story of the long assault on this problem is our Iliad and our Odyssey; it is the defining myth of our field. Just as authors throughout history have returned time and again to the classic heroic tales to reveal new aspects of the human condition, so do complexity theoreticians take up the task of retelling the story of the P versus NP question from different perspectives. Most of the main plot developments that will be recounted here have been described quite well in the surveys that I list in the first paragraph of this prologue. I will try not to duplicate the efforts of the authors of those surveys. In particular, I will not give a detailed history of the past 30 years in complexity theory; the reader should consult Ref. [9] for an excellent exposition of many of the developments that I discuss here, providing a more detailed bibliography and many interesting insights. I do not claim that this version of the story is superior to any of the recent surveys cited above—but at least it is different from all of them, and brings out certain parts of the story that need to be told. A great story is always worth retelling.

2. What Is the ''P = NP?'' Problem?

In the beginning, there was the reduction.

The story of NP-completeness begins with the story of the unreasonable effectiveness of *reducibility* as a tool to characterize the complexity of computational problems that we really care about solving. Prior to the breakthroughs of Cook [11], Levin [12], and Karp [13], many of the fundamental properties of computational complexity had already been worked out. For instance:

- The complexity of a problem f can be measured in terms of the size of the smallest circuit computing f [14] or in terms of the running time of a program computing f [15].
- ''Most'' functions on n input bits require circuits of nearly maximal (exponential) size [14].

- For ''nice'' time bounds $t < T$, programs running in time T can accomplish more than programs running in time t [15] (and the restriction to ''nice'' time bounds is essential [16]).
- It is tricky (but not impossible) to formalize the notion of a ''tight lower bound'' on the running time required to compute a function [17], because some problems provably have nothing remotely like an ''optimal'' algorithm [18].

All of this was lovely and important—but something vital was missing. Nobody had any idea how to say anything concrete about the computational complexity of any *natural* problem. (The word ''natural'' appears over and over again in the literature of computational complexity, although it seems impossible to give an adequate definition of what people mean when they refer to problems being ''natural.'' A good rule of thumb is that a function f is a ''natural'' computational problem if you can imagine someone being paid to produce a program or a circuit that computes f. The functions f that are shown to be hard to compute via diagonalization arguments [15, 18] fail this test of ''naturalness.'') Thus the field of computational complexity, which by rights should lie at the heart of our understanding of practical computation, instead was perched perilously close to the no-man's-land of irrelevance.

This is what changed as a result of the work of Cook, Levin, and Karp [11–13]. Their work introduced an audacious new tool: *efficient reductions among computational problems*. At this point we need to do three things:

1. Explain what an efficient reduction is
2. Explain why this is an audacious notion
3. Explain why this was such a big deal

2.1 What Is an Efficient Reduction?

Let us focus on the most basic and most useful version of reducibility: An *efficient reduction* is a function (i.e., a transformation that takes a bit string as input and produces a bit string as output) that is ''easy to compute.'' Initially (in the work of Cook, Levin, and Karp) attention centered on *polynomial-time reducibility*, meaning that a reduction f was considered ''easy'' if $f(x)$ could be computed in time at most $p(n)$, where n is the length of the input string x, and p is a polynomial. We will also consider other notions of ''easy'' later on.

To talk about ''reducibility among computational problems'' using this notion of ''reduction,'' it is convenient to limit our attention to computational problems that produce a ''yes'' or ''no'' answer (such as the problem of taking a graph as input and determining if the graph is connected, or taking (the binary representation of) a number as input and determining if the number is composite). Most computational

problems can easily be restated in this way. Thus a computational problem can be viewed as a *set*: namely, we can identify a computational problem with the set of input instances x for which the correct output is ''yes.''

Given two computational problems A and B, we say that A is *efficiently reducible to* B if there is an efficient reduction f such that x is in A if and only if $f(x)$ is in B. That is, input instances of problem A can be ''easily'' encoded as input instances of problem B; knowing the answer as to whether or not $f(x) \in B$ yields the answer of whether x is in A. Many people find the name ''reduction'' confusing. The use of this term traces back to the notion of ''reducing'' fractions (such as rewriting 2/4 as 1/2). My battered edition of Webster's Dictionary [19] characterizes this as a process ''to change (an expression) to an equivalent but more fundamental expression.'' Thus a reduction f showing that A is reducible to B is a way to change an instance of A into an equivalent instance of B; we are ''distilling'' the computational essence of problem A, and showing that, at heart, it consists of nothing much more complicated than B.

The notion of reducibility is much older than complexity theory. The concept of a mapping on bit strings inducing a ''reduction'' from one computational problem to another was already firmly established as a tool for showing that certain problems could not be solved by computer programs (regardless of the running time). See any textbook on computability theory (such as [20]) for more background. The work of Cook, Levin, and Karp merely hijacked this well-known notion, and imposed time bounds.

2.2 Why Is This an Audacious Notion?

On the face of things, the notion of polynomial-time reducibility does not seem like a very promising way to forge a link between practical concerns of real-world computing and the abstract theory of computational complexity. According to this definition, a function f that takes inputs of length n and requires $n^{1,000,000}$ computational steps to compute is to be considered ''easy to compute,'' even though the sun would most likely have become extinct long before any computing device would have a chance to compute $f(x)$ for inputs x consisting of even a handful of bits. How can anyone be serious in proposing such a preposterous definition of ''easy to compute''?

The reason behind this fraud is quite simple: It is convenient to maintain the fiction that if f and g are easy to compute, then it should also be easy to compute $f\ (g(x))$. This makes ''reducibility'' a transitive relation, so that if A is reducible to B and B is reducible to C, then A is also reducible to C. There are many classes of functions that one could use that would have this property. For instance, we could consider a function to be ''easy to compute'' only if it could be computed in time linear in the input length, or we could restrict attention to running time bounded by

$O(n \log^k n)$ for different values k. These notions have been studied, but they have not turned out to be nearly as useful in characterizing ''natural'' problems as polynomial-time reducibility has.

Focusing on polynomial-time reducibility brings other benefits, too. There are lots of computer programs in wide use that run for time n^2 on inputs of size n. One would not want to exclude such transformations from the class of ''easy'' functions. But once one allows quadratic time, there is no easy way to avoid allowing arbitrary polynomial time; a function f computable in time n^2 may produce output of size n^2. Thus computing $f(f(x))$ will take time n^4, and the reader can easily see where this leads. If the composition of two ''easy'' functions is to be considered ''easy,'' then one is led quite quickly to consider all polynomial-time-computable functions as being ''easy.''

Another benefit of basing complexity theory on polynomial time is that, when we are measuring run-time, we are freed from (almost all) concerns about being specific about the type of computing device that our programs are running on, as well as what language the programs are written in, etc. This is because, in the history of computing thus far, almost all ''real-world'' notions of computing that have been proposed can easily be simulated with only a polynomial slow-down by Turing machines. This is sometimes referred to as the ''Invariance Thesis'' [21] or the ''Strong Church-Turing Thesis'' [22, 23]. (Do not worry if you do not know what Turing machines are; think of them as a particularly simple programming language and machine architecture. Occasionally one hears an objection to the Turing machine model, indicating that Turing machines are *more* powerful than physical computational devices, since they come equipped with an *infinite* memory. However, this objection misses the point. Programs are typically written to handle inputs of *any* length; if a program is run on a machine with very limited memory the program may be unable to execute properly on very large inputs, although the same program would run fine on a machine that has more memory. Turing machines are intended to model *programs*, rather than *machines*; the existence of a fast program for a given problem is equivalent to the existence of a fast Turing machine for the problem. Rather than thinking of Turing machines as being *more* powerful than physical computers, it is more accurate to think of Turing machines as being a very *restricted* class of programs.)

The Strong Church-Turing Thesis is not universally accepted. Probabilistic computation (augmenting computers with a source of random bits [24]) and quantum computing (see the work of Shor [25] for insight into the power of this model) have both been proposed as physically realizable programming paradigms that might provide more computational power. There are good reasons for considering

these and other models of computation, and there are good reasons for being skeptical about whether problems computable in polynomial time are truly feasible, but there is no doubt about the fact that the class of problems solvable in polynomial time now occupies a central position in the way that we understand computation.

Definition 1

The class P consists of all computational problems A for which there is a polynomial p and a Turing machine that takes input x of length n and determines whether x is in A, in time bounded by p(n).

It is certainly not true that all problems in P are easy to compute. The value of the definition lies in the fact that there is good reason to consider problems that do *not* lie in P to be *hard* to compute.

2.3 Why Was This Such a Big Deal?

Efficient reducibility provided the first useful abstraction that enabled us to make sense of a chaotic universe of computational problems. Prior to the development of this tool, it was known that some problems had efficient algorithms while no efficient algorithms had been discovered for other problems, but there was no way to estimate the likelihood of an efficient algorithm being found, if none was already known.

In what sense does efficient reducibility provide an abstraction? If A is efficiently reducible to B, and B is efficiently reducible to A, then in a very meaningful sense, A and B are ''equivalent''—they are merely two different ways of looking at the same problem. Thus instead of infinitely many different apparently unrelated computational problems, one can deal instead with a much smaller number of *classes* of equivalent problems. Technically, there are still infinitely many of these classes; it is known that for any problem A that lies outside P there are infinitely many classes that lie ''between'' P and A [26, 27]—but these constructions rely on ''unnatural'' computational problems, which are not the computational problems that people really care about. The amazing fact (which is also amazingly useful) is that ''natural'' computational problems tend to clump into just a few equivalence classes. This was completely unexpected. Nothing had prepared the computing community for the shocking insight that there are really just a handful of *fundamentally different* computational problems that people want to solve.

2.4 Complexity Classes

The story just keeps getting better. Not only is there a framework of classes of equivalent problems that helps us partition natural computational problems into meaningful groups, but many of these equivalence classes correspond in a meaningful way to resource bounds.

Let us illustrate this by means of an example. Consider the problem of computing optimal next moves in a game of checkers. (Checkers is played on an 8-by-8 grid; we actually consider the generalized problem that is played on an n-by-n grid.) Optimal strategies for n-by-n checkers can be computed in time exponential in n, and thus this problem lies in the class known as EXP (for ''exponential time''). It turns out that Checkers is also a canonical problem for EXP, in the following sense: every problem $A \in \mathrm{EXP}$ is polynomial-time reducible to Checkers [28]. That is, every problem in EXP can be rephrased as a problem of finding an optimal move in a game of checkers. This is a specific case of a very general phenomenon, known as ''hardness.''

Definition 2

Let C be a class of computational problems. A *set A is* hard *for C if B is polynomial-time reducible to A for every set $B \in C$.*

Definition 3

Let C be a class of computational problems. A set A is complete *for C if A is hard for C and $A \in C$.*

Thus Checkers is complete for EXP. This means that the computational complexity of this problem is fairly well understood; it can be solved in exponential time, and it cannot be solved in time much less than 2^n because

- It is known [15] that there is *some* problem A in EXP that requires time $\Omega(2^n)$.
- By completeness, A is reducible to Checkers in time n^k for some k.
- If Checkers were solvable in time less than $2^{n^{1/k}}$, the reduction from (b) would yield an algorithm for A that runs too quickly, violating the lower bound (a).

If only things were always this nice! Next, we consider a completeness theorem that produces a much less satisfactory result.

Consider the problem of determining if two regular expressions (i.e., the type of expression that is used in the tool ''grep'') are equivalent (in the sense that they denote the same regular set). This problem is complete for PSPACE (the class of problems that can be solved by Turing machines using at most $p(n)$ memory

locations on inputs of length n, for some polynomial p) [29]. This is quite similar in spirit to the claim that Checkers is complete for EXP, and we can still legitimately claim that determining equivalence of regular expressions is a canonical problem for PSPACE and in some sense is one of the most difficult problems in PSPACE—but there is a significant difference. We do not know if PSPACE is equal to P, and thus we cannot (currently) conclude *anything* about the time that is required of programs that solve this problem.

. . . But there is more to be learned from this example. Even though we cannot currently *prove* that programs require exponential time to solve the regular expression equivalence problem, we can say that it would be a dramatic breakthrough if any subexponential-time algorithm for this problem were to emerge. It would imply similarly fast algorithms for *every* PSPACE-complete problem (and many such problems are known, *all* of which have resisted fast algorithmic solution), and it would mean that *any* problem that can be solved by an algorithm that uses n^k memory (including algorithms that search through all 2^{n^k} strings in a large search space looking for possible solutions to a problem) can be solved fairly quickly, in time much less than is required to examine a search space of size 2^{n^k}. It seems inconceivable that this should be possible, although we still have no formal proof that it is really impossible. Taken together, this is very strong evidence that the regular expression equivalence problem is hard to compute, even if it falls short of the standard of a real proof.

P, PSPACE, and EXP are three important examples of *complexity classes*: classes of computational problems that consist of all of the problems that can be computed using a certain amount of some computational resource (such as time or memory). Unfortunately, many important natural problems do *not* seem to be complete for any class that is defined in terms of bounding computational resources on realistic models of computation. This motivates turning to *unrealistic* models of computation.

2.5 The Class NP

Not only did Cook, Levin, and Karp introduce efficient reducibility as a useful tool for classifying the complexity of natural problems, but they also focused attention on the one complexity class that has turned out to be more useful than any other: NP.

Unlike the complexity classes that have been discussed thus far (P, PSPACE, EXP), NP is not defined in terms of computation on machines that are intended to model real-world computing. A *nondeterministic* machine that runs in time t is provided with access to a ‘‘magic word’’ m of length t, in addition to its ordinary input x. We say that the machine *accepts* input x if there is any word m that could be

provided to it, that would cause the computation on input (x, m) to output 1. Otherwise, we say that the machine *rejects* its input x. Note that this is roughly the same thing as allowing a machine to search through all of the 2^t words of length t, looking for a solution that would cause it to output 1. However, we say that the running time of the machine is t, instead of the time 2^t that it would take to search through the entire list of possibilities deterministically. Viewed another way, a nondeterministic machine running in time t on input x starts running 2^t computations simultaneously in parallel (one computation for each different choice of the ''magic word'' m), and accepts if *any* of the 2^t computations outputs 1. It is useful to think of the ''magic word'' m as a ''proof'' that the input x should be accepted.

On the face of things, this looks like a really goofy model of computation. But it is exactly the right model of computation to use, if we want to understand a host of important computational problems.

Definition 4

The class NP *consists of all computational problems A for which there is a polynomial p and a nondeterministic machine running in time p(n) on inputs of length n, that accepts input x if and only if* $x \in A$.

A few hundred of the most important NP-complete problems can be found in the standard reference work by Garey and Johnson [30]. Some of the most familiar of these are

- SAT (the set of Boolean formulae that have a satisfying assignment)
- CLIQUE (the set of pairs (G, k), where G is a graph for which there is a subset of k vertices, all of which are connected to each other)
- 3-COLORABILITY (the set of graphs G whose vertices can be colored Red, Green, and Blue, such that no edge has endpoints with the same color)

Note that, for these three examples, it is easy to see what the ''magic word'' would be that provides a proof of membership. For SAT it would be a satisfying assignment (which can easily be checked to see that it is, indeed, a satisfying assignment); for CLIQUE it would be a set of k vertices; for 3-COLORABILITY it would be an assignment of colors to the vertices.

It is hard to overstate the usefulness of NP-completeness as a tool for understanding the apparent intractability of many problems that we would dearly love to be able to solve with computers. In 1997, Papadimitriou wrote [8]:

> *. . . about 6,000 papers each year have the term ''*NP-*complete'' on their title, abstract, or list of keywords. This is more than each of the terms ''compiler,'' ''database,''*

*''expert,'' ''neural network,'' and ''operating system.'' Even more surprising is the diversity of the disciplines with papers referring to ''*NP*-completeness'': They range from statistics and artificial life to automatic control and nuclear engineering.*

Many other important computational problems do not seem to be NP-complete, but are complete for complexity classes defined using variations on the theme of nondeterminism (such as *counting* the number of different proofs of acceptance, instead of merely asking if such a proof exists [31]).

One can easily establish the following inclusions:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP}.$$

Thus, just as we obtain no *proof* of intractability from the knowledge that a problem is PSPACE-complete, so also a proof of NP-completeness yields no proof that a problem is hard to compute, but the evidence of intractability is nearly as compelling for NP-complete problems as it is for PSPACE-complete problems. In practice, this turns out to be very useful information.

It is known that P is not equal to EXP, and thus at least one of the three inclusions above must be a proper containment, but it is not known that any one of them is proper. Most people working in the field would probably conjecture that *all* of these containments are proper—but it is risky to rely on this sort of ''intuition.'' Later in this chapter, we discuss one important example, where two classes that were thought to be distinct turned out to be the same.

One strange thing about the nondeterministic model of computation is that the tasks of accepting and rejecting an input are very different. Given any problem A, the complementary problem $\bar{A} = \{x : x \notin A\}$ is in P if and only if $A \in \mathrm{P}$. This does not seem to be true for NP. Consider $\overline{\mathrm{SAT}}$; nobody has any idea how to give a short proof that a formula x does *not* have a satisfying assignment. This gives rise to the complexity class $\mathrm{coNP} = \{A : \bar{A} \in \mathrm{NP}\}$. Clearly $\mathrm{P} \subseteq \mathrm{NP} \cap \mathrm{coNP}$.

2.6 Subclasses of P

Polynomial-time reducibility is not very useful for drawing distinctions between problems in P—but useful distinctions can be made. What is needed is a more delicate tool, defined analogously to polynomial-time reducibility, but using a more restricted class of functions. We will not provide definitions here, but merely note that logspace reducibility [32] is frequently used to define classes of complete problems inside P, as are even more restrictive notions of reducibility, defined in terms of small circuit classes [33, 34]. More information on subclasses of P can be found by consulting the references [35–37].

With very few exceptions, natural problems that are complete for NP, PSPACE, and EXP (and other complexity classes) under polynomial-time reducibility remain complete even when these more restrictive notions of reducibility are used instead. Thus, there is essentially no drawback to using the more restrictive notions of reducibility (since the problems that one wants to classify remain complete under the more restrictive reducibility, and as a bonus one is able to show that certain problems are complete for P and others are complete for interesting subclasses of P).

When one is first introduced to the notions of NP-completeness and completeness in other complexity classes, it probably seems as if completeness is a very unusual property, and that it should be rare for a problem to be complete for a complexity class. However, the opposite is true. The surprising lesson that emerges, after decades of experience in complexity theory, is that *the overwhelming majority of natural computational problems can be shown to be complete for one of about a dozen or so complexity classes*. Sometimes the definitions of these complexity classes may seem cumbersome or complicated (just as the definition of NP may strike one as being fairly unnatural at first). However, it is important to note that these classes are ''*discovered*, and not *invented*'' (quoting Papadimitriou again [8]). That is, it is *not* the case that some out-of-touch theoretician cooked up the definition of these complexity classes merely to prove a few theorems. Rather, there was an important class of computational problems out there that people were already interested in, and then complexity theoreticians were able to show that the problems were all in fact complete for some complexity class that could be described in terms of resource bounds applied to some computational model.

Let us return again to the question that begins Section 2:

2.7 What Is the ''P = NP?'' Problem?

We have introduced P and NP, so perhaps the answer is clear: The ''P = NP?'' problem is simply the question of whether those two complexity classes are one and the same. But P and NP are just two of the most important complexity classes. The P versus NP question really stands for a more fundamental and general question concerning the nature of this entire framework of complexity classes, with its partition of natural problems into classes of complete sets for various complexity classes. How much of this structure is real, and how much is an illusion? The theory of complexity classes seems to explain our inability to find fast programs for certain problems—but is this explanation real, or is it simply a convenient and comforting tale that we tell ourselves? We find it comforting, because it would mean that the reason we have not found a fast algorithm is *not* because we are too stupid to find it—it is because no fast algorithm is possible.

3. Why Is the "P = NP?" Problem Important?

Once again, it seems that we are asking a question that we have already answered. The authors of the 6000 papers per year that mention NP-completeness have their own reasons for wanting to know if these complexity classes are equal or not. The people who are trying to solve NP-complete optimization problems have a powerful financial incentive for wanting to know if these problems have fast algorithms or not. These are all significant reasons for why the P versus NP problem is important—but there are some additional reasons that should be discussed, too. That is the task we take up in the rest of this section.

3.1 Cryptography

Modern cryptography relies on the existence of *one-way functions* (functions f that are easy to compute but have the property that no efficient algorithm can take as input a string y in the range of f and produce as output a string x such that $f(x) = y$). Actually the requirements are much stronger; it is necessary that the task of finding such an x be difficult *on average*, so that for the *overwhelming majority* of the strings y in the range of f, there is no way to find x such that $f(x) = y$. For instance, the RSA cryptosystem [38] relies on multiplication being a one-way function; define f to be the function that takes as input two prime numbers u and v each having the same number of bits, and produces uv as output. If there is an efficient algorithm that can find the prime factors of a given integer, then f is not one-way and many cryptographic applications in wide use are insecure.

If P = NP, then there would seem to be no way to salvage cryptography. The problem of inverting a supposed one-way function lies in NP; if all of these problems are in P, then to rescue the notion of a one-way function one would have to hypothesize the existence of functions that are very easy to compute, but whose inverses require time n^k for very *large* values of k. Complexity theory currently offers no suggestions as to how this might be accomplished.

But in fact, complexity theory currently offers few if any useful tools that can be used to provide evidence that a given function is a cryptographically secure one-way function. Let me elaborate on this point. If we know that a problem is NP-complete, then there is a coherent theoretical framework explaining why the problem is probably hard to compute; if the problem turns out to be easy, then the entire framework comes crashing down. In contrast, consider the factoring problem. The best evidence that factoring is hard comes from the fact that people have been trying to find good factoring algorithms for a few hundred years, without success. This is not particularly strong evidence, since there are several notable examples in which

problems not previously known to reside in P have yielded to new algorithmic insights and techniques. If factoring turns out to be easy, it will have dramatic consequences for the practice of cryptography, but it will not fundamentally alter the framework of complexity theory. Similar observations hold for all of the widely considered candidate cryptographically secure one-way functions.

Another factor to keep in mind is that, if f is a *one–one* function, then the problem of computing the inverse of f lies in $\mathrm{NP} \cap \mathrm{coNP}$ [39]. Thus to have one–one one-way functions, one needs not only $\mathrm{P} \neq \mathrm{NP}$, but $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$ which seems to be a stronger hypothesis.

An alternate approach might be to design a function f that is ''complete'' in some sense for the class of all one-way functions (so that f is cryptographically secure if any function is). In fact, such a construction was presented already by Levin [40]; although, this construction holds little interest for practitioners, since the constants in the security guarantees are quite weak.

Recall that this is the section of the chapter in which we are addressing the question of why the P versus NP question is important. In order for modern cryptography to rest on a firm foundation, a proof that $\mathrm{P} \neq \mathrm{NP}$ is an absolutely essential first step—but it would be only a first step. Much stronger intractability results are required for cryptography.

3.2 Understanding the World

There are few areas of scientific enquiry that are untouched by algorithmic considerations. In biology, economics, and physics, many of the natural processes that are studied can be viewed as proceeding algorithmically. If a biological theory (or an economic theory, a theory of evolution, etc.) requires that an organism (or participants in a market, or an environmental system, respectively) arrives at an optimal state, then it had better be the case that this does not require the solution of an intractable computational problem, or else the theory lacks plausibility. But until we know if $\mathrm{P} = \mathrm{NP}$, we do not have a good understanding of which problems are intractable. Aaronson has proposed hypothesizing the intractability of NP-complete problems as a natural law, in order to judge the plausibility of certain interpretations of quantum mechanics [41]. If certain aspects of a physical theory can be shown to imply that there are efficient ways to solve NP-complete problems, then this should throw doubt on the theory (since there is no empirical evidence that these problems are amenable to efficient solution). But how compelling can such an argument be, until we know for sure that conventional programs and computers are unable to solve NP-complete problems?

The P versus NP problem can be cast as the problem of whether it is significantly harder to find a proof of a theorem, than to merely check that a proof is correct, and

thus it has profound implications for mathematics. That is, if we consider any fixed formal proof system (so that the problem of checking if a proof is correct is simply a syntactic procedure that can be automated), then the set of theorems that have proofs of a reasonable length is a problem in NP. Stated another way, if you want to know if there is a proof of some statement ϕ that is at most 60 pages long, a nondeterministic machine can determine the answer quickly (where the ''magic word'' is simply the 60 page proof). If P = NP then there is a relatively fast automatic way to find the 60 page proof, given only statement of the theorem. Long before the P versus NP question was formalized, Gödel and von Neumann discussed precisely this scenario [42]. The connection between complexity and the notion of proof has a long history and has played a crucial role in some of the most dramatic developments of the theory, as discussed later in this chapter.

For all of these reasons and more, there can be little question that the P versus NP question is important.

4. What Progress Has Been Made in the Past 30 Years?

This is the most dangerous section of the chapter to write. The dangers are (at least) twofold:

- I can omit some really important developments
- I can get carried away about one or the other line of research and include more detail than I should, resulting in a long, unreadable document

In order to steer a course between Scylla and Charybdis, I will try to keep the chapter short and easy to read, even though this means that I will leave out some great stories (and my treatment of the stories that I include will be far too brief).

4.1 Small Circuits and the Polynomial Hierarchy

In order to present some of the exciting developments of the past three decades, I need to present a bit of material that is slightly older.

In a paper published in 1976 [43], Stockmeyer introduced a hierarchy of complexity classes that sits ''right above'' NP. We have already been introduced to the lowest levels in this hierarchy: P, NP, and coNP. A number of problems related to NP optimization problems are more conveniently stated in a form that is hard for both NP and for coNP and thus is not believed to lie in either class. For example, consider the Traveling Salesperson Problem. This can be phrased as a problem in NP

in the following form: Given a graph G with ''distances'' on the edges and a number k, is there a path of length at most k that visits all of the vertices in the graph? But one might find it more natural to ask ''Given G, compute the length of the shortest path that visits all of the vertices.'' If one had a subroutine for the NP formulation of the Traveling Salesman Problem, then this value could easily be computed using binary search. Thus it lies in the class P^{NP} (the class of problems that can be solved in polynomial time using an ''oracle''—that is, a subroutine whose running time we do not count—for a problem in NP). In fact, this turns out to be a complete problem for P^{NP} [44].

Once we have defined P^{NP}, it is a short step to define $\mathrm{NP}^{\mathrm{NP}}$ and $\mathrm{coNP}^{\mathrm{NP}}$. These classes seem to bear the same relationship to P^{NP} as NP and coNP bear to P, and thus it seems that these classes provide substantial additional computational power over P^{NP}. There are natural and well-studied problems that are complete for $\mathrm{NP}^{\mathrm{NP}}$ and $\mathrm{coNP}^{\mathrm{NP}}$(e.g., [43], [101], [102]). This process can be continued, to define an infinite hierarchy of complexity classes, known as the *polynomial hierarchy*. One property of this hierarchy is that, if any two levels coincide, then the entire hierarchy collapses to that level. Thus if $\mathrm{NP}^{\mathrm{NP}} = \mathrm{coNP}^{\mathrm{NP}}$, then the entire hierarchy collapses to $\mathrm{NP}^{\mathrm{NP}}$; if $\mathrm{P} = \mathrm{NP}$, then the hierarchy collapses to P. One reason the polynomial hierarchy has come to be important in the field of complexity theory, is because the belief that the hierarchy is infinite is nearly as well rooted as the belief that $\mathrm{P} \neq \mathrm{NP}$.

The story of small circuits for NP illustrates this use of the polynomial hierarchy. Recall from Section 2 that there are two basic models of computation: programs and circuits. With programs, there is one program that works for all input lengths; with circuits, there is a different circuit for each input length. Of course, if you have an efficient way to *build* circuits for your problem, there is not much difference between these two notions. But there are many problems that have *no program at all*, but which have small circuits (e.g., consider unary encodings of the Halting Problem).

Circuit complexity is essential in order to prove that certain transformations from input to output *on a fixed input size* are intractable. This is an important point, and it is worth emphasizing.

Consider, for example, the following theorem regarding the problem of determining whether a logic formula in a certain formalism (abbreviated WS1S) is true or not.

Theorem 5 [45]

Any circuit of AND, OR, *and* NOT *gates that takes as input a WS1S formula of 610 symbols and outputs a bit that says whether the formula is true must have at least* 10^{125} *gates.*

Clearly, no such circuit can fit in the solar system. Note that this theorem is quite specific about saying that this problem is difficult at a particular input length. This is the sort of security guarantee that is required in cryptography, where one needs to pick key lengths long enough so that the problem of cracking the system is intractable; it is not enough to know that programs will require a long amount of time for ''big enough'' inputs—it is important to know just how big is ''big enough.''

Recall the Checkers problem (which is complete for EXP) from Section 2.4. We know that any program that solves Checkers must run for time 2^{n^c} for *large* inputs—but we have absolutely no proof that this problem does not have circuits of *linear* size! (If this is the case, then there is *no* input size where Checkers becomes intractable.) For all we know, perhaps all problems in EXP have circuits of polynomial size.

Thus for example, if cryptographers are ever to have any hope of using theorems of complexity theory to pick key sizes to make their cryptosystems secure, it is essential to know not only that $P \neq NP$, but that NP does not have polynomial-size circuits.

But is this question really any different from the P versus NP question? There is a long line of research that tries to relate these problems, beginning with a result by Karp and Lipton [46] showing that NP has polynomial-size circuits only if the polynomial hierarchy collapses to NP^{NP}. There has been work through the years trying to show that the collapse happens at a lower level [47, 48], but it is still not known if NP having polynomial-size circuits implies that the polynomial hierarchy collapses to P^{NP}.

There are many results in complexity theory of the form ''X is true, unless the polynomial hierarchy collapses.'' This is taken as strong evidence that X is true.

4.2 Interactive Proofs and Probabilistically Checkable Proofs

There is widespread agreement that the most significant change in our understanding of NP over the last three decades grew out of an expanded notion of ''proof'' [49, 50]. Recall that NP can be viewed as the class of sets B for which there are short ''proofs'' that $x \in B$, where a ''proof'' is just a string y so that a polynomial-time program A can read the pair (x, y) and accept if and only if y provides the information that is necessary to convince the program that x is in B. Recall from Section 2 that there is a segment of the community that contends that *probabilistic* polynomial-time algorithms are a more appropriate way to capture the notion of ''feasible computation.'' Starting from this point of view, it seems natural to consider an expanded version of NP, defined in an analogous way, but allowing a

probabilistic algorithm A to determine if y provides convincing evidence that x is in B. This leads to a complexity class called MA; the name comes from viewing the process of proving membership in B as being a conversation between a magical ''prover'' (Merlin the Wizard, who provides the string y) and a mere mortal with limited computational power, but owning a pair of dice to help him make random choices (King Arthur). Viewing things this way, Merlin speaks first, giving Arthur the string y, and then Arthur rolls his dice and does his computation on the pair (x, y).

But is there any reason why Merlin should have to speak first? One can also define the class AM, where Arthur gets string x, rolls his dice to get some random bits, and on the basis of these bits poses a question to Merlin, who then sends a string y (and Arthur can do some computation to see if Merlin's reply convinces him). One can show that MA $\subseteq$ AM. There seems to be no reason to stop at such short conversations; one can define an entire sequence of classes AMA, AMAM,. . ., as well as a class IP (for ''Interactive Proofs'') where the conversation continues for a polynomial number of rounds.

Why is this interesting?

One interesting fact is that AM = AMA = AMAM, and so on. That is, two rounds of communication are as good as any constant number of rounds [49].

Much more interesting is the fact that there is an important computational problem in AM that is not known to lie in NP: the graph nonisomorphism problem [50]. The graph isomorphism problem (given two graphs G and H, are they isomorphic?) is easily seen to be in NP. (The ''magic word'' is simply a permutation of the vertices of G, yielding the graph H.) But how can you provide a short proof that two graphs are *not* isomorphic? If Merlin is all-powerful, then here is how Arthur can be convinced. Arthur gets the graphs G and H and picks one at random, permutes the vertices in a random way and obtains a new graph K_1. He repeats this process and obtains a graph K_2, and so on, until he has 100 graphs $K_1, \ldots, K_{100}$. Arthur knows, for each K_i, if K_i is a copy of G or of H. Arthur now sends the sequence $K_1, \ldots, K_{100}$ to Merlin, and asks Merlin to tell him which graph each K_i is a copy of. Note that if G and H are not isomorphic, the all-powerful Merlin has no problem doing this. But if G and H are isomorphic, then Merlin has just a 1-in-2^{100} chance of sending Arthur the right answer. Thus if Merlin sends Arthur the correct answer, Arthur is justified in feeling quite confident that G and H are really not isomorphic.

Much of the initial interest in this type of ''proof'' came from the notion of ''zero-knowledge proofs'' which has wide application in cryptography. (This comes from the observation that Arthur gains no useful information in the preceding example, other than being convinced that the graphs are not isomorphic.) This is a huge topic that is surveyed elsewhere (e.g., [3, 9]).

The class IP was something of a mystery for a while. It was felt that this notion of ''proof'' was probably not too much stronger than the usual notion of ''proof,'' and many felt that it would be unlikely that IP would contain coNP. In fact, evidence was presented that was considered at the time to be rather compelling, arguing that ''new techniques'' would be required to show $\text{coNP} \subseteq \text{IP}$ [51].

New techniques *were* found, and a dramatic series of papers ended by showing that IP = PSPACE [52, 53]!

According to the rules of IP interaction, Merlin is allowed to give different responses to the same query from Arthur. That is, if Arthur has random sequence r_1 that causes him to ask query q at some point during his interaction with Merlin, Merlin might give a different response than he gives during the interaction that Arthur has with him when using random sequence r_2. If the rules of the game are changed, so that Merlin has to commit ahead of time to the response that he will give to each possible query q from Arthur, this turns out to give a characterization of nondeterministic exponential time (NEXP) [54]. This is an *extremely* counterintuitive characterization. Nondeterministic exponential time can be viewed as the class of problems B where membership of x in B can be demonstrated by a ''proof'' of length exponential in the length of x. This characterization means that Merlin can provide an exponential-sized proof, and Arthur can be convinced of its correctness by randomly picking just a small number of positions in the proof to examine!

Soon a similar characterization of NP was given, and various parameters in the characterization were optimized, to obtain a truly spectacular and almost unbelievable reworking of the notion of ''proof'' [55, 56]. Rather than giving a formal definition of a ''Probabilistically Checkable Proof,'' let us give an example. Suppose that you are asked to referee a paper, but you are very short of time. Rather than read the entire paper, you randomly pick a few paragraphs, and if you do not see any problem, then you decide to accept the paper. Most of us would consider this to be very irresponsible behavior. How can one have any confidence in the correctness of a proof without reading every symbol of the proof? But in fact *any* proof can be encoded in such a way that this sort of ''lazy'' refereeing is sufficient. That is, the proof can be encoded as a string y (of length not much greater than the length of the original proof), so that a verifier can randomly choose k bits of y (where k is a constant that does *not* depend on the length of y, and the verifier is using only $O(\log n)$ random bits in order to make this selection), and will always accept if the proof is correct, and will reject with very high probability if the proof is not correct [55, 56]. The proof of this characterization is one of the most complicated arguments in complexity theory, and there has been a great deal of interest in finding a simpler proof; substantial simplifications have been presented in the last few years [57, 58].

4.3 Hardness of Approximation

Since the CLIQUE problem is NP-complete, we know not to expect to be able to find the size of the largest clique in a graph. When we learn that it is hard to find an optimal solution, it is natural to adjust our goals, and settle for getting the best solution that we can. There is a huge literature on algorithms for finding approximate solutions to NP-complete optimization problems, and there were some early results showing that certain approximations could not be obtained efficiently unless P = NP (e.g., [59]). For many years, however, there was little known about which approximations could be obtained efficiently, and which ones are intractable.

The dramatic characterization of NP in terms of probabilistically checkable proofs changed all of that. Given a probabilistically checkable proof, it is either the case that *all* of the polynomially many probabilistic sequences lead to acceptance or only a *small fraction* of them do. It turns out to be possible to build on this, to reduce CLIQUE to instances where there is either a very large clique or else the largest clique is quite small, and thus CLIQUE is hard to approximate unless P = NP [60, 61]. Related approaches work on many other optimization problems in NP, and in some cases it is possible to give tight bounds, showing that a solution can be found that is at most α times the optimal for some constant α, but doing any better is NP-hard. (See Håstad [62] for one such example. There are many others.)

4.4 AM and NP

We discussed the Arthur/Merlin class AM in Section 2. It is safe to say that, back when AM was introduced [49, 50], most people in the field were inclined to think that AM was likely to contain problems that were outside of NP. In the intervening years, there was astonishing progress made in the field of *derandomization* (i.e., the study of eliminating the use of probabilistic bits in randomized algorithms). Since this survey focuses on NP, I will limit the discussion here to the probabilistic analog of NP, which is AM. Let us learn what has happened in the last two decades, to change perceptions of the likely relationship between AM and NP.

A defining characteristic of a random coin toss is that it is *unpredictable*. The outcome of a hard-to-compute function is also ‘‘unpredictable’’ in some sense (although it is not clear that there is a meaningful connection between these two settings, since a function always gives the same answer to a given question, unlike tossing a coin). A sequence of important papers (see Nisan and Wigderson [63] and Impagliazzo and Wigderson [64]) showed that this connection can be made precise and exploited. If there is a problem A that is computable in time 2^n that requires *circuits* of size $2^{\varepsilon n}$ for some $\varepsilon > 0$ (i.e., if A requires circuits of nearly maximal size), then computing A on all of the inputs of size $O(\log n)$ can be used to produce a

sequence of bits that is enough like ''noise'' that it can be used to give a deterministic simulation of a probabilistic algorithm.

Related techniques were also applied [65–67] to generate random bits that could be used to give nondeterministic simulations of AM. These techniques led to the conclusion that NP = AM if there is a problem computable in nondeterministic time 2^n that requires *nondeterministic* circuits of size $2^{\varepsilon n}$ for some $\varepsilon > 0$. We will not define nondeterministic circuits here—but we will mention that this hypothesis is considered to be reasonably likely, and hence much of the complexity-theoretic community would now conjecture that AM = NP. Note that this also would imply that the graph isomorphism problem is in $\mathrm{NP} \cap \mathrm{coNP}$.

4.5 Average-Case Complexity

Even if we assume that $\mathrm{P} \neq \mathrm{NP}$ and hence we must give up on the idea of having efficient algorithms that solve NP-complete problems, there is still a pressing need to have algorithms that perform as well as possible in solving these problems. Various heuristics have been developed for different NP-complete problems that seem to perform reasonably in various settings, and one occasionally hears the claim that a particular heuristic ''works well for instances that arise in practice.''

Such claims can be difficult to evaluate, since it is usually very difficult to say anything precise about the probability distribution on inputs in real-world settings. Sometimes it is useful to talk about the performance of an algorithm when inputs of a given length are distributed uniformly, but it is easy to give examples of problems that are very easy to solve using the uniform distribution. For example, consider the set $\{(\phi, \phi) : \phi \in \mathrm{SAT}\}$. From one perspective, this is just a simple encoding of SAT—but from another perspective, this set is trivial to solve on all but an exponentially small fraction of the inputs (since we need only reject if the first half of the string is different from the second half, and this will almost always be the case).

Levin introduced a theory of average-case complexity [68] and presented an NP-complete problem that is hard on average to solve, using *any* ''reasonable'' distribution on the inputs. (It is necessary to restrict attention to some class of distributions, since—assuming that $\mathrm{P} \neq \mathrm{NP}$—any efficient algorithm will make errors on *some* inputs, and there is always an ''unreasonable'' probability distribution that places all of its weight on those inputs where the algorithm fails.) Levin focused on distributions that are computable in polynomial time. Analogous studies have been carried out, based on so-called ''samplable'' distributions, which are distributions that can be ''generated'' by a probabilistic polynomial-time algorithm. This model makes sense, if you hypothesize that the input instances that arise in practice are in

fact generated by some feasible process. There are some excellent surveys of this type of approach to average-case complexity [69–71].

Although the uniform distribution is not always the most relevant distribution to consider, some very significant insights have been gained by considering how well algorithms can perform on the uniform distribution. A crucial step in some of the derandomization arguments that were considered in Section 4 was the proof that, if there is any problem in EXP that is not solvable by polynomial-size circuits, then there is a problem in EXP for which any polynomial-size circuit gives the wrong answer for nearly half of the inputs of length n [72]. This is called a ''worst-case-to-average-case'' reduction, because it involves showing how to compute a function A correctly on *all* inputs, by accessing any circuit that computes a related function A' correctly on a large fraction of the inputs. This sort of argument draws heavily on the theory of error-correcting codes; the truth table of A' is essentially an encoding of the truth table of A using an error-correcting code. A circuit that computes A' correctly on a reasonably large fraction of the inputs can be viewed as a corrupted version of the codeword—but there are a small enough number of errors so that the truth table of A can be recovered.

Related worst-case-to-average-case reductions are known for NP [73–76]; although, the parameters are not as good as in the corresponding results for EXP, because of the additional technical obstacles that arise when working with NP.

4.6 Time–Space Tradeoffs

Proving that $\mathrm{P} \neq \mathrm{NP}$ involves proving a superpolynomial lower bound on the run time of any algorithm for SAT. Is there any way to measure our progress toward this goal? For instance, do we know that SAT requires time n^3, or time $n\log n$?

Sadly, the answer is ''No.'' We still do not know if SAT can be recognized in *linear* time on a Turing machine. However, a series of papers beginning with Fortnow [77] (and nicely surveyed by Van Melkebeek [78]) shows that algorithms for SAT that use *small space* must run for time more than $n^{1.8}$ [79]. (These results hold not only for Turing machines, but for more general models of computation that allow random access to memory locations.)

4.7 The Isomorphism Conjecture

All NP-complete problems are equivalent in some sense. Berman and Hartmanis noticed that all of the NP-complete problems in the monograph by Garey and Johnson [30] in fact are *isomorphic* to each other, in a very strong sense [80]. Namely, they showed that, for any two of these problems A and B, there is a *bijection*

f such that both f and f^{-1} are computable in polynomial time, mapping A onto B. Thus, in a natural and appealing way, it is reasonable to say that all of the NP-complete problems in Garey and Johnson are simple re-encodings of each other. They conjectured that, in fact, this is true for *all* NP-complete problems, and not merely the ones in Garey and Johnson.

If true, this would of course imply $\mathrm{P} \neq \mathrm{NP}$, since if $\mathrm{P} = \mathrm{NP}$ there are *finite* sets that are NP-complete.

The Berman–Hartmanis conjecture fueled interest in the general question of just what can be proved about what NP-complete sets must ''look like.'' For instance, they cannot be finite unless $\mathrm{P} = \mathrm{NP}$, but can they have a ''small'' number of strings? If the isomorphism conjecture is true, any NP-complete set must have at least 2^{n^ε} strings of infinitely many lengths n, for some $\varepsilon > 0$. Can one prove that all NP-complete sets must have this many strings (assuming $\mathrm{P} \neq \mathrm{NP}$)? Can there be *sparse* NP-complete sets (i.e., sets with at most a polynomial number of strings of each length)? We now have a fairly clear answer to these questions.

Mahaney's Theorem [81] says that there are sparse NP-complete sets if and only if $\mathrm{P} = \mathrm{NP}$. Some years later, Ogihara and Watanabe gave a simpler proof of this theorem that also extends to a larger class of reducibilities [82]. Very recently, Buhrman and Hitchcock proved that the 2^{n^ε} bound (i.e., the bound that is implied by the isomorphism conjecture) is tight, unless the polynomial hierarchy collapses [83].

In spite of theorems such as this that seem to support the isomorphism conjecture, there seems to be little confidence these days that the conjecture is true. For example, if f is a cryptographically-secure one-way function, the set $f(\mathrm{SAT})$ does not appear to be isomorphic to SAT. There are a number of excellent surveys of work on the isomorphism conjecture, including Kurtz et al. [84], Mahaney [85], and Young [86].

Interestingly, when more restrictive notions of reducibility are considered, the isomorphism conjecture can be replaced by an isomorphism *theorem*. Recall from Section 6 that, in investigating subclasses of P it is useful to consider reductions computed by restricted classes of circuits (these are known as AC^0 reductions). With very few exceptions, natural problems that are known to be complete for some complexity class under any kind of reducibility can be shown to be complete under AC^0 reductions; thus the framework of complete sets that we use to understand the complexity of natural problems can be formulated entirely in terms of AC^0 reductions.

It turns out that for all complexity classes of interest, *all* the sets that are complete under AC^0 reductions are isomorphic under bijections f such that both f and f^{-1} are computable in AC^0 (and in fact the bijections can be shown to have a very restricted form) [33, 34, 87]. Thus some form of the isomorphism conjecture turns out to be true.

5. Where Are We now? (Barriers to Progress)

For many years, it was felt that radically new techniques would be needed, to make any significant progress on the P versus NP problem. This was because the ''traditional'' techniques in the complexity theorist's toolkit all ''relativized.'' What does this mean?

Recall the notion of having ''free'' access to a problem as a subroutine or ''oracle,'' as discussed in Section 4.1. If you have a class of programs or machines that characterize a complexity class such as P, NP, or EXP, and you provide each such machine with an oracle for problem A, one obtains the new ''complexity classes'' P^A, NP^A, and EXP^A. Baker, Gill, and Solovay [88] observed that all of the theorems that were proved using the usual proof techniques of the period (e.g., the theorem: $\mathrm{P} \neq \mathrm{EXP}$) would carry over relative to every oracle (and hence, for every A, $\mathrm{P}^A \neq \mathrm{EXP}^A$). They also showed that there were sets A and B such that

- $\mathrm{P}^A \neq \mathrm{NP}^A$
- $\mathrm{P}^B = \mathrm{NP}^B$

In fact, the set B can be chosen to be any PSPACE-complete set.

Over the next several years, a great many open problems in complexity theory were shown to admit contradictory relativizations in this sense. The general sense in the community was that it was generally hopeless to spend time on such questions, since they obviously required nonrelativizing proof techniques, and nobody had a good idea about how to develop such techniques. For instance, when Fortnow and Sipser presented an oracle A relative to which $\mathrm{coNP}^A \not\subseteq \mathrm{IP}^A$ [51], it convinced several people to stop thinking about trying to show that IP contained the polynomial hierarchy.

5.1 Nonrelativizing Proof Techniques

Thus the community sat up and took notice when it was shown that $\mathrm{IP} = \mathrm{PSPACE}$ [52, 53]. Finally, there was a fundamentally new set of tools to apply!

There followed an intense period of activity, where several new nonrelativizing theorems were proved. (Since the focus of this chapter is on NP and I want to avoid introducing new complexity classes, I will avoid describing these in more detail.) Recently, Aaronson and Wigderson took up the challenge of characterizing these ''new'' proof techniques, and determining what their limits are [89]. They define a new notion called ''algebrization'' (which I will not define here) and show that essentially all of the results that have been proved using nonrelativizing proof

techniques ''algebrize.'' They also show that the P versus NP problem cannot be resolved by any algebrizing proof technique (nor can the problem of showing that nondeterministic *exponential* time does not have circuits of polynomial size).

So once again we are in the position of needing fundamentally new proof techniques in order to make progress on some of the big open questions, but at least we once again know what some of the barriers are (and we also have received a healthy jolt of optimism from the experience of seeing the barrier of relativization fall a number of years ago). We shall overcome!

5.2 Natural Proofs

I would be dishonest if I were to give the impression that there is great optimism that we are on the verge of a breakthrough that will finally resolve the big open problems in complexity theory. There are many barriers to progress that have been identified.

Razborov and Rudich studied the approaches that have been followed in proving superpolynomial circuit size lower bounds on restricted classes of circuits, and observed that these approaches all fall into a certain ''natural'' approach to trying to prove circuit lower bounds [90]. They also proved that, if cryptographically secure one-way functions exist, then no such ''natural proof'' can prove that problems in NP require circuits of more than polynomial size. At one level, this was demoralizing, since it explained why some fairly modest-sounding lower bounds cannot be obtained without formulating a fundamentally new approach. On the other hand, work such as this can serve as a useful road map, helping the community to plan its assault on the big open questions in complexity theory.

There have been a number of suggestions of possible strategies to avoid the pitfalls represented by Natural Proofs and Algebrization [91–95]. I discuss some of these at more length (and provide more background and motivation) in a recent survey [96].

6. Conclusions: What Would a Solution Mean?

What would it mean to ''solve'' the P versus NP problem? How can one claim the 1 million dollar prize offered by the Clay Mathematics Institute [97]?

There seem to be three possible solutions (listed in my personal order of preference):

1. Prove that $\mathrm{P} \neq \mathrm{NP}$
2. Prove that $\mathrm{P} = \mathrm{NP}$
3. Prove that there is no proof one way or the other

Let us deal with the last option first. It is certainly the most frustrating possibility of the three. Imagine if a fast program for SAT *exists*, but there is no way to prove that it actually works correctly! (You could use such a program to provide satisfying assignments whenever it claimed that a formula was satisfiable—but how could you know it was correct in claiming that a formula was *not* satisfiable? Of course, in this case it would follow that P = coNP and thus there is some sense in which there are short ''proofs'' of unsatisfiability—but there might be no way to ''prove'' that these ''proofs'' were doing what they claimed.) I encourage the reader who wants to learn more about this possibility to read the entertaining survey on this topic written by Aaronson [98].

One point that Aaronson makes is that it is unlikely that a proof of this third possibility will surface any time soon. This is because current approaches to proving that a statement ϕ is independent of some formal system S almost always prove that ϕ is actually independent of the stronger system that results by augmenting S by all *true* first-order logic statements that contain only universal quantifiers. This is relevant, because it is known that if ''$P \neq NP$'' is independent of this stronger formal system, then one can show that SAT must have circuits of ''almost'' polynomial size. (Namely, it has circuits of size $n^{\alpha(n)}$ where α is a *very* slow-growing function [99].) That is, if ''$P \neq NP$'' is independent of this stronger formal system, then it is almost the same as SAT being easy to compute anyway.

Let us move on to the second possibility: P = NP. At one level this would be very disheartening because it would mean that the entire framework of completeness that seemed to explain so much was nothing but a glorious illusion. The optimistic way for a proof of P = NP to occur would actually yield an *efficient* (say, linear or quadratic time) algorithm for SAT. In this case, the consequences would be stunning. Mathematics could be automated. Machine learning and other tasks in artificial intelligence would become trivial. Corporate efficiency would soar as all sorts of optimization problems suddenly would become routine. Cryptography would become impossible as currently conceived. Impagliazzo describes this possible world as ''Algorithmica'' [71], and the reader is encouraged to read his description.

But this assumes that SAT has a truly efficient algorithm. The more pessimistic possibility that might emerge from a proof of P = NP is an algorithm for SAT with a running time of n^{1000}. Worse yet would be a nonconstructive proof, showing that a polynomial-time algorithm for SAT *exists* but providing no clue as to how to find such an algorithm. (There is precedent for nonconstructive proofs that problems can be solved in polynomial time [100], so this possibility cannot be dismissed out of hand.) Historically, when natural problems have been shown to lie in P, in almost all cases reasonably efficient algorithms have eventually been found. Problems that really require time n^{1000} to solve seem to be exotic oddities that nobody would really

want to try to compute anyway; natural problems seem to either have efficient algorithms or require essentially exponential time. This optimistic belief is really predicated on the computational universe not being truly perverse. We have no proof that this optimism is warranted.

Finally, let us consider the preferred outcome: someone finds a proof that $P \neq NP$. Again, there are several possibilities to consider.

One possibility is that $P \neq NP$ but that every NP-complete problem is easy on average in the sense of Levin [68]. This means that, for every fast algorithm for SAT there are some instances where the algorithm gives a wrong answer—but these instances essentially never come up in practice so you do not really notice it. This corresponds to the possible world that Impagliazzo calls ''Heuristica'' [71]. It might seem as if this outcome is indistinguishable from the case where $P = NP$, but as Impagliazzo points out [71], if $P = NP$ then every problem in the polynomial hierarchy has a polynomial-time algorithm. In contrast, if SAT is easy on average, it is not clear that the same is true for problems in the polynomial hierarchy.

Even if we are really lucky and a proof of $P \neq NP$ shows that SAT requires nearly exponential time, note that much, much more is required for some of the important applications that rely on intractability lower bounds (such as cryptography). At the very minimum, we would need *circuit size* lower bounds, in order to talk about intractability for any given input size (as discussed in Section 4).

A proof that $P \neq NP$ would not be the end of the story. It would only be the beginning.

Acknowledgment

This work was supported in part by NSF grants CCF-0830133, CCF-0832787, and DMS-0652582.

References

[1] S. Arora, B. Barak, Computational Complexity: A Modern Approach. Cambridge University Press. To appear, draft available at http://www.cs.princeton.edu/theory/complexity/.
[2] D.-Z. Du, K.-I. Ko, Theory of Computational Complexity, Wiley-Interscience, New York, 2000.
[3] O. Goldreich, Computational Complexity: A Conceptual Perspective, Cambridge University Press, Cambridge, 2008.
[4] I. Wegener, Complexity Theory: Exploring the Limits of Efficient Algorithms, Springer, Berlin, 2005.
[5] M. Sipser, The history and status of the P versus NP question, in: Proceedings of ACM Symposium on Theory of Computing (STOC), 1992, pp. 603–618.
[6] S.A. Cook, The importance of the P versus NP question, J. ACM 50 (1) (2003) 27–29.
[7] R. Impagliazzo, Computational complexity since 1980, in: Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS), Lecture Notes in Computer Science, vol. 3821, 2005, pp. 19–47.

[8] C.H. Papadimitriou, NP-completeness: A retrospective, in: International Conference on Automata, Languages, and Programming (ICALP), Lecture Notes in Computer Science, vol. 1256, 1997, pp. 2–6.
[9] A. Wigderson, P, NP and mathematics—A computational complexity perspective, Proc. ICM 2006 1 (2007) 665–712.
[10] D.S. Johnson, NP-completeness columns, Twenty-Six Columns published in J. Algorithms (1981–1992) and ACM Trans. Algorithms (2005-present), available at http://www.research.att.com/dsj/columns/.
[11] S.A. Cook, The complexity of theorem-proving procedures, in: Proceedings of ACM Symposium on Theory of Computing (STOC), 1971, pp. 151–158.
[12] L.A. Levin, Universal search problems, Probl. Inf. Transm. 9 (1973) 265–266.
[13] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller and J.W. Thatcher, (Eds.), Complexity of Computer Computations, 1972, pp. 85–104.
[14] C. Shannon, The synthesis of two-terminal switching circuits, Bell Syst. Techn. J. 28 (1949) 59–98.
[15] J. Hartmanis, R. Stearns, On the computational complexity of algorithms, Trans. Am. Mathematical Soc. 117 (1965) 285–306.
[16] A. Borodin, Computational complexity and the existence of complexity gaps, J. ACM 19 (1) (1972) 158–174.
[17] L.A. Levin, Computational complexity of functions, Theor. Comput. Sci. 157 (2) (1996) 267–271.
[18] M. Blum, A machine-independent theory of the complexity of recursive functions, J. ACM 14 (2) (1967) 322–336.
[19] Merriam-Webster, Webster's Seventh New Collegiate Dictionary, Merriam-Webster, 1969.
[20] S. Homer, A. Selman, Computability and Complexity Theory, Springer, Berlin, 2001.
[21] C.F. Slot, P. van Emde Boas, On tape versus core: An application of space efficient perfect hash functions to the invariance of space, in: Proceedings of ACM Symposium on Theory of Computing (STOC), 1984, pp. 391–400.
[22] E. Bernstein, U.V. Vazirani, Quantum complexity theory, SIAM J. Comput. 26 (5) (1997).
[23] S.A. Cook, An overview of computational complexity, Commun. ACM 26 (6) (1983) 400–408.
[24] J. Gill, Computational complexity of probabilistic turing machines, SIAM J. Comput. 6 (4) (1977) 675–695.
[25] P.W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM J. Comput. 26 (5) (1997) 1484–1509.
[26] R.E. Ladner, On the structure of polynomial time reducibility, J. ACM 22 (1) (1975) 155–171.
[27] S. Homer, Minimal degrees for polynomial reducibilities, J. ACM 34 (2) (1987) 480–491.
[28] J.M. Robson, N by N checkers is exptime complete, SIAM J. Comput. 13 (2) (1984) 252–267.
[29] A.R. Meyer, L.J. Stockmeyer, The equivalence problem for regular expressions with squaring requires exponential space, in: IEEE Symposium on Foundations of Computer Science (FOCS), 1972, pp. 125–129.
[30] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the theory of NP-completeness, W.H. Freeman and Company, New York, 1979.
[31] L.G. Valiant, The complexity of computing the permanent, Theor. Comput. Sci. 8 (1979) 189–201.
[32] N.D. Jones, Space bounded reducibility among combinatorial problems, J. Comput. Syst. Sci. 11 (1975) 68–85.
[33] M. Agrawal, E. Allender, S. Rudich, Reductions in circuit complexity: An isomorphism theorem and a gap theorem, J. Comput. Syst. Sci. 57 (1998) 127–143.

[34] M. Agrawal, The first-order isomorphism theorem, in: Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS), Lecture Notes in Computer Science, vol. 2245, 2001, pp. 70–82.
[35] R. Greenlaw, H.J. Hoover, W.L. Ruzzo, Limits to Parallel Computation: P-Completeness Theory, Oxford University Press, Oxford, 1995.
[36] H. Vollmer, Introduction to Circuit Complexity, Springer, Berlin, 1999.
[37] E. Allender, Arithmetic circuits and counting complexity classes, in: J. Krajicek, (Ed.), Complexity of Computations and Proofs, vol. 13, 2004, pp. 33–72. of *Quaderni di Matematica*.
[38] R.L. Rivest, A. Shamir, L.M. Adleman, A method for obtaining digital signatures and public-key cryptosystems (reprint), Commun. ACM 26 (1) (1983) 96–99.
[39] G. Brassard, A note on the complexity of cryptography, IEEE Trans. Inf. Theory IT-25 (1979) 232–233.
[40] L.A. Levin, One-way functions and pseudorandom generators, Combinatorica 7 (4) (1987) 357–363.
[41] S. Aaronson, Guest column: NP-complete problems and physical reality, SIGACT News 36 (1) (2005) 30–52.
[42] J. Hartmanis, Gödel, von Neumann and the P=?NP problem, in: G. Rozenberg and A. Salomas, (Eds.), Current Trends in Theoretical Computer Science, World Scientific Series in Computer Science, vol. 40, 1993, pp. 445–450. .
[43] L.J. Stockmeyer, The polynomial-time hierarchy, Theor. Comput. Sci. 3 (1) (1976) 1–22.
[44] K.W. Wagner, More complicated questions about maxima and minima, and some closures of NP, Theor Comput. Sci. 51 (1987) 53–80.
[45] L.J. Stockmeyer, A.R. Meyer, Cosmological lower bound on the circuit complexity of a small problem in logic, J. ACM 49 (6) (2002) 753–784.
[46] R. Karp, R. Lipton, Turing machines that take advice, L'Ensignement Mathématique 28 (1982) 191–210.
[47] J.-Y. Cai, $S_2^p \subseteq \mathrm{ZPP}^{\mathrm{NP}}$, J. Comput. Syst. Sci. 73 (1) (2007) 25–35.
[48] T.C. Venkatesan, S. Roy, Oblivious symmetric alternation, in: Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS), Lecture Notes in Computer Science number, no. 3884, 2006, pp. 230–241.
[49] L. Babai, S. Moran, Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes, J. Comput. Syst. Sci. 36 (2) (1988) 254–276.
[50] S. Goldwasser, S. Micali, C. Rackoff, The knowledge complexity of interactive proof systems, SIAM J. Comput. 18 (1) (1989) 186–208.
[51] L. Fortnow, M. Sipser, Are there interactive protocols for co-NP languages? Inf. Process. Lett. 28 (5) (1988) 249–251.
[52] C. Lund, L. Fortnow, H. Karloff, N. Nisan, Algebraic methods for interactive proof systems, J. ACM 39 (1992) 859–868.
[53] A. Shamir, IP = PSPACE, J. ACM 39 (4) (1992) 869–877.
[54] L. Babai, L. Fortnow, C. Lund, Non-deterministic exponential time has two-prover interactive protocols, Comput. Complexity 1 (1991) 3–40.
[55] S. Arora, C. Lund, R. Motwani, M. Sudan, M. Szegedy, Proof verification and the hardness of approximation problems, J. ACM 45 (3) (1998) 501–555.
[56] S. Arora, S. Safra, Probabilistic checking of proofs: A new characterization of NP, J. ACM 45 (1) (1998) 70–122.
[57] I. Dinur, The PCP theorem by gap amplification, J. ACM 54 (3) (2007) 12.
[58] T. Holenstein, Parallel repetition: Simplifications and the no-signaling case, in: Proceedings of ACM Symposium on Theory of Computing (STOC), 2007, pp. 411–419.

[59] M.R. Garey, D.S. Johnson, The complexity of near-optimal graph coloring, J. ACM 23 (1) (1976) 43–49.
[60] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, M. Szegedy, Interactive proofs and the hardness of approximating cliques, J. ACM 43 (2) (1996) 268–292.
[61] J. Hastad, Clique is hard to approximate within $n^{1-\varepsilon}$, Acta Mathematica 182 (1999) 105–142.
[62] J. Hastad, Some optimal inapproximability results, J. ACM 48 (4) (2001) 798–859.
[63] N. Nisan, A. Wigderson, Hardness vs. randomness, J. Comput. Syst. Sci. 49 (1994) 149–167.
[64] R. Impagliazzo, A. Wigderson, $P = BPP$ if E requires exponential circuits: Derandomizing the XOR lemma, in: Proceedings of ACM Symposium on Theory of Computing (STOC), 1997, pp. 220–229.
[65] A. Klivans, D. van Melkebeek, Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses, SIAM J. Comput. 31 (5) (2002) 1501–1526.
[66] P.B. Miltersen, N.V. Vinodchandran, Derandomizing Arthur-Merlin games using hitting sets, Comput. Complexity 14 (3) (2005) 256–279.
[67] R. Shaltiel, C. Umans, Simple extractors for all min-entropies and a new pseudorandom generator, J. ACM 52 (2) (2005) 172–216.
[68] L.A. Levin, Average case complete problems, SIAM J. Comput. 15 (1) (1986) 285–286.
[69] J. Wang, Average-case computational complexity theory, in: L. Hemaspaandra and A. Selman (Eds.), Complexity Theory Retrospective II, 1997, pp. 295–328. .
[70] J. Wang, Average-case intractible NP problems, in: D.-Z. Du and K.-I. Ko, (Eds.), Advances in Languages, Algorithms, and Complexity, 1997, pp. 313–378. .
[71] R. Impagliazzo, A personal view of average-case complexity, in: Structure in Complexity Theory Conference, 1995, pp. 134–147.
[72] L. Babai, L. Fortnow, N. Nisan, A. Wigderson, BPP has subexponential time simulations unless EXPTIME has publishable proofs, Comput. Complexity 3 (1993) 307–318.
[73] R. O'Donnell, Hardness amplification within NP, J. Comput. Syst. Sci. 69 (1) (2004) 68–94.
[74] A. Healy, S.P. Vadhan, E. Viola, Using nondeterminism to amplify hardness, SIAM J. Comput. 35 (4) (2006) 903–931.
[75] L. Trevisan, On uniform amplification of hardness in NP, in: Proceedings of ACM Symposium on Theory of Computing (STOC), 2005, pp. 31–38.
[76] P. Gopalan, V. Guruswami, Hardness amplification within NP against deterministic algorithms, in: IEEE Conference on Computational Complexity, 2008, pp. 19–30.
[77] L. Fortnow, Time–space tradeoffs for satisfiability, J. Comput. Syst. Sci. 60 (2000) 336–353.
[78] D. van Melkebeek, A survey of lower bounds for satisfiability and related problems, Found. Trends Theor.Comput. Sci. 2 (2007) 197–303.
[79] R. Williams, Time–space tradeoffs for counting NP solutions modulo integers, in: IEEE Conference on Computational Complexity, 2007, pp. 70–82.
[80] L. Berman, J. Hartmanis, On isomorphism and density of NP and other complete sets, SIAM J. Comput. 6 (1977) 305–322.
[81] S. Mahaney, Sparse complete sets for NP: Solution of a conjecture of Berman and Hartmanis, J. Comput. Syst. Sci. 25 (2) (1982) 130–143.
[82] M. Ogiwara, O. Watanabe, On polynomial-time bounded truth-table reducibility of NP sets to sparse sets, SIAM J. Comput. 20 (3) (1991) 471–483.
[83] H. Buhrman, J.M. Hitchcock, NP-hard sets are exponentially dense unless coNP $\subseteq$ NP/poly, in: IEEE Conference on Computational Complexity, 2008, pp. 1–7.
[84] S. Kurtz, S. Mahaney, J. Royer, The structure of complete degrees, in: A. Selman, (Ed.), Complexity Theory Retrospective, Springer, Berlin, 1990, pp. 108–146.

[85] S. Mahaney, The isomorphism conjecture and sparse sets, in: J. Hartmanis, (Ed.), Computational Complexity Theory, American Mathematical Society Proceedings of Symposia in Applied Mathematics #38, 1989, pp. 18–46 .
[86] P. Young, Juris Hartmanis: Fundamental contributions to isomorphism problems, in: A. Selman, (Ed.), Complexity Theory Retrospective, Springer, Berlin, 1990, pp. 28–58.
[87] M. Agrawal, E. Allender, R. Impagliazzo, R. Pitassi, S. Rudich, Reducing the complexity of reductions, Comput. Complexity 10 (2001) 117–138.
[88] T.P. Baker, J. Gill, R. Solovay, Relativizatons of the P =? NP question, SIAM J. Comput. 4 (4) (1975) 431–442.
[89] S. Aaronson, A. Wigderson, Algebrization: A new barrier in complexity theory, in: Proceedings of ACM Symposium on Theory of Computing (STOC), 2008, pp. 731–740.
[90] A. Razborov, S. Rudich, Natural proofs, J. Comput. Syst. Sci. 55 (1997) 24–35.
[91] M. Agrawal, Proving lower bounds via pseudo-random generators, in: Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS), Lecture Notes in Computer Science, vol. 3821, 2005, pp. 92–105.
[92] K. Mulmuley, M.A. Sohoni, Geometric complexity theory I: An approach to the P vs. NP and related problems, SIAM J. Comput. 31 (2) (2001) 496–526.
[93] E. Allender, M. Koucký, Amplifying lower bounds by means of self-reducibility, in: IEEE Conference on Computational Complexity, 2008, pp. 31–40.
[94] J. Friedman, Linear transformations in boolean complexity theory, in: Computation and Logic in the Real World (CiE 2007), vol. 4497, of Lecture Notes in Computer Science, 2007, pp. 307–315.
[95] T. Chow, Almost-natural proofs, in: IEEE Symposium on Foundations of Computer Science (FOCS), 2008.
[96] E. Allender, Cracks in the defenses: Scouting out approaaches on circuit lower bounds, in: Computer Science – Theory and Applications (CSR 2008), vol. 5010, of Lecture Notes in Computer Science, 2008, pp. 3–10.
[97] Clay Mathematics Institute, Millenium problems, http://www.claymath.org/millennium/.
[98] S. Aaronson, Is P versus NP formally independent? Bulletin of the EATCS 81 (2003) 109–136.
[99] S.A. Kurtz, M.J. O'Donnell, J.S. Royer, How to prove representation-independent independence results, Inf. Process. Lett. 24 (1) (1987) 5–10.
[100] M.R. Fellows, M.A. Langston, Nonconstructive tools for proving polynomial-time decidability, J. ACM 35 (3) (1988) 727–739.
[101] C. Umans, The minimum equivalent DNF problem and shortest implicants, J. Comput. Syst. Sci. 63 (2001) 597–611.
[102] M. Schaefer, Graph Ramsey theory and the polynomical hierarchy, J. Comput. Syst. Sci. 62 (2) (2001) 290–322.

Dynamically Typed Languages

LAURENCE TRATT

Bournemouth University, Poole, Dorset BH12 5BB, United Kingdom

Abstract

Dynamically typed languages such as Python and Ruby have experienced a rapid grown in popularity in recent times. However, there is much confusion as to what makes these languages interesting relative to statically typed languages, and little knowledge of their rich history. In this chapter, I explore the general topic of dynamically typed languages, how they differ from statically typed languages, their history, and their defining features.

ADVANCES IN COMPUTERS, VOL. 77
ISSN: 0065-2458/DOI: 10.1016/S0065-2458(09)01205-4

1. Introduction

As computing is often split into software and hardware, so programming languages are often split into dynamically and statically typed languages. The traditional, simplified, definition of dynamically typed languages is that they do not enforce or

check type safety at compile-time, deferring such checks until run-time. While factually true, this definition leaves out what makes dynamically typed languages interesting—for example, that they lower development costs [1] and provide the flexibility required by specific domains such as data processing [2].

For many people, dynamically typed languages are the youthful face of a new style of programming introduced in the past few years. In fact, they trace their roots back to the earliest days of high-level programming languages in the 1950s via Lisp [3]. Many important techniques have been pioneered in dynamically typed languages from lexical scoping [4] to Just-In-Time (JIT) compilation [5], and they remain a breeding ground for new ideas.

Systems programming—seen as ''serious'' and thus demanding statically typed languages—is often contrasted with scripting programming—seen as ''amateurish'' and thus needing little more than dynamically typed languages [1]. The derogative use of the term ''scripting'' led to the creation of other terms such as ''latently typed'' and ''lightweight languages'' to avoid the associated stigma. In reality, the absence or presence of static typing has a number of effects on the use and applicability of a language that simple comparisons ignore [6]. So while some tasks such as low-level systems (e.g., operating systems), resource critical systems (e.g., databases), or safety critical systems (e.g., systems for nuclear reactors) benefit from the extra rigor of statically typed languages, for many other systems the associated costs outweigh the benefits [7]. Gradually, dynamically typed languages have come to be seen as a valid part of the software development toolkit, and not merely second-class citizens [8].

From the mainstream's perspective, dynamically typed languages have finally come of age. More than ever before, they are used to build widely used real-world systems, often for the Web, but increasingly for domains that were previously the sole preserve of statically typed languages (e.g., [9]), often because of lower development costs and increased flexibility [10, 11].

It should be noted that since there is no central authority defining dynamically typed languages, there is great variation within those languages which are typically classified as dynamically typed languages; nevertheless all such languages share a great deal in common. In this chapter, I explore the general topic of dynamically typed languages, how they differ from statically typed languages, their history, and their defining features. The purpose of this chapter is not to be a cheerleader for dynamically typed languages—it is my contention that both statically typed and dynamically typed languages are required for the increasingly broad range of tasks that software is put to. Rather this chapter aims to explain what dynamically typed languages are and, by extension, to show where they may and may not be useful.

2. Defining Types

The lack of a widely understood definition of dynamically typed languages has resulted in many misunderstandings about what dynamic typing is. Perhaps because of this, alternative terms such as ''soft typing'' are sometimes used instead. Earlier I gave the simplified, and oft-heard, definition that a dynamically typed language is one that does not check or enforce type safety at compile-time. Inevitably, this simplified definition does not capture everything it should—the subtleties and variations in the use of dynamic typing preclude a short, precise definition.

In this section, I define various terms relating to dynamically typed languages, building up an increasingly accurate picture of what is meant by this term. Further reading on these topics can be found in [12, 13].

2.1 Types

At an abstract level, a type is a constraint which defines the set of valid values which *conform* to it. At the simplest level all apples conform to an ''Apples'' type and all oranges to an ''Oranges'' type. Types often define additional constraints: red apples are conformant to the ''Red Apples'' type, whereas green apples are not. Types are typically organized into hierarchies, meaning that all apples which conform to the ''Red Apples'' type also conform to the ''Apples'' type but not necessarily vice versa.

In programming languages, types are typically used to both classify values, and to determine the valid operations for a given type. For example, the `int` type in most programming language represents integers, upon which the operations +, –, and so on are valid. Most programming languages define a small number of built-in types, and allow user programs to add new types to the system. While, abstractly, most types define an infinite set, many built-in programming language types represent finite sets; for example, in most languages the `int` type is tied to an underlying machine representation of n bits meaning that only a finite subset of integers conform to it.

In many Object orientated (OO) programming languages, the notions of type and class are conflated. That is, a class ''Apple'' which defines the attribute ''pip'' and the operation ''peel'' also implicitly defines a type of the same name to which instances of the class automatically conform to. Because classes do not always define types to which the classes' instances conform [14, 15], in this chapter I treat the two notions separately. This means that, abstractly, one must define a separate type ''Apples Type'' to which instances of the ''Apple Class'' conform to. This definition of types may seem unnecessarily abstract but, as shall be seen later, the notion of type is used in many different contexts.

2.2 Compile-Time Versus Run-Time

In this chapter, I differentiate between errors which happen at *compile-time* and *run-time*. Compile-time errors are those which are determined by analyzing program code without executing it; run-time errors are those that occur during program execution.

Statically typed languages typically have clearly distinct compile-time and run-time phases, with program code converted by a compiler into a binary executable which is then run separately. In most dynamically typed languages (e.g., Converge, Perl, and Python) ''running'' a file both compiles and executes it. The blurring, from an external perspective, of these two stages often leads to dynamically typed languages being incorrectly classified as ''interpreted'' languages. Internally, most dynamically typed languages have distinct compilation and execution phases and, therefore, I use the terms compile-time and run-time identically for both statically and dynamically typed languages.

2.3 Static Typing

Before defining what dynamic typing is, it is easiest to define its ''opposite.'' Statically typed languages are those which define and enforce types at compile-time. Consider the following Java [16] code:

```
int i = 3;
String s = "4";
int x = i + s;
```

It uses two built-in Java types: `int` (representing integers) and `String` (Unicode character arrays). While a layman might expect that when this program is run, `x` will be set to `7`, the Java compiler refuses to compile this code; the compile-time error that results says that the `+` operation is not defined between values of type `int` and `String` (though see Section 2.6 to see why the opposite does in fact work). This is the essence of static typing: code which violates a type's definition is invalid and is not compiled. Such type-related errors can thus never occur in run-time code.

2.3.1 Implicit Type Declarations

Many statically typed languages, such as Java, require the explicit static *declaration* of types. That is, whenever a type is used it must be declared before hand, hence `int i = 3` and so on.

It is often incorrectly assumed that all statically typed languages require explicit type declarations. Some statically typed languages can automatically infer the

correct type of many expressions, requiring explicit declarations only when automatic inference by the compiler fails. For example, the following Haskell [17] code gives an equivalent compile-time error message to its Java cousin, despite the fact that the types of `i` and `s` are not explicitly declared:

```
let
  i = 3
  s = "4"
in
  i + s
```

In this chapter, I define the term ‘‘statically typed languages’’ to include both implicitly and explicitly statically typed languages.

2.3.2 Nominal and Structural Typing

As stated earlier, types are typically organized into hierarchies. There are two chief mechanisms for organizing such hierarchies. Nominal typing, as found in languages such as Java, is when an explicit named relationship between two types is recorded; for example, a user explicitly stating that Oranges are a subtype of Fruit. Structural typing, as found in languages such as Haskell, is when the components of two types allow a type system to automatically infer that they are related in some way. For example, the Orange type contains all the components of the Fruit type, plus an extra ‘‘peel thickness’’ component—a structurally typed system will automatically infer that all Oranges are Fruits, but that opposite is not necessarily true. Structural typing as described here is only found in statically typed languages although a similar feature—duck typing—is found in dynamically typed languages (see Section 5.7).

2.4 Dynamic Typing

Dynamic typing, at its simplest level, is when type checks are left until run-time. It is important to note that this is different than being *typeless*: both statically and dynamically typed languages are typed, the chief technical difference between them being *when* types are enforced. For example, the following Converge [18] code compiles correctly but when run, the `Int.+` function raises a run-time type exception `Expected arg 2 to be conformant to Number but got instance of String`:

```
i := 3
s := "4"
x := i + s
```

In this example, one can trivially statically analyze the code and determine the eventual run-time error. However, in general, dynamically typed languages allow code which is more expressive than any current type system can statically check [19]. For example, in non-OO languages static type systems typically prevent an individual function from having multiple return points if each returns results of differing, incompatible, types. In OO languages, on the other hand, the compiler statically determines the set of methods (considering subtypes) that an object method call refers to; in dynamically typed languages the method lookup happens at run-time. This run-time lookup is known as *late binding* and allows objects to dynamically alter their behavior, allowing greater flexibility in the manipulation of objects, the price being that lookups can fail as in the above example.

2.5 Safe and Unsafe Typing

Programs written with static types are often said to be *safe*[1] in the sense that type-related errors caught at compile-time cannot occur at run-time. However, most statically typed languages allow user programs to *cast* (i.e., force) values of one type to be considered as conformant to another type. For example, in C one can cast an underlying `int` value to be considered as an Orange, even if this is semantically nonsensical; instances of the two types are unlikely to share the same memory representation, and indeed may use different quantities of memory. Programs which abuse this feature can crash arbitrarily. Languages whose type systems can be completely overruled by the user are said to have an *unsafe* typing system.

In contrast to unsafe typing, languages with a safe type system do not allow the user to subvert it. This can be achieved either by disallowing casting (e.g., Haskell) or inserting run-time checks to ensure that casts do not subvert the type system (e.g., Java). For example, an object which conforms to the Red Apple type can always be cast to the Apple type. However, objects which conform to the Apple type can only be cast to the Red Apple type if the object genuinely conforms to the Red Apple type (or one of its subtypes); attempting to cast a Green Apple object to the Red Apple type will cause a run-time check to fail and an exception to be raised.

The concept of safe and unsafe type systems is orthogonal to that of static and dynamic typing. Static type systems can be safe (Java) or unsafe (C); all dynamically typed languages of which I am aware are safe.[2]

[1] This terminology is not universal, with ‘‘strong’’ and ‘‘weak’’ sometimes used in place of ‘‘safe’’ and ‘‘unsafe,’’ respectively.

[2] Note that assembly languages are often classified as dynamically and weakly typed; such languages fall considerably outside the scope of this chapter and are not considered herein.

2.6 Implicit Type Conversions

In many languages—both statically and dynamically typed—a number of implicit type conversions (also known as ''coercions'') are defined. This means that, in a given context, values of an ''incorrect'' type are automatically converted into the ''correct'' type. For example, in Perl [20], the addition of a number and a string evaluates to a number as the string is implicitly converted into a number; in contrast in Python [21] a run-time type error is raised. The C language defines a large number of implicit type conversions between number types. At the extreme end of the spectrum, the TCL language implicitly converts every type into a string [22]. Implicit type conversions need not be symmetrical; for example, in Java, adding a string to a number gives a compile-time warning (see Section 2.3 for an example) while adding a number to a string returns a string.

2.7 Terminology Summary

Table I shows a comparison of a number of languages with respect to the terms defined in this section. As is clearly shown, languages utilize types in almost every conceivable combination, making the traditional ''hard'' distinction between statically and dynamically typed languages seem very simplistic. Both classes of languages are typed, the chief technical difference between them being *when* types are enforced. The terms ''statically typed'' and ''dynamically typed'' are the source of much confusion but are sufficiently embedded within the community that it is unlikely that they will be superseded—hence, why I use those terms in this chapter. However, readers may find it more helpful to think of ''static typing'' as that performed at compile-time and dynamic typing that performed at run-time. This can help understand the real world, where most ''statically typed'' languages also utilize run-time type checking, and where some ''dynamically typed'' languages allow optional compile-time type checking.

TABLE I
LANGUAGE COMPARISON WITH RESPECT TO TYPING (''n/a'' MEANING ''NOT APPLICABLE'')

	C	Converge	Haskell	Java	Perl	Python	Ruby
Compile-time type checking	●	○	●	●	○	○	○
Run-time type checking	○	●	○	●	●	●	●
Safe typing	○	●	●	●	●	●	●
Implicit typing	○	n/a	●	○	n/a	n/a	n/a
Structural typing	○	n/a	●	○	n/a	n/a	n/a
Run-time type errors	○	●	○	●	●	●	●
Implicit type conversions	●	○	○	●	●	○	●

3. Disadvantages of Static Typing

The advantages of static typing are widely known [23] and include:

- Each errors detected at compile-time prevents a run-time error.
- Types are a form of documentation/comment.
- Types enable many forms of optimization.

Taken at face value, the first of these is a particularly compelling argument: why would anyone choose to use less reliable languages? In reality, the absence or presence of static typing has a number of effects on the use and applicability of a language that are not explained by the above. In particular, because the overwhelming body of research on programming languages has been on statically typed languages, the disadvantages of statically typed languages are rarely enumerated. In this section, I enumerate some of the weaknesses of static typing and why it is, therefore, not equally applicable to every programming task.

3.1 Static Types Are Inexpressive

As defined in Section 2.1, types are constraints. In practice, programming language types most closely conform to the intuitive notion of ''shape'' or ''form.'' Perhaps surprisingly, in some situations types can be too permissive and in others too restrictive (for an extreme example of this duality, see overloading in Java [24]). Furthermore, as static types need to be checked at compile-time, by definition they lack run-time information about values, further limiting their expressivity (interestingly, the types used in dynamically typed languages are virtually identical in expressivity to those used in statically typed languages, probably due to cultural expectations rather than technical issues).

3.1.1 Overly Permissive Types

Consider the following Java code which fails at run-time with a `division by zero` exception:

```
int x = 2;
int y = 0;
int z = x/y;
```

Looking at this, programmers of even moderate experience can statically spot the cause of the error: the divisor should not be zero. Java's compiler cannot statically

detect this error because the `int` type represents real numbers including zero; thus the above code is statically type correct according to Java's types. Not only is there not a type in Java which represents the real numbers excluding zero, there is no mechanism for defining such a type in a way that would result in equivalent code leading to a compile-time error. This limitation is shared by virtually all statically typed languages.

As suggested above, the static types available in today's mainstream languages are particularly inexpressive. Though research languages such as Haskell contain more advanced type systems, they still have many practical limitations. Consider the `head` function, which takes a list and returns its first element; given an empty list, `head` raises a run-time exception. Taking the head of an empty list is a common programming error, and is particularly frustrating in programming languages such as Haskell whose run-time error reporting makes tracking down run-time errors difficult [25]. It is possible to make a new list type, and a corresponding `head` function, which can statically guarantee that the head of an empty list will never be taken [26]; however, this only works for lists whose size is always statically known. Lists that are created on the basis of user input—a far more likely scenario—are highly unlikely to be statically checkable. Trying to use a type system in this way adds significant complexity to user programs with only minimal benefits.

Because of the general inexpressiveness of static types, an entirely separate strand of research tries to statically analyze programs to detect errors that escape static type checkers (see, e.g., [27] for work directly related to the `head` function).

3.1.2 Overly Restrictive Types

Since any practical type system needs to be both decidable and sound, they are not complete; in other words, certain valid programs will be rejected by the type checker [28, 29]. For example, type systems provide a fixed, typically small (or even empty), number of ways of relating types, with Object orientated languages allowing types to be defined as subtypes of others allowing a certain kind of polymorphism. However, programmers often need to express relationships between types that static types prevent, even in research languages with advanced type systems such as ML [19].

3.1.3 Type System Complexity

From a pragmatic point of view, relatively small increases in the expressivity of static type systems cause a disproportionately large increase in complexity [2, 30]. This can be seen clearly in Abadi and Cardelli's [15] theoretical work which defines static type systems of increasing expressiveness for Object orientated languages; their latter systems, though expressive, are sufficiently complex that, to the best of my knowledge, they have never been implemented in any language.

3.2 Types Are Represented by a Separate Language

Since most of us are used to the presence of explicit static types, it is easy to overlook the fact that they are represented by an entirely different language from the base programming language. In other words, when learning the syntax and semantics of programming *X*, one must also learn the syntactically and semantically distinct static type language *XT*. That *X* and *XT* are, at heart, separate languages can be seen by the very different types of errors that result from violating each one's semantics. While programming languages have developed various mechanisms when presenting error information to aid programmers, the error messages from static type systems are often baroque and hard to understand [31].

3.3 Type Systems' Correctness

Static type systems are often the most complex parts of a programming language's specification. Because of this, it is easy for them to contain errors which then result in ''impossible'' run-time behavior [12].

A famous example comes from Eiffel [32], one of the first ''mainstream'' Object orientated languages. Eiffel allows overridden methods to use subtypes of the parameters in the superclass. Consider classes `A1`, `A2`, `B1`, `B2`, and `B3`, where `A2` subclasses `A1`, and `B3` subclasses `B2` which subclasses `B1`. In Object orientated languages, in general, instances of subclasses (e.g., `A2`) can be considered as instances of superclasses (e.g., `A1`); intuitively, this is because subclasses have type-identical versions of everything in the superclass plus, optionally, extra things. Eiffel subtly changes this, so that subclasses can contain type-compatible versions of everything in the superclass plus, optionally, extra things. Therefore, in Eiffel, one can define a method `m(p1:B2)` (meaning that m has a parameter `p1` of type `B2`) in class `A1` that is overridden in class `A2` by `m(p1:B3)`. If an instance of `A2` is considered to be an instance of its superclass `A1`, then an instance of `B2` can validly be passed to `A2::m` which may then attempt to access an attribute present only in instances of the subclass `B3`. Such *covariant* typing is unsafe and programs which utilize it can crash arbitrarily at run-time despite it satisfying Eiffel's type-safety rules [33].

As the Eiffel example suggests, and despite their formal veneer, the vast majority of static type systems are not proved correct; some are sufficiently complex that a full proof of correctness is impractical or impossible [23]. Eiffel again gives us a good example of the subtleties that type systems involve: counterintuitively type theory shows that `A2::m` could safely use supertypes of the parameter types in `A1::m` (i.e., *contravariant* typing), so `A2::m(p1:B1)` is type-safe [34].

Flaws discovered in type systems are particularly invidious, because changes to type systems will typically break most extant programs; for this reason, even modern versions of Eiffel contain the above flaw (whilst alleviating it to some extent).

3.4 System Ossification

Virtually all software systems are changed, often continuously, and rarely in a planned or anticipated manner, after their original development [35]. It is, therefore, an implicit requirement that software be amenable to such change, which further implies that programming languages facilitate such change.

When changing a program, it is often desirable to change small sections at a time and see the effect of that change on that particular part of the program, so that any new errors can be easily related to the change; when performing such changes it is often expected that the program as a whole may not work correctly. Static type systems often prevent this type of development, because they require that the system as a whole is always type correct: it is not possible to temporarily turn off static type checking. As static types make changing a system difficult, they inevitably cause systems to prematurely ossify, making them harder to adapt to successive changes [36].

3.5 Run-Time Dynamicity

Software is increasingly required to inspect and alter its behavior at run-time, often in the context of critical systems that are expected to run without downtime, which must be patched whilst still running [37]. Traditionally, statically typed languages' compilers have discarded most information about a program's structure, its types, and so on during the compilation process, as they are not considered central to the program's execution. This means that most such languages are incapable of meaningful *reflection* [38]. Of those that do (e.g., Java), the ability to change the run-time behavior of a program is relatively limited because of the possibility of subverting the type system. This means that statically typed languages have typically proved difficult to use in systems that require run-time dynamicity [36].

4. History

Dynamically typed languages have a long and varied history. While few dynamically typed languages have had a direct impact on the programming mainstream, they have had a disproportionate effect on programming languages in general. Perhaps because of their inherently flexible nature, or the nature of the people attracted to them, dynamically typed languages have pioneered a bewildering

array of features. Thus, the history of dynamically typed languages is intertwined with that of statically typed programming languages which, often after a significant delay, have incorporated the features pioneered in dynamically typed languages.

To the best of my knowledge, a history of dynamically typed languages has not yet been published, although the History of Programming Languages (HOPL) conferences[3] include histories of several of the most important languages (see, e.g., [39–41]). A full history is far beyond the scope of this chapter. However, there have been several important innovations and trends which explain the direction that dynamically typed languages have taken and why current dynamically typed languages take the shape they do. The initial history of dynamically typed languages is largely of individual languages—Lisp and Smalltalk in particular—while the more recent history sees groups of languages—such as so-called ''scripting'' languages including Perl, Python, and Ruby—forging a common direction. Therefore, this section enumerates, in approximately chronological order, the major points in the evolution of dynamically typed languages.

4.1 Lisp and Its Derivatives

Arguably, the first dynamically typed language certainly the oldest still in use, and without doubt the most influential dynamically typed language is Lisp [3]. Created in the 1950s, Lisp was originally intended as a practical notation for the λ-calculus [42]. Lisp is notable for its minimal syntax, the smallest of any extant programming language used in the real world, allowing it a similarly small and uniform semantics. This simplicity—it was quickly discovered that it is possible to specify a minimal Lisp interpreter in a single page of Lisp code—made its implementation practical on machines of the day. That the innovations pioneered by, and within, Lisp are too many to mention can be inferred from its introduction of the `if-then-else` construct now taken for granted in virtually all programming languages.

Simply labeled, Lisp is an impure functional language. To modern eyes, Lisp is unusual because its concrete syntax uses prefix notation as can be seen from this simple example of a Fibonacci function:

```
(defun fib (n)
  (if (= n 0)
    0
    (if (= n 1)
      1
      (+ (fib (- n 1)) (fib (- n 2))))))
```

[3] http://research.ihost.com/hopl/

Lisp's minimal syntax allows it to be naturally represented by Lisp lists. Since lists can be inspected, altered, and created this led to what is arguably Lisp's most distinctive feature: macros. A macro is effectively a special function which, at compile-time, generates code. Macros allow users to extend a programming language in ways unforeseen by its creators [43]. Macros have, therefore, been a key facilitator in Lisp's continued existence, as they allow the spartan base language to be seamlessly extended: a typical Lisp implementation will implement most of its seemingly ''primitive'' control structures through macros (see Section 5.3.2). Despite many attempts, it was not until the late 1990s that a syntactically rich, statically typed language gained a practical macro-like facility broadly equivalent to Lisp's (see [44, 45]).

Lisp invented the concept of garbage collection [46] where memory allocation and deallocation is handled automatically by the Lisp interpreter or VM. Lisp was also the first language whose implementations made significant efforts to address performance concerns [47]; many of the resulting implementation techniques have become standard parts of subsequent language implementations.

4.1.1 Scheme

Lisp has spawned many dialects, the most significant of which is Scheme [4]. For the purposes of this chapter, Scheme can be thought of as a version of Lisp with a minimalist aesthetic, particularly with regard to its libraries. While Lisp has seen reasonable industrial usage (particularly in the 1980s, when it was the language of choice for artificial intelligence work), Scheme has largely been a research language, albeit a very influential one.

Scheme was the first language to introduce closures, allowing full lexical scoping, simplifying many types of programming such as graphical user interface (GUI) programming. It also popularised the concept of continuations, allowing arbitrary control structures to be constructed by the user [48]. Scheme also showed that functions and continuations could be treated as first-class objects. Much of the foundational work on safe, powerful, macros was done in Scheme (see, e.g., [49, 50]).

4.2 Smalltalk

Smalltalk is Lisp's nearest rival in influence. Put simply, Smalltalk is a small, uniform Object orientated language, heavily influenced by Lisp and Simula [51]. Compared to later languages, Smalltalk's syntax is small and uncomplicated (though not as minimalistic in nature as Lisp's); however, in most other ways, Smalltalk-80 [52] (the root of all extant Smalltalk's) is recognizably a modern, Object orientated, imperative programming language.

Smalltalk pioneered the idea of ‘‘everything is an object’’ where even primitive values (integers, etc.) appear as normal objects whose classes are part of the standard class hierarchy. Smalltalk has extensive metaprogramming abilities. Reflection allows programs to query and alter themselves [53]. A metaobject protocol (MOP) [54] allows objects to change the way they behave; from the perspective of this chapter, the most significant of these abilities is metaclasses [55] (see Section 5.3.1).

In Smalltalk, every object can be queried at run-time to find out its type. In common with most Object orientated languages, a Smalltalk class also implicitly defines a type (see Section 2.1), so the ‘‘type’’ of an object is the `Class` object which created it. A metaclass is simply the type of a class. In Smalltalk, the default metaclass for a class is called `Metaclass`; a cycle is created in the type hierarchy so that `Metaclass` is its own type. Metaclasses allow Smalltalk to present a uniform, closed world where every object in a running system is typed by an object in the same running system. Only a small amount of bootstrapping is needed to create this powerful illusion (later proposals have shown how the metaclass concept can be further simplified [56]).

4.3 Text-Processing Languages

Text processing is a perennial programming task, and several languages have been wholly or mostly designed with this in mind. This domain has been dominated by dynamically typed languages, because the processing of unstructured data benefits greatly from the flexibility afforded by such languages [2].

The first languages aimed at these tasks, most noticeably SNOBOL4 [57], were effectively domain-specific languages (DSLs) for text processing, and were not suitable for more general tasks [58]. One of SNOBOL4’s direct successor languages was Icon [59], which introduced a unique expression evaluation system which dispenses with Boolean logic and allows limited backtracking within an imperative language. This allows one to express complex string matching which can naturally evaluate multiple possibilities.

Sed and AWK [60] represent an entirely different strand of text-processing languages from SNOBOL and Icon. They can be thought of as enhanced UNIX shell languages, with AWK extending Sed with a number of more general programming language constructs. Perl [20] represents the final evolution of this family of languages. Reflecting its role as a tool for ad hoc development, it integrates a bewildering number of influences to an AWK base, and is notable for having arguably the most sophisticated—or, depending on ones point of view, complex—syntax of any programming language.

Most of the above languages are not, in the widely understood sense, general purpose languages. Icon is the most obviously general purpose language, although

because of the many idioms it encompasses, Perl has been used in many domains. Because of the ubiquity of Sed and AWK and, in the early years of the Web, Perl's dominance of server side processing, these languages have been more widely used than any other category of dynamically typed languages.

4.4 Declarative Languages

Although dynamically typed languages are often implicitly assumed to be imperative languages, dynamic typing is equally applicable to declarative languages which, for the purposes of this chapter, I define to mean logic and ''pure'' functional languages (i.e., those without side effects). Prolog [61] was among the first, and remains the most widely used, logic language. Logic languages are very unlike ''normal'' languages, with the user declaring relations among data, and then stating a goal over this which the language engine then attempts to solve—the order in which statements in the language are executed is nonlinear.

Pure functional languages[4] have largely been confined to the research lab and have tended to be coupled with exotic static type systems. Although Erlang [62] started existence as a distributed variant of Prolog, it has since evolved to become one of the few dynamically typed pure functional languages. This perhaps reflects its industrial origins where it was designed to implement robust, scalable, distributed systems, particularly telephony systems [63]. Erlang is arguably the most successful pure functional language yet with several million LoC systems. By eschewing static types, it is able to focus on the hard issues surrounding distributed systems, including a number of unique concepts relating to message passing and fault tolerance.

4.5 Prototyping Languages

Object orientated languages derived from SIMULA such as Smalltalk are class-based languages: objects are created by instantiating classes. While everything in Smalltalk is an object, practically speaking classes are a very distinguished type of object from the user's perspective. Self [64] aimed to distill the Object orientated paradigm down to its bare essentials: objects, methods, and message sends. In particular, Self removed classes as a fundamental construct; new objects are created by *cloning* another object. The notion of type in Self, and other prototyping languages, is thus subtly different than in other languages.

[4] The ''pure'' name is a misnomer, since a truly side effect free program would be incapable of input/output. Informally, ''pure'' is generally used to mean ''no explicit side effects such as assignment.''

Because of their minimalistic nature, raw prototyping languages tend to be particularly inefficient. Self pioneered a number of important implementation techniques [65] that ultimately allowed Self to become one of the highest performing dynamically typed languages. Much of this work has found its way into other languages, including statically typed languages such as Java [5].

4.6 Modern ''Scripting'' Languages

The resurgence of interest in dynamically typed languages is largely due to what were originally dismissively called ''scripting'' languages [1], which had their roots in text-processing languages such as Sed and AWK (see Section 4.3). Unlike many of the languages described earlier in this section, these languages were not designed with innovation as a primary goal, and instead emphasized consolidation and popularization. They have, therefore, focused on practical issues such as portability, and shipping with extensive libraries. TCL [22] was the first such language which gained reasonable popularity in large part because of its bundled GUI toolkit. Python and Ruby [66]—fundamentally very similar languages once surface syntax issues are ignored—can be seen as modernized, if less internally consistent, versions of Smalltalk. Because of their inherent flexibility, such languages were initially often used to ''glue'' other systems together, but have increasingly seen to be useful for a wide range of programming tasks, such as Web programming tasks. Lua [67] is a smaller language (both conceptually, and in its implementation) than either Python and Ruby, and has been more explicitly designed as an embeddable programming language; it has been used widely in the computer games industry to allow the high-level definition and extension of games [68].

While this subcategory of dynamically typed languages has not greatly advanced the state of the art, it has been the driving factor in validating dynamically typed languages and making them a respected part of a programmer's toolbox. Most new systems written using dynamically typed languages use this category of languages.

5. Defining Features

In previous sections, I have defined the fundamental terms surrounding types and programming languages, and presented a brief history of dynamically typed languages. In this section, I enumerate the defining features and characteristics of dynamically typed languages, and explain why they make such languages interesting and useful. Some of these features and characteristics have recently found

their way into new statically typed languages, either as a core feature or as library add-ons. However, no statically typed language contains all of them, nor is that likely to occur for both technical and cultural reasons.

5.1 Simplicity

A defining characteristic of virtually all dynamically typed languages is conceptual simplicity. Fundamentally, dynamically typed languages are willing to trade run-time efficiency for programmer productivity. Such simplicity makes both learning and using dynamically typed languages simpler, in general, than statically typed languages since there are less ''corner cases'' to be aware of. At its most extreme, Lisp's minimal syntax means that a full interpreter written in Lisp can fit on one page. Although most dynamically typed languages include as standard a greater degree of syntax and control structures than Lisp, this general principle remains.

At the risk of stating the obvious, dynamically typed languages do not contain constructs relating to static types. This is a significant form of simplification, as although static typing is sometimes considered to be the simple ''tagging'' of variables with a given type name, static typing has a much more pervasive effect on a language. For example, static typing requires an (often significant) extension to a language's grammar to allow type ''tags'' to be expressed and requires concept(s) allowing static types to be related to one another (e.g., the Java concept of interface).

The learning curve of dynamically typed languages is considerably shallower than for most statically typed languages. For example, in many dynamically typed languages, the classic ''hello world'' program is simply `print "Hello world!"` or a minor syntactic variant. In Java, at the other extreme, it requires a seven-line program—in a file whose name must exactly match the class contained within it—using a bewildering array of unfamiliar concepts. While programming beginners obviously struggle with the complexity that a language like Java forces on every user, it is widely known that programming professionals find it easier to learn new dynamically typed languages [1].

5.2 High-Level Features

Dynamically typed languages pioneered what are often informally known as ''high-level features''—those which abstract away from low-level machine concerns.

5.2.1 Built-in Data Types

Whereas many statically typed languages provide only very simple built-in data types—integers and user-defined structures—dynamically typed languages typically provide a much richer set. The two universal data types are lists (automatically

resizing arrays) and strings (arbitrary character arrays); most dynamically typed languages also provide support for dictionaries (also known as associative arrays or hash tables; fast key/value lookup) and sets. These data types are typically tightly integrated into the main language, often with their own syntax, and used consistently and frequently throughout libraries. In contrast, most statically typed languages defer most such data types to libraries; consequently, they are rarely as consistently or frequently used.

Complex data structures are often naturally expressed using just built-in data types. For example, the following Converge code shows how dictionaries of sets representing room numbers and employees are naturally represented:

```
  x := Dict{10 : Set{"Fred", "Sue"}, 17 : Set{"Barry",
"George", "Steve"}, 18 : Set{"Mark"}}
  x[10].add("Andy")
  x[17].del("Steve")
```

After the above has been evaluated the dictionary referenced by `X` looks as follows:

```
  Dict{10 : Set{"Fred", "Andy", "Sue"}, 17 : Set{"Barry",
"George"}, 18 : Set{"Mark"}}
```

Using built-in data types not only improves programmer productivity, but also execution speed as built-in data types is highly optimized.

5.2.2 Automatic Memory Management

Manual memory management—when the programmer must manually allocate and free memory—wastes programmer resources (consuming perhaps around 30–40% of a programmer's time [69]) and is a significant source of bugs [46]. Lisp was the first programming language to introduce the concept of garbage collection, meaning that memory is automatically allocated and freed by the language run-time, largely removing this burden from the programmer. Virtually, all dynamically typed languages (and, more recently, most statically typed languages) have followed this lead.

5.3 Metaprogramming

Metaprogramming is the querying, manipulation, or creation of one program by another; often a program will perform such actions upon itself. Metaprogramming can occur at either, or both of, compile-time or run-time. Dynamically typed languages have extensive metaprogramming abilities.

5.3.1 Reflection

Formally, reflection can be split into three main aspects [70, 71]:

1. *Introspection*: the ability of a program to examine itself.
2. *Self-modification*: the ability of a program to alter its structure.
3. *Intercession*: the ability of a program to alter its behavior.

For the purposes of this chapter, reflection is considered to be a run-time ability. For example, in Smalltalk, programs can perform deep introspection on objects at run-time to determine their types (see Section 4.2). In the following Smalltalk examples, ''→'' means ''evaluates to'':

```
2 + 2 → 4
(2 + 2) class → SmallInteger
(2 + 2) class class → SmallInteger class
(2 + 2) class class class → Metaclass
```

Self-modification allows behavior to be added, removed, or changed at run-time. For example, in Smalltalk if a variable `ie` references an appropriate method (the definition of which is left to the reader), then it can be added to the `Number` class, so that all numbers can easily test whether they are odd or even:

```
3 isEven → Message not understood
Number addSelector: #isEven withMethod ie → Adds method
  isEven to Number
3 isEven → false
```

Unfettered run-time modification of a system is dangerous, since it can have subtle, unintended consequences. However, careful use of reflection allows programmers to bend a language to their particular circumstances rather than the other way around. Most dynamically typed languages are capable of introspection; many are capable of self-modification; relatively few are capable of intercession (Smalltalk being one of the few). While a few statically typed languages such as Java support the introspective aspects of reflection, few are as consistently reflective as Smalltalk and its descendants, and none allow the level of manipulation as shown above.

Some OO languages have a MOP [54] which allows intercession, as objects can alter the way they respond to message sends. For example, in Python, objects can override the `__getattribute__` function which receives a message `name` and returns an object of its choosing. The following example code (although too simple

for production use) shows how Python objects can be made to appear to automatically have automatic ‘‘getter’’ methods if they do not exist:

```
class C(object):
  x = 2
  def __getattribute__(self, name):
    if name.startswith("get_"):
      v = object.__getattribute__(self, name[4:])
      return lambda : v
    else:
      return object.__getattribute__(self, name)

i= C()
print i.x
print i.get_x()
```

In this example, both `i.x` and `i.get_x()` evaluate to the same result. Similar tricks can be played with the setting and querying of object slots. While delving into the MOP can easily introduce complications such as infinite loops, it can be useful, as in this example, to allow one object to emulate the behavior of another, allowing otherwise incompatible frameworks and libraries to interact. Reflection also allows much deeper changes to a system such as allowing run-time modification of whole program aspects [72].

5.3.2 Compile-Time Metaprogramming

Compile-time metaprogramming allows the user to interact with the compiler to allow the construction of arbitrary program fragments. Lisp’s macros are the traditional form of compile-time metaprogramming and are used extensively to extend the minimal base language. For example, the `when` control structure is a specialized form of `if`, taking a condition and a list of expressions; if the condition holds, `when` evaluates all expressions, returning the result of the final expression. In Common Lisp [73] (alongside Emacs Lisp, one of the major extant Lisp implementations) `when` can be implemented as follows:

```
(defmacro when (cond &rest body)
  `(if ~cond (progn ~@body)))
```

Whenever a ‘‘function call’’ to `when` is encountered during compilation, the above macro is executed and the resultant generated code statically replaces the ‘‘function call.’’ The two major features in the above are the quote’ which in essence returns the quoted expression as an abstract syntax tree (AST) (i.e., without evaluating it) and the insertion ~ which inserts one Lisp AST in another.

Because macros in Lisp are often considered to rely on some of Lisp's defining features—in particular its minimal syntax which means that Lisp ASTs are simply lists of lists—subsequent dynamically typed languages did not have an equivalent system. In a rare occurrence, the statically typed languages MetaML [44] and then Template Haskell [45] showed how a practical compile-time metaprogramming system could be naturally integrated into a modern syntactically rich language. Compile-time metaprogramming is slightly more generic in concept than traditional macros, as it allows users to interact with the compiler, where such interactions may not always lead to the generation of code. Converge (created by this chapters author) integrates a Template Haskell-like system into a dynamically typed language, and uses it to implement a syntax extension feature which allows syntactically distinct DSLs to be embedded into normal programs.

5.3.3 Eval

Colloquially referred to by its short name, ''eval'' refers to the ability, almost wholly confined to dynamically typed languages, to evaluate arbitrary code expressions as strings at run-time. In other words, code fragments can be received from, for example, end users, evaluated and the resulting value used for arbitrary purposes. Note that eval is very different from compile-time metaprogramming, since expressions are evaluated at run-time, not compile-time, and any value can be returned (not just ASTs). While eval has many obvious downsides—allowing arbitrary code to be executed at run-time has severe security implications—when used carefully (e.g., in configuration files) it can reduce the need for arbitrary mini-programming languages to be implemented within a system.

5.3.4 Continuations

First popularised in Scheme, continuations remain a relatively exotic construct, with support only found in a handful of other languages, noticeably including Smalltalk. At a high-level, they can be thought of as a generalized form of coroutine [48] which allows a safe way of defining ''goto'' points, capturing a certain part of the current program state and allowing that part to be suspended and later resumed. Continuations are sufficiently powerful that all other control structures can be defined in terms of them.

The low-level power of continuations, and the fact that they subvert normal expectations of control flow, has meant that they have been talked about rather more than they have been used. However, they have recently shown to be a natural match for Web programming, where the back button in Web browsers causes huge problems because it is effectively an ''undo''; most Web systems give unpredictable

and confusing results if the back button is used frequently. Continuations can naturally model the chain of resumption points that represent each point in the users browsing history, as can be seen in the Smalltalk Seaside framework [74]. This means that Web systems respect user's intuition when the back button is used, but are not unduly difficult to develop.

5.4 Refactoring

Refactoring is the act of applying small, behavior-preserving transformations, to a system [75]. The general aim of refactoring is to maintain, or restore, the internal quality of a system after a series of changes so that further changes to the system are practical. A key part of the refactoring definition ''behavior-preserving'': it is vital that refactorings do not introduce new errors into a system. In practice, two distinct types of refactorings can be identified:

1. Small, tightly defined, and automatable refactorings. Exemplified by the ''move method'' refactoring where a method is moved from class `C` to `D`.
2. Larger, typically project-specific, nonautomatable refactorings. A typical example is splitting a module or class into two to separate out functionality.

Statically typed languages have an inherent advantage over dynamically typed languages in the first type of refactoring because of the extra information encoded in static types. However, static types are a burden in the second type of refactoring because they always require the entire system to be type correct. This means that it is not possible to make, and test, small local changes to a subsystem when such changes temporarily violate the type system; instead, the entire refactoring must be implemented in one fell swoop which means that any resulting errors are difficult to relate to an individual action. Counterintuitively, perhaps, static types inhibit large-scale refactorings, tending to ossify a program's structure (see Section 3.4). The flexibility of dynamically typed languages, on the other hand, encourages continual changes to a system [36], though it is often wise to pair it with a suitable test suite to prevent regressions (see Section 6.2).

5.5 ''Batteries Included'' Libraries

Traditionally, many statically typed languages—from Algol to Ada—have been designed as paper standards, detailing their syntax and semantics, but typically agnostic as to libraries. Such languages are then implemented by multiple vendors, each of which is likely to provide different libraries. In contrast, most dynamically typed languages—with the notable exception of the Lisp family—have been defined by their initial implementation and its accompanying libraries. The majority of

modern dynamically typed languages (see Section 4.6) come with a rich set of standard libraries—the so-called ''batteries included'' approach[5] [76]—which encompass enough functionality to be suitable for a majority of common programming tasks. Implicit in this is the assumption that if the initial implementation is replaced, the standard library will be provided in a backward-compatible fashion; in comparison to paper-based standards, it is often difficult to distinguish between the language and its libraries. Furthermore, due to the emphasis on a rich set of standard libraries, it is relatively easy to define new, external libraries without requiring the installation of many dependent libraries.

As described in Section 6.1, the performance of dynamically typed languages varies from slightly to significantly slower than statically typed languages; however, suitable use of libraries (which are typically highly optimized) can often significantly diminish performance issues.

5.6 Portability

Portable software is that which runs on multiple target platforms. For the purposes of this chapter, a platform can be considered to be a combination of hardware and operating system.[6] For most nonspecialized purposes, users wish their software to run on as many platforms as practical.

One way of achieving portability is to allow programs to deal, on an as-needs basis, with known variations in the underlying platform; the other is to provide abstractions which abstract away from the hardware and the operating system [77]. Since dynamically typed languages aim to present a higher-level view of the world to programs (see, e.g., Section 5.2), they follow this latter philosophy. There are many examples of such abstractions, but two in particular show the importance of abstracting away from the hardware and the operating system. First, ''primitive types'' such as integers will typically automatically change their representation from an efficient but limited machine type to a variably sized container as necessary, thus preventing unintended overflow errors. Second, file libraries provide simple `open` and `read` calls (note that garbage collection typically closes files automatically in dynamically typed languages, so explicit calls to `close` are less important) which abstract away from the wide variety of file-processing calls found in different operating systems. By providing such abstractions, dynamically typed programs

[5] While this phrase originated in the Python community, it reflects a common belief among most dynamically typed languages.

[6] A precise definition of platform would have to cope with many ontological difficulties, such as the Java Virtual Machine which defines a ''platform-independent'' platform of its own.

are typically more portable than most statically typed languages because there is less direct reliance on features of the underlying platform.

5.7 Unanticipated Reuse

A powerful type of reuse is when functionality is composed from smaller units in ways that are reasonable and valid, but not anticipated by the authors of each subunit. Ousterhout shows how, by using untyped text as its medium and lazy evaluation as its process, the UNIX shell can chain together arbitrary commands with pipes [1]. For example, the following command counts how many lines the word ''dynamic'' occurs in `.c` files:

```
find . -name "*.c" | grep -i dynamic | wc -l
```

The enabling factor in such reuse is the loose contracts placed on input and output data: if the UNIX shell, for example, forced data passed through pipes to be statically typed it is unlikely that such powerful chains of commands could be created as commands would not be as easily reusable.

Dynamically typed languages allow similar reuse to the UNIX shell, but with a subtle twist. While most Unix shell commands demand nothing of input text (which may be empty, all on one line, etc.), and statically typed languages demand the complete typing of all inputs, dynamically typed languages allow shades of gray in-between. Essentially, the idea is that functions should demand (and, possibly, check) the minimum of any inputs to ensure correct functionality, thus allowing functions to operate correctly on a wide range of seemingly unrelated input. This philosophy, while longstanding, has recently acquired the name *duck typing* to reflect the intuitive notion that if an input ''talks like a duck and quacks like a duck, it is a duck''—even if other aspects of the input may not look like a duck [78]. Duck typing can be seen as the run-time, dynamically typed equivalent of structural typing (see Section 2.3.2). A good example of the virtues of duck typing can be found in Python where functions that deal with files often expect only a simple read method in any input objects; this allows programs to make many nonfile objects (e.g., network streams) appear as files, thus reducing the number of cases where specialized functions must be created for different types.

5.8 Interactivity

Virtually, all dynamically typed languages are interactive, in the sense that users can execute commands on a running instance of the system and, if desired, perform further interactive computations on the result. Arguably, the most powerful interactive systems are for Smalltalk, where systems are generally developed within an

interactive GUI system containing both system tools (the compiler, etc.) and the users code [52]. Most languages, however, provide such interactivity via a command-line interface which allows normal expressions to be entered and immediately evaluated. This allows the run-time system presented by the language to be explored and understood. For example, the following session shows how the Python shell can be used to explore the type system and find out help on a method:

```
>>> True.__class__
<type 'bool'>
>>> True.__class__.__class__
<type 'type'>
>>> dir(True.__class__.__class__)
['__base__', '__bases__', '__basicsize__', '__call__',
'__class__', '__cmp__', '__delattr__', '__dict__',
'__dictoffset__', '__doc__', '__flags__',
'__getattribute__', '__hash__', '__init__',
'__itemsize__', '__module__', '__mro__', '__name__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__str__', '__subclasses__',
'__weakrefoffset__', 'mro']
>>> help(True.__class__.__class__.mro)
mro(...)
  mro() -> list
  return a type's method resolution order
>>>
```

By providing an interactive interface, dynamically typed languages encourage exploration of the run-time system, and also allow small examples to be worked on without any ‘‘compile link’’ overhead.

5.9 Compile–Link–Run Cycle

In the majority of programming languages—with the notable exception of Smalltalk and languages directly influenced by it such as Self (see Section 8)—programs are stored in one or more files. To run a program in a statically typed language, one must typically compile each individual file of the program, and link them together to produce a binary, which can then be run. This process is know as the ‘‘compile–link–run’’ cycle. Because statically typed languages are relatively complex to compile and link, this is often a lengthy process—even on modern machines, large applications can

take several hours to compile and link from scratch. This is often a limiting factor in rapid application development [79].

In contrast, most dynamically typed languages conflate the compile–link–run cycle, allowing source files to be directly ''run.'' As compilation of individual modules is often done on an ''as-needs'' basis, and since the compilation and linking of dynamically typed languages is much simpler since no static types need to be checked, this means the user experiences a much shorter compile–link–run cycle.

5.10 Run-Time Updates

With the increasing trend of software providing long-running services (e.g., switches, financial applications), it is necessary to upgrade software without stopping it [37]. This means replacing or augmenting values in the run-time system, typically with data and functionality in the ''old'' system existing side-by-side with the ''new.''

While it is possible to perform limited run-time updates with statically typed languages, the general requirement to retain the type safety of the running system (without which random low-level crashes are likely), and the difficulty of migrating data, makes this extremely challenging in such languages (see Section 3.5). Dynamically typed languages have two significant advantages in such situations. First, reflection allows arbitrary manipulation and emulation of data. Second, there is no absolute requirement to maintain type safety in the updated system as, at worse, any type errors resulting from updating data or functionality will result in a standard run-time type error (in contrast, subverting the type system of a statically typed language is likely to lead to a low-level crash). Erlang makes heavy use of these features to allow extensive run-time updating in a way that allows resultant systems to keep running for very long periods of time [62].

6. Disadvantages of Dynamic Typing

6.1 Performance

Much has been said and written about the relative performance of various programming languages over the years; regrettably, much has been based on superstition, supposition, or unrepresentatively small examples. There is little doubt that, in practice, equivalent programs in dynamically typed languages are slower than in statically typed languages. While on certain macro benchmarks some language implementations (typically Lisp or Smalltalk implementations) can achieve approximate parity with statically typed languages, a general rule of

thumb is that the most finely tuned dynamically typed language implementations are approximately two times slower than the equivalent statically typed implementation[7].

The performance gap between dynamically typed and statically typed languages has lowered over recent years, in large part due to innovations surrounding JIT compilation [5]—the difference in speed between dynamically typed language implementations with and without JIT compilation is typically a factor of 3–5. Currently, the performance between different dynamically typed language implementations varies wildly, with languages such as Ruby an order of magnitude slower than leading Lisp's. As there are few technical reasons for such differences, and given recent trends such as common virtual machines and the awareness of the benefits of JIT compilation, it is likely that the performance gap between implementations will narrow considerably.

Arguably, more important than absolute performance measured in minutes and seconds is the performance relative to requirements: in other words, does the program ''run fast enough?'' Thanks in part to the advancements of commodity computers, for most real-world purposes, this question is often redundant. For certain tasks, particularly very low-level tasks, or those on low-performance computers such as some embedded systems, statically typed languages retain an important advantage. However, it is interesting to note that in certain data-intensive and performance-sensitive domains such as scientific computing dynamically typed languages have proved to be very successful (see, e.g., [76, 80]). There are two explanations for this. First, the high-level nature of dynamically typed languages allows programmers to focus on improving algorithms rather than low-level coding tricks. Second, dynamically typed languages typically come with extensive, highly optimized libraries to which the most performance critical work is often deferred (the so-called ''batteries included'' approach [76]).

6.2 Debugging

A fundamental difference between statically and dynamically typed languages is that the former can detect and prevent certain errors at compile-time (see Section 2.3). Logically, this implies that dynamically typed programs are inherently more error-prone than statically typed languages. This is potentially a real problem,

[7] As shown by ''The Computer Language Benchmarks Game'' http://shootout.alioth.debian.org/ which, despite its stated limitations, is one of the best attempts to compare performance, and is notable for the variety of language implementations it includes.

hence why it is included in the ''disadvantages'' section. However in practice, run-time type errors in deployed programs are exceedingly rare [79].

There are three main reasons why run-time type errors are rarely an issue. First, type errors represent a small, generally immediately obvious, trivially fixed class of errors and are thus typically detected and fixed quickly during development. Second—as shown in Section 3—static types do not capture many of the more important and subtle errors that one might hoped would have been detected; such errors thus occur with equal frequency in statically and dynamically typed programs. Third, automated testing will tend to detect most type errors. This last point is particularly interesting. Unit testing is when a test suite is created that can, without user intervention, be used to check that a system conforms to the tests. Unit tests are often called ''regression suites'' to emphasize that they are intended to prevent errors creeping back into a system. The first unit test suite was for Smalltalk [81], but virtually all languages now have an equivalent library or facility, for example, Java [82]. As this suggests, unit testing allows developers to make guarantees of their programs that are considerably in excess of anything that static typing can provide.

6.3 Code Completion

Many modern developers make use of sophisticated integrated development environments (IDEs) to edit programs. One feature associated with such tools is code completion. In particular when a variable of type `T` is used in a slot lookup, the functions and attributes of the type are automatically displayed. This feature makes use of static types to ensure that (modulo any use of reflection) its answers are fully accurate. A fully equivalent feature is not possible for dynamically typed languages since it is not possible to accurately determine the static type of an arbitrary expression.

6.4 Types as Documentation

Since most statically typed languages force users to explicitly state the types that functions consume and return, statically typed programs have an implicit form of documentation within them, which happens to be machine checkable [23]. There is little doubt that this form of documentation is often useful and that dynamically typed languages do not include it. However, since it is possible to informally notate the expected types of a function in comments, or associated documentation strings processed by external tools, this is not a major disadvantage; furthermore, some dynamically typed languages include optional type systems (see Section 7.2) that allow code to be annotated with type declarations when desired.

7. Variations

In the majority of this chapter, I have described a homogenized picture of dynamically typed languages, emphasizing the culturally common aspects of most languages. Inevitably, this smooths over some important differences and variations between languages; this section details some of these.

7.1 Non-OO and OO Languages

Dynamically typed languages come in both OO (e.g., Converge, Python) and non-OO (e.g., Lisp) flavors. Unsurprisingly, older dynamically typed languages tend to be non-OO, with languages of the past decade or more almost exclusively OO. Interestingly, the transition between these two schools can be seen in languages such as Python (and, to a lesser extent, Lua) which started as non-OO languages but which were subsequently retrofitted with sufficient OO features that their early history is only rarely evident. The general principles are largely the same in both cases, and in most of this chapter I have avoided taking an exclusively OO or non-OO approach.

OO does, however, introduce some new differentiating factors between statically and dynamically typed languages. In particular, static typing allows OO languages to introduce new ways of method dispatch (such as method overloading) due to polymorphism. While metaprogramming allows dynamically typed languages to introduce analogous features, they are not tightly integrated into the language, or frequently used. In part, because of this, it is generally easier to move between non-OO and OO programming styles in dynamically typed languages such as Python than to attempt the same in a statically typed OO language such as Java.

It is notable that dynamically typed languages have played a major part in the continued development of OO. For example, languages such as Self introduced the notable concept of prototyping [64]; Smalltalk has been used as the workbench for innovations such as traits [83] which defines an alternative to inheritance for composing functionality.

7.2 Optional Types

In most of this chapter, dynamic and static typing have been talked about as if they are mutually exclusive—and in most current languages this is true. While not integrated into any mainstream language, there is a long history of work which aims to utilize the benefits of both approaches [2] and blur this distinction. There are three main ways of achieving this. First, one can add a ‘‘dynamic type’’ to a

statically typed language, meaning that most data are statically typed, with some ‘‘dynamically typed’’ (see, e.g., [84, 85]). Second, and of greater interest to this chapter, one can add an *optional type* system to a dynamically typed language.

Intuitively, optional typing is easily defined: static types can be added at selected points in a program, or discovered through type inference, and those types are statically checked by a compiler. Optional typing thus means that portions a program can be guaranteed not to have type errors. Exactly how much of a program needs to be statically typed varies between approaches, for example, some proposal require whole modules to be fully statically typed [86] where others allow a free mixture of dynamic and static typing [87]. Optional types have two further advantages: they offer the possibility that extra optimizations can be used on statically typed portions [19]; they also provide a machine-checkable form of documentation within source code (see Section 6.4).

Optional typing raises two particularly important questions:

1. Are type violations fatal errors (as they are in fully statically typed languages), or merely informative warnings?
2. Should static typing effect the run-time semantics of the system?

There is currently no agreement on either of these points. For example, as described in Section 7.1 static typing in OO languages can affect method dispatch, meaning that OO programs could perform method dispatch differently in statically and dynamically typed portions. Because of this, one possibility is to make optional types truly optional, in that their presence or absence does not affect the run-time semantics of a program [23]. Taking this route also raises the possibility of using different type systems within one program.

For the purposes of this chapter, optional typing is considered to subsume a number of related concepts—including gradual typing, soft typing, and pluggable typing. As this may suggest, optional typing in its various form is still relatively immature and remains an active area of research.

7.3 Analysis

One approach to validating the correctness of a program is analysis. Static analysis involves analyzing the source code of a system for errors, and is capable of finding various classes of errors, not just type errors. Static analysis is a well-established technique in certain limited areas, such as safety critical systems, where developers are prepared to constrain the systems they write in order to be assured of correctness. Such a philosophy is at odds with that of dynamically typed languages, which emphasize flexibility. Furthermore, the inherent flexibility of dynamically typed languages would lead to a huge increase in the search space.

Therefore, static analysis is unlikely to be a practical approach for analyzing dynamically typed programs. Another approach to analysis is to perform it at runtime—dynamic analysis—when virtual machines, libraries, and so on are augmented with extra checks which aim to detect many errors at the earliest possible point, rather than waiting until a program crashes. Although such tools are in their infancy some, such as the Dialyzer system which performs such analysis for Erlang systems [88], are in real-world use.

8. The Future

Definitively predicting the future of dynamically typed languages is impossible since there is no central authority, or single technology, which defines such languages. Nevertheless, certain trends are currently evident. The increasing popularity of dynamically typed languages mean a revived interest in performance issues; while languages such as Self have shown that dynamically typed languages can have efficient implementations, few current languages have adopted such techniques. As dynamically typed languages continue to be used in the real world, increasingly for larger systems, users are likely to demand better performance. Experimentation in optional typing is likely to continue, with optional type systems eventually seeing real use in mainstream languages. The cross-fertilization of ideas between statically and dynamically typed languages will continue, with language features such as compile-time metaprogramming crossing both ways across the divide. It is also likely that we will see an increase in the number of dynamically typed DSLs, since such languages tend by nature to be small and ‘‘lightweight’’ in feel.

9. Conclusions

In this chapter, I detailed the general philosophy, history, and defining features of dynamically typed languages. I showed that, while a broad banner, such languages share much in common. Furthermore, I have highlighted their contribution to the development of programming languages in general and, I hope, a sense of why they are currently enjoying such a resurgence.

Acknowledgment

I am grateful to Éric Tanter who provided insightful comments on a draft of this chapter. All remaining errors and infelicities are my own.

References

[1] J. Ousterhout, Scripting: higher-level programming for the 21st century, Computer 31 (3) (1998) 23–30.

[2] E. Meijer, P. Drayton, Static typing where possible, dynamic typing when needed: the end of the cold war between programming languages, in: Proc. OOPSLA'04 Workshop on Revival of Dynamic Languages, October 2004.

[3] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine (Part I), Commun. ACM 3 (4) (1960) 184–195.

[4] G. Sussman, G. Steele, Jr., Scheme: an interpreter for extended lambda calculus, Technical Report AI Lab Memo AIM-349, MIT AI Lab, December 1975.

[5] J. Aycock, A brief history of Just-In-Time, ACM Comput. Surv. 35 (2) (2003) 97–113.

[6] L.D. Paulson, Developers shift to dynamic programming languages, Computer 40 (2) (2007) 12–15.

[7] R.P. Loui, In praise of scripting: real programming pragmatism, Computer 41 (7) (2008) 22–26.

[8] D. Spinellis, V. Guruprasad, Lightweight languages as software engineering tools, in: USENIX Conference on Domain-Specific Languages, October 1997, USENIX Association, Berkeley, CA, pp. 67–76.

[9] S. Karabuk, F.H. Grant, A common medium for programming operations-research models, IEEE Software 24 (5) (2007) 39–47.

[10] O.L. Madsen, B. Magnusson, B. Mølier-Pedersen, Strong typing of Object orientated languages revisited, in: Proc. OOPSLA, 1990, ACM, Ottawa, Canada, pp. 140–150.

[11] P. Norvig, Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp, Morgan Kaufmann, San Mateo, CA, 1992.

[12] L. Cardelli, Type systems, in: The Computer Science and Engineering Handbook. CRC Press, Boca Raton, FL, 1997, pp. 2208–2236.

[13] B.C. Pierce, Types and Programming Languages, MIT Press, Cambridge, MA, 2002.

[14] W. Cook, W. Hill, P. Canning, Inheritance is not subtyping, in: Proc. 17th Symposium on Principles of Programming Languages, 1990, pp. 125–135.

[15] M. Abadi, L. Cardelli, A Theory of Objects, Springer-Verlag, New York, 1996.

[16] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, second ed., Addison-Wesley, Boston, MA, 2000.

[17] S.P. Jones, Haskell 98 Languages and Libraries: The Revised Report, Cambridge University Press, New York, 2003.

[18] L. Tratt, Converge Reference Manual, July 2007 (http://www.convergepl.org/documentation/, accessed 3 June 2008).

[19] R. Cartwright, M. Fagan, Soft typing, in: Proc. SIGPLAN'91 Conference on Programming Language Design and Implementation, 1991, pp. 278–292.

[20] L. Wall, T. Christiansen, J. Orwant, Programming Perl, O'Reilly, third ed., 2000.

[21] G. Rossum, Python 2.3 Reference Manual, (http://www.python.org/doc/2.3/ref/ref.html, accessed 3 June 2008).

[22] J. Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley, Reading, MA, 1994.

[23] G. Bracha, Pluggable type systems, in: OOPSLA'04 Workshop on Revival of Dynamic Languages, October 2004.

[24] D. Ancona, E. Zucca, S. Drossopoulou, Overloading and inheritance, in: Workshop on Foundations of Object Orientated Languages (FOOL8), 2001.

[25] M. Shields, T. Sheard, S.P. Jones, Dynamic typing as staged type inference, in: Proc. Symposium on Principles of Programming Languages, January 1998, pp. 289–302.

[26] H. Xi, F. Pfenning, Eliminating array bound checking through dependent types, in: Proc. Conference on Programming Language Design and Implementation, 1998, pp. 249–257.
[27] N. Mitchell, C. Runciman, Unfailing Haskell: a static checker for pattern matching, in: Proc. Symposium on Trends in Functional Programming, 2005, pp. 313–328.
[28] A. Aiken, E.L. Wimmers, T.K. Lakshman, Soft typing with conditional types, in: Proc. Symposium on Principles of Programming Languages, 1994, ACM, New York, pp. 163–173.
[29] D.C.J. Matthews, Static and dynamic type checking, in: Advances in Database Programming Languages, 1990, pp. 67–73.
[30] D.B. MacQueen, Reflections on standard ML, in: Functional Programming, Concurrency, Simulation and Automated Reasoning, vol. 693 of LNCS, Springer-Verlag, New York, 1993, pp. 32–46.
[31] E. Meijer, Confessions of a used programming language salesman, SIGPLAN Notices 42 (10) (2007) 677–694.
[32] B. Meyer, Eiffel: The Language, Prentice-Hall International, London, 1992.
[33] W.R. Cook, A proposal for making Eiffel type-safe, Comput. J. 32 (4) (1989) 305–311.
[34] G. Castagna, Covariance versus contravariance: conflict without a cause, ACM Transactions on Programming Languages and Systems, May 1995, pp. 431–447.
[35] M.M. Lehman, L.A. Belady, Program Evolution: Processes of Software Change, Academic Press, London, 1985.
[36] O. Nierstrasz, A. Bergel, M. Denker, S. Ducasse, M. Gälli, R. Wuyts, On the revival of dynamic languages, in: Proc. Software Composition 2005, vol. 3628 of LNCS, 2005, pp. 1–13.
[37] M. Hicks, S.M. Nettles, Dynamic software updating, ACM Trans. Program. Lang. Syst. 27 (6) (2005) 1049–1096.
[38] F.N. Demers, J. Malenfant, Reflection in logic, functional and Object orientated programming: a short comparative study, in: Proc. IJCAI'95 Workshop on Reflection and Metalevel Architectures and Their Applications in AI, August 1995, pp. 29–38.
[39] G.L. Steele, R.P. Gabriel, The Evolution of Lisp, 1996, pp. 233–330.
[40] A.C. Kay, The Early History of Smalltalk, 1996, pp. 511–598.
[41] R.E. Griswold, M.T. Griswold, History of the Icon Programming Language, Addison-Wesley, 1996, Reading, MA, 1996, pp. 599–624.
[42] J. McCarthy, History of LISP, in: Proc. History of Programming Languages, 1978, ACM, New York, pp. 173–185.
[43] C. Brabrand, M. Schwartzbach, Growing languages with metamorphic syntax macros, in: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, 2000, ACM, San Diego, CA.
[44] T. Sheard, Using MetaML: a staged programming language, in: Advanced Functional Programming, September 1998, pp. 207–239.
[45] T. Sheard, S.P. Jones, Template meta-programming for Haskell, in: Proc. Haskell Workshop 2002, ACM, New York.
[46] R. Jones, R. Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, New York, 1999.
[47] R.P. Gabriel, Performance and Evaluation of LISP Systems, MIT Press, Cambridge, MA, 1986.
[48] C.T. Haynes, D.P. Friedman, M. Wand, Continuations and coroutines, in: Proc. Symposium on LISP and Functional Programming, 1984, ACM, Austin, TX, pp. 293–298.
[49] E. Kohlbecker, D.P. Friedman, M. Felleisen, B. Duba, Hygienic macro expansion, in: Proc. Symposium on Lisp and Functional Programming, 1986, ACM, Austin, TX, pp. 151–161.
[50] W. Clinger, J. Rees, Macros that work, in: Proc. 19th ACM Symposium on Principles of Programming Languages, January 1991, ACM, New York, pp. 155–162.

[51] O.J. Dahl, K. Nygaard, An Algol-based simulation language, Commun. ACM 9 (9) (1966) 671–678.
[52] A. Goldberg, D. Robson, Smalltalk-80: The Language, Addison-Wesley, Reading, MA, 1989.
[53] P. Maes, Concepts and experiments in computational reflection, in: Proc. OOPSLA, 1987, ACM, New York, pp. 147–155.
[54] G. Kiczales, J. Rivieres, D.G. Bobrow, The Art of the Metaobject Protocol, MIT Press, Cambridge, MA, 1991.
[55] I.R. Forman, S.H. Danforth, Putting Metaclasses to Work: A New Dimension in Object Orientated Programming, Addison-Wesley, Reading, MA, 1998.
[56] P. Cointe, Metaclasses are first class: the ObjVLisp model, in: Proc. Object Orientated Programming Systems, Languages, and Applications, October 1987, pp. 156–162.
[57] R.E. Griswold, J.F. Poage, I.P. Polonsky, The SNOBOL4 Programming Language, second ed., Prentice-Hall, Englewood Cliffs, NJ, 1971.
[58] R.E. Griswold, A history of the SNOBOL programming languages, SIGPLAN Notices 13 (8) (1978) 275–308.
[59] R.E. Griswold, M.T. Griswold, The Icon Programming Language, Peer-to-Peer Communications, third ed. 1996.
[60] A.V. Aho, B.W. Kernighan, P.J. Weinberger, The AWK Programming Language, Addison-Wesley, Reading, MA, 1998.
[61] L. Sterling, E. Shapiro, The Art of Prolog, second ed., MIT Press, Cambridge, MA, 1994.
[62] R. Virding, C. Wikstrom, M. Williams, J. Armstrong, Concurrent Programming in Erlang, Prentice Hall, New York, 1996.
[63] J. Armstrong, A history of Erlang, in: Proc. History of Programming Languages, 2007, ACM, New York, .
[64] D. Ungar, R.B. Smith, Self: the power of simplicity, in: Proc. OOPSLA, October 1987, pp. 227–241.
[65] C. Chambers, D. Ungar, Customization: optimizing compiler technology for SELF, a dynamically-typed Object orientated programming language, SIGPLAN Notices 24 (7) (1989) 146–160.
[66] D. Thomas, A. Hunt, Programming Ruby: A Pragmatic Programmer's Guide, Addison-Wesley, Reading, MA, 2000.
[67] R. Ierusalimschy, Programming in Lua, second ed., Lua.org, 2006.
[68] R. Ierusalimschy, L.H. de Figueiredo, W. Celes, The evolution of Lua, in: Proc. History of Programming Languages, 2007, ACM, New York.
[69] P. Rovner, On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language, Technical Report CSL-84-7, Xerox Parc, 1985
[70] G. Bracha, D. Ungar, Mirrors: design principles for meta-level facilities of Object orientated programming languages, in: Proc. OOPSLA, 2004, ACM, New York, pp. 331–344.
[71] S. Mostinckx, T. Cutsem, S. Timbermont, E.G. Boix, É. Tanter, W. Meuter, Mirror-based reflection in AmbientTalk, Software Pract. Exper. 2009.
[72] F. Ortin, J.M. Cueva, Dynamic adaptation of application aspects, J. Syst. Software 71 (2004) 229–243.
[73] G.L. Steele Jr., Common Lisp the Language, second ed., Digital Press, Newton, MA, 1990.
[74] S. Ducasse, A. Lienhard, L. Renggli, Seaside: a flexible environment for building dynamic Web applications, IEEE Software 24 (5) (2007) 56–63.
[75] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Reading, MA, 1999.
[76] T.E. Oliphant, Python for scientific computing, Comput. Sci. Eng. 9 (3) (2007) 10–20.
[77] H. Spencer, G. Collyer, #ifdef considered harmful, or portability experience with C News, in: Proc. Summer 1992 USENIX Conference, 1992, San Antonio, TX, pp. 185–198.

[78] A. Koenig, B.E. Moo, Templates and duck typing, Dr. Dobb's, 2005.
[79] L. Tratt, R. Wuyts, Dynamically typed languages, *IEEE Software* 24 (5) (2007) 28–30.
[80] X. Cai, H.P. Langtangen, H. Moe, On the performance of the Python programming language for serial and parallel scientific computations, Sci. Program. 13 (1) (2005) 31–56.
[81] K. Beck, Simple Smalltalk testing: with patterns, 1994 (http://www.xprogramming.com/testfram.htm, accessed 14 July 2008).
[82] J. Link, P. Fröhlich, Unit Testing in Java: How Tests Drive the Code, Morgan Kaufmann, San Mateo, CA, 2003.
[83] N. Schärli, S. Ducasse, O. Nierstrasz, A.P. Black, Traits: composable units of behaviour? in: Proc. ECOOP, vol. 2743 of LNCS, July 2003, pp. 248–274.
[84] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin, Dynamic typing in a statically typed language, ACM Trans. Program. Lang. Syst. 13 (2) (1991) 237–268.
[85] F. Henglein, Dynamic typing: syntax and proof theory, Sci. Comput. Program. 22 (3) (1994) 197–230.
[86] S. Tobin-Hochstadt, M. Felleisen, The design and implementation of typed Scheme, SIGPLAN Notices 43 (1) (2008) 395–406.
[87] J.G. Siek, M. Vacharajani, Gradual typing with unification-based inference, Dynamic Languages Symposium, 2008.
[88] T. Lindahl, K. Sagonas, Detecting software defects in telecom applications through lightweight static analysis: a war story, in: C. Wei-Ngan (Ed.), Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04), vol. 3302, of LNCS, November 2004, Springer-Verlag, New York, pp. 91–106.

Factors Influencing Software Development Productivity—State-of-the-Art and Industrial Experiences

ADAM TRENDOWICZ

Fraunhofer Institute for Experimental Software Engineering, Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany

JÜRGEN MÜNCH

Fraunhofer Institute for Experimental Software Engineering, Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany

Abstract

Managing software development productivity is a key issue in software organizations. Business demands for shorter time-to-market while maintaining high product quality force software organizations to look for new strategies to increase development productivity.

Traditional, simple delivery rates employed to control hardware production processes have turned out not to work when simply transferred to the software domain. The productivity of software production processes may vary across development contexts dependent on numerous influencing factors. Effective productivity management requires considering these factors. Yet, there are thousands of possible factors and considering all of them would make no sense from the economical point of view. Therefore, productivity modeling should focus on a limited number of factors with the most significant impact on productivity.

In this chapter, we present a comprehensive overview of productivity factors recently considered by software practitioners. The study results are based on the review of 126 publications as well as international experiences of the

ADVANCES IN COMPUTERS, VOL. 77
ISSN: 0065-2458/DOI: 10.1016/S0065-2458(09)01206-6

Fraunhofer Institute, including the most recent 13 industrial projects, four workshops, and eight surveys on software productivity. The aggregated results show that the productivity of software development processes still depends significantly on the capabilities of developers as well as on the tools and methods they use.

1. Introduction

Rapid growth in the demand for high-quality software and increased investment in software projects show that software development is one of the key markets worldwide [2, 122]. Together with the increased distribution of software, its variety and complexity are growing constantly. A fast changing market demands software products with ever more functionality, higher reliability, and higher performance. Software project teams must strive to achieve these objectives by exploiting the impressive advances in processes, development methods, and tools. Moreover, to stay competitive and gain customer satisfaction, software providers must ensure that software products with a certain functionality are delivered on time, within budget, and to an agreed level of quality, or even with reduced development costs and time. This illustrates the necessity for reliable methods to manage software development productivity, which has traditionally been the basis of successful software management. Numerous companies have already measured software productivity [3] or planned to measure it for the purpose of improving their process efficiency, reducing costs, improving estimation accuracy, or making decisions about outsourcing their development.

Traditionally, the productivity of industrial production processes has been measured as the ratio of units of output divided by units of input [4]. This perspective was transferred into the software development context and is usually defined as *productivity* [5] or *efficiency* [6]. As observed during an international survey performed in 2006 by the Fraunhofer Institute for Experimental Software Engineering, 80% of software organizations adapt this industrial perspective on productivity to the context of software development, where inputs consist of the effort expended to produce software deliverables (outputs). The assumption those organizations make is that measuring software productivity is similar to measuring any other forms of productivity. Yet, software production processes seem to be significantly more difficult than production processes in other industries [7, 8, 120]. This is mainly because software organizations typically develop new products as opposed to fabricating the same product over and over again. Moreover, software development is a human-based ("soft") activity with extreme uncertainties from the outset. This leads to many difficulties in the reliable definition of software productivity. Some of the most critical practical consequences are that software development productivity measures based simply on size and effort are hardly comparable [7, 125], and that size-based software estimates are not adequate [9].

These difficulties are related to a variety of practical issues. One of these is software sizing.[1] Large numbers of associated, mutually interacting, and usually unknown factors influencing software productivity (besides size) are another critical issue. Even if reliable and consistent size measurement is in place, the key question of software productivity measurement usually remains and may be formulated as follows: "What makes software development productivity vary across different contexts?" In other words, which characteristics of the project environment capture the reasons why projects consumed different amounts of resources when normalized for size.

To answer this question, characteristics that actually differentiate software projects and their impact on development productivity have to be considered. Those characteristics include personnel involved in the project, products developed, as well as processes, technologies and methods applied.

Yet, there are several pragmatic problems to be considered when analyzing software project characteristics and their impact on productivity. One is, as already mentioned, the practically infinite quantity of potential factors influencing software productivity [11]. Even though it is possible to select a limited subset of the most significant factors [12, 13], both factors and their impact on productivity may differ depending on productivity measurement level and context [7]. This also means that different project characteristics may have different impacts on the levels of single developer, team, project, and whole organization. Moreover, even if considered on the same level, various productivity factors may play different roles in the embedded software, in Web applications, and in waterfall or incremental development. This implies, for example, that factors and their ratings covered by the COCOMO model [14] would require reappraisal when applied in a context other than that in which the model was built [15].

To address this issue, considerable research has been directed at identifying factors that have a significant impact on software development productivity. This includes (1) studies dealing directly with identifying productivity factors [16]; (2) studies aiming at building cost and productivity modeling and measurement techniques, methods, and tools [8]; (3) data collection projects intending to build software project data repositories useful for benchmarking productivity and predicting cost [17–20]; and (4) studies providing influence factors hidden in best practices to reduce development cost (increase productivity) [13, 123].

[1] Software sizing (size measurement) as a separate large topic is beyond the scope of this chapter. For more details on the issue of size measurement in the context of productivity measurement, see, for example, [10].

Although a large amount of work has been directed recently at identifying significant software productivity factors, many software organizations actually still use the simplified definition of productivity provided in [12]. In the context of high-maturity organizations [3], where production processes are stable and projects largely homogeneous, a simple productivity measure may work, provided that it is used to compare similar projects. Yet, in the case of crosscontext measurement of heterogeneous projects with unstable processes, using simple productivity may lead to serious problems. In consequence, organizations that failed to measure simplified productivity find it either difficult to interpret and/or entailing little benefit [7].

Software organizations do not consider productivity factors because they often do not know which ones they should start with—which are the most significant ones. Existing publications provide hardly any support in this matter. Published results are distributed over hundreds of publications, which in many cases lack a systematic approach to presenting influence factors (e.g., context information, relative impact on productivity) and/or present significant factors implicitly as cost drivers in a cost model, project attributes in a data repository, or best practices for improving development processes.

In this study, we try to provide a systematic overview of those productivity factors that have the most significant impact on software productivity. In doing so, we assume that software cost is a derivative of software productivity and, therefore, we do not treat factors influencing cost (also called cost drivers) separately. Besides commonly available published studies investigating productivity factors, the overview also includes internal experiences gained in recent years at the Fraunhofer Institute for Experimental Software Engineering (IESE) in several industrial projects, workshops, and surveys performed in the area of software productivity measurement and cost modeling. Yet, due to confidentiality reasons, we do not disclose some sensitive data such as company name or titles of respective internal reports that may indicate the names of the involved industry partners. Instead, we refer to certain characteristics of the companies. Such context information might be useful when selecting relevant factors for similar contexts, based on the overview presented in this chapter.

The remainder of the chapter is organized as follows. Section 2 presents the design of the study. Section 3 provides brief definitions of the terminology used. Sections 4 and 5 provide a summary of productivity factors based on the literature review and the authors' industrial experiences gained at Fraunhofer IESE, respectively. An in-depth discussion on selected productivity factors presented in the literature is given in Section 6. Section 7 provides an overview of several practical issues to be considered when identifying and productivity factors and using them to model software development productivity and cost. Finally, Section 8 summarizes the presented results and discusses open issues.

2. Design of the Study

The review of productivity factors presented in this chapter consists of two parts. First, we present a review of the authors' individual experiences gained during a number of industrial initiatives. Here, we would include factors identified for the purpose of cost and productivity modeling as well as factors acquired during surveys and workshops performed by Fraunhofer IESE in the years 1998–2006 (also referred to as *IESE studies*). The second part presents an overview of publications regarding software project characteristics that have an influence on the cost and productivity of software development.

2.1 Review of Industrial Experiences

The review includes the following industrial initiatives:

- International commercial projects regarding cost and/or productivity modeling performed in the years 1998–2006 at medium and large software organizations in Europe (mainly Germany, e.g., [21]), Japan, India, and Australia (e.g., [22]).
- International workshops on cost and/or productivity modeling performed in the years 2005–2006 in Germany and Japan (e.g., [7, 24, 124]). The workshop results include factors explicitly encountered by representative of various, international software organizations. The considered factors consist of both factors that were considered by experts as having a significant impact on productivity but not already measured, and factors already included in the organizational measurement system for managing development cost and productivity.
- Surveys on productivity measurement practices performed in various international companies in the years 2004–2006. These surveys concerned productivity/cost modeling and included questions about the most significant factors influencing software productivity. The 2004 study was a state-of-the-practice survey performed across 12 business units of a global software organization dealing mainly with embedded software. The four 2005 studies were performed across Japanese units of internationally operating software organizations dealing both with embedded and business applications. Finally, the 2006 survey was performed across 25 software organizations all over the world and covered various software domains.

2.2 Review of Related Literature

2.2.1 Review Scope and Criteria

The design of the review is based on the guidelines for structural reviews in software engineering proposed by Kitchenham [25]. Yet, based on the conclusions of Jørgensen and Shepperd [10], we relied on a manual rather than an automatic search of relevant references. The automatic search through the INSPEC repository was complemented by a manual search through references found in reviewed papers and using a generic Web search machine (http://www.google.com). The automatic search criteria were specified as follows:

INSPEC: (SOFTWARE AND ((COST OR EFFORT OR PRODUCTIVITY) WITH (FACTORS OR INDICATORS OR DRIVERS OR MEASURE))). TX. = **417 documents**

The available papers[2] were selected for inclusion (full review) based on the title and abstract. The review was done by one researcher. The criteria used to decide whether to include or exclude papers were as follows:

- *Availability and age*. The papers had to be published no earlier than 1995. Since the type and impact of influence factors may change over the time [26], we limit our review to the past decade.

We made an *exception* for papers presenting software project data sets. Although indirectly, we did include those that were collected and first published before 1995 but that were applied to validate cost/productivity models published after 1995.

- *Relevancy*. The papers had to report factors with a potential impact on software development cost and/or productivity. Implicit and explicit factors were considered. *Explicit* factors are those considered in the context of productivity modeling/measurement. *Implicit* factors include those already included in public-domain cost models and software project data repositories. *Algorithmic cost estimation models*, for example, include so-called cost drivers to adjust the gross effort estimated based only of software size (e.g., [121]). *Analogy-based*

[2] Unfortunately, some publications could not be obtained though the library service of Fraunhofer IESE.

methods, on the other hand, use various project characteristics found in the distance measure, which is used to find the best analogues to base the final estimate on. Common software project data repositories indirectly suggest a certain set of attributes that should be measured to assure quality estimates.

- *Novelty*. We did not consider studies that adopt a complete set of factors from other reference and do not provide any novel findings (e.g., transparent model) on a factor's impact on productivity. We do not, for instance, consider models (e.g., those based on neural networks) that use as their input the whole set of COCOMO I factors and do not provide any insight into the relative impact of each factor on software cost.
- *Redundancy*. We did not consider publications presenting the results of the same study (usually presented by the same authors). In such cases, only one publication was included in the review.
- *Perspective*. As already mentioned in the introduction, there are several possible perspectives on productivity in the context of software development. In this review, we will focus on project productivity, that is, factors that make development productivity differ across various projects. Therefore, we will, for example, not consider the productivity of individual development processes such as inspections, coding, or testing. This perspective is the one most commonly considered in the reviewed literature (although not stated directly) and is the usual perspective used by software practitioners [7].

Based on the aforementioned criteria, 136 references were included in the full review. After a full review, 122 references in total were considered in the results presented in this chapter. Several references (mostly books) contained well-separated sections where productivity factors were considered in separate contexts. In such cases, we considered these parts as separate references. This resulted in the final count of 142 references. For each included reference, the following information was extracted:

- *Bibliographic information*: title, authors, source and year of publication, etc.;
- *Factor selection context*: implicit, explicit, cost model, productivity measure, project data repository;
- *Factor selection method*: expert assessment, literature survey, data analysis;
- *Domain context*: embedded software (Emb), management information systems (MIS), and Web applications (Web);
- *Size of factor set*: initial, finally selected;
- *Factor characteristics*: name, definition, weighting (if provided).

2.2.2 Study Limitations

During the literature review, we faced several problems that may limit the validity of this study and its conclusions. First, the description of factors given in the literature is often incomplete and limited to a factor's name only. Even if both name and definition are provided, factors often differ with respect to their name; although according to the attached definition, they refer to the same phenomenon (factor).

Second, published studies often define factors representing a multidimensional phenomenon, which in other studies is actually decomposed into several separate factors. For instance, the *software performance constraints* factor is sometimes defined as time constraints, sometimes as storage constraints, and sometimes as both time and storage constraints.

Moreover, instead of factors influencing development productivity, some studies considered a specific factor's value and its specific impact on productivity. For instance, some studies identified the life cycle model applied to developing software as having an impact of productivity, while others directly pointed out incremental development as increasing overall development productivity.

Finally, factors identified implicitly within cost models (especially data-driven models) are burdened with a certain bias, since in most cases, data-driven models are built using the same commonly available (public) data repositories (e.g., [1, 14, 19, 27, 28]). In consequence, factors identified by such cost models are limited a priori to specific set of factors covered by the data repository used.

To moderate the impact of those problems on the study results, we undertook the following steps:

- We excluded from the analysis those models that were built on one of the publicly available project repositories and simply used all available factors. We focused on models that selected a specific subset of factors or were built on their "own," study-specific data.
- We compared factors with respect to their definition. If this was not available, we used a factor's name as its definition.
- In our study, we decomposed a complex factor into several base factors according to the factor's definition.
- We generalized factors that actually referred to a specific factor's value to a more abstract level. For instance, the "iterative development" factor actually refers to a specific value *development life cycle model* factor—we classified this factor as "life cycle model."
- Finally, we excluded (skipped) factors whose meaning (based on the name and/or definition provided) was not clear.

2.2.3 Demographical Information

Among the reviewed references, 33 studies identified factors directly in the context of development productivity modeling, 82 indirectly in the context of cost modeling (e.g., estimation, factor selection), 14 in the context of a project data repository (for the purpose of cost/productivity estimation/benchmarking), nine in the context of software process improvement, two in the context of schedule modeling, and, finally, two in the context of schedule estimation (see Fig. 1).

Regarding the domain, 27 references considered productivity factors in the context of embedded software and software systems, 50 in the context of management information systems, and four in the context of Web applications (see Fig. 2). The remaining references either considered multiple domains (18) or did not explicitly specify it (43).

With respect to development type, 19 references analyzed productivity factors in the context of new development, 44 in the context of enhancement/maintenance, and 15 in the context of software reuse and COTS development (see Fig. 3).

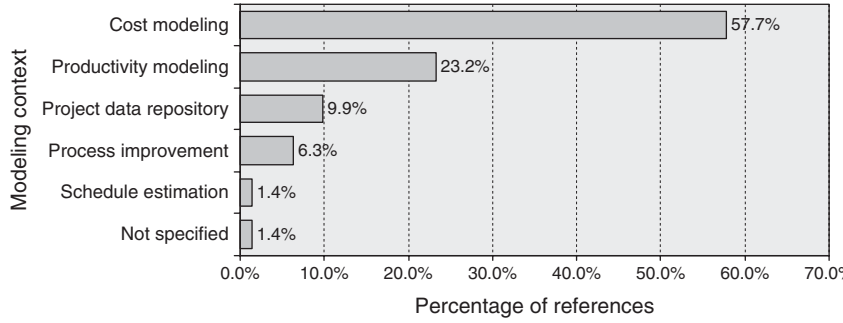

FIG. 1. Literature distribution regarding modeling context.

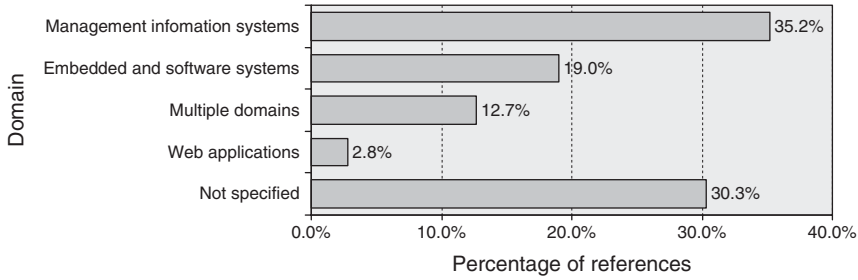

FIG. 2. Literature distribution regarding domain.

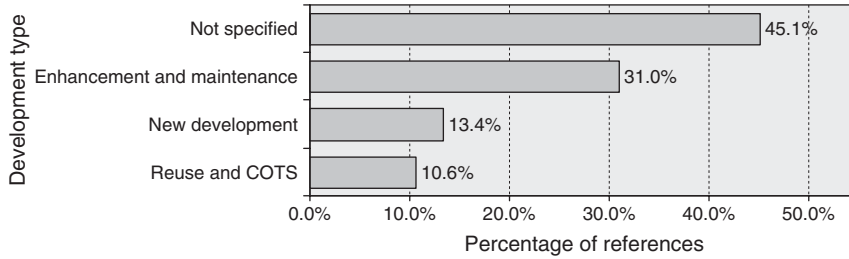

FIG. 3. Literature distribution regarding development type.

Most of the analyzed studies started with some initial set of potential productivity factors and applied various selection methods to identify only the most significant ones. The analyzed references ranged from 1 to 200 initially identified factors that were later reduced down to the 1–31 most significant factors.

In total, the reviewed references mentioned 246 different factors, with 64 representing abstract phenomena (e.g., *team experience and skills*) and the remaining 178 concrete characteristics (e.g., *analyst capability* or *domain experience*).

2.3 Aggregation of the Review Results

Due to space limitations, we do not provide the complete lists of factors we identified in this study. Instead, we report those factors that are most commonly considered as having the greatest impact on software development productivity.

Both in the literature and in the IESE studies, various methods were used to determine the significance of the considered factors. Several authors present a simple ranking of factors with respect to their significance [21]; others used various weighting schemas to present the relative impact of each considered factor on productivity or cost [16].

We aggregated the results presented in various studies in the following procedure:

- We ranked factors with respect to the weightings they were given in the source study. The factor having the greatest weight (impact on productivity) was ranked with "1," the next one as "2," and so on. If the source study reported factors without distinguishing their impact/importance, we considered all factors as having rank "1." The disadvantage of the applied aggregation procedure is that the information regarding the "distance" between factors with respect to their weight was lost.

- We aggregated the results for each factor using two measures (1) number of studies a given factor was considered in (*Frequency*) and (2) median factor's rank[3] over all the studies it was considered in (*Median rank*).

3. Related Terminology

3.1 Context Versus Influence Factors

In practice, it is difficult to build a reliable productivity model that would be applicable across a variety of environments. Therefore, usually only a limited number of factors influencing productivity are considered within a model; the rest is kept constant and described as the so-called *context* for which the model is built and in which it is applicable. Building, for instance, a model for business application and embedded real-time software would require covering a large variety of factors that play a significant role in both domains. Alternatively, one may build simpler models for each domain separately. In that case, the factor "application domain" would be constant for each model. We would refer to factors that describe a modeling context as *context factors*. On the other hand, factors that are included in the model to explain productivity variance within a certain context will be called *influence factors*. Moreover, we would say that context factors determine influence factors, that is, dependent on the project context, different factors may have a different impact on productivity and may interact with each other differently.

3.2 Classification of Influence Factors

To gain better visibility, we further categorized (after several other authors [22, 30, 31]) the identified influence factors into four groups: product, personnel, project, and process factors.

Product factors cover the characteristics of software products being developed throughout all development phases. These factors refer to such products as software code, requirements, documentation, etc. and their characteristics, such as complexity, size, volatility, etc.

Personnel factors reflect the characteristics of personnel involved in the software development project. These factors usually consider the experience and capabilities of such project stakeholders as development team members (e.g., analysts, designers, programmers, project managers) as well as software users, customers, maintainers, subcontractors, etc.

[3] We assume ranks to be on the equidistant ordinal scale, which allows applying median operation [29].

Project factors regard various qualities of project management and organization, development constraints, working conditions, or staff turnover.

Process factors are concerned with the characteristics of software processes as well as methods, tools, and technologies applied during a software development project. They include, for instance, the effectiveness of quality assurance, testing quality, quality of analysis and documentation methods, tool quality and usage, quality of process management, or the extent of customer participation.

4. Overview of Factors Presented in Literature

In total, 246 different factors were identified in the analyzed literature. This section presents the most commonly used productivity factors we found in the reviewed literature. First, we present factors that are commonly selected across all development contexts. Then, we present the most popular factors selected in the context of a specific model, development type, and domain. Additionally, factors specific for software reuse are analyzed.

Regarding the studies where project data repositories were presented, we analyzed publicly available data sets that contain more than only size and effort data (see, e.g., [32] for an overview).

4.1 Crosscontext Factors

Table I presents top four productivity factors and three context factors that were found to be the most significant ones according to all studies reviewed.

The most commonly identified factors represent complex phenomena, which might be decomposed to basic subfactors. For each complex factor, we selected (at most) three most commonly identified subfactors. *Team capability and experience*, for instance, is considered to be the most influential productivity factor. Specifically, *programming language experience*, *application experience and familiarity*, and *project manager experience and skills* are those team capabilities that were most often selected in the reviewed studies.

4.2 Context-Specific Factors

In Tables II–IV, each cell contains information in the form *X*/*Y*, where *X* is the number of studies where a certain factor was selected (*Frequency*) and *Y* means the median rank given to the factor over those studies (*Median rank*). Moreover, the most significant factors for a certain context are marked with bold font and cells

TABLE I
TOP CROSSCONTEXT PRODUCTIVITY FACTORS

Influence factors	Frequency (no. of references)	Median rank
Team capabilities and experience	64	1
Programming language experience	*16*	*1*
Application experience and familiarity	*16*	*1*
Project manager experience and skills	*15*	*1*
Software complexity	42	1
Database size and complexity	*9*	*1*
Architecture complexity	*9*	*1*
Complexity of interface to other systems	*8*	*1.5*
Project constraints	41	1
Schedule pressure	*43*	*1*
Decentralized/multisite development	*9*	*1*
Tool usage and quality/effectiveness	41	1
CASE tools	*12*	*1*
Testing tools	*5*	*1*
Context factors		
Programming language	29	1
Domain	14	1
Development type	11	1

containing factors that were classified as the most significant ones in two or more contexts are gray-filled. An empty cell means that a factor did not appear in a certain context at all.

For each considered context number of relevant references is given in the table header (in brackets).

4.2.1 Model-Specific Factors

Table II presents the most common factors selected in the context of cost modeling (CM), productivity measurement (PM), project data repositories (DB), and studies on software process improvement (SPI).

4.2.2 Development-Type-Specific Factors

Table III presents the most common productivity factors selected in the context of *new development* (Nd) and *maintenance/enhancement* projects (Enh/Mtc). We considered maintenance and enhancement together because the considered references did not make a clear difference between those two types of development.

TABLE II
TOP MODEL-SPECIFIC PRODUCTIVITY FACTORS

Influence factors	PM (33)	CM (82)	DB (14)	SPI (9)
Team capabilities and experience	***14/1***	***39/1***	***6/1***	***3/1***
Overall personnel experience	***5/1***	*4/1*	–	*1/1*
Project manager experience and skills	***3/2***	*6/7*	***3/1***	***3/1***
Task-specific expertise	***3/6***	*2/9.5*	–	–
Application experience and familiarity	*2/1*	***12/1***	*1/1*	–
Programming language experience	*3/9*	***11/1***	***2/1***	–
Tool experience	*1/1*	*4/1*	***2/1***	–
Teamwork capabilities	*1/1*	*2/1*	***2/1***	–
Training level	–	*4/1*	*1/1*	***2/1***
Tool usage and quality/effectiveness	***12/2***	***22/1***	*5/1*	*2/1*
CASE tools	***4/5.5***	***3/1***	*4/1*	*1/1*
Testing tools	*1/1*	***4/3.5***	–	–
Team size	***8/1***	*14/1*	*5/1*	*1/1*
Reuse	***8/1***	*9/1*	*5/1*	*2/2.5*
Reuse level	***5/1***	*7/1*	*3/1*	*2/2.5*
Quality of reused assets	***2/1***	–	–	–
Software complexity	*7/2.3*	***25/1***	***8/1***	*1/1*
Architecture complexity	*2/6.5*	***6/1***	*1/1*	–
Complexity of interface to other systems	*1/1*	***6/2***	*1/1*	–
Database size and complexity	*1/1*	***4/10***	***4/1***	–
Code complexity	*1/1*	*2/1*	***4/1***	–
Team organization	*6/2*	***21/1***	*3/1*	***3/1***
Team cohesion/communication	*3/3*	***14/1***	–	*1/3*
Staff turnover	–	***11/1***	*2/1*	***2/1.5***
Team structure	*4/1*	***4/3.5***	*1/1*	***2/1***
Project constraints	*9/3*	***21/3***	***6/1***	***3/1***
Schedule pressure	*9/1*	***16/5***	***5/1***	***2/1***
Decentralized development	*1/16*	***5/7***	***2/1***	*1/1*
Process maturity and stability	*2/1*	*7/1*	–	**4/1**
Methods usage	7/6	16/1	5/1	**3/1**
Reviews and inspections	*2/11*	*4/1*	*2/1*	***2/1***
Testing	*1/5*	*2/1*	*1/1*	***2/1***
Requirements management	–	*2/1*	*1/1*	***2/1***
Context factors				
Programming language	**8/1**	**12/1**	**8/1**	1/1
Life cycle model	**2/1**	3/1	**3/1**	1/1
Domain	**2/4.5**	7/**1**	**4/1**	1/1
Development type	–	**7/1**	**3/1**	1/1
Target platform	1/1	3/1	**3/1**	–

TABLE III
TOP DEVELOPMENT-TYPE-SPECIFIC PRODUCTIVITY FACTORS

Influence factors	Nd (19)	Enh/Mtc (43)
Team capabilities and experience	**8/1**	**26/1**
Task-specific experience	***3/11***	*1/5*
Application experience and familiarity	***1/1***	***11/1***
Programmer capability	***1/1***	*5/3*
Programming language experience	–	***10/1.5***
Analyst capabilities	–	***6/3***
Project constraints	**7/6**	**17/1**
Schedule pressure	***7/5***	***13/2***
Distributed/multisite development	***3/8***	***4/1***
Reuse	**5/1**	11/4
Reuse level	***4/1***	*7/4*
Quality of reusable assets	***1/1***	*1/1*
Management quality and style	**5/1**	5/8
Team motivation and commitment	**5/1**	4/4.5
Product complexity	5/7	**17/2**
Interface complexity to hardware and software	–	***6/2***
Architecture complexity	*1/1*	***6/2***
Database size and complexity	–	***4/10***
Logical problem complexity	–	***3/1***
Required software quality	4/2	**15/4**
Required reliability	*4/2*	***13/4***
Required maintainability	–	***3/4***
Tool usage	3/1	**17/2**
Testing tools	–	***3/5***
Tool quality and effectiveness	*1/6*	***2/1.5***
Method usage	2/4.3	**13/4**
Review and inspections	*1/8*	***4/1***
Testing	–	***3/1***
Context factors		
Programming language	**4/1**	**10/1**
Target platform	**1/1**	3/1
Domain	–	**8/3**
Development type	–	**6/1**

4.2.3 Domain-Specific Factors

Table IV presents the most common productivity factors selected in the context of specific software domains: embedded systems (Emb), management and information systems (MIS), and Web applications (Web).

TABLE IV
TOP PRODUCTIVITY FACTORS USED TO MODEL SOFTWARE COST

Influence factors	Emb (27)	MIS (50)	Web (4)
Team capabilities and experience	**14/1**	**20/1**	**3/3**
Programming language experience	***5/1***	*4/3*	*1/3*
Training level	***3/1***	*1/1*	–
Application experience and familiarity	*1/1*	***9/1***	–
Programmer capability	*1/2*	***6/3.5***	–
Design experience	–	*1/3*	***1/1***
IT technology experience	*1/1*	*1/1*	***1/1***
Team size	**8/1**	**9/1**	1/3
Reuse	3/1	**9/1**	1/5
Reuse level	*2/3.5*	***6/1***	*1/5*
Tools usage and quality	**12/1**	**9/1**	**3/2**
CASE tools	***3/1***	***4/1***	–
Methods usage	**12/1**	8/1.5	–
Reviews and inspections	***4/1***	*1/1*	–
Testing methods	***2/3***	*1/1*	***1/2***
Modern programming practices	***2/6***	*2/5*	–
Project constraints	**9/1**	**16/2.2**	–
Schedule pressure	***7/1***	***12/3***	–
Distributed/multisite development	*1/1*	***5/7***	–
Requirements quality	**8/1**	9/2	1/2
Requirements quality	***4/1***	–	–
Requirements volatility	***4/1***	*8/2.5*	*1/2*
Required software quality	**8/1**	**12/3.5**	2/4.5
Required software reliability	***7/1***	***10/4***	–
Required software maintainability	–	***4/3.5***	–
Software complexity	3/1	**17/1**	**3/3**
Database size and complexity	–	***7/1***	–
Source code complexity	*1/1*	***4/1***	–
Complexity of interface to other systems	*3/1*	*2/2.5*	***1/3***
Architecture complexity	*3/1*	*3/1*	***1/1***
Context factors			
Programming language	**7/1**	**10/1**	**1/1**
Domain	**4/1**	4/2	–
Target platform	2/1	**4/1**	–
Life cycle model	2/1	3/1	–
Development type	1/1	**4/1**	**1/1**
Business sector	1/14	**4/1**	–

TABLE V
TOP REUSE-SPECIFIC PRODUCTIVITY FACTORS

Factor	Frequency (no. of references)	Median weight
Quality of reusable assets	12	1
Maturity	*7*	*1*
Volatility	*5*	*1*
Reliability	*3*	*1*
Asset complexity	9	1
Interface complexity	*5*	*1*
Architecture complexity	*4*	*1*
Team capabilities and experience	6	1
Integrator experience and skills	*3*	*1*
Integrator experience with the asset	*3*	*1*
Integrator experience with COTS process	*3*	*1*
Product support (from supplier)	6	1
Supplier support	*6*	*1*
Provided training	*5*	*1*
Required software quality	4	1
Required reliability	*3*	*1*
Required performance	*3*	*1*
Context factors		
Life cycle model	2	1
Programming language	1	1
Domain	1	1

4.3 Reuse-Specific Factors

From among the reviewed references, 15 focused specifically on reuse-oriented software development, that is, development with and for reuse (including COTS—commercial-off-the-shelf components). Table V summarizes the most common factors influencing development productivity in that context.

4.4 Summary of Literature Review

The review of the software engineering publications presented here shows that software development productivity still depends on the capabilities of people and tools involved (Fig. 4).

The productivity factors selected by researchers and software practitioners confirm requirements specification, coding, and testing as the traditionally acknowledged essential phases for the success of the whole development process. Moreover, the high importance of *project constraints* and *project manager's skills* suggests project management as another key factor for project success. Finally, as might have

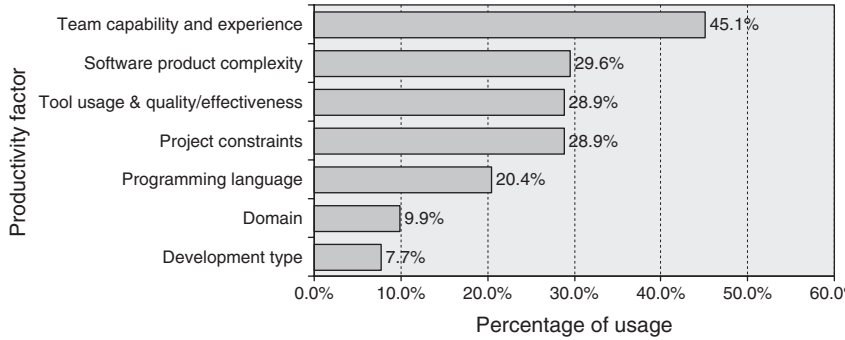

FIG. 4. Literature review: most common productivity factors.

been expected, the internal (architecture, data) and external (interfaces) complexity of software is a significant determinant of development productivity.

Yet, software complexity and programming language are clearly factors preferred in the context of project data repositories. This most probably reflects a common intuition of repository designers that those factors have a significant impact on development productivity and numerous software qualities (e.g., reliability, maintainability).

As already mentioned in the introduction, the importance of a certain productivity factor varies depending on the project context. The skills of software programmers and analysts, for instance, seem to play a more important role in enhancement/maintenance projects. Similarly, tool/method usage seems to be less important in new development projects. On the other hand, software development that is not a continuation of a previous product/release seems to significantly rely on the quality of project management and team motivation.

The results presented in the literature support the claim made by Fenton and Pfleeger [30] who suggest that software complexity might have a positive impact on productivity in the context of new development (and thus is not perceived as a factor worth considering) and a negative impact in case of maintenance and enhancement.

What might be quite surprising is that the domain is not considered in the new development project at all (Fig. 5).

Regarding software domain-specific factors, there are several significant differences between factors playing an important role in the embedded and MIS domains. The productivity of embedded software development depends more on *tools and methods used*, whereas that of MIS depends more on the *product complexity* (Fig. 6).

Finally, *reuse* is not as significant a productivity factor as commonly believed. Less than 17% of publications mention reuse as having a significant influence on development productivity. This should not be a surprise, however, if we consider the complex nature of the reuse process and the numerous factors determining its success (Fig. 7).

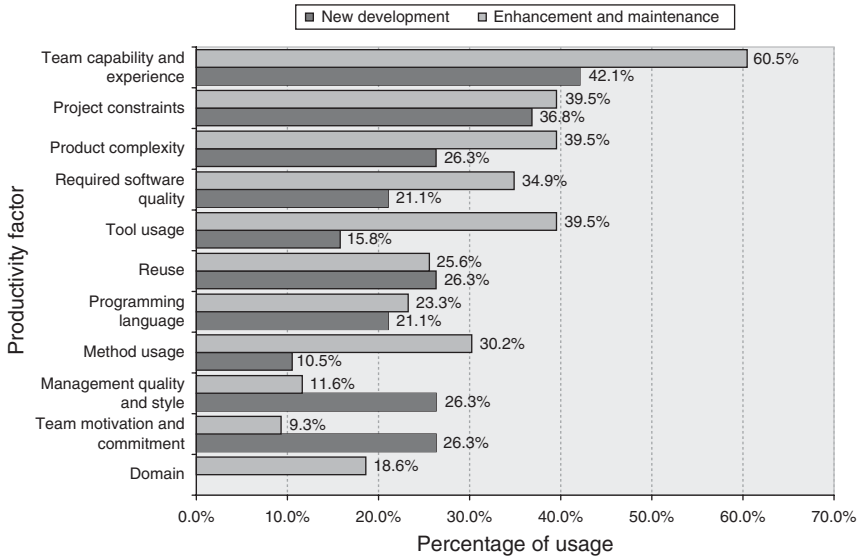

FIG. 5. Literature review: development-specific factors.

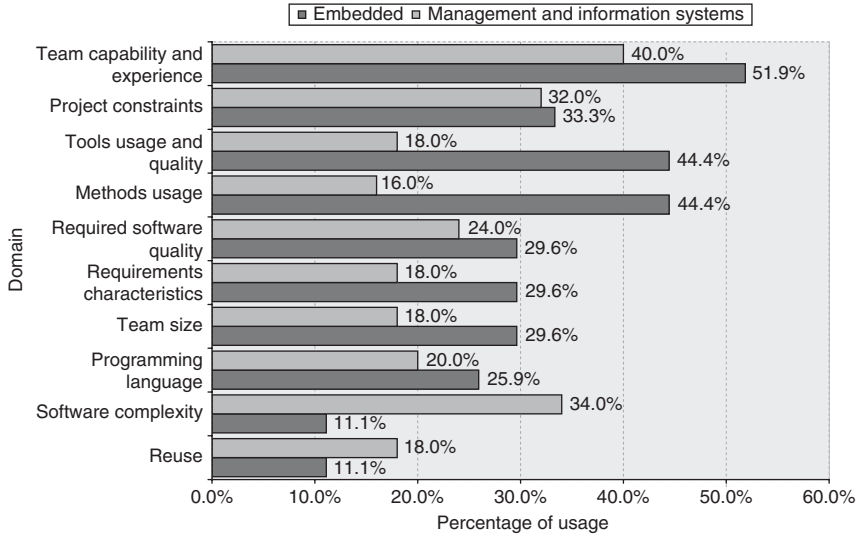

FIG. 6. Literature review: domain-specific factors.

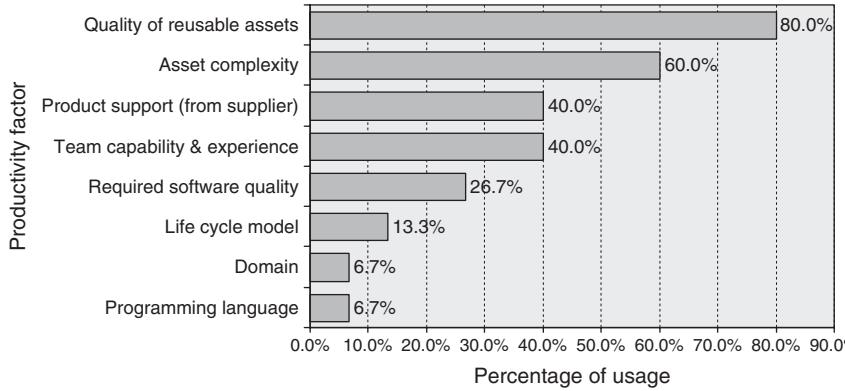

FIG. 7. Literature review: reuse success factors.

According to the reviewed studies, the complexity and quality of reusable assets are key success factors for software reuse. Furthermore, the support of the asset's supplier and the capabilities of development team (for integrating the asset into the developed product) significantly influence the potential benefits/losses of reuse (see Section 6.2.4 for a comprehensive overview of key reuse success factors).

5. Overview of Factors Indicated by Industrial Experiences

This section summarizes experiences regarding the most commonly used productivity factors indicated by our industrial experiences gained in recent years at Fraunhofer IESE. The studies summarized here are subject to a nondisclosure agreement, thus only aggregated results without any details on specific companies are presented.

5.1 Demographics

The IESE studies considered in this section include:

- Thirteen industrial projects on cost and productivity modeling performed for international software organizations in Europe (mostly), Japan, India, and Australia.
- Four international workshops on software cost and productivity modeling, which took place in the years 2005–2006 in Germany and Japan. The workshop

participants came from both academic (universities, research institute) and industrial (software and system) contexts.

- Eight worldwide surveys on cost and productivity modeling. This includes one crosscompany survey where we talked to a single representative of various software organizations, and seven surveys where we talked to a number of software engineers within a single company.

The studies considered here covered a variety of software domains (Fig. 8) and development types (Fig. 9).

In total, we identified 167 different factors, with 31 of them representing complex phenomena and 136 basic concepts (subfactors).

5.2 Cross-Context Factors

Table VI and Figure 10 present these productivity factors that were selected most often in the context of all IESE studies (commercial projects, surveys, and workshops).

Similarly to published studies, *development team capabilities, project constraints*, and *method usage* were considered as the most significant factors influencing

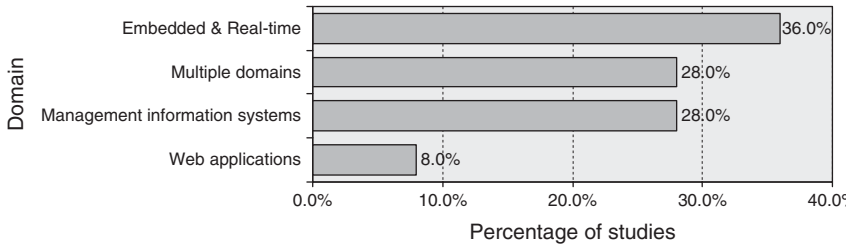

FIG. 8. Industrial experiences: application domains considered.

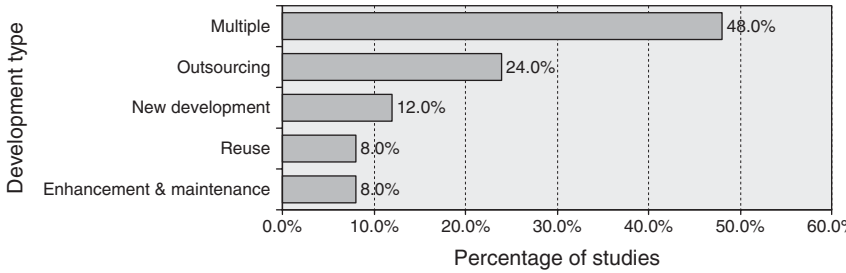

FIG. 9. Industrial experiences: development types considered.

TABLE VI
INDUSTRIAL EXPERIENCES: CROSS-CONTEXT PRODUCTIVITY FACTORS

Factor	Frequency (no. of references)	Median rank
Requirements quality	23	1
Requirements volatility	*20*	*1*
Requirements novelty	*11*	*1*
Team capabilities and experience	23	3
Project manager experience and skills	*10*	*4.5*
Programming language experience	*10*	*17*
Teamwork and communication skills	*9*	*3*
Domain experience and knowledge	*9*	*5*
Project constraints	20	4.6
Schedule pressure	*13*	*2*
Distributed/multisite development	*11*	*6*
Customer involvement	18	2
Method usage and quality	18	4.3
Requirements management	*10*	*2.5*
Reviews and inspections	*7*	*5*
Required software quality	18	8.5
Required software reliability	*16*	*4*
Required software maintainability	*10*	*15*
Context factors		
Life cycle model	8	3
Development type	3	5
Domain	2	1.5

software development productivity. Unlike other practitioners, software engineers involved in IESE undertakings selected *requirements novelty and stability, customer involvement*, and *required software reliability and maintainability* as high-importance productivity factors. High importance of requirements volatility and customer involvement might be connected to the fact that most of the IESE studies regarded outsourcing projects where stable requirements and communication between software supplier and contractor are essential for project success.

The fact that *programming language* does not occur at all as a context factor results from the homogeneous context of most IESE studies (most of them regard a single business unit or group where a single programming language is in use).

5.3 Context-Specific Factors

In this section, we look at the variances in selected productivity factors across different contexts they were selected in.

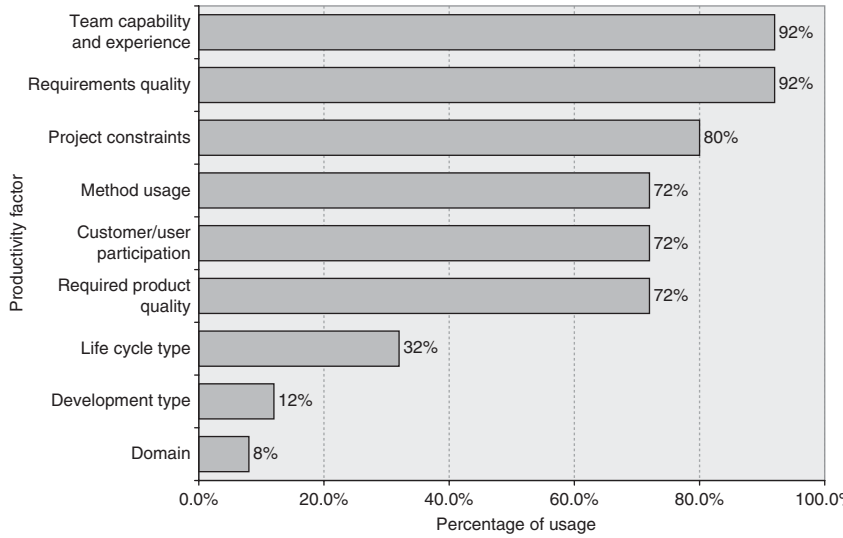

FIG. 10. Industrial experiences: cross-context factors.

Each cell of Tables VII and IX contains information in the form X/Y, where X is the number of studies where a certain factor was selected (*Frequency*) and Y means the median rank given to the factor over those studies (*Median rank*). Moreover, the most significant factors for a certain context are marked with bold font and cells containing factors that were classified as the most significant ones in two or more contexts are gray-filled. An empty cell means that a factor did not appear in a certain context at all.

For each context considered, the number of relevant references is given in the table header (in brackets).

5.3.1 Study-Specific Factors

In Table VII, the most commonly selected factors across various study types are presented.

Notice that practically no context factors were considered in the context of commercial projects, which, in fact, took place in a homogeneous context where all usually significant context factors were constant (thus had no factual impact on productivity).

Moreover, product qualities and constraints such as reliability, maintainability, and performance (execution time and storage) are also mentioned only in the context of specific commercial projects.

TABLE VII
INDUSTRIAL EXPERIENCES: STUDY-SPECIFIC PRODUCTIVITY FACTORS

Factor	C&P (13)	Srv (8)	Wsk (4)
Requirements quality	**13/3**	**8/1**	2/1
Requirements volatility	***12/1***	***6/1***	*2/1*
Requirements novelty	***7/9***	***3/1***	*1/1*
Required software quality	**13/13.5**	4/2	1/1
Required software reliability	***12/5***	*3/3*	*1/1*
Required software maintainability	***9/16***	*1/1*	–
Required software usability	***7/19***	–	–
Project constraints	**12/7.5**	5/3	**3/1**
Schedule pressure	***6/2.5***	*5/3*	***2/1***
Distributed/multisite development	***5/14***	*4/3.5*	***2/1***
Budget constraints	–	–	***2/1***
Team capability and experience	**12/7.3**	**8/1**	**3/1**
Programming language experience	***8/20.5***	*1/15*	*1/1*
Project manager experience and skills	***7/8***	*2/4.5*	*1/1*
Platform (virtual machine) experience	***5/14***	–	–
Teamwork and communication skills	*3/4*	***6/2***	–
Domain experience and knowledge	*4/18.5*	***4/3***	*1/1*
Training level	*3/19*	*1/4*	***2/1***
Software product constraints	**11/12**	2/8	1/1
Execution time constraints	***11/11***	*2/7*	–
Storage constraints	***6/18.5***	*1/1*	–
Customer/user involvement	10/2	**7/2**	1/1
Method usage	10/8.5	**7/3**	1/1
Reviews and inspections	*3/9*	***4/1.5***	–
Requirements management	*6/1.5*	***4/3***	–
Tool usage and quality	9/10	**6/4.7**	1/1
Testing tools	*1/35*	***2/3***	–
Project management tools	*1/42*	***2/10.5***	–
Reuse	10/14	4/2	**3/1**
Reuse level	*6/13.5*	*3/3*	***2/1***
Required software reusability	*6/12*	*1/18*	***2/3***
Context factors			
Life cycle model	**2/18**	**3/5**	**3/1**
Domain	–	**2/1.5**	–
Development type	–	**2/6**	**1/1**

5.3.2 Development-Type-Specific Factors

For most of the studies considered here, the development type was either not clear (not explicitly stated) or productivity factors were given as valid for multiple development types. Since there are too little data to analyze productivity factors for most of the development types considered, we only take a closer look at factors that are characteristic for organizations outsourcing their software development (Table VIII).

It might be no surprise that in outsourcing projects, *requirements quality* and *customer involvement* are key factors driving development productivity. Stable requirements and communication with the software customer (especially during early development phases) are commonly believed to have crucial impact on the success of outsourcing projects.

5.3.3 Domain-Specific Factors

Unfortunately, most of the IESE studies considered either regarded multiple domains or the domain was not clearly (explicitly) specified. Therefore, we analyzed only factors specific for the embedded systems (Emb) and management and information systems (MIS) domains, for which a reasonable number of inputs were available.

Similar to what we found in the related literature, the *usage of tools* (especially testing) is characteristic of the embedded domain. Regarding *method usage*,

TABLE VIII
INDUSTRIAL EXPERIENCES: PRODUCTIVITY FACTORS IN THE OUTSOURCING CONTEXT

Factor	Frequency (no. of references)	Median weight
Requirements quality	6	6.2
Requirements volatility	*6*	*1.5*
Requirements novelty	*5*	*11*
Customer/user involvement	6	4
Project constraints	6	8.5
Distributed/multisite development	*3*	*14*
Concurrent hardware development	*3*	*12*
Legal constraints	*3*	*30*
Software complexity	6	12.5
Database size and complexity	*6*	*22*
Complexity of device-dependent operations	*2*	*8.5*
Complexity of control operations	*2*	*9*
Context factors		
Life cycle model	1	18

different methods play an important role in the embedded and MIS domains. Use of early quality assurance methods (reviews and inspections) is regarded as having a significant impact on productivity in the MIS domain, whereas use of late methods, such as testing, counts more in embedded software development.

Requirements management as well as configuration and change management activities are significant productivity factors only in the MIS domain. Software practitioners do not relate them to productivity variance in the embedded domain because they are usually consistently applied across development projects. According to our observation, however, the effectiveness of those activities varies widely and therefore should be considered as a significant influence factor (Table IX).

5.4 Summary of Industrial Experiences

The productivity factors observed in the most recent IESE studies do not differ much from those indicated in the reviewed literature. *Capabilities of development team, project constraints*, and *methods usage* are the main impact factors. The outsourcing character of the projects considered in the majority of the IESE studies gave priority to some additional factors such as *requirements quality* (volatility, complexity, and novelty), *required product quality* (reliability, maintainability, and performance), and *customer involvement*.

There were slight differences between the factors considered across various domains. As in the reviewed literature, *tool and method usage* were regarded as more significant in the embedded domain.

The IESE studies considered referred to a rather narrow context (organization, business unit, group, etc.), where such characteristics as domain and programming language were quite homogeneous. That is most probably why common context factors from the literature (see Section 4.1) do not actually count in the context of IESE studies. Constant factors do not introduce any variance across projects and thus do not explain productivity variance, which in practice makes their consideration useless (Fig. 10).

One quite surprising observation is that the implementation of early quality assurance techniques such as reviews and inspections does not seem to have a deciding impact on development productivity. In only 6 out of 25 studies, this factor was stated explicitly as having a significant impact on productivity. At the same time, our practical experiences indicate that those activities are usually ineffective or are not applied at all. This contradicts the common belief that early quality assurance activities contribute significantly to the improvement of development productivity (through decreased testing and rework effort).

Another interesting observation is that while *requirements volatility* is considered to have a significant impact on development productivity, *requirements management*

TABLE IX
INDUSTRIAL EXPERIENCES: DOMAIN-SPECIFIC PRODUCTIVITY FACTORS

Factor	Em (9)	MIS (7)
Requirements quality	**9/5**	**7/1**
Requirements volatility	***9/1***	***7/1***
Requirements novelty	***6/10***	–
Requirements complexity	–	***2/1***
Software complexity	**9/5**	5/9
Architecture complexity	***3/7***	–
Database size and complexity	***3/22***	*2/16.5*
Team capabilities and experience	**9/8.4**	**7/3**
Programming language experience	***7/19***	–
Project manager experience and skills	***5/9***	***3/3***
Teamwork and communication skills	*2/1*	***5/3***
Customer/user involvement	**8/2**	**7/2**
Involvement in design reviews	***2/19.5***	–
Tool usage and quality	**8/9**	3/15
Testing tools	***2/18***	–
Method usage	**8/9.6**	**7/3**
Requirements management	3/3	**7/2**
Reviews and inspections	2/6.5	**4/7**
Testing methods	4/5.5	2/8
Documentation methods	5/20	–
Configuration management and change control	2/22.5	**3/4**
Project constraints	7/9	**6/3.7**
Schedule pressure	3/9	**5/2**
Distributed/multisite development	3/14	**4/8.5**
Context factors		
Life cycle model	**3/14**	**1/9**
Domain	**1/1**	**1/2**
Application type	–	**1/2**

is not consistently regarded as having such an impact. It is, however, believed that proper quality management may moderate the negative influence of unstable requirements on development productivity.

Finally, an interesting and, at the same time, surprising observation we made is that very mature organizations (e.g., two organizations at CMMI L-5 and one ISO-9001 certified) consider as very significant factors such factors as *Clarity of development team roles and responsibilities*, which are actually common for rather immature organizations.

6. Detailed Comments on Selected Productivity Factors

This section presents a brief overview of publications presenting in-depth investigations on rationales underlying the influence of certain factors on software development productivity as well as dependencies between various factors. The summary presented here includes empirical research studies as well as industrial experiences.

6.1 Comments on Selected Context Factors

This section presents an overview of literature regarding the experiences with context factors, that is, factors that are used to limit the context of productivity modeling and analysis (see Section 3.1).

6.1.1 Programming Language

The analysis of the factors presented in the reviewed literature showed that programming language is the most common context factor (Table I). The impact of a programming language on development productivity is considered to be so large that some authors present average development productivities for certain languages, independent of other potential factors [33]. Moreover, a single organization or business unit usually uses a limited number of distinct languages. Therefore, considering this factor when analyzing productivity within a single organization usually makes no sense. This conclusion is confirmed several data analysis studies (e.g., [34, 35, 126]) where programming language has significant impact on productivity when analyzed on crossorganization data whereas no influence was observed on organization-specific data.

6.1.2 Application Domain

According to the literature review presented (Table I), the application domain is considered as the second most significant context factor.

The studies presented in the literature provide evidence not only that factors influencing productivity vary across different application domains, but also that, in principle, the magnitude of productivity alone varies across different domains. Putnam and Myers [36, 119], for example, analyzed the QSM data repository [37] and found out that there is a common set of factors influencing software process

productivity. They also found out that the analyzed projects tend to cluster around a certain discrete value of productivity,[4] and that except for a limited number of projects (outliers), each cluster actually represents a certain application domain. Therefore, although an exact productivity measurement would require considering other influence factors within a certain context (cluster), general conclusions about productivity within a certain context can already be drawn. The authors provided evidence that projects in a real-time domain tend to be less productive, in general, than in the business systems domain. Jones combines productivity variance across domains with different levels of project documentation generally required in each domain. He observed [31] that the level of documenting varies widely across various software domains. The productivity of military projects, for instance, suffers due to the extremely high level of documentation required (at least three times more documentation per function point is required than in MIS and Software Systems domains) [31].

6.1.3 Development Type

Development type is another important factor that determines how other project characteristics influence development productivity.

Fenton and Pfleeger [30] suggest, for instance, that software complexity might have a positive impact on productivity in the context of new development and a negative one in case of maintenance and enhancement. The development team is able, for example, to produce more output (higher productivity) even though the output is badly structured (high complexity). Yet, one may expect that if producing badly structured output might be quite productive, then maintaining it might be quite difficult and thus unproductive. This could be, for instance, observed well in the context of software reuse, where the required quality and documentation of reusable artifacts have a significant impact on the productivity of their development (negative impact) and reuse (positive impact).

In the context of maintenance projects, the purpose of software change has a significant impact on productivity. Bug fixes, for instance, are found to be more difficult (and therefore less productive) than comparably sized additions of new code by approximately a factor of 1.8 [38]. Bug fixes tend to be more difficult than additions of new code even when additions are significantly larger.

Corrective maintenance (bug fixes), however, seems to be much more productive than perfective maintenance (enhancements) [39]. This supports the common intuition that error corrections usually consist of a number of small isolated changes,

[4] The standard deviation of process productivity within clusters ranged from ±1 to ±4.

while enhancements include larger changes to the functionality of the system. In both types of changes, the extent of coupling between modified software units may amplify or moderate the impact of change type on productivity. Changing a certain software component requires considering the impact of the change on all coupled components. This usually includes reviewing, modifying, and retesting related (coupled) components. Therefore, the more coupled components the more additional work is required (lower productivity).

The time spam between software release and change (so-called "aging" or "decay" effect) is also considered as a significant productivity factor in enhancement and maintenance projects. Graves [38], for instance, confirms the findings of several earlier works. He provides statistically significant evidence that a one-year delay in change would cost 20% more effort than similar change done earlier.

Finally, maintenance productivity has traditionally been influenced by the personal capabilities of software maintainers. It was, for example, found that one developer may require 2.75 times as much effort as another developer to perform comparable changes [38]. Yet, it is not always clear exactly which personnel characteristics are determinant here (overall experience, domain experience, experience with changed software, extent of parallel work, etc.).

6.1.4 Process Maturity

Process maturity is rarely identified directly as a factor influencing development productivity. Yet, numerous companies use project productivity as the indirect measure of software process maturity and/or use productivity improvement as a synonym of process improvement. Rico [40] reports, for instance, that 22% of all measures applied for the purpose of software process improvement are productivity measures. Another 29% are development effort, cost, and cycle time measures, which, in practice, are strongly related to development productivity.

A common belief that pushes software organizations toward process improvement is that high-maturity organizations (e.g., as measured according to the CMMI model [41]) are characterized by high-productivity processes[5] [3, 17, 42, 43]. In fact, there are several studies that provide quantitative evidence of that belief. Diaz and King [17] report, for instance, that moving a certain software organization from CMM level 2 to level 5 in the years 1997–2001 consistently increased project productivity by the factor 2.8. A threefold increase in productivity within organizations approaching higher CMM levels has been confirmed by several other software companies

[5] Putnam [42] analyzed the QSM database and showed that there seems to be a strong correlation between an organization's CMM level and its productivity index.

worldwide [3]. Harter et al. [44] investigate the relationships between process maturity measured on the CMM scale, development cycle time, and effort for 30 software products created by a major IT company over a period of 12 years. They found that in the average values for process maturity and software quality, a 1% improvement in process maturity leads to a 0.32% net reduction in cycle time, and a 0.17% net reduction in development effort (taking into account the positive direct effects and the negative indirect effects through quality).

Yet, according to other studies (e.g., [45, 46]), overall organization maturity can probably not be considered as a significant factor influencing productivity. One may say that high maturity entails a certain level of project characteristics (e.g., CMMI key practices) positively influencing productivity; however, which characteristics influence productivity and to which extent varies most probably between different maturity levels. In that sense, process maturity should be considered as a context rather than as an influence factor. Influence factors should then refer to single key practices rather than to whole maturity levels.

Overall maturity improvement should be selected if the main objective is a long-term one, for example, high-quality products delivered on time and within budget. In that case, increased productivity is not the most important benefit obtained from improved maturity. The more important effects of increased maturity are stable processes, which may, in turn, facilitate the achievement of short-term objectives, such as effective productivity improvement (Fig. 11). One must be aware that although increasing the maturity of a process will not hurt productivity in the long-term perspective, it may hurt it during the transition to higher maturity levels. It has been commonly observed (e.g., [48]) that the introduction of procedures, new tools, and methods is, in the short-term perspective, detrimental to productivity (so-called *learning effect*). Boehm and Sullivan [47], for instance, illustrate productivity behavior when introducing new technologies (Fig. 11).

This learning effect can be moderated by introducing a so-called *delta team* consisting of very skilled personnel who are is able to alleviate the short-term, negative effect on productivity of implementing certain key process areas (KPAs). This is, however, nothing else than preventing the productivity decrease caused by investments on implementing certain KPAs by improving factors that have proved to have a significant positive impact on productivity (in this case, team capabilities). Benefits from process improvement and from introducing new technologies can also be gained faster by sharing experiences and offering appropriate training and management support [49, 50].

An alternative way of moderating the short-term, negative impact of process improvement on productivity would be to first implement KPAs that have a short-term, positive impact on productivity (e.g., team capabilities, personnel

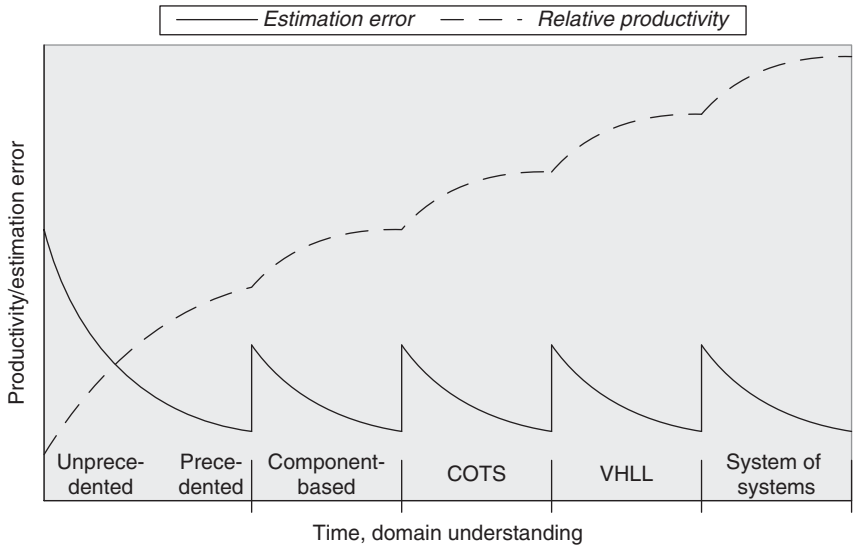

FIG. 11. Short-term effect on productivity of introducing new technology [47].

continuity) in order to offset the negative impact of implementing other KPAs. Example practices (KPAs) that have proved to make the greatest contribution to organizational maturity (highest benefits) can be found in [3].

Yet, a certain maturity level (optionally confirmed by a respective certification) is rather a "side effect" (a consequence) of process improvement (e.g., driven by a high-productivity objective) rather than an objective for its own sake—which is the right sequence [51].

Finally, the maturity of measurement processes should be the subject of special attention since inconsistent measurements usually lead to misleading conclusions with respect to productivity and its influencing factors. In that sense, one may say that mature and rigorous measurement processes have a significant impact on productivity [52]. Boehm and Sullivan [47] claim that there is about a $\pm 15\%$ range of variation in effort estimates between projects and organizations due to the counting rules for data. Niessink and van Vliet [53] observed, in turn, that the existence of a consistently applied (repeatable) process is an important prerequisite for a successful measurement program. If, for instance, a process exists in multiple variants, it is important to identify those factors that differentiate various variants and to know which variant is applied when.

6.1.5 Development Life Cycle

Only a few studies selected the development life cycle model as a significant factor influencing software productivity. This factor is, however, usually addressed indirectly as a context factor. Existing empirical studies and in the field experience (e.g., [54]) confirm, for instance, the common belief that iterative and incremental software development significantly increases development productivity (as compared to the traditional waterfall model).

6.2 Comments on Selected Influence Factors

In this section, we present an overview of comments on and experiences with selected factors influencing software development productivity.

6.2.1 Development Team Characteristics

Team size and structure is an important team-related factor influencing development productivity. In principle, the software engineering literature indicates significant benefits from using small teams. According to Camel and Bird [55], a common justification for small teams is not that a small team is so advantageous, but that a large team is disadvantageous. However, this does not seem to be consistent for the whole development life cycle. Brooks [56] says, for instance, that when we add people at the project's end, the productivity goes down. Blackburn et al. [57] support this thesis claiming that the small "tiger teams" in later stages of software development (after requirements specification) tend to increase productivity. Those experiences support the manpower Rayleigh buildup curve proposed in [58] and adapted in [36] where a larger team is used during early development phases and the team size drops during later phases.

However, the impact of team size on productivity has much to do with the communication and coordination structure [30, 59, 60]. In other words, the impact of team size depends significantly on the degree to which team members work together or separately, and on the communication structure [61] rather than simply on the number of people in a team. It has been observed that the main reason why large teams are usually so unproductive is primarily the burden of maintaining numerous communication links between the team members working together [55]. Thus, even large teams, where developers work mostly independently, do not seem to have a significant negative impact on productivity compared to small teams [61].

Even if large teams are indispensable, their negative influence on productivity can be moderated by an effective team (communication) structure and communication

media. For instance, even in large-scale projects, minimizing the level of work concurrency by keeping several small, parallel teams (with some level of independence) instead of one large team working together seems to be a good strategy for preventing a drop in productivity [62]. That is, for example, why agile methods promoting two-person teams working independently (pair programming) may be (however only in certain conditions) more productive than large teams working concurrently [63]. Yet, as further reported in [63], well-defined development processes are not without significance in agile development. A large industrial case study proved, for example, that even pairs of experienced developers working together need a collaborative, role-based protocol to achieve productivity [63].

Observing social factors in hyperproductive organizations, Cain et al. [64] found the developer close to the core of the organization. Average and low productive organizations exhibit much higher prestige values for management functions than for those who directly add value to the product (developers). Collaborations in these organizations almost always flow from the manager in the center to other roles in the organization. Even though a *star structure* ("chief surgical team") is mostly the favored communication structure, it seems that its impact on productivity depends on who plays the central role, managerial (*outward control flow*) or technical (*inward control flow*) staff. The authors point to architect as an especially prestigious role in a team within highly productive organizations.

The use of proper *communication media* may also significantly improve team communication and, in consequence, development productivity. Face-to-face communication still seems to be the most efficient way of communication. Carey and Kacmar [65], for instance, observed that although simple tasks can be accomplished successfully using electronic communication media (e.g., telephone, email, etc.) complex tasks may result in lower productivity and larger dissatisfaction with electronic medium used. Agile software development considers one of the most effective ways for software developers to communicate to be standing around a whiteboard, talking, and sketching [66]. Ambler [51] goes further and proposes to define a specific period during the day ($\sim$5 h) during which team members must be present together in the workroom. Outside that period, they may go back to their regular offices and work individually. A more radical option would be to gather the development team in a common workroom for the whole duration of the project. Teasley et al. [67] conclude their empirical investigation by saying that having development teams reside in their own large room (an arrangement called *radical collocation*) positively affected system development. The collocated projects had significantly higher productivity and shorter schedules than both the performance of similar past projects considered in the study and industry benchmarks. Yet, face-to-face communication does not guarantee higher productivity and may still vary widely dependent on the team communication structure (especially for larger teams).

Finally, it was observed that, besides a core team structure, close coupling to the quality assurance staff and the customer turned out to be a pattern within highly productive software organizations [64].

Staff turnover is another team-related project characteristic having a major impact on development productivity. Collofello et al. [68] present an interesting study where, based on a process simulation experiment, they investigated the impact of various project strategies on team attrition. No action appeared to be the most effective strategy when schedule pressure is high and cost containment is a priority. Replacing a team member who left a team alleviates the team exhaustion rate (lightens increased attrition). Even though overstaffing is an expensive option, replacing a team member who left with several new members minimizes the duration of the project. Thus, this strategy should be considered for projects in which the completion date has been identified as the highest priority. It is, however, not clear how those results (especially regarding overstaffing) relate to Brooks' real-life observation that putting more staff at the end of a project makes it even more late [56].

Task assignment is the next reported human-related factor differentiating software development productivity. Boehm [43], for instance, introduces in his COCOMO II model the task assignment factor to reflect the observation that proper assignment of people to corresponding tasks has a great impact on development productivity. Hale et al. [61, 69] go further and investigate related subfactors such as *intensity, concurrency*, and *fragmentation*. They observed that the more time is spent on the task by the same developer (*intensity*), the higher the productivity. Further, the more team members work on the same task (*concurrency*), the lower the productivity (communication overhead disproportionately larger than task size). Finally, the more fragmented the developer's time over various tasks (*fragmentation*), the lower the productivity (due to overhead to switch work context).

Finally, *team capabilities and skills* is traditionally acknowledged as the most significant personnel characteristic that influences development productivity. Team capabilities also have an indirect impact on productivity by influencing other factors with a potential impact on development productivity. The impact of the team structure on productivity, for instance, usually assumes highly skilled technical people. Moreover, the full benefits of tool support and technological improvements cannot be achieved without capable personnel [70]. On the other hand, the impact of team capabilities on productivity may be amplified or moderated by other factors. An appropriate management and communication structure can, for instance, moderate the negative impact of a low-experienced team on productivity; it can, however, never compensate for the lack of experience [57].

Researchers and practitioners agree about the high importance of team capabilities regarding development productivity. Yet, it is not clear exactly which skills

have an impact on productivity, and which do not. Presented in Sections 4 and 5 suggests that programming language skills, domain experience, and project management skills are essential characteristics of a highly productive development team. Krishnan [59], however, claims that whereas higher levels of domain experience of the software team is associated with a reduction in the number of field defects in the product (higher quality), there is no significant direct association between either the language or the domain experience of the software team and the dollar costs incurred in developing the product. This does not, however, necessary imply no impact on productivity. Software teams with low experience (<2 years) may cost more due to lower productivity, whereas highly experienced teams (>10 years) may cost more due to higher salaries. Here we can again see that when observing development productivity and its influence factors, a careful definition of respective measures is crucial for the validity of the conclusions drawn.

Although not reflected by the results presented in this chapter (Sections 4 and 5), significant team capabilities do not only cover technical skills. There are several nontechnical skills that are found to be essential with respect to development productivity. Examples are ability and willingness to learn/teach quickly, multiarea (general) specialization, knowledge of fundamentals, and flexibility [71]. It has been observed (especially in a packaged software development environment) that it is the innovation, creativity, and independence of the developers that determine development productivity and project success [69, 72]. Yet, as warned in [73], an emphasis on creativity and independence creates an atmosphere that contributes to a general reluctance to implement accurate effort and time reporting. This, in turn, limits the reliability of the collected productivity data and may lead to wrong conclusions regarding productivity and its impact factors.

6.2.2 Schedule/Effort Distribution

The right schedule and work distribution are the next significant parameters influencing development productivity. Looking at the published investigations, one may get the impression that the development schedule should be neither too long nor to short. Thus, both schedule over- and underestimation may have a negative impact on productivity.

Several authors have observed the negative impact of schedule compression on productivity (so-called *deadline effect*) [43, 74]. A schedule compression of 25% (which is considered as *very low* compression) may, for instance, lead to a 43% increase in development costs [75]. Schedule compression is recognized as a key element by practically all of the most popular commercial cost estimation tools (e.g., PRICE-S [76], SLIM [36, 119], SEER-SEM [77], and COCOMO I/II [14, 43]), which implement various productivity penalties related to it [75].

Others show that schedule expansion also seems to have a negative impact on development productivity [26, 57, 75]. One of the possible explanations of this phenomenon is the so-called Parkinson's low, which says that "the cost of the project will expand to consume all available resources" [78].

Yet, the right total schedule is only one part of success. The other part is its right distribution across single development phases. An investigation on productivity shown in [57] concludes that some parts of the process simply need to be "unproductive," that is, should take more time. This empirical insight confirms the common software engineering wisdom that more effort (and time) spent on early development stages bears fruit, for example, through less rework in later stages and finally increases overall project productivity. Apparently, the initial higher analysis costs of highly productive projects are more than compensated for by the overall shorter time spent on the total project [23, 79]. If the analysis phase, for example, is not done properly in the first place, it takes longer for programmers to add, modify, or delete codes. The Norden's Rayleigh curve [58] may here be taken as a reference shape of the most effective effort and time distribution over the software development life cycle. Yet, the simple rule of "the more the better" does not apply here and after exceeding a certain limit of time spent on requirements specification, the overall development productivity (and product quality) may drop [16]. The results of a case study at Ericsson [80] determined, for example, that the implementation phase had the largest improvement potential in the two studied projects, since it caused a large faults-slip-through to later phases, that is, 85 and 86% of the total improvement potential of each project.

6.2.3 Use of Tools and Methods

The effect of the usage of certain tools and technologies on software development productivity is widely acknowledged (see Sections 4 and 5), but at the same time, paradoxically not well understood [81]. The opinions regarding the impact of tool usage vary largely across individual studies. Following the factors in Boehm's COCOMO II model [43], numerous organizations consider tool usage as a significant means for improving development productivity. Yet, empirical studies have reported their positive and negative effects [57] as well as little (no significant) direct effect [60]. This paradox might be explained by findings that the effect of tool usage on development productivity is coupled with the impact of other factors and that isolated use of tools makes little difference in the productivity results [1, 71, 82].

It has been observed, for instance, that due to the significant costs involved in learning the CASE tools, the effect of tools in some projects is negative, that is, they increase the effort required for developing software products [83]. Other significant

factors affecting the relationship between tool usage and development productivity are the degree of tool integration, tool maturity, tool training and experience, appropriate support when selecting and introducing corresponding tools (task-technology fit), team coordination/interaction, structured methods use, documentation quality, and project size [54, 82–85]. For instance, when teams receive both tool-specific operational training and more general software development training, higher productivity is observed [83]. The same study reports on a 50% increase in productivity if tools and formal structured methods are used together.

Abdel-Hamid [1] observed, moreover, that studies conducted in laboratory settings using relatively small pilot projects tend to report productivity improvements of 100–600%, whereas when the impact of CASE tools is assessed in real organizational settings, much more modest productivity gains (15–30%) or no gains at all were found. The author relevantly summarizes that project managers are the main source of failed CASE tools application, because most often they do not institute rudimentary management practices needed before and during the introduction of new tools. What they usually do is to look for a solution they can buy. Actually, in a simulation presented by Abdel-Hamid, almost half the productivity gains from new software development tools were squandered by bad management. Moreover, Ambler [51] underlines being flexible regarding tool usage. He observed that each development team works on different things, and each individual has different ways of working. Forcing inappropriate tools on people will not only hamper progress, it can destroy team morale.

The impact of project size and process maturity on the benefits gained from introducing tools is shown in [86]. The authors observed at IBM software solutions that in the context of advanced processes, productivity gains through requirements planning tools may vary between 107.9% for a small project (five features) and −23.5% for a large project (80 features). Yet, in the context of a small project (10 features) such a gain may vary between −26.1 and 56.4% when regular and advanced processes are applied, respectively. A replicated study confirmed this trend—productivity decreases with growing project size and higher process complexity. The productivity loss in the larger project was due to additional overhead for processing and maintaining a larger amount of data produced by a newly introduced tool. Higher process complexity, on the other hand, brings more overhead related to newly introduced activities (processes). Yet, measuring at the macrolevel makes it difficult to separate the impact of the tool from other confounding variables (such as team experience and size/complexity of a single feature). Therefore, the results of Bruckhaus [86] should be interpreted cautiously [87].

The use of software tools to improve project productivity is usually interpreted in terms of automated tools that assist with project planning, tracking, and management. Yet, nonautomated tools such as checklists, templates, or guidelines that

help software engineers interpret and comply with development processes can be considered as supporting the positive impact of high-maturity processes on improved project productivity [71].

6.2.4 Software Reuse

Reuse of software products, processes, and other software artifacts is considered the technological key to enabling the software industry to achieve increased levels of productivity and quality [7, 88].

Reuse contributes to an increase in productivity in both new development and software maintenance [89]. There are two major sources of savings generated by reuse (1) less constructive work for developing software (reuse of similar and/or generic components) and (2) less analytical work for testing software and rework to correct potential defects (reuse of high-quality components). In other words, software reuse catalyzes improvements in productivity by avoiding redevelopment and improvements in quality by incorporating components whose reliability has already been established [90]. That is, for example, why reuse benefits are greater for more complex reusable components [91] (however, this positive effect levels off beyond a certain component size).

Nevertheless, reuse is not for free and may, at least at the beginning, bring negative savings (productivity loss). A high-quality, reusable (generic) software component first needs to be developed, which usually costs more that developing specific software (not intended to be reused). Populating the repository of reusable artifacts alone contributes to an initial loss of productivity [91]. The repository must not expand indefinitely, due to additional maintenance costs. On the one hand, creating and maintaining rarely used, small repositories of small components tends to cost more than the reuse savings they generate. As the repository size increases, opportunities for reuse tend to increase, generating more development savings. On the other hand, maintaining and searching through very large repositories may again generate negative reuse savings.

Later on, to actually reuse a certain component, it has to be identified as available in the repository and retrieved. Then, the feasibility of the component to be reused in a specific context has to be determined (a component has to be analyzed and understood). Even if accepted, a component often cannot be reused as is, but has to be modified in order to integrate it into the new product. At the end, it has to be (re)tested in a new context. In each step of reuse, a number of factors influencing its productivity have to be considered. Finally, if the total cost for retrieving, evaluating, and integrating a component is less than the cost of writing it from scratch, it makes economic sense to reuse the component.

The success of reuse depends on numerous factors. Rinie and Sonnemann [92] used an industrial survey to identify several leading reuse success factors (so-called *reuse capability indicators*):

- Use of product-line approach
- Architecture that standardizes interfaces and data formats
- Common software architecture across the product line
- Design for manufacturing approach
- Domain engineering
- Management that understands reuse issues
- Software reuse advocate(s) in senior management
- Use of state-of-the-art tools and methods
- Precedence of reusing high-level software artifacts such as requirements and design versus just code reuse
- Tracing end-user requirements to the components that support them

Atkins et al. [87, 93] confirm part of these results in an empirical study where the change effort of large telecommunication software was reduced by about 40% by using a version-sensitive code editor and about four times when the domain engineering technologies were applied.

The impact of reuse on development productivity, like most other influence factors, is strongly coupled with other project characteristics. It should thus not be simplified by taking as granted the positive impact of reuse on productivity. Frakes and Succi [94] observed, for instance, a certain inconsistency regarding the relationship between reuse and productivity across various industrial data sets, with some relationships being positive and others negative.

One of the factors closely coupled with reuse is the characteristics of the personnel involved in reuse (development of reusable assets and their reuse). Morisio et al. [95] observed that the more familiar developers are with reused, generic software (framework), the more benefit is gained when reusing it. The authors report that although developing a reusable asset may cost 10 times as much as "traditional" development, the observed productivity gain of each development where the asset is reused reached 280%. Thus, the benefit from creating and maintaining reusable (generic) assets increases with the number of its reuses.

Another factor influencing the impact of reuse on development productivity is the existence of defined processes. Soliman [96], for example, identifies the lack of a strategic plan available to managers to implement software reuse as a major factor affecting the extent of productivity benefits gained from reuse. Major issues for managers to consider include commitments from top management, training for

software development teams, reward systems to encourage reuse, measurement tools for measuring the degree of success of the reuse program, necessary reuse libraries, rules regarding the development of reusable codes, domain analysis, and an efficient feedback system.

Software reuse in the context of *object-oriented (OO) software development* is one specific type of reuse that proved to be especially effective in increasing development productivity [97, 98]. Yet, the common belief that using the OO paradigm is sufficient for improving development productivity does not have much quantitative evidence supporting it [98]. OO reuse requires considering several aspects, such as the type of reuse or the domain context of reuse. Results of experiments in the context of C++ implementation show, for instance, that since *black-box reuse* (class reuse without modification) increases programmer productivity, the benefits of *white-box reuse* (reuse by deriving a new class from an existing one through inheritance) are not so clear (especially for reuse of third-party library classes) [29]. Finally, introducing the OO paradigm, like introducing any new technology, requires considering and often adjusting the business context (model) to gain the expected productivity benefits [74]. For example, inappropriate schedule planning (constrained or too long) or lack of process control (e.g., over the effects of newly introduced technology) may completely level down any benefit from the applied new technology, including the object-oriented paradigm [74].

The use of *COTS* components is closely related to software reuse and might, in practice, be classified as a subclass of software reuse. Yet, software reuse differs from COTS software in three significant ways [1]: (a) reuse components are not necessarily able to operate as standalone entities (as is assumed to be the case with most components defined as COTS software); (b) reuse software is generally acquired internally within the software developing organization (by definition, COTS components must come from outside); and (c) reused software usually requires access to the source code, whereas with COTS components, access to the source code is rare (references to so-called "white-box" COTS notwithstanding). From that perspective, the productivity of software development in the context of software reuse and use of COTS is influenced by overlapping sets of factors.

In principle, the factors presented in the literature focus on the productivity of integrating COTS into developed software [1]. Abts [99] postulates, for instance, that increasing the number of COTS components is economically beneficial only up to a certain point, where the savings resulting from the use of COTS components break even with the maintenance costs arising from the volatility of such components.

Finally, *software product lines* and *domain engineering* as the most recent incarnations of the reuse paradigm should be considered as a potential factor influencing productivity of software development in general, and software changes in particular. In fact, they may potentially reduce the cost of software change

three- to fourfold (telecommunication switch system) [100, 101]. Such a gain is in line with the generally accepted view that domain engineering techniques improve software productivity two- to 10-fold. Yet, the cost of change is also influenced by other factors. Besides, quite obviously, the size and complexity of change, the type of change also has a substantial impact on change productivity. Similarly to [38], the authors observed, for instance, that changes that fix bugs are more difficult than comparably sized additions of new code (see also Section 6.1.3).

Concluding, there are various aspects of software reuse (besides simply the level of reuse) that have to be taken into account when considering reuse as a way to increase development productivity in a certain context. In practice, it is recommended to collect organization-specific data related to reuse and productivity, and use these data to guide reuse projects. However, one has to be careful when defining both productivity and reuse measures for that purpose, since improper measure definition might lead to wrong conclusions [102].

6.2.5 Software Outsourcing

Outsourcing has recently been gaining a lot of attention from the software community as a means for reducing development costs. According to SPR data [13], the level of outsourcing tripled in the years 1989–2000. In 2000, about 15% of all information systems were produced under contract or outsource agreements (2% international outsource).

Now, it is not clear if reduced cost is related to lower manpower costs or higher development productivity at organizations providing outsourcing services. Analyzing the ISBSG data [19], Carbonneau [103] confirmed the results of earlier studies (e.g., [104]) that outsourced projects do not have significantly different productivity (in terms of functionality delivered per effort invested) than in-source projects. Therefore, software development outsourcing will only lead to significant cost savings when the outsourcing provider has access to significantly cheaper labor. This is consistent with prior research [105], which concludes that "an external developer must have a considerable cost advantage over an internal developer in order to have the larger net value."

Nowadays, companies are more and more interested not only in lower labor cost but also in increased productivity of outsourcing projects. A study performed across 15 business units of a large international software organization performed by Fraunhofer IESE showed that 42% of the respondents plan to use productivity measurement to actually manage outsourcing projects. For that purpose, they need to identify factors that influence productivity in the context of outsourcing.

In case of outsourcing projects, communication between software provider and outsourcing organization seems to be a crucial aspect influencing development

productivity. As already mentioned in Section 6.2.1, the number of involved people (team size) as well as communication structure have a significant impact on development productivity. This is especially true for outsourced projects [46, 106], since outsourcing projects usually suffer from a geographical and mental distance between the involved parties. Software might be outsourced to an organization in the same country (near-shore) or abroad (far-shore). Due to geographical, temporal, and cultural distances, international outsourcing is found to be between 13 and 38% more expensive than national outsourcing [107]. In that context, communication facilities play an essential role. A summary of communication means in the context of offshoring projects can be found, for instance, in Moczadlo [108] (Fig. 12).

Team and task distribution is another project aspect related to communication between project stakeholders (see also Section 6.2.1) and thus should be considered in the context of software outsourcing. de Neve and Ebert [109] strongly advise building coherent and collocated teams of fully allocated engineers. *Coherence* means splitting the work during development according to feature content and assembling a team that can implement a set of related functionality. *Collocation* means that engineers working on such a set of coherent functionality should sit in the same building, perhaps within the same room. *Full allocation* implies that engineers working on a project should not be distracted by different tasks in other projects.

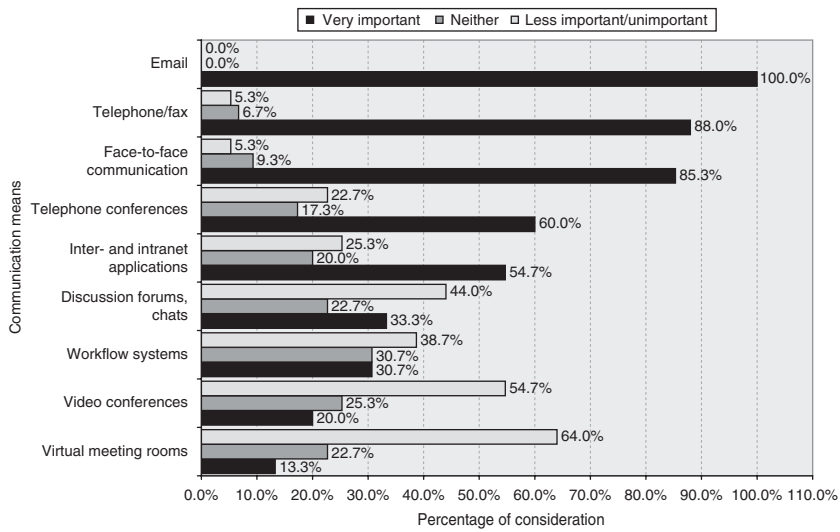

FIG. 12. Importance of communication channels [108].

Another already known factor that influences the productivity of outsourcing projects is *project management*, including realistic, detailed, and well-structured project planning [109, 110]. Those aspects cover factors such as schedule pressure and distribution (see Section 6.2.2), as well as task assignment (see Section 6.2.1).

Customer/contractor involvement, which is less important in case of in-house development (see results in Sections 4 and 5), becomes essential in outsourcing, where a software product is developed completely by a service provider [110]. It is recommended, for instance, that "at least one technical staff from the contracting organization should be involved in the details of the work, to form a core of technical expertise for in-house maintenance" [110].

7. Considering Productivity Factors in Practice

There are several essential issues that must be considered when selecting factors for the purpose of modeling software development productivity, effort/cost, schedule, etc.

This section presents a brief overview of the most important aspects to be considered in practice.

7.1 Factor Definition and Interpretation

Despite similar naming, the productivity factors presented in the literature usually differ significantly across various studies with respect to the phenomenon they actually represent. Moreover, our experience (e.g., [111]) is that, in practice, even if factor definitions suggest similar underlying phenomena, these may be interpreted differently by various experts—dependent on their background and experiences. Software reliability, for instance, is sometimes defined (or at least tacitly interpreted) as including safety and security and sometimes not.

Furthermore, a high-level view on abstract productivity factors that aggregate a number of specific indicators within a single factor may mask potential productivity problems, as poor results in one area (indicator) can be offset by excellence in another [73]. Therefore, it is recommended identifying project aspects influencing productivity on the level of granularity ensuring identification of potentially offsetting factors. Team capabilities, for instance, may cover numerous specific skills such as communication or management skills. It may thus be that low communication skills of developers are not visible because they are compensated by great management skill of the project manager. In such a case, considering team capabilities as a productivity indicator would not bring much value.

Therefore, to maximize the validity of the collected inputs (factor's significance, factor's value, and factor's impact on productivity), the exact definition of the factors as well as related measures has to be done in the first place. Blindly adopting published factors and assuming that everyone understands them may consistently (and usually does) lead to large disappointments. Unclear factor definition results in invalid data and inaccurate and instable models. In consequence, after investing significant effort, software organizations finally completely give up quantitative project management.

7.2 Factor Selection

One of the major purposes of the overview presented in this chapter is to support software practitioners in selecting factors relevant in their specific context. We are, however, far from suggesting uncritical adoption of the "top" factors presented here. This chapter is supposed to increase a software practitioner's understanding of possible sources of productivity drivers rather than bias his thinking while selecting factors relevant in his specific context. The overview presented here should, at most, be taken as a starting point and reference for selecting context-specific factors.

As with any decision support technology, the prerequisite when selecting productivity factors is that they maximize potential benefit while minimizing related costs. On the business level, they should then contribute to the achievement of organizational goals, such as productivity control and improvement, while generating minimal additional costs. On the software project level, the selected factors should cover (explain) a possibly large part of the observed productivity variance and be of minimal quantity to assure an acceptable cost of collecting, interpreting, and maintaining respective project data.

Therefore, an effective factor selection approach is an essential step in modeling productivity or any productivity-related phenomena.

Selecting an optimal[6] set of productivity factors is not a trivial task. In principle, we distinguish three major selection approaches: based on experts' assessments, based on data analysis, and a hybrid approach where both data- and expert-based methods are combined. Industrial experiences (e.g., [111]) indicate that none of the first two methods is able to provide us with an optimal set of factors.

Data-based factor selection techniques provide a subset of already measured factors that has usually been selected arbitrarily, for example, based on one of the popular cost estimation models [112, 113]. Data-based selection can usually be

[6] A minimal amount of factors that would meet specified cost- and benefit-related criteria, for example, effective productivity control at minimal modeling cost.

easily automated and thus does not cost much (in terms of manpower[7]). One significant limitation is that data-based selection simply reduces the set of factors given a priori as input. This means in practice that if input data does not cover certain relevant productivity factors, a data-based approach will not identify them. It may at most exclude irrelevant factors. Maximizing the probability of covering all relevant factors would require collecting a significant amount of data, hopefully covering all relevant factors, which would most probably disqualify such an approach due to high data collection costs.

On the other hand, *expert-based factor selection techniques* seem to be more robust, since experts are able to identify (based on their experience) factors unmeasured so far. However, experts tend to be very inconsistent in their assessments, depending, for example, on personal knowledge and expertise. Across 17 IESE studies where we asked experts to rank identified factors with respect to their impact on productivity and where we measured Kendall's coefficient of concordance $W \in (0, 1)$ [115] to quantify experts' agreement, in half of the cases (46%) experts disagreed significantly ($W \leq 0.3$ at a significance level $p = 0.05$).

Hybrid approaches to selecting productivity factors seem to be the best alternative. In the reviewed literature, however, merely 6% of the studies directly propose some kind of combined selection approach. Most of the published studies (45%) select productivity factors based on experts' opinion or already published factors (with COCOMO factors [43] coming out on top). A successful attempt at combining data- and expert-based approaches within an iterative framework has been made, for example, in [111].

7.3 Factor Dependencies

In practice, productivity factors are not independent of each another. Identification of reciprocal relationships between productivity factors is a crucial aspect of productivity modeling. On the one hand, explicit consideration of factors' dependencies provides software practitioners with a more comprehensive basis for decision making. On the other hand, the existence of relationships between productivity factors limits the applicability of certain modeling methods (e.g., multiple regression models) and thus must be known in advance, before applying certain modeling methods.

In practice, we may distinguish between several kinds of between-factor relationships. Factors may be in a causal relationship, which means that a factor's change (*cause*) leads to (*causes*) a change (*effect*) to a related factor. A factor may also

[7] Factor selection techniques are easy to automate; however, many of them contain NP-hard algorithms (e.g., [114]), which actually limits their practical applicability.

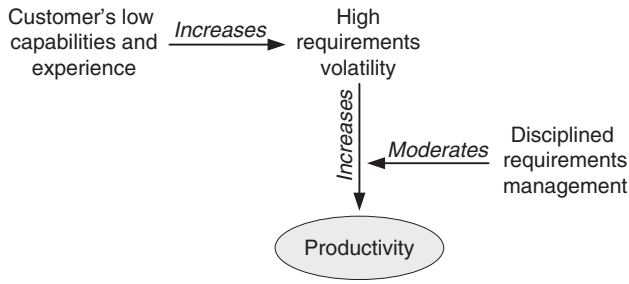

FIG. 13. Types of factor relationships.

be correlated, which means that changes to a factor go in parallel with changes to another factor. Although factors linked in a causal relationship should be correlated, correlation does not imply causal association.

Moreover, besides influencing another factor, a factor may also influence the strength of another factor's impact on productivity. For example (Fig. 13), *Customer's low capabilities and experiences* contributes to *Higher requirements volatility*, which in turn leads to higher development effort. Now, the negative impact of *Higher requirements volatility* can be moderated (decreased) by *Disciplined requirements management*.

There are several practical issues to be considered regarding factor dependencies. The first one is how to identify various kinds of dependencies? Data-based (analytical) methods are able at most to identify correlations. Cause–effect relationships can be identified by experts, who, however, tend to disagree in their subjective assessments. The next issue is what should we do with the identified dependencies? Should we explicitly model or better eliminate them? What alternative techniques exist to model or eliminate the identified relationships? Existing publications on productivity measurement/modeling do not explicitly pay much attention to those issues.

7.4 Model Quantification

Quantitative productivity management requires systematic collection of related project data. This entails quantifying selected productivity factors and their impact on productivity.

Proper definition of measures for each identified factor later on contributes to the cost and reliability of collecting the data upon which quantitative productivity management will take place. Noncontinuous factors (nominal, ordinal), especially,

require a clear and unambiguous definition of scales, reflecting distinct (orthogonal) classes. Wrong measurement processes and improper scales definition usually lead to unreliable, messy data, which significantly limits the applicability of certain analysis methods and the reliability of the obtained analysis results (see, e.g., [111]).

Finally, the impact each identified factor has on productivity and, if explicitly modeled, on other factors should be quantified. Some statistical approaches, for example, regression, provide certain weights that reflect each factor's impact on productivity. Data mining provides a set of techniques dedicated to weight a factor's impact (e.g., [114]). One major disadvantage of those methods is that the weights they provide are rather hard to interpret by experts. There are also several approaches based on expert assessments to quantify a factor's impact on productivity. The COBRA method [111], for instance, uses a percentage change of productivity caused by the worst-case factor value, which is very intuitive and easy to asses by experts. A less intuitive measure is used within Bayesian Belief Nets [116], where the conditional probability of a certain productivity value given the values of the influencing factors is assessed.

8. Summary and Conclusions

This chapter has presented a comprehensive overview of the literature and of experiences made within Fraunhofer IESE regarding the most common factors influencing software development productivity.

The major outcome of the study is that the success of software projects still relies upon humans.

The second most commonly considered factors are tool and method. However, even the best tool or method alone is not a silver bullet and cannot be a substitute for highly skilled people and effective work coordination. Investing in people is still considered as bringing more benefit than investing in tools and methods only [57]. Tools and methods should therefore be considered as human aid that amplifies the positive impact of highly skilled and well-coordinated teams on development productivity [60].

Some productivity factors refer to using or not using certain activities. Yet, a certain activity may be consistently applied across development projects and therefore, at first glance, not by considered as having an impact on development productivity. However, when we consider the effectiveness of a certain activity, it may occur that it still has a significant impact on productivity. This calls for considering software development methods, processes, and tools in terms of their effectiveness rather than simply using or not using them.

Moreover, any software development effort, even if staffed with skilled individuals, is likely to be unsuccessful if it does not explicitly account for how people work together [60]. A software development environment is a complex social system that may squander the positive impact of skillful individuals as well as software tools and methods if team communication and coordination fail [1].

Factors facilitating team communication and work coordination are particularly important in the context of software outsourcing. Geographical and, often, mental distance between the involved parties (e.g., outsourcing company, software provider, etc.) require dedicated managerial activities and communication facilities to maintain a satisfactory level of productivity.

The most commonly selected factors support the thesis that schedule is not a simple derivative of project effort. The negative impact of project schedule on productivity, however, is considered only in terms of schedule constraints (compression). Parkinson's low ("cost of the project will expand to consume all available resources") seems not to be considered in daily practice.

Several "top" factors support the common intuition regarding the requirements specification as the key development phase. First of all, requirements quality and volatility are considered to be essential drivers of development productivity. Several further factors are considered as either contributing to the quality and volatility of requirements or moderating the impact of already instable requirements on productivity. Distribution of the project effort (manpower) focusing on the requirements phase as well as significant customer involvement in the early phases of the development process are the factors most commonly believed to increase requirements quality and stability. The impact of already instable requirements may, on the other hand, be moderated by disciplined requirements management as well as early reviews and inspections.

Finally, the results obtained here do not support the traditional belief that software reuse is the key to productivity improvements. It seems that the first years of enthusiasm also brought much disappointment. A plethora of factors that should be considered to gain the expected benefits from reuse might explain this situation. Ad hoc reuse, without any reasonable cost-benefit analysis and proper investments to create a reuse environment (e.g., creation and maintenance of high-quality reusable assets, integration support, appropriate team motivation, and training) usually contributes to a loss in productivity.

The factors presented in this chapter result from a specific aggregation approach that reflects current industrial trends. However, it must be considered that the analyzed studies usually differ widely with respect to the identified factors, their interdependencies, and their impact on productivity. Therefore, each organization should consider potential productivity factors in its own environment ("what is good for them does not have to necessarily be good for me"), instead of uncritically

adopting factors used in other contexts (e.g., COCOMO factors) [51]. Moreover, since software is known as a very rapidly changing environment, selected factors should be reviewed and updated regularly.

Selecting the right factors is just a first step toward quantitative productivity management. The respective project data must be collected, analyzed, and interpreted from the perspective of the stated productivity objectives [117, 118]. Inconsistent measurements and/or inadequate analysis methods may, and usually do, lead to deceptive conclusions about productivity and its influencing factors [102]. In that sense, one may say that rigorous measurement processes and adequate analysis methods also have a significant impact on productivity, although not directly [52]. Therefore, such aspects as clear definition and quantification of selected factors, identification of factor interdependencies, as well as quantification of their impact on productivity has to be considered.

ACKNOWLEDGMENTS

We would like to thank Sonnhild Namingha from the Fraunhofer Institute for Experimental Software Engineering (IESE) for reviewing the first version of this chapter.

REFERENCES

[1] T.K. Abdel-Hamid, The slippery path to productivity improvement, IEEE Software 13 (4) (1996) 43–52.

[2] Gartner, Inc. press releases, Gartner Says Worldwide IT Services Revenue Grew 6.7 Percent in 2004, 8 February 2005 (http://www.gartner.com/press_releases/pr2005.html).

[3] M.C. Paulk, M.B. Chrissis, The 2001 High Maturity Workshop, Special Report, CMU/SEI-2001-SR-014, Carnegie Mellon Software Engineering Institute, Pittsburg, PA, 2002.

[4] National Bureau of Economic Research, Inc., Output, Input, and Productivity Measurement. Studies in Income and Wealth, vol. 25 by the Conference on Research in Income and Wealth, Technical Report, Princeton University Press, Princeton, NJ, 1961.

[5] IEEE Std 1045–1992, IEEE Standard for Software Productivity Metrics, IEEE Computer Society Press, Los Alamitos, CA, 1992.

[6] K.G. van der Pohl, S.R. Schach, A software metric for cost estimation and efficiency measurement in data processing system development, J. Syst. Software 3 (1983) pp. 187–191.

[7] N. Angkasaputra, F. Bella, J. Berger, S. Hartkopf, A. Schlichting, Zusammenfassung des 2. Workshops "Software-Produktivitätsmessungen" zum Thema Produktivitätsmessung und Wiederverwendung von Software [Summary of the 2nd Workshop "Software Productivity Measurement" on Productivity Measurement and Reuse of Software], IESE-Report Nr, 107.05/D, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, 2005 (in German).

[8] L.C. Briand, I. Wieczorek, Software resource estimation, in: Encyclopedia of Software Engineering, (J.J. Marciniak, Ed.), vol. 2. John Wiley & Sons, New York, NY, 2002, pp. 1160–1196.

[9] T. Meznies, Z. Chen, D. Port, J. Hihn, Simple software cost analysis: Safe or unsafe? in: Proc. International Workshop on Predictor Models in Software Engineering, St. Louis, MO, 15 May 2005.

[10] M. Jørgensen, M. Shepperd, A systematic review of software development cost estimation studies, IEEE Trans. Software Eng. 33 (1) (2007) 33–53.
[11] T. Noth, M. Kretzschmar, Estimation of Software Development Projects, Springer-Verlag, Berlin, 1984, (in German).
[12] F.J. Heemstra, M.J.I.M. van Genuchten, R.J. Kusters, Selection of Cost Estimation Packages, Research report EUT/BDK/36, Eindhoven University of Technology, Eindhoven, Netherlands, 1989.
[13] C. Jones, Software Assessments, Benchmarks, and Best Practices, Addison-Wesley Longman, Inc., New York, NY, 2000.
[14] B.W. Boehm, Software Engineering Economics, Prentice Hall PTR, Upper Saddle River, NJ, 1981.
[15] B.A. Kitchenham, N.R. Taylor, Software project development cost estimation, J. Syst. Software 5 (1985) 267–278.
[16] T.C. Jones, Estimating Software Cost, McGraw-Hill, New York, NY, 1998.
[17] D. Diaz, J. King, How CMM impacts quality, productivity, rework, and the bottom line, CrossTalk: J. Defense Software Eng. 15 (3) (2002) 9–14.
[18] D. Greves, B. Schreiber, K. Maxwell, L. Van Wassenhove, S. Dutta, The ESA initiative for software productivity benchmarking and effort estimation, Eur. Space Agency Bull. 87 (1996).
[19] ISBSG Data Repository. Release 9, International Software Benchmarking Group, Australia, 2005.
[20] Software Technology Transfer Finland (STTF). (http://www.sttf.fi/index.html).
[21] L.C. Briand, K. El Emam, F. Bomarius, COBRA: A hybrid method for software cost estimation, benchmarking and risk assessment, in: Proc. 20th International Conference on Software Engineering, April 1998, pp. 390–399.
[22] M. Ruhe, R. Jeffery, I. Wieczorek, Cost estimation for Web applications, in: Proc. 25th International Conference on Software Engineering, Portland, OR, 3–10 May 2003, pp. 285–294.
[23] C. Andersson, L. Karlsson, J. Nedstam, M. Höst, B. Nilsson, Understanding software processes through system dynamics simulation: A case study, in: Proc. 9th Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 8–11 April 2002, pp. 41–50.
[24] J. Heidrich, A. Trendowicz, J. Münch, A. Wickenkamp, Zusammenfassung des 1st International Workshop on Efficient Software Cost Estimation Approaches, WESoC'2006, IESE Report 053/06E, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, April 2006 (in German).
[25] B. Kitchenham, Procedures for Performing Systematic Reviews, Technical Report TR/SE-0401, Keele University, Keele, UK, 2004.
[26] K.D. Maxwell, L. Van Wassenhove, S. Dutta, Software development productivity of European space, military, and industrial applications, IEEE Trans. Software Eng. 22 (10) (1996) 706–718.
[27] J.M. Desharnais, Analyse statistique de la productivite des projects de development en informatique apartir de la technique des points des function, Master's Thesis, University of Montreal, Canada, 1989 (in French).
[28] C.F. Kemerer, An empirical validation of software cost estimation models, Commun. ACM 30 (1987) 416–429.
[29] M. Lattanzi, S. Henry, Software reuse using C++ classes. The question of inheritance, J. Syst. Software 41 (1998) 127–132.
[30] N.E. Fenton, S.L. Pfleeger, Software Metrics. A Rigorous and Practical Approach, second ed., International Thomson Computer Press, London, 1997.
[31] C. Jones, Software cost estimating methods for large projects, CrossTalk: J. Defense Software Eng. 18 (4) (2005) 8–12.
[32] C. Mair, M. Shepperd, M. Jorgensen, An analysis of data sets used to train and validate cost prediction systems, in: Proc. International Workshop on Predictor Models in Software Engineering, St. Louis, MO, 15 May 2005, pp. 1–6.

[33] K. Kennedy, C. Koelbel, R. Schreiber, Defining and measuring productivity of programming languages, Int. J. High Performance Comput. Appl. 11 (4) (2004) 441–448.
[34] E. Mendes, C. Lokan, R. Harrison, C. Triggs, A replicated comparison of cross-company and within-company effort estimation models using the ISBSG database, in: Proc. International Metrics Symposium, Como, Italy, 2005, pp. 36–46.
[35] S. Vijayakumar, Use of historical data in software cost estimation, Comput. Control Eng. J. 8 (3) (1997) 113–119.
[36] L.H. Putnam, W. Myers, Measures for Excellence: Reliable Software on Time, Within Budget, Yourdon Press, Upper Saddle River, NJ, 1992.
[37] The QSM Project Database, Quantitative Software Management, Inc., McLean, VA (http://www.qsm.com/database.html).
[38] T.L. Graves, A. Mockus, Inferring change effort from configuration management databases, in: Proc. 5th International Software Metrics Symposium, Bethesda, MD, 1998, pp. 267–273.
[39] V. Basili, L. Briand, S. Condon, Y.M. Kim, W.L. Melo, J.D. Valen, Understanding and predicting the process of software maintenance releases, in: Proc. 18th International Conference on Software Engineering, Berlin, Germany, 1996, pp. 464–474.
[40] F. Rico, Using Cost Benefit Analyses to Develop Software Process Improvement (SPI) Strategies, A DACS State-of-the-Art Report, ITT Industries Advanced Engineering & Sciences Division, New York, NY, 2000.
[41] CMMI Project Team, CMMISM for Software Engineering, Version 1.1, Staged Representation, Technical Report CMU/SEI-2002-TR-029, Carnegie Mellon Software Engineering Institute, Pittsburg, PA, 2002.
[42] L.H. Putnam, Linking the QSM Productivity Index with the SEI Maturity Level. Version 6, Quantitative Software Management, Inc., McLean, VA, 2000 (http://www.qsma.com/pdfs/LINKING6.pdf).
[43] B.W. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D. Refer, B. Steece, Software Cost Estimation with COCOMO II, Prentice-Hall PTR, Upper Saddle River, NJ, 2000.
[44] D.E. Harter, M.S. Krishnan, S.A. Slaughter, Effect of process maturity on quality, cycle time, and effort in software product development, Manage. Sci. 46 (4) (2000) 451–466.
[45] B.K. Clark, Quantifying the effects of process improvement on effort, IEEE Software 17 (6) (2000) 65–70.
[46] J.D. Herbsleb, A. Mockus, An empirical study of speed and communication in globally distributed software development, IEEE Trans. Software Eng. 29 (6) (2003) 481–494.
[47] B.W. Boehm, K.J. Sullivan, Software economics: A roadmap, in: Proc. International Conference on Software Engineering, Limerick, Ireland, 2000, pp. 319–343.
[48] J. Griffyth, Human factors in high integrity software development: A field study, in: Proc. 15th International Conference on Computer Safety, Reliability and Security, Vienna, Austria, 23–25 October 1997, Springer-Verlag, London, 1997.
[49] P. Tomaszewski, L. Lundberg, Software development productivity on a new platform: An industrial case study, Inform. Software Technol. 47 (4) (2005) 257–269.
[50] W.K. Vaneman, K. Trianfis, Planning for technology implementation: An SD(DEA) approach, in: Proc. Portland International Conference on Management of Engineering and Technology, Technology Management in the Knowledge Era, PICMET-Portland State University, Portland, OR, 2001.
[51] S.M. Ambler, Doomed from the start: What everyone but senior management seems to know, Cutter IT J. 17 (3) (2004) 29–33.

[52] S.B. Hai, K.S. Raman, Software engineering productivity measurement using function points: A case study, J. Inf. Technol. Cases Appl. 15 (1) (2000) 79–90.
[53] F. Niessink, H. van Vliet, Two case studies in measuring software maintenance effort, in: Proc. International Conference on Software Maintenance, Bethesda, MD, 16–20 November 1998, IEEE Computer Society Press, Los Alamitos, CA, 1998, pp. 76–85.
[54] G.H. Subramanian, G.E. Zarnich, An examination of some software development effort and productivity determinants in ICASE tool projects, J. Manage. Inform. Syst. 12 (4) (1996) 143–160.
[55] E. Carmel, B.J. Bird, Small is beautiful: A study of packaged software development teams, J. High Technol. Manage. Res. 8 (1) (1997) 129–148.
[56] F.P. Brooks, The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary ed., Addison-Wesley, Reading, MA, 1995.
[57] J.D. Blackburn, G.D. Scudder, L. Van Wassenhove, Concurrent software development, Commun. ACM 43 (4) (2000) 200–214.
[58] P.V. Norden, Curve fitting for a model of applied research and development scheduling, IBM J. Res. Dev. 3 (2) (1958) 232–248.
[59] M.S. Krishnan, The role of team factors in software cost and quality: An empirical analysis, Inform. Technol. People 11 (1) (1998) 20–35.
[60] S. Sawyer, P. Guinan, Software development: Processes and performance, IBM Syst. J. 34 (7) (1998) 552–569.
[61] R.K. Smith, J.E. Hale, A.S. Parrish, An empirical study using task assignment patterns to improve the accuracy of software effort estimation, IEEE Trans. Software Eng. 27 (3) (2001) 264–271.
[62] M. Cusumano, R. Selby, How Microsoft builds software, Commun. ACM 40 (6) (1997) 53–61.
[63] A. Parrish, R. Smith, D. Hale, J. Hale, A field study of developer pairs: Productivity impacts and implications, IEEE Software 21 (5) (2004) 76–79.
[64] B.G. Cain, J.O. Coplien, N.B. Harrison, Social patterns in productive software development organizations, Ann. Software Eng. 2 (1) (1996) 259–286.
[65] J.M. Carey, C.J. Kacmar, The impact of communication mode and task complexity on small group performance and member satisfaction, Comput. Hum. Behav. 13 (1) (1997) 23–49.
[66] A. Cockburn, Agile Software Development, Addison-Wesley Professional, Boston, MA, 2001.
[67] S.D. Teasley, L.A. Covi, M.S. Krishnan, J.S. Olson, Rapid software development through team collocation, IEEE Trans. Software Eng. 28 (7) (2002) 671–683.
[68] J. Collofello, D. Houston, I. Rus, A. Chauhan, D.M. Sycamore, D. Smith-Daniels, A system dynamics software process simulator for staffing policies decision support, in: Proc. 31st Annual Hawaii International Conference on System Sciences, vol. 6, Kohala Coast, HI, 6–9 January 1998, pp. 103–111.
[69] J. Hale, A. Parrish, B. Dixon, R.K. Smith, Enhancing the COCOMO estimation models, IEEE Software 17 (6) (2000) 45–49.
[70] I.R. Chiang, V.S. Mookerjee, Improving software team productivity, Commun. ACM 47 (5) (2004) 89–93.
[71] R. Bechtold, Reducing software project productivity risk, CrossTalk: J. Defense Software Eng. 13 (5) (2000) 19–22.
[72] E. Carmel, S. Sawyer, Packaged software teams: What makes them so special? Inform. Technol. People 11 (1) (1998) 6–17.
[73] Anonymous, Above average(s): Measuring application development performance, Intranet Networking Strategies Rep. 8 (3) (2000) 1–4.
[74] T.E. Potok, M.A. Vouk, The effects of the business model on object-oriented software development productivity, IBM Syst. J. 36 (1) (1997) 140–161.

[75] Y. Yang, Z. Chen, R. Valerdi, B.W. Boehm, Effect of schedule compression on project effort, in: Proc. 5th Joint International Conference & Educational Workshop, the 15th Annual Conference for the Society of Cost Estimating and Analysis and the 27th Annual Conference of the International Society of Parametric Analysts, Denver, CO, 14–17 June 2005.
[76] R. Park, The central equations of the PRICE software cost model, in: Proc. 4th COCOMO Users' Group Meeting, , Software Engineering Institute, Pittsburgh, PA, November 1988.
[77] R.W. Jensen, in: An improved macrolevel software development resource estimation modelAn improved macrolevel software development resource estimation model, in: Proc. 5th International Society of Parametric Analysts Conference, St. Louis, MO, 26–28 April 1983, pp. 88–92.
[78] C.N. Parkinson, Parkinson's Law and Other Studies in Administration, Houghton Mifflin Company, Boston, MA, 1957.
[79] M.A. Mahmood, K.J. Pettingell, A.I. Shaskevich, Measuring productivity of software projects: A data envelopment analysis approach, Decision Sci. 27 (1) (1996) 57–80.
[80] L.O. Damm, L. Lundberg, C. Wohlin, Faults-slip-through—A concept for measuring the efficiency of the test process, Software Process Improv. Practice 11 (1) (2006) 47–59.
[81] C.F. Kemerer, Software Project Management Readings and Cases, McGraw-Hill, Chicago, IL, 1997.
[82] R. Bazelmans, Productivity—The role of the tools group, ACM SIGSOFT Eng. Notes 10 (2) (1985) 63–75.
[83] P. Guinan, J. Cooprider, S. Sawyer, The effective use of automated application development tools, IBM Syst. J. 36 (1) (1997) 124–139.
[84] J. Baik, B.W. Boehm, B.M. Steece, Disaggregating and calibrating the CASE tool variable in COCOMO II, IEEE Trans. Software Eng. 28 (11) (2002) 1009–1022.
[85] C.D. Cruz, A proposal of an object oriented development cost model, in: Proc. European Software Measurement Conference, Technologisch Instituut VZW, Antwerp, Belgium, 1998, pp. 581–587.
[86] T. Bruckhaus, N.H. Madhavii, I. Janssen, J. Henshaw, The impact of tools on software productivity, IEEE Software 13 (5) (1996) 29–38.
[87] D.L. Atkins, T. Ball, T.L. Graves, A. Mockus, Using version control data to evaluate the impact of software tools: A case study of the Version Editor, IEEE Trans. Software Eng. 28 (7) (2002) 625–637.
[88] V. Basili, H.D. Rombach, The TAME project: Towards improvement-oriented software environments, IEEE Trans. Software Eng. 14 (6) (1988) 758–773.
[89] V. Basili, Viewing maintenance as reuse-oriented software development, IEEE Software 7 (1) (1990) 19–25.
[90] R.W. Selby, Enabling reuse-based software development of large-scale systems, IEEE Trans. Software Eng. 31 (6) (2005) 495–510.
[91] D.L. Nazareth, R.A. Rothenberger, Assessing the cost-effectiveness of software reuse: A model for planned reuse, J. Syst. Software 73 (2004) 245–255.
[92] D.C. Rine, R.M. Sonnemann, Investments in reusable software. A Study of software reuse investment success factors, J. Syst. Software 41 (1) (1998) 17–32.
[93] D.L. Atkins, A. Mockus, H.P. Siy, Measuring technology effects on software change cost, Bell Labs Tech. J. 5 (2) (2000) 7–18.
[94] W.B. Frakes, G. Succi, An industrial study of reuse, quality, and productivity, J. Syst. Software 57 (2001) 99–106.
[95] M. Morisio, D. Romano, C. Moiso, Framework based software development: Investigating the learning effect, in: Proc. 6th IEEE International Software Metrics Symposium, Boca Raton, FL, 4–6 November 1999, pp. 260–268.
[96] K.S. Soliman, Critical success factors in implementing software reuse: A managerial prospective, in: Proc. International on Information Resources Management Association Conference, Anchorage, AK, 21–24 May 2000, pp. 1174–1175.

[97] V.R. Basili, L.C. Briand, W.L. Melo, How reuse influences productivity in object-oriented systems, Commun. ACM 39 (10) (1996) 104–116.
[98] J.A. Lewis, S.M. Henry, D.G. Kafura, An empirical study of the object-oriented paradigm and software reuse, in: Proc. Conference on Object-Oriented Programming Systems, Languages and Applications, 1991, pp. 184–196.
[99] C.M. Abts, B.W. Boehm, COTS Software Integration Cost Modeling Study, University of Southern California Center for Software Engineering, Los Angeles, CA, 1997.
[100] A. Mockus, D.M. Weiss, P. Zhang, Understanding and predicting effort in software projects, in: Proc. 25th International Conference on Software Engineering, Portland, OR, 3–10 May 2003, IEEE Computer Society Press, Los Alamitos, CA, 2003, pp. 274–284.
[101] H. Siy, A. Mockus, Measuring domain engineering effects on software change cost, in: Proc. 6th International Symposium on Software Metrics, Boca Raton, FL, IEEE Computer Society Press, Los Alamitos, CA, 1999, pp. 304–311.
[102] P. Devanbu, S. Karstu, W. Melo, W. Thomas, Analytical and empirical evaluation of software reuse metrics, in: Proc. 18th International Conference on Software Engineering, 1996, p. 189.
[103] R. Carbonneau, Outsourced Software Development Productivity, Report MSCA 693T. John Molson School of Business, Concordia University, Montreal, Canada, 2004.
[104] M.J. Earl, The risks of outsourcing IT, Sloan Manage. Rev. 37 (3) (1996) 26–32.
[105] E.T.G. Wang, T. Barron, A. Seidmann, Contracting structures for custom software development: The impacts of informational rents and uncertainty on internal development and outsourcing, Manage. Sci. 43 (12) (1997) 1726–1744.
[106] J.D. Herbsleb, A. Mockus, T.A. Finholt, R.E. Grinter, An empirical study of global software development: Distance and speed, in: Proc. 23rd International Conference on Software Engineering, Toronto, Canada, 2001.
[107] M. Amberg, M. Wiener, Wirtschaftliche Aspekte des IT Offshoring [Economic Aspects of IT Offshoring], Arbeitspapier. 6, Universität Erlangen-Nürnberg, Germany, 2004 (in German).
[108] R. Moczadlo, Chancen und Risiken des Offshore Development. Empirische Analyse der Erfahrungen deutscher Unternehmen [Opportunities and Risks of Offshore Development. Empirical Analysis of Experiences Made by German Companies], FH Pforzheim, Pforzheim, Germany, 2002.
[109] P. de Neve, C. Ebert, Surviving global software development, IEEE Software 18 (2) (2001) 62–69.
[110] J. Herbsleb, D. Paulish, M. Bass, Global software development at siemens: Experience from nine projects, in: Proc. 27th International Conference on Software Engineering, St. Louis, MO, 2005.
[111] A. Trendowicz, J. Heidrich, J. Münch, Y. Ishigai, K. Yokoyama, N. Kikuchi, Development of a hybrid cost estimation model in an iterative manner, in: Proc. 28th International Conference on Software Engineering, Shanghai, China, 2006, pp. 331–340.
[112] Z. Chen, T. Menzies, D. Port, B. Boehm, Finding the right data for software cost modeling, IEEE Software 22 (6) (2005) 38–46.
[113] C. Kirsopp, M.J. Shepperd, J. Hart, Search heuristics, case-based reasoning and software project effort prediction, in: Proc. Genetic and Evolutionary Computation Conference, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2002, pp. 1367–1374.
[114] M. Auer, A. Trendowicz, B. Graser, E. Haunschmid, S. Biffl, Optimal project feature weights in analogy-based cost estimation: Improvement and limitations, IEEE Trans. Software Eng. 32 (2) (2006) 83–92.
[115] D. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures, third ed., Chapman & Hall/CRC, Boca Raton, FL, 2003.
[116] N. Fenton, W. Marsh, M. Neil, P. Cates, S. Forey, M. Tailor, Making resource decisions for software projects, in: Proc. 26th International Conference on Software Engineering, May 2004, pp. 397–406.

[117] V.R. Basili, Software Modeling and Measurement: The Goal Question Metric Paradigm, Computer Science Technical Report Series, CS-TR-2956 (UMIACS-TR-92-96), University of Maryland, College Park, MD, 1992.
[118] R. Basili, D.M. Weiss, A methodology for collecting valid software engineering data, IEEE Trans. Software Eng. SE-10 (6) (1984) 728–737.
[119] L.H. Putnam, W. Myers, Executive Briefing: Managing Software Development, IEEE Computer Society Press, Los Alamitos, CA, 1996.
[120] CHAOS Chronicles, The Standish Group International, Inc., West Yarmouth, MA, 2007.
[121] A.J. Albrecht, Measuring application development productivity, in: Proc. IBM Applications Development Symposium, Monterey, CA, 14–17 October 1979, pp. 83–92.
[122] Gartner, Inc., press releases, Gartner Survey of 1,300 CIOs Shows IT Budgets to Increase by 2.5 Percent in 2005, 14 January 2005 (http://www.gartner.com/press_releases/pr2005.html).
[123] D. Herron, D. Garmus, Identifying your organization's best practices, CrossTalk: J. Defense Software Eng. 18 (6) (2005) 22–25.
[124] N. Angkasaputra, F. Bella, S. Hartkopf, Software Productivity Measurement—Shared Experience from Software-Intensive System Engineering Organizations, IESE-Report No. 039.05/E Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, 2005.
[125] B.A. Kitchenham, E. Mendes, Software productivity measurement using multiple size measures, IEEE Trans. Software Eng. 30 (12) (2004) 1023–1035.
[126] K. Maxwell, L. Van Wassenhove, S. Dutta, Performance evaluation of general and company specific models in software development effort estimation, Manage. Sci. 45 (6) (1999) 787–803.
[127] J.P. Mclver, E.G. carmines, J.L. Sullivan, Unidimensional Scaling, Sage Publications, Beverly Hills, CA, 2004.

Evaluating the Modifiability of Software Architectural Designs

M. OMOLADE SALIU

Performance Management Practice, Online Business Systems, Calgary, Canada

GÜNTHER RUHE

Computer Science Department, University of Calgary, Calgary, Canada

MIKAEL LINDVALL

Fraunhofer Center for Experimental, Software Engineering, College Park, Maryland, USA

CHRISTOPHER ACKERMANN

Fraunhofer Center for Experimental, Software Engineering, College Park, Maryland, USA

Abstract

In this chapter, we propose an architectural design evaluation technique called EBEAM (Expert-Based Evaluation of Architecture for Modifiability) that assists experts in articulating their knowledge of architectural designs and expressing the knowledge in measurable terms. EBEAM supports the evaluation of different architectural design versions for modifiability. In addition, EBEAM supports relative comparison between these design versions and the target design. We develop EBEAM as a generalized technique that is reusable for evaluating other architectural design attributes, apart from modifiability. We discuss EBEAM in detail and report on two case studies that investigate its applicability, and one study that validates the results of the evaluations made using EBEAM.

ADVANCES IN COMPUTERS, VOL. 77
ISSN: 0065-2458/DOI: 10.1016/S0065-2458(09)01207-8

1. Introduction

In this chapter, we propose an architectural design evaluation technique called expert-based evaluation of architecture for modifiability (EBEAM) that assists experts in articulating their knowledge of architectural designs and expressing the knowledge in measurable terms. EBEAM supports the evaluation of different architectural design versions for modifiability. In addition, EBEAM supports relative comparison between these design versions and the target design. We develop EBEAM as a generalized technique that is reusable for evaluating other architectural design attributes, apart from modifiability. We discuss EBEAM in detail and report on two case studies that investigate its applicability. We also present another empirical study to validate the results of the evaluations made using EBEAM.

2. Evaluating Software Architectural Designs

According to Lehman's laws of software evolution [1], software systems must be continually adapted, or they become progressively less satisfactory to use in their environment. Therefore, we often need to extend the functionalities of an operational system by adding new features or removing defects discovered during usage of the software system [2]. The ease of adding features in the system depends on the modifiability of its architectural design. Bengtsson [3] defines modifiability as:

> Modifiability is the ease with which a system can be adapted to changes in the functional specification, in the environment, or in the requirements.

Improving the modifiability of an architectural design implies that future changes will take less time and will be less costly to implement. When compared to a less modifiable design, the architectural design with improved modifiability will allow

for smoother evolution. Thus, new upgraded versions of the system can be delivered earlier to the customers. Software that lacks modifiability is sometimes re-architected to increase the modifiability. Typically, re-architecting does not necessarily add user value, because it does not provide new functionalities. Since re-architecting could also be expensive, it is desirable to have a technique to assess whether the re-architected software achieves the desired improvement in modifiability. It is also important to assess the potential improvement in modifiability, in case there are several candidate architectures available that address the improvements in different ways. Information about the modifiability would also be useful for determining the amount of effort required and the risks involved in implementing new features. In the literature, there are several objective metrics that are defined on architectural design characteristics to correlate them with quality attributes (e.g., modifiability) [4].

For several reasons, using only architectural design metrics is not sufficient for measuring the improvement in modifiability. First, architectural design metrics have not been able to combine measures obtained for several architectural design characteristics, because of the problem of aggregating measures that are based on different scales. Second, several aspects of software architectures are difficult to measure, and can only be assessed by the experts that designed the architectures. For example, it is difficult for someone unfamiliar to the system to measure how well the names of the components convey their role in the system, but an expert could answer such a question. A major problem, however, pertains to how experts can be systematically involved in such evaluations. One option is to have a review and evaluation of the architectural design candidates by experts, but it is difficult and expensive to hire experts in the field for architectural reviews (especially for small projects). The current architects (and other people in the development team) understand the system and the various architectural design candidates, but are somewhat biased, especially if they contributed to the definitions of the various candidates.

Given the aforementioned difficulties, it is important to have a methodology that assists experts in articulating their knowledge of an architectural design and expressing this knowledge in measurable terms, and also in relation to another architectural version of the same system. We describe an evaluation technique named EBEAM that assists experts in articulating their knowledge of architectural design candidates in a systematic manner.

We focus on modifiability of architectural designs, because the architecture has a significant effect on the overall system costs. However, the same technique could be used for any other attribute desired of an architectural design. The case study described in this chapter examines the particular case when a redesign and reimplementation of a software architectural design have been conducted. Nevertheless,

EBEAM can also be used earlier in the lifecycle when changes have yet to be conducted. During the case study, we used EBEAM to evaluate the improvement of the new version of a system relative to the old version, and also in relation to the overall architectural goals (i.e., the target design).

The results obtained from EBEAM can be used to determine the level of improvement in the modifiability of the architectural design, and whether the improvement is close to the target architectural goals. This information together with information about the (actual or estimated) cost to achieve the improvement can be used to determine the cost for further improvement.

3. Overview of the EBEAM

EBEAM is a three-stage evaluation technique that is based on a systematic elicitation of the judgment from experts and transforming the results into quantitative measures. We adopt and adapt the analytic hierarchy process (AHP) [5] as part of our three-stage approach to help experts articulate their judgments. EBEAM allows the participation of multiple experts in the evaluation process. Li and Smidts [6] discuss the importance of using multiple experts in making judgments.

The activities involved in the three stages of EBEAM, shown in Fig. 1, are as follows:

Stage I: Evaluation of Design Characteristics

This stage of EBEAM requires experts that are knowledgeable in architectural design to identify the characteristics that influence the modifiability of architectural designs. Then, these experts evaluate and rank the relative importance of these characteristics to modifiability. Using aggregation schemes established in the literature and discussed by Forman and Peniwati [7] and Clemen and Winkler [8], we combine the judgments made by all the different experts into a unified measure. Any expert that has knowledge and experience in architectural designs could participate in this phase. Thus, the experts do not necessarily have to be familiar with the candidate architectural designs. This model can be improved over time by having more experts add their knowledge and experience to it.

Stage II: Evaluation of Architectural Designs

In Stage II, the experts with knowledge about the candidate architectural designs (we refer to them as the local experts) individually compare each candidate in relation to the other candidates or the specific architectural goals or both. This comparison is performed based on how the candidate architectural designs handle each of the characteristics evaluated in the

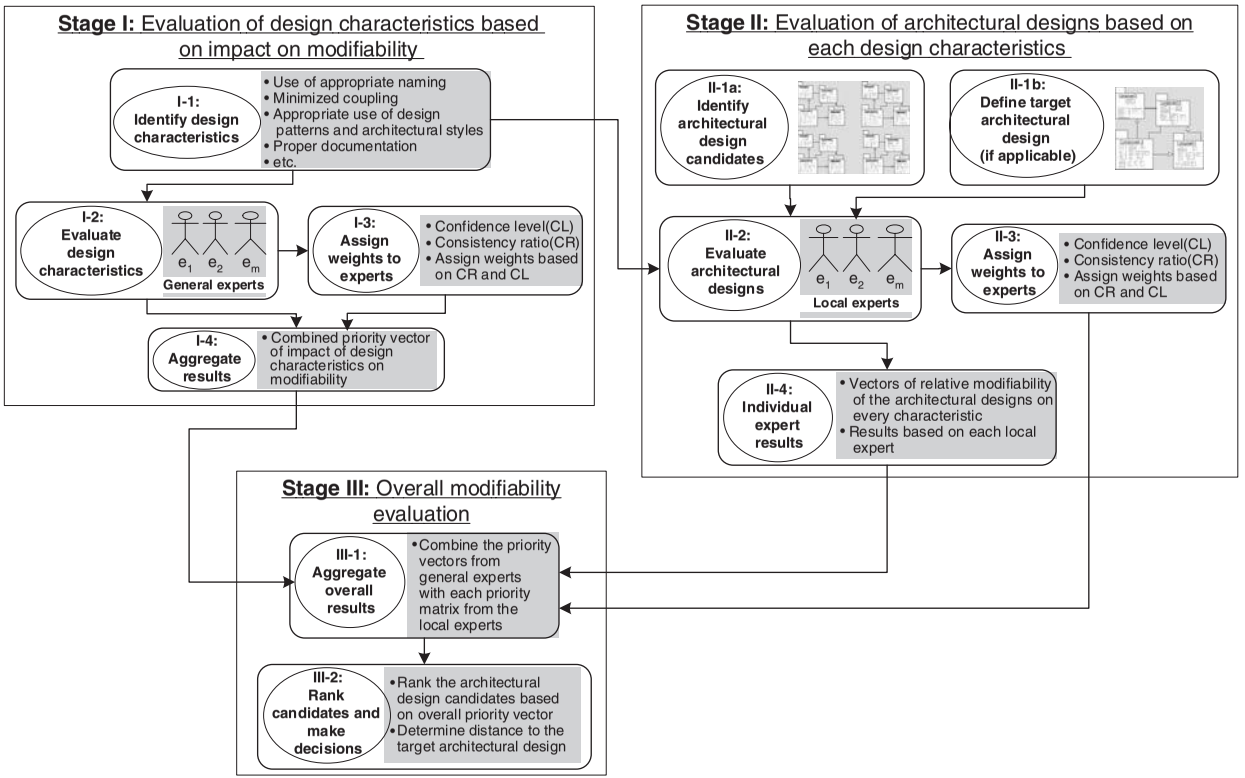

FIG. 1. Overview of the EBEAM framework.

first stage. By using the knowledge and experience of the local experts, we can evaluate important characteristics that are difficult or even impossible to measure objectively. This is further simplified because we use relative measures instead of absolute measures.

Stage III: Overall Architectural Modifiability Evaluation

In Stage III, we consolidate the results from the general experts in Stage I with the judgments from the local experts in Stage II to produce the relative modifiability of the architectural designs. The consolidation approach we adopt aims to reduce the bias from local experts.

It should be noted that there can be overlap between the general experts and the local experts, because some experts could posses both general architectural design knowledge as well as the knowledge about the specific architectural design candidates.

We have published an earlier version of this evaluation technique in [9]. EBEAM builds on the idea in this previous work, but focuses on software architectures and uses a more fine-grained set of architectural design characteristics. The previous work also involves a straightforward application of the AHP. On the other hand, EBEAM defines a new three-stage process that adapts the AHP, especially with regards to the aggregation of the judgments from multiple experts. The EBEAM also introduces a new and more robust weighting scheme that considers the consistency in the judgments made by each expert and their familiarity/confidence in making the judgments.

4. STAGE I: Evaluation of Design Characteristics

This stage is concerned with the identification of the design characteristics that contribute to modifiability and also the evaluation of their relative contributions.

4.1 Identifying Design Characteristics (I-1)

Every major post-implementation activity (e.g., implementing change requests) requires that the technical development team first develops an understanding of the existing architectural design. An architectural design that adheres to tested design principles and practices would facilitate this understanding to a higher degree than would an architectural design that does not follow such principles. These design principles (e.g., minimized coupling), when adhered to, can result in an architectural design that facilitates understandability. We focused on adherence to design principles as a way to measure the goodness of an architectural design. Because

irrespective of the domain of a software system, adherence to characteristics of good and tested design principles and practices may result in a more modifiable system.

The first task is to identify the desired characteristics of software architectural designs that could be considered good practices. There are several software architectural design ideas that are considered to be good practices [10]. There are also discussion forums (e.g., Hacknot [11]) where software architectural design experts discuss the characteristics that a good software architectural design should possess.

There is no doubt that there is a long list of architectural design characteristics to explore [4]. In the process of coming up with the characteristics of a good architectural design, we explored the vast literature on software architectural design, extracted the generally accepted architectural design practices and refined the resulting set. To achieve this, we organized two workshops involving participants drawn from experienced architectural designers and researchers at the Fraunhofer Center in Maryland (FC-MD). The first workshop resulted in an initial list that we later used to conduct a pilot study on architectural evaluation. During the second workshop, we discussed the results of the pilot study, identified the coverage and applicability of the initial list of characteristics, and refined the characteristics. The final list of characteristics is based on experiences of the software developers, existing software architectural design literature, and experiences of the researchers involved in the study. Some of the characteristics that made the initial list, but were later excluded, are size of the system, size of files, size of folders making up the system, and use of less-complex algorithms. A discussion of their exclusion appears in Section 4.2. The chosen design characteristics that influence modifiability include the following.

4.1.1 Use of Appropriate and Representative Naming

The names of the components, classes, methods, parameters, attributes, and all other constructs of a system should closely correspond to the functionality and the roles they represent in the system. Appropriate naming allows programmers to quickly understand the structure of a software system, identify components and their responsibilities, and relationships of a component with other components. The importance of using representative naming is discussed in [10].

4.1.2 Minimized Coupling

Coupling is a measure of the interdependence between entities in a system, for example, components, classes, modules, and so on. It measures the number of connections from and to a component [12]. The quality of a component increases as

component coupling decreases [13]. Tight coupling can magnify the implementation effort of new features since the dependencies to the rest of the program can lead to ripple effects [14]. Tight coupling also makes it difficult to comprehend the role of a component.

A special case of coupling is the coupling to Libraries. It is important to encourage reuse of libraries to prevent the re-implementation of functionality that the components in the library provide. However, coupling from libraries and other common components and functions to application-specific components must be avoided.

4.1.3 Minimized Coupling to COTS (Including Languages)

Another special case of coupling is the coupling to COTS components and coupling to programming languages and vendor unique libraries. While it is desirable to reuse such components, it is often important that the COTS components are wrapped so that the coupling is to the wrapper rather than to the COTS component itself. High direct coupling to COTS components can create problems with modifiability, since replacing the COTS component increases with increased coupling. This characteristic, for example, is motivated by a recent project at FC-MD in which coupling to COTS was analyzed [15].

4.1.4 Maximized Cohesion

Cohesion is the strength or logical unit of software components [16]. Entities that provide a certain function should be bundled together to facilitate programmers' identification of components (classes and packages) that are related to each other. Even if we do not desire coupling, it is a known fact that we cannot live without coupling. It is better to have couplings inside components than couplings between components. Couplings inside components could indicate the coming together of related classes, which inherently translates to high cohesion.

Even though there are definitions for how to measure cohesion, it requires an expert to judge whether the right components are grouped together, especially from the perspective of functionality.

4.1.5 Appropriate Use of Design Patterns and Architectural Styles

Design patterns represent frequently used ways to combine classes or associate objects to achieve a certain purpose [17]. Most software developers agree on the benefits of design patterns and try to build software based on them to provide an

easily modifiable software system. Nevertheless, incorrect implementation of design patterns leads to confusion, because developers tend to make wrong assumptions that the existence of a design pattern means the pattern has been implemented correctly. If design patterns are not implemented correctly, the software architecture may be difficult to understand and it may not readily accommodate future changes. For example, empirical studies in Hochstein and Lindvall [12] indicate that the goal of developing a modifiable system based on design patterns is seldom reached because of incorrect implementation.

One of the questions requiring difficult decisions when building the architectural design of a system is the selection and use of appropriate architectural style(s) [18]. Architectural styles constrain the roles of architectural elements (i.e., components, connectors, and data) that may be used to compose a system or subsystem, and also the pattern of relationships among the elements [19]. Examples of architectural styles include pipe and filter model, layered model, and client–server. Decisions on architectural styles choices determine the suitability of the architecture in addressing the problem and affect the desired functionality and performance [18].

The need to choose appropriate style and the correct implementation thereof is of high interest from modifiability perspective; using appropriate architectural styles helps to understand the roles of the components and how they interact.

Appropriate usage of architectural styles and design patterns is difficult to measure, but an expert who knows about the roles of the system components and how they interact can determine whether suitable patterns are used.

4.1.6 Proper Use of Information Hiding (Including Interfaces)

One major criterion suggested for decomposing systems into modules is the information hiding principle [20]. In making design decisions, this principle requires that, "system details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change [21]." In essence, it requires hiding design decisions to protect other parts of a software architectural design from changing, if the design decisions change. Proper use of information hiding ensures that changes are easier to perform because the changes are typically local rather than global. Applying this principle perfectly is not always easy because of the difficulty of estimating the likelihood of change [21].

The use of interfaces also facilitates reuse because in order to use the component only the interface class needs to be understood. In addition, it is a good design practice to maintain narrow interfaces. We define narrow interfaces by three criteria:

(1) low number of available methods, (2) low number of parameters, and (3) use of simple parameters. Even though some rules of thumbs, such as the ones above, do exist, it is difficult to measure these characteristics. Notwithstanding, an expert can determine not only whether the size, but also whether the structure of the interface is appropriate.

4.1.7 Maximized Modularity

Modularization is one of the techniques proposed for the structural decomposition of software systems. The goal of modularization is to allow modules to be designed and revised independently [21], thereby improving the ease of change. A proper modular structure facilitates change, because it isolates some changes to a small part of a software system [20]. Measuring modularity is difficult; at the same time an expert with knowledge about the software system can recognize whether the system is decomposed into appropriate modules.

4.1.8 Minimized Duplication and Redundancy

Each design problem is expected to be solved once, instead of providing different solution instances for the same problem. Such duplication may cause unnecessary overhead of understanding two or more different solutions [11]. There are many reasons why duplicated functionality decreases modifiability. One reason is that it takes less effort to make a change in one place than in many places. Another reason is that it is easy to forget changing many different but similar design decisions. In addition, when there are many instances of the same entity, it is confusing and difficult to understand why they all exist and which one to use. It is also difficult to maintain consistency between the different solutions [11]. Several duplications could exist in an architectural design, including duplication of algorithms.

4.1.9 Minimized Concurrency and Threads

A system that only has one thread is easier to understand than a system with many threads and concurrent processes. Implementing concurrency and threads makes the design difficult to understand, because there are a lot of codes that distract from the implementation of the functionalities. We may not be able to avoid this kind of implementation mechanisms for some problems, but it constitutes a good design principle to reduce their usage as much as possible [11]. Concurrency is not bad, but the constructs for implementing concurrency could make it difficult to understand the design.

4.1.10 Proper Documentation

Documentation plays an important role for modifiability because it facilitates understanding of the architectural design. According to Clements and colleagues [22], an architecture must be documented in order for the architecture to achieve its effectiveness. Documentation provides explanations about the design in a way that enables designers new to the design or code base of the system to get a good overview quickly. Documentation provides clarifications on design decisions already made, which could help during modification tasks, thereby increasing the ease of modification (EoM) by enhancing understandability.

4.2 Comments on the Excluded Characteristics

A number of software design characteristics were not included in this list for various reasons. We did not include size as an attribute of modifiability because size can be seen as an effect of other characteristics. Modifiability can be increased by adding comments, by using longer and more expressive names, and by dividing complex algorithms into smaller pieces, but all these could also increase the size. On the other hand, getting rid of duplications often means decreased size. Thus, an increase in size may not necessarily mean that the code resulting from the design would be more difficult to modify.

We have also not included number of files and number of folders, because redesigning a system to increase modifiability often means adding more structures, classes, files, and so on. Since our interest is to compare two or more architectural design versions of the same system, we believe size, number of files and folders may not really change much, and even if they change it will not have big relative impact on the modifiability of the system.

By comparing two versions of a system, we also avoid other context-oriented characteristics such as nonreal-time versus real-time systems and nonembedded versus embedded systems. These factors definitely have huge impact on modifiability, but they are irrelevant when comparing two version of the same system. By comparing two versions of the same system, we also avoid discussions regarding essential and accidental complexity [23]. The essential complexity relates to the complexity of the problem to be solved, while accidental complexity relates to the complexity of the solution. Accidental complexity refers to the extra complexity in the solution to a design problem, which results from the specific design approach adopted in solving a problem. A good design is expected to minimize accidental complexity. Thus, focusing our characteristics selection on good design principles subsumes minimization of accidental complexity. On the other hand, the unavoidable

software complexity resulting from the nature of the problem being solved is an essential complexity [23].

Thus, it is safe to conclude that the set of characteristics we have selected and refined would apply to any type of system, as long as we are evaluating different design versions of the same system.

4.3 Evaluating Design Characteristics (I-2)

4.3.1 Definition of Goal and Alternatives

To evaluate the contribution of design characteristics to modifiability, EBEAM first structures the problem in a hierarchical form as shown in Fig. 2. At the top is the goal, that is, the relative contribution of the design characteristics to modifiability. In the middle layer are the architectural design characteristics that contribute to the goal (in AHP, they are also known as the criteria). The experts (i.e., $e_1, e_2, \ldots, e_m$) that would evaluate the relative contribution of each characteristic to the goal are represented in the third layer. We adopt this hierarchical structuring from the AHP, but adapted the evaluation process using a new aggregation and weighting scheme for the participating experts.

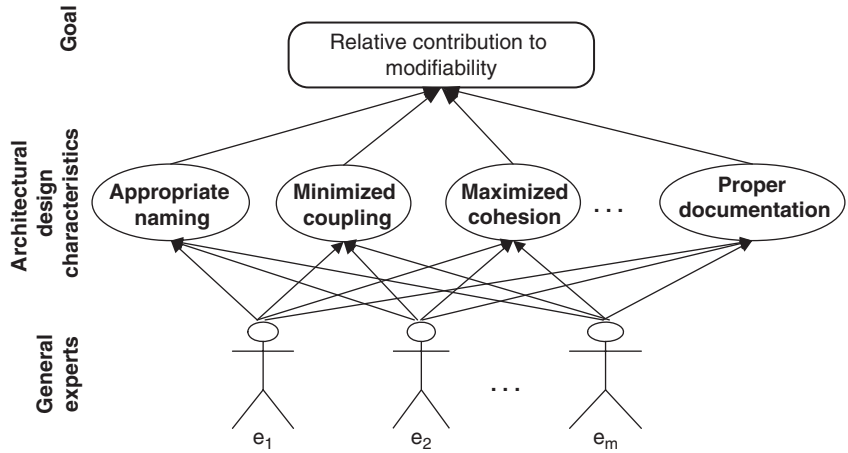

FIG. 2. Evaluation of architectural design characteristics.

4.3.2 Prioritization of Design Characteristics

Each expert, selected according to their general knowledge and experience in architectural designs, performs pairwise comparison between the design characteristics. Pairwise comparison is carried out to determine which of the characteristics contribute more to modifiability than another. The weighting scale used for the purpose of this pairwise comparison is shown in Table I. Since we have 10 characteristics, we need a 10×10 matrix that contains the list of these characteristics in the rows and columns. This matrix enables us to relatively compare each characteristic from the row to all the other characteristics in the column. For each pair of design characteristics (starting with "appropriate and representative naming" and "minimized coupling," e.g.), their relative contribution to modifiability is described by the "intensity of contribution." Using the comparison scale in Table I (adapted from AHP [5]), the expert inserts into the comparison matrix, the number representing his chosen importance intensity in the cell that appears in the intersection of the two properties being compared. The question the expert is asking herself or himself while comparing each pair of characteristics to get the intensity of importance value is "How does characteristic-*i* (row) contribute to the modifiability of software designs when compared to characteristic-*j* (column)?"

On completing the pairwise comparison, the aggregated eigenvalues computation establishes a *priority vector*, $V_e(z)$ $(1 \leq z \leq Z)$, $(0 \leq V_e(z) \leq 1)$, which represents

TABLE I
SCALE FOR PAIRWISE COMPARISON OF ARCHITECTURAL DESIGN CHARACTERISTICS

Intensity of contribution	Definition	Explanation
1	Equal contribution	The two characteristics (*i* and *j*) are of equal importance
3	Moderate contribution	Experience slightly favor one characteristic over another
5	Strong contribution	Experience strongly favors one characteristic over another
7	Very strong contribution	A characteristic is strongly favored and its dominance demonstrated in practice
9	Extreme contribution	The evidence favoring one over another is of highest possible order of affirmation
2, 4, 6, 8	Intermediate values between two adjacent judgments	When compromise is needed
Reciprocals	If characteristic *i* has one of the above numbers assigned to it when compared with characteristic *j*, then *j* has the reciprocal value when compared with *i*	

the relative contribution of all the Z characteristics from the perspective of each expert e. It is guaranteed that the entries in the vector $V_e(z)$ satisfy $\sum_{z=1,\ldots,Z} V_e(z) = 1$.

4.4 Assigning Weights to Experts (I-3)

Having different knowledge and experience in architectural design research and practice, the general experts would be expected to exhibit different confidence levels (CL_e) in making judgments on the characteristics. Each expert is asked to rate his confidence level (CL_e) in making the judgments on a scale of 1 (lowest) to 9 (highest). Their confidence level (CL_e) shows the extent to which their judgments can be trusted. But given that a highly confident expert might not be necessarily consistent when making the comparison judgments, we also need to consider the consistency of the comparisons that an expert makes. The consistency ratio (CR_e) for an expert e defines the accuracy of the pairwise comparisons carried out by the expert. The lower the CR_e of expert e, the higher the accuracy of the judgment s(he) made. A consistency ratio of 0.10 or less is considered acceptable [5].

We derive the weight of the experts by integrating both CL_e and CR_e. The consequence of this premise is that the more confident an expert is about his ability to make judgment on the characteristics, and the more consistent he performs the relative comparison tasks, the more the influence he wields in determining the overall contribution of each characteristic to modifiability. We employ the consistency formulas discussed in Saaty [5] to compute the consistency ratio from the comparison matrix prepared by each expert. Because a lower CR_e is desirable, we use $1 - \mathrm{CR}_e$ as a multiplicative factor of the confidence level, so that we do not penalize a highly consistent expert. To consider the impact an expert has on the final priority vector, we compute the weight W_e $(1 \leq e \leq E)$ for each expert using the formula:

$$W_e = \frac{\mathrm{CL}_e(1 - \mathrm{CR}_e)}{\sum_{e=1,\ldots,E} \mathrm{CL}_e(1 - \mathrm{CR}_e)}, \tag{1}$$

where $0 \leq W_e \leq 1$. The weights of all the experts satisfy $\sum_{e\,=\,1,\ldots,E} W_e = 1$.

4.5 Aggregating the Contribution of Design Characteristics to Modifiability (I-4)

In this step, we aggregate the different priority vectors resulting from the evaluation of all the experts. This results in a single overall vector of relative contribution of each characteristic from the perspectives of all the experts. During

this aggregation procedure, the weight of each expert is a multiplicative weighting factor of the vectors of relative contribution of the characteristics, as given in Equation 2.

$$V(z) = \sum_{e=1,\ldots,E} V_e(z) W_e, \quad 1 \leq z \leq Z \tag{2}$$

This type of aggregation scheme is referred to as the weighted arithmetic mean of priorities (AIP) [7]. There are evidences in the literature to support the fact that weighted linear combination of the judgment of experts, such as the one in Equation 2, performs better than other more mathematically complex aggregation methods [8]. Besides, they are more easily understood.

5. STAGE II: Evaluation of Architectural Designs

In this stage of EBEAM, we evaluate the relative modifiability of the architectural design candidates, and their relative modifiability with respect to the target design.

5.1 Identify Architectural Design Candidates and Define the Target Design (II-1)

The first task here is to identify the architectural design candidates, which are different design versions of the same system. If applicable, the software architects also define a specification of the target architectural design. Lindvall et al. [24] discuss the idea of specifying a target (or ideal) design for an existing system architecture. For example, existing versions of the design for Client–Server architecture may currently allow direct communication between the clients and the server. In an ideal case, the target design specification could be defined such that no Client component except the Mediator should be allowed to contact the Server directly. Such a target design may define some goals that are yet to be implemented in the existing design versions, probably due to lack of resources.

5.2 Evaluation of Design Candidates for a Fixed Characteristic (II-2)

Each local expert with knowledge of the architectural candidates performs relative pairwise comparison of the architectural design candidates and the target design with respect to each of the design characteristics. (In AHP, the architectural design candidates and the target design would be known as the alternatives

to compare.) The goal is to evaluate the relative EoM of any pair of candidates being compared based on the manner the architectural design candidates handle the characteristic under consideration. We structure the problem (similar to the one for the design characteristics) as shown in Fig. 3. The question to ask when comparing two designs, based on the characteristic under consideration, is of the following form: "How much easier to modify would design-*i* be when compared to design-*j*, if we only consider their use of *minimized coupling*?" The same question applies to all the other characteristics that contribute to modifiability. The rating scale used is given in Table II.

Each local expert develops one comparison matrix for each characteristic on which the candidates are compared. It should be noted that the target design (if applicable) also constitutes one of the candidates, for comparison purposes. For each of these matrices developed we also compute the consistency ratio for the purpose of assigning weights during aggregation. During this step, we also compute the priority vector for each comparison matrix developed. The priority vector shows the relative modifiability of the candidate designs based on each characteristic, and from the perspective of each expert.

Let us suppose we have N architectural design candidates (including the target design) to compare. Combining all the priority vectors for one expert would result in a $Z \times N$ matrix that shows, for every entry (z, n), the relative modifiability of each design in column n with respect to the attribute in row z. Higher value in a cell of the matrix translates to higher modifiability of the design in the column relative to other

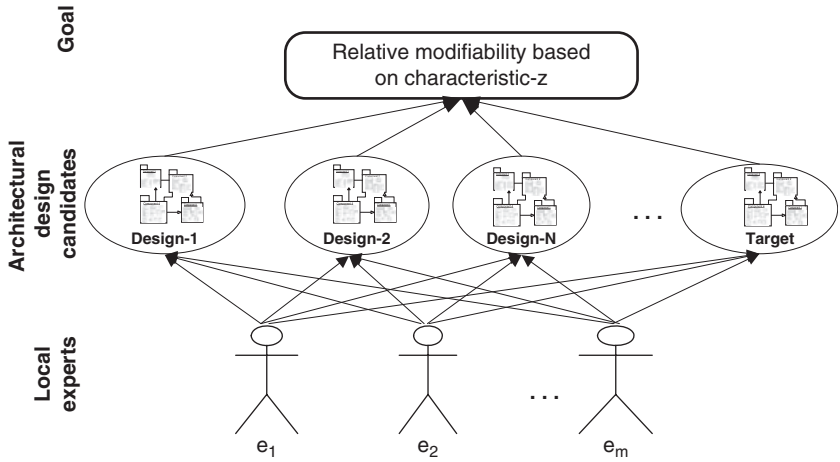

FIG. 3. Evaluation of architectural designs.

TABLE II
SCALE FOR PAIRWISE COMPARISON OF ARCHITECTURAL DESIGNS

Intensity of contribution	Definition	Explanation
1	Equal contribution	The two architectural designs (i and j) are of equal importance
3	Moderate contribution	Experience slightly favor one architectural design over another
5	Strong contribution	Experience strongly favors one architectural design over another
7	Very strong contribution	An architectural design is strongly favored and its dominance is demonstrated in practice
9	Extreme contribution	The evidence favoring one architectural design over another is of highest possible order of affirmation
2, 4, 6, 8	Intermediate values between two adjacent judgments	When compromise is needed
Reciprocals	If architectural design i has one of the above numbers assigned to it when compared with architectural design j, then j has the reciprocal value when compared with i	

design candidates, and consequently the easier it is to modify the design. The matrix by each expert e is described by:

$$\phi_e(z,n), \quad 1 \leq e \leq E, \quad 1 \leq n \leq N, \quad 1 \leq z \leq \mathrm{Z}, \tag{3}$$

where each row z (representing one characteristic) in the priority matrix $\phi_e(z,n)$ for each expert e must satisfy the inequality $\sum_{n=1,\ldots,N}\phi_e(z,n) = 1$.

5.3 Computing the Weight for Local Experts (II-3)

For the purpose of aggregating the judgment by all the experts, weights must be assigned to the experts to reflect the influence that each expert wields on the final matrix. The process of weight assignment is similar to the one described by Equation 1. One difference here is that we replace confidence level in Equation 1 with a measure of domain familiarity (DF). Also, the DF is assessed on per-design basis for each expert e (i.e., $\mathrm{DF}_{e,n}$)—an expert maybe more familiar with one architectural design candidate than the other. Thus, he could specify different familiarity level for different design candidates on a scale of 1 (lowest) to 9 (highest). Also, from the previous step, it is clear that the CR is also derived on per-characteristic basis for each expert (i.e., $\mathrm{CR}_{e,z}$), because each expert develops one comparison matrix for each characteristic. Thus, a CR must be computed for each comparison matrix. As a result, the weight $W_{e,n}$ for each expert e on design n is computed as:

$$W_{e,n} = \frac{\mathrm{DF}_{e,n}\left(1 - \mathrm{CR}_{e,z}\right)}{\sum_{e=1,\ldots,E}\mathrm{DF}_{e,n}\left(1 - \mathrm{CR}_{e,z}\right)}, \tag{4}$$

where$1 \leq n \leq N, 1 \leq z \leq Z$. Equation 4 shows that we derive the relative weight of each expert for specific characteristic and on specific design candidate during aggregation.

6. STAGE III: Overall Modifiability Evaluation

During this stage of EBEAM, we consolidate the results from the local experts with the judgment from the general experts (i.e., results from Stage I) to produce the relative modifiability of the architectural design candidates, and their closeness to the target design.

6.1 Combining the Design Modifiability Matrices of All Experts (III-1)

Using the priority matrix (i.e., described by Equation 3) developed by each expert, we have to combine all their priority matrices into a single aggregated matrix, in order to establish the overall relative priorities of the design candidates from the perspectives of all the experts. Unlike the aggregation carried out in Stage I (see Equation 2), the aggregation required here is a little tricky. First, because we have to combine design modifiability matrices and not vectors. Second, because we have to consider the weighting vector $W_{e,n}$ defined in Equation 4, and also augment it with the priority vector $V(z)$ of the characteristics. The combined matrix is described by:

$$\theta(z,n) = \sum\nolimits_{e} V(z)\phi_e(n,z)W_{e,n}. \tag{5}$$

Since we are combining matrices and vectors, we perform a cell by cell computation. This explains the weighting scheme described by Equation 4. On expanding Equation 5 further, the resulting priority matrix $\theta(z, n)$ is given by:

$$\theta(z,n) = V(z)\left[\frac{\sum_{e=1,\ldots,E}\phi_e(n,z)\mathrm{DF}_{e,n}\left(1 - \mathrm{CR}_{e,z}\right)}{\sum_{e=1,\ldots,E}\mathrm{DF}_{e,n}\left(1 - \mathrm{CR}_{e,z}\right)}\right], \tag{6}$$

where $1 \leq n \leq N$, $1 \leq z \leq Z$.

Now, we have a $Z \times N$ priority matrix that aggregates the judgment of all the experts. From this priority matrix, we can determine the overall relative modifiability of each design candidate in terms of each characteristic that contributes to

modifiability. The information derived from here is useful in helping us determine which subset of characteristics is actually contributing to the low modifiability of a design. Such information could assist architects when making refactoring or redesign decisions.

6.2 Rank Architectural Designs (III-2)

6.2.1 Rank the Designs Based on Modifiability Values

We derive a priority vector from the aggregated priority matrix $\theta(z, n)$ developed in the previous step by summing over each column (i.e., each column represent a design candidate) of the matrix. The higher the priority value assigned to a design, the easier it is to modify the design relative to the other candidates, and consequently the higher the modifiability of the design. Thus, we define the EoM based on these modifiability values, as follows:

$$\mathrm{EoM}(n) = \sum\nolimits_{z=1,\ldots,Z} \theta(z, n), \tag{7}$$

where $\theta(z, n)$ is the $Z \times N$ matrix described by Equation 6. Each entry in the modifiability vector is normalized, where $0 \leq \mathrm{EoM}(n) \leq 1$ and $\sum_n \mathrm{EoM}(n) = 1$.

While $\theta(z, n)$ gives the overall modifiability of each design with respect to each of the characteristics, the $\mathrm{EoM}(n)$ gives the overall modifiability over all the characteristics. All the design candidates can then be ranked based on their modifiability values.

6.2.2 Determine the Distance from the Target Design

To measure the distance from the target, we *idealize* the modifiability values in the priority matrix $\theta(z, n)$ and also in the priority vector $\mathrm{EoM}(n)$. By idealizing, we mean selecting the design with the largest modifiability value (i.e., definitely the target design) and dividing all the other values in the priority matrix (respectively, the priority vector) by this largest value. Thus, the target design now has a modifiability value 1, then the modifiability value for every other design candidate would be proportionately less than 1. These idealized priorities can help in road mapping, as we would be able to measure how far we are from the goal specified for the target design. For instance, if the target design has modifiability value of 1 on a specific characteristic, and the best design version has a modifiability value of 0.75 on the same characteristic, it implies that the best of the design versions has achieved 75% of the goal defined for that design characteristic.

However, if there is no target architectural design specified for the system, we simply exclude target design from consideration in the EBEAM evaluation process. Thus, the modifiability evaluation is conducted relative to the available design versions.

7. CASE STUDY I: The Application of EBEAM to TSAFE Designs

The application of EBEAM to a real life software project was conducted at the FC-MD in the summer of 2006. We used EBEAM to evaluate the architectural design versions of the air traffic control piece of software—the tactical separation-assisted flight environment (TSAFE) system. We discuss the study in details.

7.1 Context

The system under consideration is a prototype of the TSAFE software system defined by NASA Ames Research Center [25] and implemented by Dennis [26]. This implementation was later turned into a testbed by FC-MD [27]. TSAFE was proposed as a principal component of a larger Automated Airspace Computing system that shifts the burden from human controllers to computers. The TSAFE prototype checks conformance of aircraft flights to their flight plans, predicts future trajectories, and displays results on a geographical map. We refer to this original prototype as TSAFE I. Figure 4 shows the high-level structure of TSAFE I with its four main components: the client, the parser, the database, and the engine. TSAFE runs in two independent threads: the parsing thread and the main thread. In the parsing thread, the parser reads data from a radar feed, extracts flight information, and sends it to the database component. In the main thread, a timer in the client

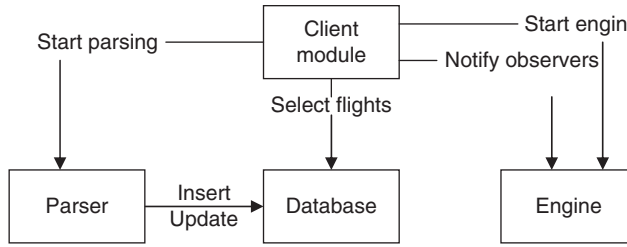

FIG. 4. Conceptual view of TSAFE I.

component initiates the process of updating the flight data every 3 s. The client requests data from the database and sends it to the engine component for computation. The engine sends the computation results to the client to be displayed on a graphical user interface (GUI).

Since TSAFE I was a prototype, it only implemented the most basic functions from the original NASA description [25]. An earlier analysis study was conducted at FC-MD using TSAFE I as the basis for an experimental software testbed [28]. During the experimental study, new set of features that would be required in TSAFE in the future were described. These features are related to demands that would be placed on TSAFE I when operating in a real environment.

After several analyses conducted during the experiments in Ackermann and Lindvall [28], it was discovered that it would be difficult to incorporate the new set of features into TSAFE I because the architecture is not easily modifiable. This prompted the need to redesign the system in order to fix the modifiability problems and then create structures to accommodate the implementation of the change requests. The newly created design of TSAFE is now referred to as TSAFE II. Both TSAFE I and TSAFE II have different structures but their external GUI and behavior are identical. Details regarding how the new architectural design was created from the old design are discussed in Ackermann and Lindvall [28]. The high-level design models of TSAFE I and TSAFE II are given by Figs. A.1 and A.2 in Appendix A.

During the design of TSAFE II, the designers focused on improving the manner in which a subset of design characteristics (i.e., naming, coupling, cohesion, and use of design patterns) is handled. Having completed the design of TSAFE II, it is important to be able to show whether there is improvement in modifiability. It is also necessary to show the extent of the improvement, and in terms of which design characteristics. In addition, it is desirable to determine how the current versions of the architectural designs measure up to the target design. With these issues under consideration, EBEAM was seen as a systematic method that could help with the evaluation and communication of results to third party. Most importantly, EBEAM could generate quantitative measures without ignoring the knowledge and experience of the local experts.

7.2 Evaluating TSAFE Designs Using EBEAM

The goal of this evaluation is to compare the architectural designs of TSAFE I and TSAFE II with respect to modifiability. Apart from comparing the two implemented design versions (i.e., TSAFE I and TSAFE II), it is important to determine how close the current designs are to realizing the projected design goals in terms of other design characteristics. The projected design goals are encapsulated in a conceptual

design known as the Target TSAFE. To perform the modifiability evaluation using EBEAM, we treat all the three designs (TSAFE I, TSAFE II, and Target TSAFE) as candidate designs. From the evaluation results, we can determine the relative modifiability of the two versions as well as their closeness to the target architecture design in terms of all the design characteristics.

During the case study, three experts participated in Stage I of the EBEAM evaluation process. The requirements for selecting these experts were based on their experience in architectural design in general. With their experiences, these experts could evaluate the relative contribution of the design characteristics to modifiability, without having particular design candidates on mind. For Stage II, only two experts that are familiar with the TSAFE system and its different design versions participated. These two experts (i.e., Architect-1 and Architect-2) compared the three candidate designs to evaluate the relative modifiability of TSAFE I and TSAFE II, and the closeness of the two versions to the Target TSAFE. The two experts that evaluated the design candidates in Stage II also participated in the evaluation of the design characteristics in Stage I. The participation of these experts in both stages is not a requirement. In fact, it would be more desirable to have the local experts participate only in the evaluation of architectural design candidates to reduce bias. For the case study, however, there were not enough independent experts for Stage 1 of the evaluation process. This is not a limitation of the EBEAM technique.

7.3 Results and Discussions

7.3.1 Results from Stage I: Contribution of Design Characteristics to Modifiability

On completing the first stage of the evaluation, the priority vectors resulting from each of the three experts, which show the relative contribution of design characteristics to modifiability, are given in Table III. We have not included the comparison matrix completed by each expert or the process of computing the priority vectors, because they adhere to the EBEAM process. In Table III, the columns named "Rank" represent the relative ranking of the contribution of each design characteristic to modifiability, based on each expert's evaluation. For example, expert e_1 ranked "minimized concurrency and threads" as the most important characteristic that determines how modifiable an architectural design would be, while ranking "minimized duplication" as the least important characteristic. The results in Table III shows that the experts do not completely agree on how each attribute relatively contribute to modifiability. However, a perfect correlation is not expected, because EBEAM is supposed to support the experts in articulating their individual

TABLE III
CONTRIBUTION OF THE CHARACTERISTICS TO MODIFIABILITY FROM THE PERSPECTIVES OF THE THREE PARTICIPATING EXPERTS

		Expert e_1		Expert e_2		Expert e_3		Aggregated result		
Z	Design characteristics	Priority vector $V1(z)$	Rank	Priority vector $V2(z)$	Rank	Priority vector $V3(z)$	Rank	Priority vector $V(z)$	Priority vector (%)	Rank
1	Minimized coupling	0.1591	R(2)	0.2345	R(1)	0.2564	R(1)	0.2151	21.51	R(1)
2	Maximized cohesion	0.1441	R(3)	0.1925	R(2)	0.1233	R(4)	0.1513	15.13	R(2)
3	Maximized modularity	0.1092	R(6)	0.1294	R(4)	0.1438	R(3)	0.1271	12.71	R(3)
4	Minimized concurrency and threads	0.1709	R(1)	0.0433	R(6)	0.0856	R(6)	0.1036	10.36	R(4)
5	Proper information hiding (including interfaces)	0.0309	R(9)	0.1050	R(5)	0.1743	R(2)	0.1024	10.24	R(5)
6	Appropriate use of design patterns and architectural styles	0.1108	R(5)	0.0402	R(7)	0.0993	R(5)	0.0858	8.58	R(6)
7	Minimized coupling to COTS (including language)	0.0537	R(8)	0.1766	R(3)	0.0209	R(10)	0.0789	7.89	R(7)
8	Proper documentation	0.1387	R(4)	0.0219	R(10)	0.0261	R(9)	0.0651	6.51	R(8)
9	Use of proper and representative naming	0.0655	R(7)	0.0276	R(9)	0.0363	R(7)	0.0442	4.42	R(9)
10	Minimized duplication	0.0172	R(10)	0.0292	R(8)	0.0340	R(8)	0.0266	2.66	R(10)
	Confidence level—CL =	8		7		8				
	Consistency ratio—CR =	0.0494		0.0976		0.0836				
	1−CR =	0.9506		0.9024		0.9164				
	CL×(1 − CR) =	7.6048		6.3168		7.3312				
	W_e =	0.3578		0.2972		0.345				

Sorted based on ranking of their contribution to modifiability.

knowledge and experience, which we cannot expect to be the same, although some levels of agreements exist. For example, the results from expert e_2 and expert e_3 agree that "minimized coupling" contributes to EoM the most. The results from expert e_1 almost agree with this ranking as well, as expert e_1 also ranked minimized coupling as the second most important characteristic.

The consistency ratio (CR_e), confidence level (CL_e), and the weight computed for each expert e are also given in Table III. This weight is computed for each expert using Equation 1. Although expert e_1 and expert e_3 claimed to exhibit the same confidence level, but expert e_1 achieved a higher CR when performing the evaluation of the design characteristics. The higher CR for e_1 indicates that we can trust the judgments of expert e_1 more than the judgments of expert e_3, which invariably translates to the higher weight assigned to e_1. Expert e_2 has the lowest weight because the expert has lower CL and lower CR than the other two experts. The weights are used in the aggregation scheme for unifying the different priority vectors from each expert's evaluation into a single priority vector. The resulting priority vector shows the relative contribution of each architectural design characteristic to modifiability. The aggregation scheme is defined in Equation 2.

The priority vector reflecting the aggregated views of all the experts is also shown in Table III. For easy interpretation, we also present the priority vectors as percentages. The percentages enable us to reason in terms of what percentage of the relative modifiability values that a specific characteristic assumes. The column named "Rank" in the aggregated result represents the relative ranking of each characteristic based on their impact on modifiability. (*Note*: we have sorted Table III according to these final rankings.) The aggregated result ranked "minimized coupling" as the architectural design characteristic that has the highest contribution to the modifiability of architectural designs (i.e., rank R(1)). The interpretation is that, if only one choice is possible, all the experts agree that they would rather select an architectural design that minimizes coupling over another design that minimizes duplication. The reasoning supporting such decision is that, it would be easier to modify a candidate with minimized coupling than another without minimized coupling, but which exhibits minimized duplication. The final aggregated priority vector given here would be used in Stage III when aggregating the results with those from TSAFE designs evaluation.

Our interpretation of results here should be treated with caution, however, because it only reflects the judgment of the participating experts based on their experience. It could turn out to be different ranking if the evaluation is performed by different architectural design experts. For the purpose of our study, this is not a limitation, because the experts have been drawn from the same environment where we need the results to evaluate specific design candidates.

7.3.2 Results from Stage II: Evaluation of TSAFE Designs

Results from the evaluation of TSAFE architectural designs, according to the perspectives of the two architects (i.e., the local experts), are given as part of Table IV. Table IV also contains the results of the consistency ratios ($CR_{e,z}$). The $CR_{e,z}$ is computed for each expert, using each comparison matrix developed for the design candidates with respect to each characteristic. In the last row of the table, we have the domain familiarity of each architect with respect to each of the design candidates. The $DF_{e,n}$ entries show that the architects claimed the same level of familiarity on all the design candidates, but their consistency ratio for each comparison would ensure the architects do not have the same influence on the overall modifiability result. In addition, Table IV has a column named "Priorities of design characteristics." This column represents the overall priority vector shown in Table III which already unifies the design experts' evaluation of the relative contribution of each characteristic to modifiability. This overall priority vector would be used as a multiplicative factor of each of the priority matrices from the architects' evaluation of the design candidates.

In Table IV, there are two $Z \times N$ priority matrices with each of the matrices representing the result from one architect. Every entry (z, n) in each priority matrix represents the relative modifiability of the architectural design candidate in column n with respect to the characteristic in row z. This priority matrix is described by Equation 3. For example, looking at the entries for "minimized coupling" in the first row for Architect-1, the architect evaluated the modifiability of TSAFE I as having a value of 0.0567 compared to the modifiability of TSAFE II that stands at 0.2946, and that of Target TSAFE which is 0.6486.

These modifiability values have been evaluated with respect to the way in which the architectural design candidates handle "minimized coupling" from the perspective of Architect-1. But what do the numbers really mean? In a simple term, suppose Architect-1 is given 100 points to distribute among the three candidates based on the way the candidates handled coupling, TSAFE I would receive 5.67% while TSAFE II and Target TSAFE would receive 29.46 and 64.86%, respectively. Therefore, TSAFE II is a major improvement over TSAFE I and it is more modifiable than TSAFE I in terms of reduced coupling. Notwithstanding the improved modifiability of TSAFE II, it is still far from the Target TSAFE design in terms of minimizing coupling in the architecture. This implies that Architect-I knows that the coupling would still need to be reduced considerably in order to achieve the target coupling level envisaged for the Target TSAFE. For example, suppose we have client/server architecture and TSAFE II implements it in such a way that there are bi-directional calls between the server and the client components, but the target is to

TABLE IV
EVALUATION OF TSAFE DESIGNS FROM THE PERSPECTIVES OF THE TWO ARCHITECTS

		Architect-1 (e_1)					Architect-2 (e_2)					Priorities of design characteristics
Z	Design characteristics	TSAFE I	TSAFE II	Target TSAFE	Consistency ratio (CR)	$1 - CR$	TSAFE I	TSAFE II	Target TSAFE	Consistency ratio (CR)	$1 - CR$	$V(z)$
1	Minimized coupling	0.0567	0.2946	0.6486	0.0701	0.9299	0.0548	0.3583	0.5869	0.0320	0.9680	0.2151
2	Maximized cohesion	0.0869	0.2737	0.6393	0.0466	0.9534	0.0548	0.3583	0.5869	0.0320	0.9680	0.1513
3	Maximized modularity	0.1111	0.4444	0.4444	0.0000	1.0000	0.0581	0.2299	0.7120	0.1453	0.8547	0.1271
4	Minimized concurrency and threads	0.3333	0.3333	0.3333	0.0000	1.0000	0.1250	0.1250	0.7500	0.0000	1.0000	0.1036
5	Proper information hiding (including interfaces)	0.1226	0.3202	0.5571	0.0158	0.9842	0.0543	0.3059	0.6399	0.0944	0.9056	0.1024
6	Appropriate use of design patterns and architectural styles	0.1096	0.3092	0.5813	0.0032	0.9968	0.0556	0.2424	0.7020	0.1867	0.8133	0.0858
7	Minimized coupling to COTS (including language)	0.3333	0.3333	0.3333	0.0000	1.0000	0.2500	0.2500	0.5000	0.0000	1.0000	0.0789
8	Proper documentation	0.2000	0.2000	0.6000	0.0000	1.0000	0.1111	0.1111	0.7778	0.0000	1.0000	0.0651
9	Use of proper and representative naming	0.0819	0.3431	0.5750	0.0251	0.9749	0.0577	0.3468	0.5955	0.0188	0.9812	0.0442
10	Minimized duplication	0.3333	0.3333	0.3333	0.0000	1.0000	0.0605	0.2009	0.7386	0.1460	0.8540	0.0266
	Domain familiarity (DF)	8	8	8			8	8	8			

We sort in descending order of the contribution of priorities of design characteristics.

have just the client call the server for services rather than allowing bi-directional calls. Then, it would be necessary to reduce the coupling in TSAFE II by a factor of 2 to attain the objectives defined for Target TSAFE. This interpretation is straightforward because the values derived by the architect for each of the candidates evaluated are relative to one another. Similar interpretation applies to all the other entries in the two priority matrices.

7.3.3 Results from Stage III: Overall Modifiability of TSAFE Designs

To aggregate the views of the architects, as shown in Table IV, we combine the priority matrices, the priority of design characteristics, the consistency ratios, and the domain familiarity using Equation 6. Instead of giving the raw values as matrices and vectors, we present the results as figures to ease understanding. Figure 5 shows the aggregated result that represents the relative modifiability of each candidate with respect to each architectural design characteristic. Figure 5 generally shows that TSAFE II constitutes a major improvement over TSAFE I across all the design characteristics, but TSAFE II still falls short of the specification envisaged for the Target TSAFE. This evaluation does not end only in the design comparisons, as it also allows us to know what characteristics of the design to focus on when making refactoring or redesign decisions.

Looking at the first characteristic (i.e., minimized coupling) in Fig. 5, for example, TSAFE II has achieved a major improvement over TSAFE I, but it is still far from achieving the target design in terms of coupling. But looking at the fourth characteristic (i.e., minimized concurrency and threads), TSAFE II is not any better than TSAFE I based on the evaluation of both architects. In addition, the two TSAFE design versions are not yet close to the Target TSAFE design in terms of their use of minimized concurrency and threads. The two TSAFE versions are also not better than each other in terms of documentation and coupling to COTS.

Figure 6 shows the final overall modifiability values that are computed from the fine-grained values given in Fig. 5, using Equation 7. The figure gives a more coarse-grained evaluation that allows us to easily rank the architectural designs and make decisions about which design is more modifiable overall, without looking at each design characteristic. This is more useful when we only have to choose from a set of candidate designs, regardless of the fine-grained information on the design characteristics.

The radar diagram in Fig. 7 is the transformation of the aggregated modifiability result shown in Fig. 5 into its idealized form, as discussed in Section 6.2.2. The Target TSAFE now represents the current goal, which is to attain a level 1

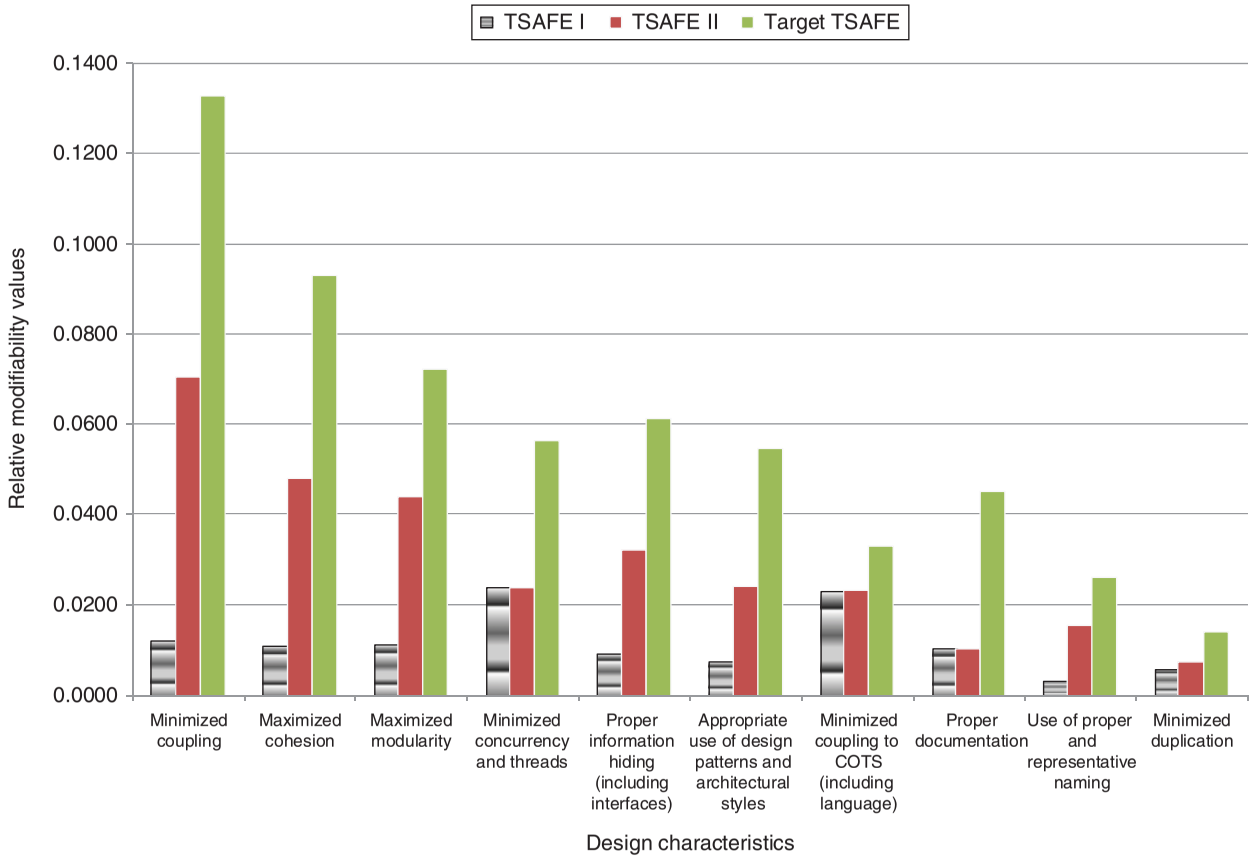

FIG. 5. Aggregated TSAFE modifiability with respect to each characteristic.

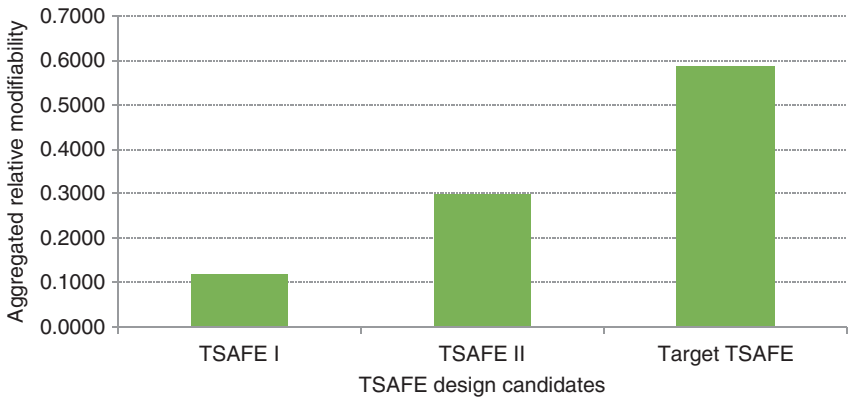

FIG. 6. Overall modifiability evaluation of the candidates.

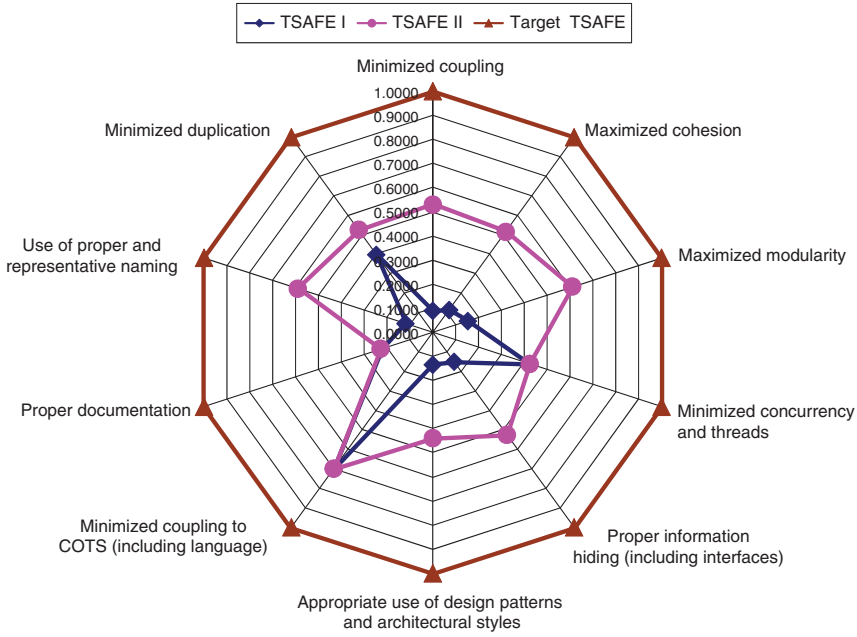

FIG. 7. Idealized representation of the aggregated modifiability with respect to each characteristic.

(i.e., 100%) on all design characteristics. The results shown in the radar diagram is useful for road mapping, since it is possible to determine how far we are from the goals specified for the target design. In Fig. 7, the target architectural design (Target TSAFE) is used as a reference. For instance, the radar diagram shows TSAFE II to be in the mid-way to achieving the specified goal in terms of maximized cohesion. Coupling to COTS is the design characteristic for which the current versions of the architectural designs have made the most improvement in terms of modifiability. Conversely, there is still no proper documentation available for TSAFE II. Documentation remains the characteristic for which the current design is farthest from reaching the target design goals.

8. Empirical Validation

The goal of this empirical validation is to verify the efficacy of EBEAM in assisting the experts to correctly articulate their knowledge of architectural designs. We defined and selected some architectural design metrics, collected data on the design candidates in terms of the metrics, and compared the results with the results from expert judgment. This empirical validation study was conducted in Fall 2007; about 3 months after the case study in which the experts articulated their knowledge of the architectural designs using EBEAM and results collected. This implies that the results from design metrics were not available during the case study discussed above, thereby reducing the possibility of the metrics results introducing bias into the judgments of the experts.

8.1 Defining and Selecting the Objective Metrics

Several design metrics have been discussed in the literature. For objective metrics collection, we chose a subset of the design characteristics discussed in Section 4.1. We only chose a subset of the characteristics, because it is not possible to define objective metrics on most of the characteristics. Since we only need the objective metrics for validation purposes, it is not even necessary to define metrics on all the characteristics. The design characteristics we have chosen include coupling, duplication, and information hiding/interfaces. These metrics can be computed from the unified modeling language (UML) models, except for measure of duplication that requires specialized tools. In the sequel, we discuss the metrics we have defined and those we have selected for use in this study.

8.1.1 Coupling Metrics

Although there are many existing coupling metrics [29], we focus on defining metrics that can easily be captured at both coarse-grained and fine-grained levels. The two coupling metrics provided here are based on measuring the relationship between elements that belong to different components/modules of a software system. It should be noted that we use the terms module and component interchangeably.

8.1.1.1 High-Level System Coupling Metric.

This metric is aimed at capturing the number of calls that exist between all the components of the architectural design of a system. The metric does not consider the individual classes in the components. The purpose is to have a high-level view of the number of components as well as the number of relationships that exist among them. The system coupling metric (SCM) is measured as a count of the number of directional inter-component references. We do not ignore the direction of the arrows because we want to capture the number of components as well as the different calls that exist.

From the UML model of TSAFE I in Fig. A.1 (see Appendix A), we compute the SCM(TSAFE I) = 9, while UML model in Fig. A.2 (Appendix A) gives SCM (TSAFE II) = 4. These values are computed by adding all the arrows in each UML model. For example, we have just three components in TSAFE II, but the Client and the Server components are involved in bi-directional calls for services. These coupling measures do not include the number of coupling to libraries. The lower the value of SCM the easier to modify the corresponding architectural design candidate is assumed to be, and vice versa. According to the results here, TSAFE II is more modifiable than TSAFE I.

8.1.1.2 Fan-In/Fan-Out Metric.

The fan-in/fan-out coupling metric proposed by Henry and Kafura [30] is a directional measure of coupling. The fan-in/fan-out captures the inbound and outbound relationship to and from the elements in a component. Given a component C, the fan-in is the number of components that call it and the number of global data elements (or libraries) from which C retrieves information. The fan-out for a component C is the number of components called by C, including the number of data elements (or libraries) that C alters.

The results of the fan-in/fan-out metrics for each component of TSAFE I and TSAFE II as computed from the low level views captured in Figs. A.3 and A.4 (see Appendix A) are given in Table V. The lower the value of fan-in/fan-out the easier it is to modify the components as well as the corresponding architectural design of the system, and vice versa.

TABLE V
COUPLING MEASURES FOR TSAFE I AND TSAFE II

TSAFE I			TSAFE II		
Components	Fan-in	Fan-out	Components	Fan-in	Fan-out
Main	0	10	Main	0	5
Client	1	6	Client	3	2
Engine	12	0	Server	5	1
Database	10	0			
Feed	4	11			

8.1.2 Interfaces Metrics

We define two metrics for interfaces. The first metric is the number of interfaces (NOI) which counts the number of classes from different components that directly assess classes in a particular component C. The second metric is the classes calling interfaces (CCI) which is a count of the classes in a component that directly assess the classes in other components. The NOI and CCI calculation for each component C and the sum for the entire system for TSAFE I and TSAFE II appear in Table VI. The interface metrics was computed from the low level views of the UML models of the architectures shown in Figs. A.3 and A.4 (Appendix A) for TSAFE I and TSAFE II, respectively.

We compute the NOI metric for each component of both TSAFE I and TSAFE II. Suppose we consider the *Main* component of TSAFE I and that of TSAFE II, these two *Main* components do not receive any direct call from any class outside of it. Thus, the NOI for the *Main* component of TSAFE I is 0 and that of TSAFE II is 0. We also compute the NOI for the rest of the components; the results appear in Table VI. We can see in Table VI that the NOI(TSAFE I) = 10 and NOI(TSAFE II) = 5, respectively.

In addition, we compute the CCI for both TSAFE I and TSAFE II. The results of the CCI also appear in Table VI. For example, if we look at the *Main* component in TSAFE I architectural design model in Fig. A.3, we can see that six classes in the *Main* component call several other classes in the other components. On the other hand, the *Main* component of TSAFE II architectural design has only one class that issues calls to classes in the other components. The goal is always to try to minimize the total number of these calls in a system. For example, instead of six classes in the *Main* component directly accessing data from the classes in the other components of the system, it is better to provide one interface through which these communications can take place. Table VI contains this measure for all the components in the system, where the total for all the components of TSAFE I is given as CCI(TSAFE I) = 14, and that of TSAFE II is given as CCI(TSAFE II) = 4.

TABLE VI
INTERFACES MEASURES FOR TSAFE I AND TSAFE II

TSAFE I			TSAFE II		
Components	Number of interfaces (NOI)	Classes calling interfaces (CCI)	Components	Number of interfaces (NOI)	Classes calling interfaces (CCI)
Main	0	6	Main	0	1
Client	1	2	Client	2	2
Engine	5	0	Server	3	1
Database	1	0	Total	5	4
Feed	3	6			
Total	10	14			

The lower the total NOI computed for an architectural design candidate, the easier it would be to modify the architectural design, and vice versa. Thus, TSAFE II is generally assumed to be more modifiable than TSAFE I according to the results.

8.1.3 Duplication

The tool we used to measure duplication in the code implementation of TSAFE I and TSAFE II is the Simian UI tool [31]. The Simian UI is a tool for finding and removing duplicated regions of code from within the Eclipse IDE. The measure is based on blocks that include more than six lines of code. For this measure, the result is that TSAFE I and TSAFE II contain the same amount of duplicated regions, namely 1321 blocks each.

8.2 Expert-Judgment Using EBEAM on Selected Design Characteristics

Since we focused on defining metrics for three characteristics, it is important that we also choose the same three characteristics from the expert judgment evaluation using EBEAM.

The good thing about EBEAM is that, we can easily extract the judgment of the experts on a subset of the design characteristics, without requiring the experts to perform the evaluation all over. The relative values obtained for each of the characteristics would be different from what we have in the results discussed in the case study because the relative comparison would be based on just three characteristics and not on all 10 characteristics considered earlier. The result of the

TABLE VII
EVALUATION OF TSAFE I AND TSAFE II DESIGNS ON THREE CHARACTERISTICS

		Architect-1 (e_1)		Architect-1 (e_2)		
Z	Design characteristics	TSAFE I	TSAFE II	TSAFE I	TSAFE II	Priorities of design characteristics $V(z)$
1	Minimized coupling	0.1250	0.8750	0.1111	0.8889	0.6519
2	Proper information hiding (including interfaces)	0.2500	0.7500	0.1111	0.8889	0.2704
3	Minimized duplication	0.5000	0.5000	0.1667	0.8333	0.0777
	Priority vector	0.1879	0.8121	0.1154	0.8846	

TABLE VIII
AGGREGATED TSAFE EVALUATION FROM ALL ARCHITECTS

Attributes	TSAFE I	TSAFE II
Minimized coupling	0.0770	0.5749
Proper information hiding including interfaces	0.0488	0.2216
Minimized duplication	0.0259	0.0518
Final priority vector	0.1517	0.8483

evaluation by each of the architects is shown in Table VII, while an aggregated result is shown in Table VIII. Table VIII compares TSAFE I with TSAFE II, with respect to each characteristic from modifiability perspective. Table VIII also shows the final overall modifiability of the two architectural designs, with TSAFE I having a relative modifiability value of 0.1517 while the relative modifiability value of TSAFE II is 0.8483. The result clearly shows that the modifiability, measured in this way, is higher for TSAFE II than for TSAFE I.

8.3 Comparing the Objective and Subjective Measures on TSAFE Architectural Designs

It is important that we have a basis for comparing the results from design metrics to those obtained from expert judgment. There is no straightforward way to accomplish this, because the results from the objective metrics defined on different characteristics are not on the same scale or unit, for all three characteristics considered. We do not have this problem with EBEAM, however. Notwithstanding, we can define approximate mappings that would ease the comparison. Table IX contains all the modifiability results computed using design metrics. The reader should note that we sum up the

TABLE IX
TSAFE EVALUATION USING METRICS

Metrics	TSAFE I	TSAFE II
System coupling metric	9	4
Fan-in/fan-out	54	16
Number of interfaces	10	5
Classes calling interfaces	14	4
Duplication	1321	1321

fan-in/fan-out for each of the TSAFE architectural designs, because we need to compare the architectural designs and not the components that make up the designs—although Henry and Kafura [30] did not perform such additions in their work.

Because the results from EBEAM are normalized measures (i.e., between 0 and 1), comparison between results from expert judgment and the metrics would only make sense if we normalize the results from design metrics. The easiest way to do this is to find the ratio of each architectural design measure relative to the measures within its metric category.

Thus, given two architectural designs with values A and B, we would compute their metric values $m(A)$ and $m(B)$ using the following formulas:

$$m(A) = 1 - \left(\frac{A}{A+B}\right) \tag{8}$$

$$m(B) = 1 - \left(\frac{B}{A+B}\right) \tag{9}$$

In Equations 8 and 9, we subtract from 1 to change the interpretation of the metric values such that higher values mean higher modifiability, and vice versa for lower values. This is necessary to ensure that the interpretation of results obtained using design metrics is consistent with those results obtained from expert-judgment using EBEAM. Recall that assigning higher values to an architectural design on a particular characteristic in EBEAM implies higher relative modifiability on that characteristic. On applying Equations 8 and 9 to the measures given in Table IX, we obtained the results given in Table X.

To compare the results from objective and subjective measures, we can choose any of the two coupling metrics and any of the two interfaces metrics in Table X to represent the objective metrics for these two characteristics (i.e., coupling and interfaces). This decision is justified, because the results from different metrics that measures the same characteristic produced correlated results. For example, both SCM and fan-in/fan-out consistently ranked TSAFE II as better than TSAFE I

TABLE X
NORMALIZED METRICS FOR TSAFE EVALUATION

Metrics	TSAFE I	TSAFE II
System coupling metric	0.3077	0.6923
Fan-in/fan-out	0.2286	0.7714
Number of interfaces	0.3333	0.6667
Classes calling interfaces	0.2222	0.7778
Duplication	0.5000	0.5000

in terms of reduced coupling. Thus, whether we discuss results in terms of SCM or in terms of fan-in/fan-out is immaterial; the relative preference between the candidates in terms of coupling remains the same. The same argument applies to the measures derived from NOI and CCI, as both measures agree in their ranking of the two design candidates.

Now, suppose we choose SCM and NOI as the representative metrics for coupling and interfaces, respectively. Figure 8 shows the bar charts of the results from metric-based approach. It is easy to deduce the relative modifiability of the two architectural design candidates with respect to each characteristic. Except for duplication measures in which the two architectural designs perform equally, TSAFE II is clearly better than TSAFE I.

Now, looking at the results from expert judgment using EBEAM (see Table VIII), we see that the values for TSAFE I and TSAFE II fall between 0 and 1 for each characteristic. However, the modifiability values of the two architectural designs do not add to 1 for each characteristic, as we now have for the metrics results in Table X. Thus, we need to apply Equations 8 and 9 to Table VIII without subtracting 1 from the results. The normalized results using the EBEAM are given in Table XI with the corresponding bar chart in Fig. 9.

The results shown for the metrics-based approach in Fig. 8 and the results from EBEAM in Fig. 9 are both consistent in their conclusions that TSAFE II is overall a superior architectural design candidate in terms of modifiability TSAFE I. Although the two figures reveal that we do not have perfect correlation in the exact values resulting from the two approaches in terms of each characteristic considered, but this should not be expected. The most important issue is whether EBEAM could assist expert in making consistent judgment that reflects reality. By consistently ranking the design candidates, EBEAM allows easy judgment between competing design candidates. Meanwhile, the difference in the conclusion drawn by expert assessment and design metrics with respect to duplication could be an indication of the limitation of human expert in coping with more fine-grained assessments. Especially, for a task as

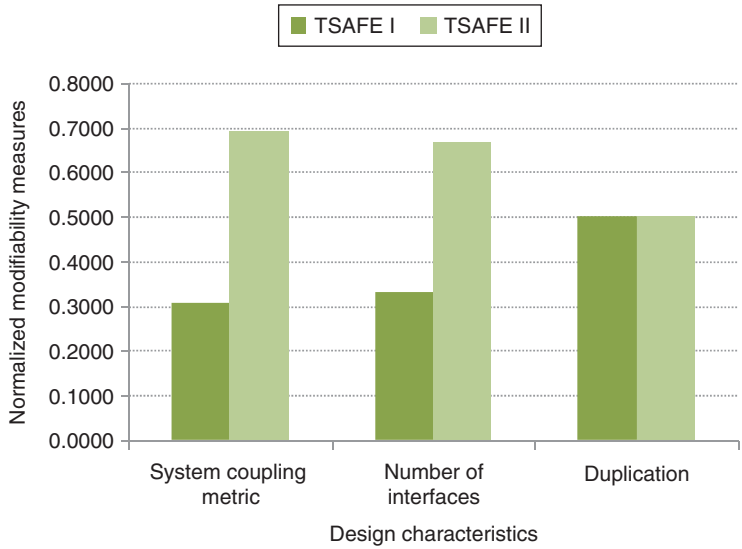

FIG. 8. Metrics results on TSAFE I and TSAFE II.

TABLE XI
NORMALIZED TSAFE EVALUATION USING EBEAM

Attributes	TSAFE I	TSAFE II
Minimized coupling	0.1181	0.8820
Proper information hiding (including interfaces)	0.1806	0.8195
Minimized duplication	0.3334	0.6667

arduous as sifting through codes that implements a design to count duplicated entities, only a rough approximation could be expected from an expert, at the very best.

9. CASE STUDY II: The Application of EBEAM to CGS Designs

EBEAM was employed in evaluating the architectural designs of the Common Ground Software (CGS) developed at the John Hopkins University Applied Physics Laboratory (JHU/APL) Space Department. This study, which is not as detailed as the

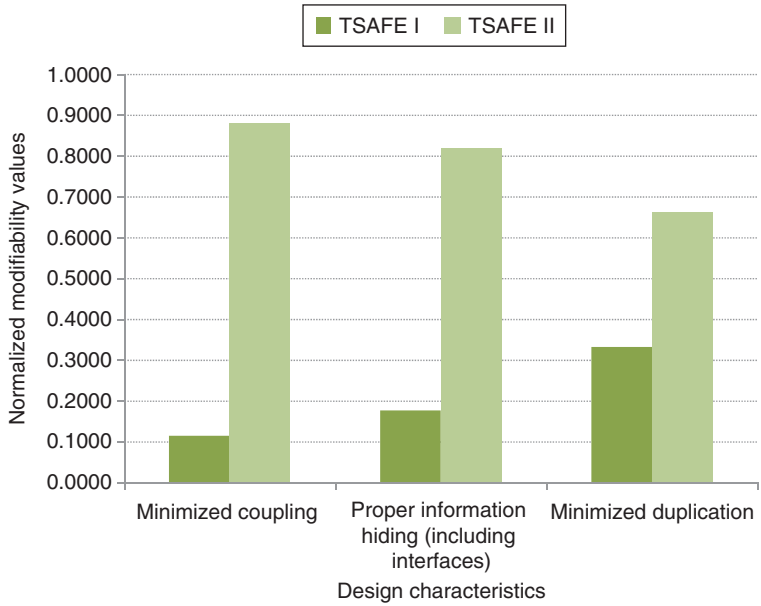

FIG. 9. Results of EBEAM on TSAFE I and TSAFE II.

one conducted with the TSAFE architects due to participation of experts, evaluates the applicability of the architecture evaluation method in a different domain.

9.1 Context

The JHU/APL Space Department develops Mission Operations Center (MOC) system software using a shared software architecture called Common Ground for all JHU/APL-supported NASA missions. The Common Ground architecture is 10 years old; it is difficult to maintain for current missions and to evolve for reuse in future missions. For instance, the CGS has a number of dependencies on technologies that have become obsolete. To avoid further maintenance and evolution problems, the Software Architecture Visualization and Evaluation (SAVE) tool and process developed by FC-MD was applied to the CGS [32]. SAVE allows JHU/APL to align the actual architecture (as currently defined in the source code) of CGS with the planned architecture for CGS (as defined by the architectural goals).

The process of applying SAVE to the CGS involves generating a high-level description of the *actual* architecture from the source code, defining a *planned*

architecture including architectural goals and design rational, identifying deviations between the planned and actual architecture, creating a new *target* architecture based on updated architecture goals, and creating a roadmap to align on-going system development and maintenance with the new target architecture. For a detailed discussion of the infusion of SAVE tool and process into the CGS, the reader can consult Stratton et al. [15, 32]. The most interesting aspect of this technology infusion project to us in this study was the availability an actual (or current), target, and planned (or ideal) architectural design candidates. The possibility of having two concrete versions of the CGS architectural design and a planned (or ideal) version immediately shows that applicability of EBEAM in evaluating the different versions. Without going into the details of the evaluation process, which is clear from Case Study I, we briefly discuss the EBEAM application to CGS and the results.

9.2 Evaluating CGS Designs and Results

This study comprised the comparison of architectural design of Current CGS and Target CGS with respect to modifiability. It also involves the comparison of these two design candidates to the ideal design in order to determine the closeness of existing candidates to the envisioned ideal architectural design. A JHU/APL senior software engineer who has extensive CGS experience conducted the study. Given that the Stage I of EBEAM is generic, we concentrated the study on Stage II of EBEAM. The specific knowledge and expertise of the senior software engineer on the CGS architectural design is the important aspect of the study, while the priorities of architectural design characteristics (see Table III) obtained in the first case study can be applied.

Table XII shows the results of the evaluation conducted on the CGS designs. Each entry ($z \times n$) in the priority matrix represents the modifiability of the CGS architectural design candidates in column n with respect to the characteristics in row z. The priorities of design characteristics are also represented in the table. Detail interpretation of these results follows from the discussion of results in Section 7 presented earlier.

The results from aggregating priorities of design characteristics and the evaluation of design candidates by the software engineer are shown in Fig. 10. Similarly, Fig. 11 shows the graphical radar representation of the idealized analysis that captures the extent to which the current architectural design candidates for CGS are close to realizing the envisioned Ideal Design. The interpretation of results is straightforward, given the previous discussion of Section 7. For example, Fig. 10 shows that the Target CGS architecture has improved significantly in terms of minimizing duplication and concurrency and threads. In fact, the software engineer performing the evaluation is of the opinion that the Target CGS is close to the envisioned Ideal CGS in terms of these two design characteristics (i.e., minimizing

TABLE XII
EVALUATION OF CGS DESIGNS

	Software engineer						
	Attributes	Current CGS	Target CGS	Ideal CGS	Consistency ratio (CR)	1 − CR	Priority of design characteristics $V(z)$
1	Minimized coupling	0.1429	0.1429	0.7143	0.0000	1.0000	0.2151
2	Maximized cohesion	0.1429	0.1429	0.7143	0.0000	1.0000	0.1513
3	Maximized modularity	0.2000	0.2000	0.6000	0.0000	1.0000	0.1271
4	Minimized concurrency and threads	0.1429	0.4286	0.4286	0.0000	1.0000	0.1036
5	Proper information hiding (including interfaces)	0.1429	0.1429	0.7143	0.0000	1.0000	0.1024
6	Appropriate use of design patterns and architectural styles	0.0612	0.2157	0.7231	0.1035	0.8965	0.0858
7	Minimized coupling to COTS (including language)	0.0612	0.2157	0.7231	0.1035	0.8965	0.0789
8	Proper documentation	0.2000	0.2000	0.6000	0.0000	1.0000	0.0651
9	Use of proper and representative naming	0.3333	0.3333	0.3333	0.0000	1.0000	0.0442
10	Minimized duplication	0.0909	0.4545	0.4545	0.0000	1.0000	0.0266

duplication and concurrency/threads). The other characteristics in which significant improvements have been made in the Target CGS over the Current CGS are minimized coupling to COTS and appropriate use of design patterns and architecture styles. In most of the other design characteristics, except for use of proper naming, no significant improvement exists in the Target CGS. The implication of these results is that, the architects can focus their redesign effort on improving the four design characteristics that have seen no improvement in the target CGS.

With the Ideal CGS used as the benchmark in the idealized analysis representation of Fig. 11, we can easily deduce the closeness or otherwise of Target CGS to the design goal from the perspectives of different design characteristics evaluated.

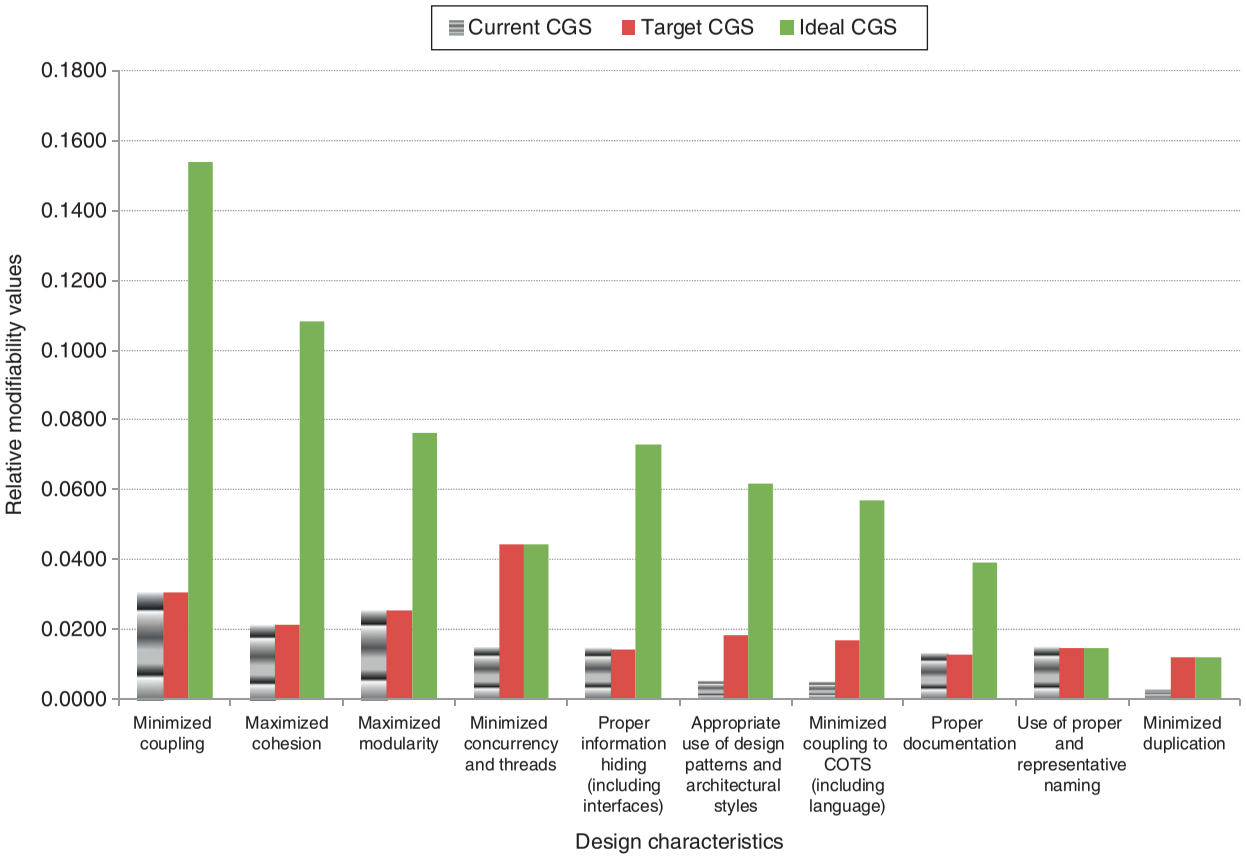

FIG. 10. Aggregated CGS modifiability with respect to design characteristic.

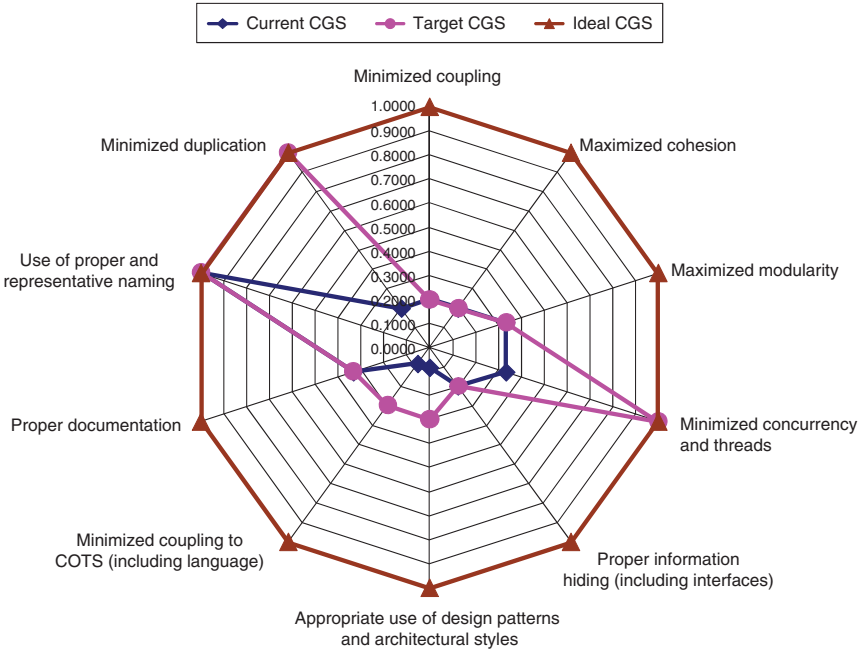

FIG. 11. Idealized modifiability of CGS with respect to design characteristic.

For example, maximized cohesion, minimized coupling, and proper information hiding remain the three designed characteristics that the least improvement has been achieved in the Target CGS. These are the three characteristics on which the Target CGS is farthest from the goal.

This second study essentially demonstrates the applicability of EBEAM across different domains. It also shows the usefulness of EBEAM even when we do not have multiple experts to perform the evaluations.

10. Applicability of the EBEAM Technique

Observations from the results obtained using both EBEAM and design metrics show that the two approaches are not competing, but rather complementary. Therefore, expert judgment can be employed to evaluate candidate architectures for characteristics that cannot be measured using metrics, and vice versa. If there are no mechanisms to collect metrics, then expert judgment would suffice for evaluating

all the architectural design characteristics. Especially, if all we have are high-level architectural designs without the codes that implement the architectures, then it would be hard to even consider using some of the existing metrics. Example of this is the conceptual view of the target architectural design that we considered during the case study that we discuss in this chapter.

In fact, there are situations in which objective metrics exist for measuring specific characteristics, but such metrics may not be able to capture the detail meaning of the characteristics. If we consider characteristics like *use of proper and representative naming*, one could define a metric that uses the Thesaurus to check for appropriate naming. This type of metrics would only confirm the use of meaningful names, but it cannot measure whether such names are representative. During the first case study presented, the lead architect that worked on redesigning TSAFE I shared his experience on the use of representative naming. He discovered a particular component that was named "Feed" in TSAFE I, which took him several days to decipher what the component actually does. This component was later discovered to be a component that handles parsing tasks, which would have been easier to understand if named "Parser." The EoM of such component would be inhibited, because the naming convention is not representative of the tasks the component performs, even though the naming convention is meaningful. A metric would show the name "Feed" as proper because it is meaningful. It is only through expert judgment evaluation that we can establish whether the naming convention adopted across the architectural designs are representative. Closely related to this is the fact that we can easily determine whether documentation exists, but we cannot use metrics to determine how proper and useful the documentation is.

In concluding our discussions on EBEAM and the empirical studies presented, we summarize the benefits, limitations, and threats to validity of using the EBEAM for architectural design evaluation.

10.1 Benefits of EBEAM

The major benefit of the EBEAM technique presented in this chapter is that it does not just give a high-level measure that simply identifies which architectural candidate is better than another. It also allows the architects to see which architectural characteristic needs to be re-examined to increase the modifiability of the candidate. It is not enough to say that an architecture restructuring has been performed and modifiability has been improved. It is important to know what aspects have been improved in the redesigned architecture. Thus, the results derived from EBEAM can serve as a pointer to what aspects of the existing architectural design require further improvement to increase modifiability.

Because of the flexibility and adaptability properties of EBEAM, the results from Stage I (i.e., evaluating the contribution of characteristics to modifiability) can be

updated with new knowledge and experience of new experts, whenever such become available. The new experts simply have to carry out their own evaluation, and the aggregated results are recomputed to accommodate the new knowledge, if desired.

There are several other benefits that would justify the use of EBEAM over design metrics. These benefits include the following:

- EBEAM is a generic modifiability evaluation technique that can be applied to any architectural design, regardless of the domain of the software system as demonstrated by the two case studies. Some metrics are not defined to be generic, and would work within certain programming paradigms only. For example, some metrics properties defined by Weyuker [33] have been found to be irrelevant to object-oriented design metrics, especially the structural metrics [34, 35]. There are hosts of other metrics that are targeted at only object-oriented designs, like those by Chidamber and Kemerer [36]. Metrics defined on datasets collected from a specific software product cannot always be generalized to other products, especially if the software products are from different domains.
- The results from EBEAM are based on ratio scale measures. The aggregation of the judgment of experts also retains ratio scale measures. The measures from EBEAM are also normalized relative measures that enable easy comparison of results. In the case of design metrics, however, raw measurements taken on different design characteristics would all be on different range of values and scales/units. Thus, it is not possible to aggregate these measures into a single measure to support decision making. And consequently, it complicates the comparison of different candidates using raw metric results [37]. Some metrics have been criticized in the literature because of the way the metrics assume measurement scale types [38] that contradict standard measurement practices [38, 39].
- EBEAM can easily be adopted and adapted to assess other architectural design attributes, even though we have developed it with modifiability as the focus. The experts only need to identify the characteristics that influence such attributes, and the nature of influence that each characteristic have on the attributes. Nevertheless, the evaluation process remains the same.

10.2 Limitations

We identified the following limitations of the EBEAM technique and threats to the experiments discussed in the case study:

- The EBEAM presents a repeatable evaluation process, but it does not guarantee that an expert will always give the same relative ranking to each characteristic and the architectural design candidates, if the expert is not consistent in

his judgment. The consistency of an expert in making judgments depends on how focused the expert during the comparison procedures.

- We consider as an approximation the choice of design characteristics for evaluating modifiability. Modifiability would always be influenced by several other issues. But using this approximation for assessment would suffice for the task of comparing any two or more architectural design candidates.
- The EBEAM is a heavy-weight technique. Adopting EBEAM requires that software architects devote time to the process of articulating their knowledge.
- As part of EBEAM, experts are asked to rate their confidence level (CL) in making judgment on design characteristics and also their domain familiarity (DF) with the architectural design of the system being evaluated. Some experts may assign higher ratings for themselves on these two factors, even if they do not posses high CL or DF. This could happen if the experts do not want to reveal that they posses little knowledge of the design characteristics, and also little knowledge of the architectural design candidates. We cannot independently verify their ratings in this case, because it is difficult to ascertain that the ratings they give for CL and DF are consistently in sync with their levels of expertise.

11. Related Work

To determine how the characteristics of architectural designs compare to the desired characteristics of a system, techniques for evaluating architectural designs are important. Architecture evaluation can be performed at different stages in the development life cycle. Lindvall et al. [24] distinguish between early and late software architecture evaluation. Early evaluation is carried out on architectures that are yet to be implemented. These architectures are typically a description of the system to be built. Although it is impossible to fully understand the actual design characteristics of a system until it is built and tested, a model of the system can help characterize the architectural design for evaluation purposes. On the other hand, late evaluation is used to evaluate software architectures once an implemented version of the system exists. Late architecture evaluation is useful because it helps in identifying deficiencies in the existing architecture, and guides the reconstruction of the architecture to address these deficiencies. An example of a late architecture evaluation method and tool is the SAVE [40]. The tool extracts the architecture from a system implementation and visualizes it in a diagram. The evaluation is done by comparing the extracted diagram to another diagram that represents the

design specification. The output is a diagram annotated with deviations between the implemented architecture and its design.

SAVE, like most architecture evaluation approaches, assesses properties of the static structure of a software system. Only few approaches focus on the system's runtime structure, that is, the dynamic architecture. For example, Lindvall et al. [41] analyzed the differences in how the system is structured during design time and during runtime [41]. Dynamic architectures are becoming increasingly important with the emerging trend of distributed systems and framework-based architectures because dependencies are often established only during runtime and not visible during design time. Object-based distributed frameworks, such as CORBA and Java RMI, are more and more widely used for implementing even smaller applications with the goal to divide the services of a system into independent components. OSGI is a framework-based architecture that serves a similar purpose, but is based on the idea of plug-ins [42]. In light of this development, the analysis of dynamic architectures becomes an increasingly important issue. An approach for evaluating dynamic dependencies for framework-based architectures has been introduced by Ganesan et al. [43]. The authors have developed strategy for constructing an architecture model via Colored Petri Nets (CPN) by monitoring events during system execution.

In addition to structural characteristics, software architectures also possess behavioral properties. The behavior describes the way architecture components interact [44]. UML Sequence Diagrams are a common way to model system behavior in the design phase, but is also widely used to illustrate the implemented behavior in various reverse engineering approaches [45, 46]. However, the goal of most reverse engineering approaches is to derive a general understanding of the system's behavior but they do not provide capabilities to evaluate it to a given design. An exception is the work presented in Ackermann et al. [47]. The authors discuss an approach for evaluating the adherence of several behavioral properties—namely sequence, parameter, and timing—to a behavior specification. The result of this evaluation is a graphical representation of the deviations.

Abowd et al. [48] categorize existing architecture evaluation methods into two—*questioning* and *measuring* techniques. Questioning techniques generate qualitative questions to be asked about architecture, while measuring techniques are based on quantitative measurements on the architecture to determine a specific quality attribute. Instead of providing ways to generate questions that should be asked about an architectural design, the measuring techniques provide answers to existing questions that the evaluation team already has about particular qualities of the architecture. Questioning techniques include scenarios, checklists, and questionnaires, while measuring techniques include metrics, simulations, prototypes, and experiences.

Most of the existing works on architecture evaluation are based on questioning techniques, with scenarios being the most widely used questioning technique.

Scenarios are used to capture events that could happen during the life of a system. Some of the architectural evaluation techniques that are based on questioning techniques include the Architecture Tradeoff Analysis Method (ATAM) [49], Cost-Benefit Analysis Method (CBAM) [50], Active Reviews for Intermediate Design (ARID) [51], Scenario-Based Architecture Analysis Method (SAAM) [52] and its variants, Architecture-Level Prediction of Software Maintenance (ALPSM) [53], Software Architecture Evaluation Model (SAEM) [54], and Architecture Level Modifiability Analysis (ALMA) [3]. Each technique has different views about architecture evaluation and presents different approaches to assessing architectural designs. Measuring techniques for architecture evaluation are relatively rare, though. Shereshevsky et al. [55] and Lindvall et al. [24] are two examples of research in this area. Existing measuring techniques use coupling or cohesion measures or both, as the basis for their evaluation. Measurement-based techniques have some limitations: they are always strongly dependent on the context, it is difficult to determine the right set of metrics, and the interpretation of pure measurement data is often unclear (what does it mean if a value is 4 instead of 5). For a more detailed discussion of existing evaluation techniques in the literature, the reader should refer to the survey by Dobrica and Niemela [51] which discusses various early evaluation techniques, and the comparison framework by Babar et al. [56] and Kazman et al. [57].

The existing techniques discussed above are mostly targeted at evaluating a single architecture [58]. These techniques are used to determine the suitability of architectural designs with respect to quality attributes of a software system. The evaluation techniques could clarify whether a given architecture satisfies certain properties, but they do not directly result in criteria collation and analysis. EBEAM is useful for evaluating a single architecture from the perspective of a chosen quality attribute, and also useful for selecting among competing architectural design candidates.

While we require experts to answer questions about the goodness characteristics of architectural designs when making their judgments, the resulting judgments are transformed to measurable quantities to better support decision-making. Therefore, EBEAM is a bridge between measuring and questioning techniques (see chapter "Advances in Computer Displays" for a discussion of these techniques). By combining both measurement-based approach and questioning-based approach, we take advantage of the benefits that each approach has to offer. Each architectural design candidate can be compared based on quantitative results generated with respect to different design characteristics considered. Of all the existing techniques, the closest to EBEAM is the architecture evaluation technique proposed by Svahnberg et al. [58]. In contrast to our work, Svahnberg and his colleagues simply focused on comparing candidate architectures based on the presence of competing quality attributes. Their technique is mostly useful for early architectural design evaluation, and they do not support fine-grained evaluation based on selected characteristics.

The EBEAM presented in this chapter is more general, because it can be used during both early and late phases of architectural designs.

12. Summary

In this chapter, we have discussed our expert judgment-based technique (i.e., EBEAM) for evaluating the modifiability of software architectural designs. We have reported a case study that uses EBEAM to evaluate the architectural designs of the prototype system for NASA flight assistance known as TSAFE. The architectural designs of TSAFE that were evaluated include the initial prototype, a redesigned version, and the target design. To validate the results obtained from EBEAM on TSAFE design evaluations, we defined a set of design metrics, evaluated the modifiability of the architectural designs of TSAFE using these metrics, and compared the results. Both EBEAM and the metrics show correlation in their results. We briefly presented another case study that applied EBEAM to the CGS developed at the JHU/APL.

We developed EBEAM as three-stage evaluation technique that can be performed independently, depending on the relevance of the knowledge that a participating expert has at that stage of the evaluation process. The three-stage formulation of the technique ensures that, the bias of individual experts in the first stage of the assessment would not be propagated down to the subsequent stages of the evaluation process. While we acknowledge that the bias of the experts cannot be completely eliminated, reducing the bias is an initial attempt. We can only verify the extent to which bias influences results after an extensive longitudinal study involving the application of EBEAM to different projects, and also in different development environments.

In developing EBEAM, we have also proposed a new weighting scheme to assign importance to experts. The weighting scheme considers the consistency with which the experts make their judgments about the characteristics or the architectural candidates being compared. Our empirical study did not reveal how much influence the inclusion of the consistency of judgment exerts on the final results, because the consistency ratio of the experts that participated in the study at FC-MD was close. But this is not expected to be the case all of the time, because varying degrees of inconsistency could occur, especially when there are more experts involved.

The contributions of EBEAM that we have proposed in this chapter include:

1. Proposing architectural design evaluation technique (i.e., EBEAM) that assists experts in articulating their knowledge and experience, and transforming these into quantifiable results.

2. Combining measurement-based approach with questioning-based approach in EBEAM, thereby taking advantage of the benefits that each approach has to offer.
3. Conducting exploratory study involving detailed analyses of design characteristics that influence the modifiability of software architectural designs.
4. Describing the way EBEAM can be adopted for making architectural comparison and selection decisions.
5. Demonstrating the applicability of EBEAM in a case study using a prototype of the TSAFE system defined by the NASA Ames Research Center [25, 26] and another study that used the CGS developed at the JHU/APL.
6. Evaluating the correlation between the results obtained from expert judgment using EBEAM and the results from design metrics.

Appendix A: UML Models for TSAFE Architectural Designs

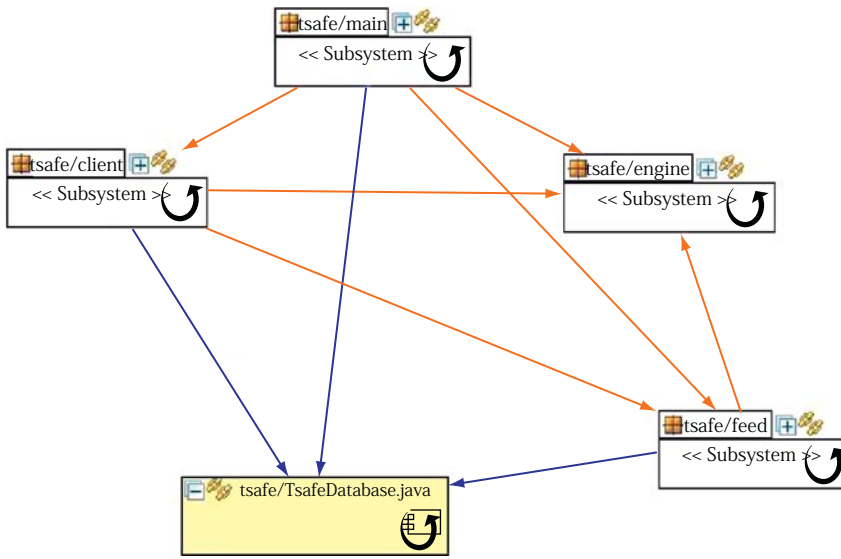

FIG. A.1. High level coupling in TSAFE I.

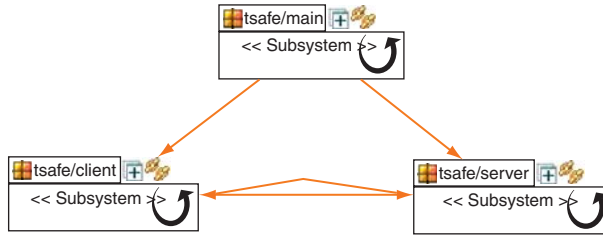

FIG. A.2. High level coupling in TSAFE II.

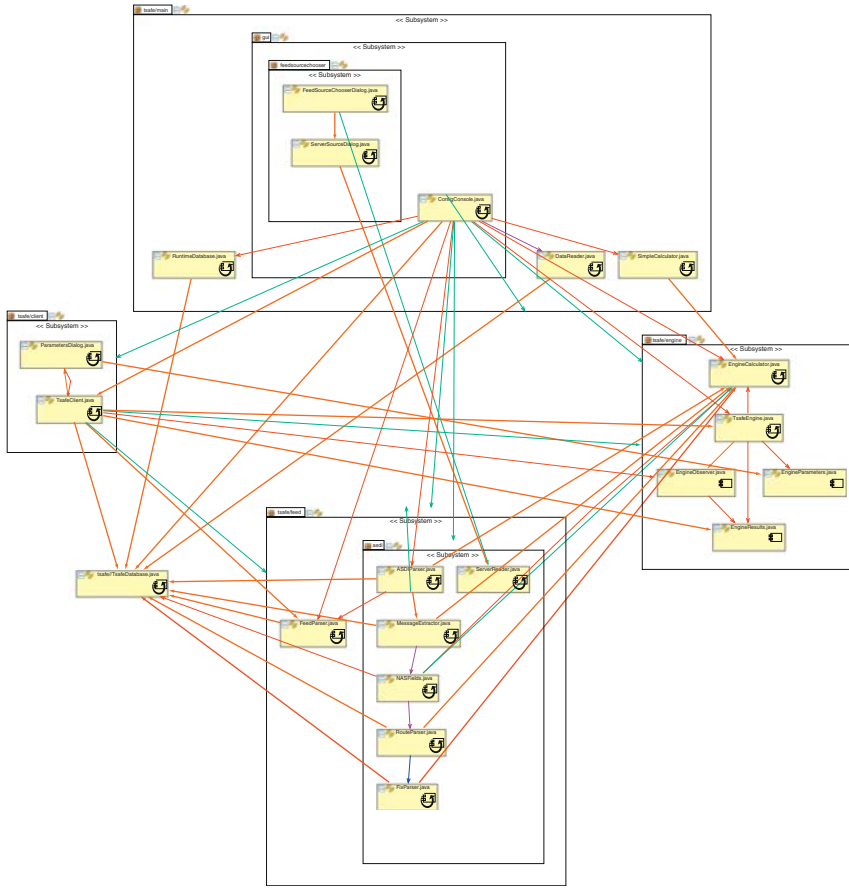

FIG. A.3. Low level view of TSAFE I.

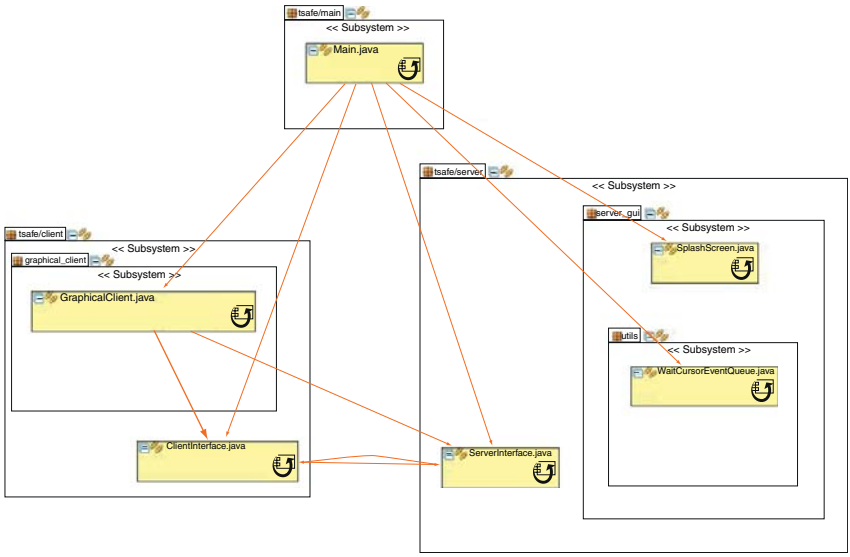

FIG. A.4. Low level view of TSAFE II.

ACKNOWLEDGMENTS

This research is partially funded by the National Science and Engineering Research Council of Canada (NSERC) and Informatics Circle of Research Excellence (iCORE). This work is partially sponsored by NSF grant CCF0438933, "Flexible High Quality Design for Software." We thank FC-MD for using their facilities during the workshops and the empirical studies, Patricia Costa for contributing her knowledge and expertise, William C. Stratton for facilitating the study conducted at JHU/APL, and Jens Knodel for stimulating discussions and helpful comments on earlier drafts of the paper.

REFERENCES

[1] M.M. Lehman, Laws of software evolution revisited, in: Proceedings of 5th European Workshop on Software Process Technology (EWSPT'96), Nancy, France, Oct. 9–11, 1996, pp. 108–124.
[2] P. Grubb, A.A. Takang, Software Maintenance: Concepts and Practice, second ed., World Scientific, New Jersey, 2003.
[3] P. Bengtsson, N. Lassing, J. Bosch, H. van-Vliet, Architecture-level modifiability analysis (ALMA), J. Syst. Softw. 69 (1–2) (2004) 129–147.
[4] D. Kelly, A study of design characteristics in evolving software using stability as a criterion, IEEE Trans. Softw. Eng. 32 (5) (2006) 315–329.
[5] T.L. Saaty, The Analytic Hierarchy Process, McGraw-Hill, New York, 1980.
[6] M. Li, C. Smidts, A ranking of software engineering measures based on expert opinion, IEEE Trans. Softw. Eng. 29 (9) (2003) 811–824.

[7] E. Forman, K. Peniwati, Aggregating individual judgments and priorities with the analytic hierarchy process, Eur. J. Oper. Res. 108 (1) (1998) 165–169.
[8] R.T. Clemen, R.L. Winkler, Combining probability distributions from experts in risk analysis, Risk Anal. 19 (2) (1999) 187–203.
[9] O. Saliu, G. Ruhe, Software release planning for evolving systems, Innov. Syst. Softw. Eng. NASA J. 1 (2) (2005) 189–204.
[10] H. Erdogmus, What's good software, anyway? IEEE Softw. 24 (2) (2007) 5–7.
[11] Hacknot, The top 10 elements of good software design, http://www.hacknot.info/hacknot/action/showEntry?eid=54 (last accessed: Aug. 6, 2006).
[12] L. Hochstein, M. Lindvall, Diagnosing architectural degeneration, in: Proceedings of 28th NASA/IEEE Software Engineering Workshop (SEW-28), Greenbelt, MD, USA, Dec. 3–4, 2003, pp. 137–142.
[13] H. Dhama, Quantitative models of cohesion and coupling in software, J. Syst. Softw. 29 (1) (1995) 65–74.
[14] S.S. Yau, J.S. Collofello, Some stability measurements for software maintenance, IEEE Trans. Softw. Eng. 6 (6) (1980) 545–552.
[15] W. Stratton, D. Sibol, M. Lindvall, P. Costa, Technology infusion of the SAVE tool into the common ground software development process for NASA missions at JHU/APL, in: Proceedings of 2007 IEEE Aerospace Conference, Big Sky, MT, USA, March 3–10, 2007.
[16] J. Krauskopf, Elemental concerns (software design), IEEE Potentials 9 (1) (1990) 13–15.
[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Design, Addison-Wesley, Reading, MA, 1994.
[18] M. Shaw, Comparing architectural design styles, IEEE Softw. 12 (6) (1985) 27–41.
[19] M. Shaw, P. Clements, Toward boxology: Preliminary classification of architectural styles, in: Proceedings of Second International Software Architecture Workshop, San Francisco, USA, 1996, pp. 50–54.
[20] D. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (12) (1972) 1053–1058.
[21] D. Parnas, P. Clements, D. Weiss, The modular structure of complex systems, in: Proceedings of 7th International Conference on Software Engineering (ICSE'84), Orlando, FL, USA, 1984, pp. 408–417.
[22] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond, Addison-Wesley, Reading, MA, 2002.
[23] F.P. Brooks, No silver bullet: Essence and accidents of software engineering, Computer 20 (4) (1987) 10–19.
[24] M. Lindvall, R. Tesoriero, P. Costa, An empirically-based process for software architecture evaluation, Empir. Softw. Eng. 8 (1) (2003) 83–108.
[25] H. Erzberger, The automated airspace concept, in: Proceedings of 4th USA/Europe Air Traffic Management R & D Seminar, Santa Fe, NM, USA, Dec. 3–7, 2001, pp. 1–15.
[26] G. Dennis, Building a Trusted Computing Base for Air Traffic Control Software, Master's Thesis, Massachusetts Institute of Technology, Boston, MA, 2003.
[27] M. Lindvall, I. Rus, P. Donzelli, A. Memon, M. Zelkowitz, A. Betin-Can, T. Bultan, C. Ackermann, B. Anders, S. Asgari, V. Basili, et al. Experimenting with software testbeds for evaluating mew technologies, Empir. Softw. Eng. Int. J. 12 (4) (2007) 417–444.
[28] C. Ackermann, M. Lindvall, Understanding change requests in order to predict software impact, in: Proceedings of 30th NASA/IEEE Software Engineering Workshop (SEW 2006), Columbia, MD, USA, April 25–26, 2006.

[29] L.C. Briand, J.W. Daly, J.K. Wust, A unified framework for coupling measurement in object-oriented systems, IEEE Trans. Softw. Eng. 25 (1) (1999) 91–121.
[30] S. Henry, D. Kafura, Software structure metrics based on information flow, IEEE Trans. Softw. Eng. 7 (5) (1981) 510–518.
[31] Simian, Simian UI—The Code Similarity Analyzer, Redhill Consulting Pty Ltd., http://www.integility.com/simian_ui (last accessed: Nov. 15, 2006).
[32] W.C. Stratton, D.E. Sibol, M. Lindvall, P. Costa, The SAVE tool and process applied to ground software development at JHU/APL: An experience report on technology infusion, in: Proceedings of 31st IEEE Software Engineering Workshop (SEW-31), Columbia, MD, March 6–8, 2007, pp. 187–193.
[33] E.J. Weyuker, Evaluating software complexity measures, IEEE Trans. Softw. Eng. 14 (9) (1988) 1357–1365.
[34] J. Alghamdi, M.O. Saliu, Analysis and theoretical validation of object-oriented coupling metrics, in: Proceedings of IASTED International Multi-Conference on Applied Informatics (AI'03), Innsbruck, Austria, Feb. 10–13, 2003, pp. 1145–1152.
[35] Gursaran, G. Roy, On the applicability of Weyuker property 9 to object-oriented structural inheritance complexity metrics, IEEE Trans. Softw. Eng. 27 (4) (2001) 381–384.
[36] S.R. Chidamber, C.F. Kemerer, A metrics suite for object-oriented design, IEEE Trans. Softw. Eng 20 (6) (1994) 476–493.
[37] A.P. Nikora, J.C. Munson, Determining fault insertion rates for evolving software systems, in: Proceedings of 9th International Symposium on Software Reliability Engineering, 1998, pp. 306–315.
[38] B. Kitchenham, S.L. Pfleeger, N. Fenton, Towards a framework for software measurement validation, IEEE Trans. Softw. Eng. 21 (12) (1995) 929–944.
[39] N. Fenton, Software measurement: a necessary scientific basis, IEEE Trans. Softw. Eng. 20 (3) (1994) 199–206.
[40] P. Miodonski, T. Forster, J. Knodel, M. Lindvall, D. Muthig, Evaluation of Software Architectures with Eclipse, Technical Report IESE-Report 107.04/E, Institute of Experimental Software Engineering, Kaiserslautern, Germany, 2004.
[41] M. Lindvall, C. Ackermann, W.C. Stratton, D.E. Sibol, A. Ray, L. Yonkwa, J. Kresser, S. Godfrey, J. Knodel, Using sequence diagrams to detect communication problems between systems, in: Proceedings of IEEE Aerospace Conference, March 1–8, 2008, pp. 1–11.
[42] OSGI, OSGI—The dynamic module system for Java, http://www.osgi.org (last accessed: Nov. 30, 2008).
[43] D. Ganesan, T. Keuler, Y. Nishimura, Architecture compliance checking at runtime: An industry experience report, in: Proceedings of 8th International Conference on Quality Software (QSIC'08), Oxford, UK, Aug. 12–13, 2008, pp. 347–356.
[44] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, second ed., Addison-Wesley Professional, Reading, 2003.
[45] L.C. Briand, Y. Labiche, J. Leduc, Toward the reverse engineering of UML sequence diagrams for distributed Java software, IEEE Trans. Softw. Eng. 32 (9) (2006) 642–663.
[46] W. DePauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang, Visualizing the execution of Java programs 2269 (2001) 151–162.
[47] C. Ackermann, D. Sibol, W. Stratton, M. Lindvall, S. Godfrey, An analysis framework for inter-system interaction behavior, in: Proceedings of 19th IEEE International Symposium on Software Reliability Engineering (ISSRE), Redmond, WA, Nov. 11–14, 2008.

[48] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, A. Zaremski, Recommended Best Industrial Practice for Software Architecture Evaluation. CMU/SEI-96-TR-025, Carnegie Mellon University, Pittsburgh, USA, 1997, pp. 1–34.

[49] R. Kazman, M. Klein, M. Barbacci, T.H. Longstaff, L.J. Carriere, The architecture tradeoff analysis method, in: Proceedings of International Conference on Engineering of Complex Computer Systems (ICECCS98), 1998, pp. 68–78.

[50] R. Kazman, J. Asundi, M. Klein, Quantifying the costs and benefits of architectural decisions, in: Proceedings of 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Canada, 2001, pp. 297–306.

[51] L. Dobrica, E. Niemela, A survey on software architecture analysis methods, IEEE Trans. Softw. Eng. 28 (7) (2002) 638–653.

[52] R. Kazman, L. Bass, M. Webb, G. Abowd, SAAM: A method for analyzing the properties of software architectures, in: Proceedings of 16th International Conference on Software Engineering-Sorrento, Italy, May 16–21, 1994, pp. 81–90.

[53] P. Bengtsson, J. Bosch, Architecture level prediction of software maintenance, in: Proceedings of 3rd European Conference on Software Maintenance and Reengineering (CSMR), Amsterdam, Netherlands, 1999, pp. 139–147.

[54] J.C. Duenas, W.L. de-Oliveira, J.A. de-la-Puente, A software architecture evaluation model, in: Proceedings of 2nd International Workshop On Development and Evolution of Software Architectures for Product Families, Spain, 1998, pp. 148–157.

[55] M. Shereshevsky, H. Ammari, N. Gradetsky, A. Mili, H.H. Ammar, Information theoretic metrics for software architectures, in: Proceedings of International Computer Software and Applications Conference (COMPSAC 2001), Chicago, IL, USA, 2001.

[56] M.A. Babar, L. Zhu, R. Jeffrey, A framework for classifying and comparing software architecture evaluation methods, in: Proceedings of Australian Software Engineering Conference, Melbourne, Australia, April, 2004, pp. 309–319.

[57] R. Kazman, L. Bass, M. Klein, T. Lattanze, L. Northrop, A basis for analyzing software architecture analysis methods, Softw. Qual. J. 13 (4) (2005) 329–355.

[58] M. Svahnberg, C. Wohlin, L. Lundberg, M. Mattsson, A quality-driven decision support method for identifying software architecture candidates, Int. J. Soft. Eng. Knowl. Eng. 13 (5) (2003) 547–573.

The Common Law and Its Impact on the Internet

ROBERT AALBERTS

Department of Finance, University of Nevada, Las Vegas, 4505 Maryland Parkway, Las Vegas, Nevada 89154, USA

DAVID HAMES

Department of Management, University of Nevada, Las Vegas, 4505 Maryland Parkway, Las Vegas, Nevada 89154, USA

PERCY POON

Department of Finance, University of Nevada, Las Vegas, 4505 Maryland Parkway, Las Vegas, Nevada 89154, USA

PAUL D. THISTLE

Department of Finance, University of Nevada, Las Vegas, 4505 Maryland Parkway, Las Vegas, Nevada 89154, USA

Abstract

This chapter begins with a discussion of the English common law, the basis for much of American law today, and its evolving role in regulating the cyberworld. The common law, based on case precedents, has proven to be very adept at resolving new and difficult problems, at times more quickly and competently than legislation. This same mechanism is being used today to solve problems to protect Internet users from destructive "cyberevils" such as spamming and other kinds of cybertorts. In the second part of our chapter, we contrast the cases of *Doe v. XYC Co.* and *Delfino v. Agilent* to illustrate the application of common law doctrines in managing risks arising from employees' use of the Internet.

ADVANCES IN COMPUTERS, VOL. 77
ISSN: 0065-2458/DOI: 10.1016/S0065-2458(09)01208-X

1. Introduction

When the Normans invaded England in 1066 AD, the bow and arrow was the most high-tech hardware they possessed. Still, the Normans leveraged this technology to vanquish its Anglo-Saxon enemy and impose a new legal regime called the common law. Today, we look to the same system to provide us with both a sword and a shield to thwart spam, adware, spyware, and other enemies lurking in a dangerous virtual world.

Still, how can such an ancient legal system offer protections in a nonphysical world the Normans could never have imagined? The common law legal system was devised in the eleventh century in an environment of great fear and repression. Few rules were in place. Yet, it has survived for many centuries and has evolved into arguably the most practical, adaptive, and functional legal system the world has ever seen. Indeed, in a seminal study by LaPorta et al. [1], in which they examined 49 countries with both common law and civil law legal systems, the authors concluded that the flexibility afforded by the common law, which uses independent judges and juries and applies law on a case-by-case basis, creates an environment in which there

are stronger property rights, less corruption, and more efficiency.[1] Moreover, the common law's flexibility, versus the rigidity of statutory and regulatory law used exclusively in the civil law system, is measurably better at creating new rules for protecting investors and other stakeholders.

The common law's importance in protecting activities on the Internet cannot be understated. The Internet is one of the greatest innovations and generators of wealth the world has ever seen. However, to be truly effective it cannot function properly in a lawless environment. Property rights need to be protected or the incentives necessary for the creation of wealth will be undermined. The common law has proven for centuries that it is up to the challenge.

This chapter will first explore how and why this thousand-year-old legal system has risen to such towering accomplishments and where it will likely evolve in its new role of protecting the cyberworld. The second part will discuss a case study concerning the liability of employers for the wrongful acts of their employees on the Internet at work. The aim of this section is to demonstrate how the courts applying the common law of two states, New Jersey and California, met the challenge. The lesson that was learned from these two cases is that the common law creates rights and duties that the users of the Internet must understand to protect their personal and property interests.

1.1 A Brief Primer on the Common Law

The common law's foundation was based on a very simple but inspired premise. After the successful invasion of England, a succession of Norman kings sent their most trusted administrators, later called judges, out into the shires or counties of the newly vanquished English realm. Here, they were instructed to set up a system to manage and pacify this new but very hostile frontier.

The judges quickly learned that pursuing a peaceful approach to settling disputes could avoid bloody uprisings and conflicts. They discovered that the local Anglo-Saxon inhabitants, in many cases, already possessed customs and common sense solutions that they could successfully draw upon to solve many local problems as well as to placate those who may pose a threat to them in the future.

Indeed, today much like the approach used by the old common law judges, the idea of listening first to those who are aggrieved and then hammering out a solution based on evolving common law principles has and will continue to solve many of the

[1] In the civil law system, legislation is primary source of law and courts base their decisions on provisions of the relevant codes and/or statutes. The civil law system is used in continental Europe and much of Latin America, East Asia, and Africa. In the common law system, cases have traditionally been the primary source of law and courts base their decisions on prior case law. The common law system tends to be used in countries, like the U.S., whose legal traditions originated in Great Britain.

Internet's most dangerous threats. After all, the Internet is, if anything, a new frontier marked by conflict and in a constant state of change. The more devious inhabitants of cyberspace (hackers, spammers, etc.), much like the local and hostile Anglo-Saxons of the eleventh century, are not only clever and destructive, but can regroup and strike quickly. The common law came into being as a system to do precisely what we are trying to do today in pacifying the Internet—manage a threatening and hostile adversary in a new, albeit nonspatial, environment called cyberspace.

The common law's flexibility offers a number of advantages for managing cyberspace. The judge-made, common law can, in many instances, not only react faster, but has often performed historically more competently than legislatures. This has especially been the case in confronting the unpredictable threats posed by new technologies. Legislative bodies, such as Congress, are slow and often riddled with special interests. As a result, these elective bodies are reluctant to address issues that are too politically sensitive or are overly influenced by special interests. Moreover, the laws they produce, in the form of statutes and codes, are often narrow, as well as vague. Statutes can take years to decipher and often yield unintended consequences. Courts, on the other hand, are able to intervene and apply well-settled, centuries-old rules of law (i.e., trespass, nuisance, etc.) to the specific facts arising in modern conflicts. This allows courts the flexibility of adapting long-standing and well- understood legal principles to the many new problems created by modern technological developments.

The structure of the common law can also bestow the benefits of learning from the past while constructing a better-managed future. This key attribute can help guide the Internet. One of the bases of the common law is the deference given to precedents. Once a precedent is established other judges generally must, because of the doctrine of *stare decisis*, apply these precedents. Indeed, the common law has enjoyed widespread credibility and functionality due to its strong adherence to *stare decisis*. Courts are able to impart certainty, fairness, and predictability upon those who are seeking their guidance. Of course, judges can overrule established precedent. But if they do, they must be prepared to articulate compelling socioeconomic, political, or technological reasons. Overruling precedent, in the words of Chief Justice of the Supreme Court John Roberts during his confirmation hearings, inflicts a "jolt" on the legal system [2]. Tradition and practicality dictate an almost sacred respect for this venerable doctrine.

1.2 Common Law Actions and the Internet

In recent years, the common law has protected Internet users from a variety of threats and intrusions often originating from the broad sweep of tort law.[2] We will begin by discussing one very old common law action in tort called trespass to chattels

[2] A tort is defined by West's Legal Dictionary as "A civil wrong (other than a breach of contract) that has caused harm to person or property."

and then explain its present legal utility to managing the Internet. After that, we will turn to another venerable common law action, nuisance. One important doctrinal development concerning the applicability of trespass to chattels and nuisance to the Internet will also be presented to demonstrate the dynamic and flexible nature of the common law. To this end, our chapter will hopefully reveal how the common law will, in a number of key instances, offer some of the protections for combating the Internet problems of the future.

1.2.1 Trespass to Chattels

The common law action of trespass to chattels[3] was created centuries ago to battle those who were transporting or carrying off another's personal property. Often, this was a farmer's livestock—hence the word chattel, the Norman French word for cattle. Today, it encompasses all direct interferences with another's personal property [3].

For years, trespass to chattels was overshadowed by its more popular legal brother—conversion[4]—which often afforded better monetary relief. The Internet, however, has spawned a legal renaissance for this long ignored action. As one commentator notes: "[Trespass to chattels is] a centuries-old. . .theory that languished for years in the dusty archives of obscure legal doctrine learned and then promptly forgotten in the first year of law school, which has unexpectedly found new life courtesy of the Internet" [4].

The efficacy of trespass to chattels in combating Internet incursions was signaled in a 1996 case titled *Thrifty-Tel, Inc. v. Bezenek* [5]. In *Thrifty-Tel*, the defendants were accused of "phreaking"—the practice of cracking a phone network, usually as a way of making free long-distance calls. Since existing state or federal statutes did not, at least at that time, address phreaking, the court was forced to rely on the common law. Importantly, the court ruled that the flow of electrons was sufficiently tangible to constitute physical contact, a required element of a trespass to chattels.

The significance of *Thrifty-Tel* cannot be understated. Just 1 year later, in *CompuServe, Inc. v. Cyber Promotions, Inc.* [6], an Ohio court followed the same logic but applied it for the first time to spammers. The court argued that science now afforded the opportunity to quantify what was, in the past, not possible. This includes gases, shockwaves, and particulates. Furthermore, the electronic message sent through CompuServe's ISP by Cyber Promotions caused harm to CompuServe's ISP because

[3] Trespass to chattels is defined by West's Legal Dictionary as a "Tort with the following elements (a) personal property-Chattel, (b) plaintiff is in possession of the chattel or is entitled to immediate possession, (c) intent to dispossess or to intermeddle with the chattel, (d) dispossession, impairment, or deprivation of use for a substantial time, (e) causation of the dispossession, impairment, or deprivation."

[4] Conversion is defined by West's Legal Dictionary as the "unauthorized exercise of dominion or control over someone's personal property (chattel)."

the "multitudinous electronic mailing" demands the disk space and drains the processing power of plaintiff's computer equipment. This, in turn, caused "the value of that equipment to CompuServe [to be] diminished even though it is not physically damaged by defendants' actions" [7].

Possibly the most important trespass to chattels case involving the Internet was *eBay v. Bidder's Edge, Inc*. [8]. In *eBay* the defendant Bidder's Edge infiltrated eBay's Web site with search "spiders" which "screen scraped" eBay's Web site and placed the information on its own Web site. eBay argued that Bidder's Edge caused its Web site to lose the use of a section of its computer space and therefore its efficiency. This, it claimed, constituted a trespass to chattels. Although Bidder's Edge scraped a relatively small load of 1.53% off of eBay's listing servers, the court maintained that if its actions were not stopped, more companies would do the same. In the aggregate, the court argued, "eBay would be brought to its knees by what would be then a debilitating load" [8]. Trespass to chattels, however, hit a jurisprudential detour a few years later. While some may view this as a weakness, the dynamic that unfolded further showed how adaptive the common law can be as it solves the complexities of the cyberworld. In the now famous case of *Intel Corp. v. Hamidi* [9], a disgruntled Intel ex-employee sent out, over a 2-year period, in excess of 30,000 inflammatory and uncomplimentary emails about his former employer to Intel employees. Hamidi's behavior, while troublesome, did not reach the point where he could be prosecuted under existing statutes. He did not, for example, breach computer security barriers. When asked, he obligingly removed recipients from his mailing list. His barrage of emails also did not cause physical damage or a disruption of Intel's computer system nor were employees kept from using their computers. The havoc that Hamidi wreaked was more pedestrian. His emails sparked a swell of water cooler talk setting off productivity and morale issues at Intel. Apparently Hamidi's email struck some sensitive nerves at the chip giant.

What the *Hamidi* case did trigger legally, however, is a repositioning of the common law concerning a trespass to chattels and the future emergence of the common law action of nuisance as a legal weapon. Under California's common law, trespass to chattels requires that there must be a showing of actual damage to Intel's servers, such as slowing it down or causing a loss of a quantifiable amount of its system. This, as stated, did not occur in the *Hamidi* case.

1.2.2 Will Nuisance Emerge as a Viable Action in Internet Law?

The outcome of the *Hamidi* case was not completely unanticipated, at least by legal scholars and commentators. Before the decision was issued, some worried that trespass to chattels, especially after the *eBay* and *CompuServe* rulings, might be traveling in an

unreasonable and dangerous direction. For example, how chilling would it be to Internet activity if anyone who sends out information that any recipient thinks is spam, could be sued? These kinds of questions were taken seriously in the *Hamidi* decision. Undoubtedly, other courts felt the same need for an alternative action.

The early, but decidedly savvy common law lawyers were, if anything, able to seek out alternative methods for solving their clients' troubles. They were some of the earliest practitioners of arguing fine distinctions and analogies for advancing their cases. Times have not changed. The narrowing of trespass to chattels has only opened up the action of nuisance as another means for solving Internet woes. Interestingly enough, it was the *Hamidi* court that has helped this process along.

Nuisance, like trespass to chattels, is also a very old common law action. It is invoked when one landowner unreasonably interferes with the use and enjoyment of another's land.[5] Unlike a trespass, however, it does not require an actual physical invasion. Invisible incursions, like foul smells, fumes, and electromagnetic energy, can constitute a nuisance.

Possibly the greatest contribution that nuisance may offer the Internet is a method for determining fair outcomes. Nuisance requires a balancing of the extent of the harm caused versus its social utility as a means to determine whether it is reasonable to outlaw a purported nuisance. Or as one economic commentator explains "Unreasonableness alternatively may exist if the activity is meritorious but the defendant fails properly to internalize the costs of his activity, thereby imposing a negative externality on society in addition to whatever social utility his activity provides" [10].

The *Hamidi* court consistently hinted that a burden/benefit approach may be the best way to counter the various injurious Internet activities—especially those that cannot be quantified. For example, the court noted that "Creating an absolute property right to exclude undesired communications from one's email and Web servers might help force spammers to internalize the costs they impost on ISPs and their customers. But such a property rule might also create substantial new costs, to email and e-commerce users and to society generally, in lost ease and openness of communication and in lost network benefits" [9]. For these reason, a case-by-case analysis is likely the best method for making these determinations.

1.2.3 The Future of Pop-up Ads, Spyware, and Adware

Pop-up ads are quite possibly the most irritating features we face on the Internet. Can trespass to chattels or nuisance take on these annoyances? One argument is that pop-ups, including those embedded on our hard drives by spyware and adware, can

[5] Nuisance is defined by West's Legal Dictionary as the "an unreasonable interference with the use and enjoyment of land."

slow down a system or even cause a computer to crash. This could occur, for example, when a user's project takes up most of his computer's processing power, memory, and hard drive space only to be lost by the uninvited arrival of pop-ups, not to mention spiders, Webcrawlers, and robots. In such a case, valuable data would be lost if the system crashes. However, such an outcome is probably rare. Thus, nuisance may be a better bet.

1.2.4 Spam and Electronic Nuisance

Although nuisance has not been directly applied to spam, it could likely provide computer users with legal protection. And even though it has been traditionally tied to invasions to real property, nuisance has already been successfully invoked in at least two cases—one involving a series of unwanted and intrusive phones calls and another when someone caused another to have electronic disturbances to his TV reception. Thus, interference with personal property located in, say, a house can be a nuisance to its inhabitants. Expanding the doctrine to computers in homes is a foreseeable extension.

As discussed, nuisance employs a burden/benefit framework to determine its social utility. Assuming that spam is defined as illegitimate commercial solicitations, spam received over time can be not only highly annoying, but is often used to sell fraudulent products and services. Moreover, it may even threaten the proper functioning of a system as well as individual computers. Its benefit, of course, is economic. Spam helps unscrupulous individuals make huge amounts of money—a decidedly small amount of social utility.[6]

1.3 The Common Law Versus Statutes

State and federal legislative bodies have, of course, addressed some of these Internet evils. There can be problems with statutes, however. Statutes are typically created to target a specific problem raised by special interest groups ranging from corporations to consumer protection groups and then brought to the attention of lawmakers. A good example of this is Congress' 2003 CAN-SPAM Act. The Act outlaws unsolicited commercial email sent by illegitimate marketers. However, it has been criticized as ineffective for several reasons. First, a private person cannot

[6] Based on data from the National Technology Readiness Survey [11], Aalberts, Poon, and Thistle [12] estimate the cost of the approximately one trillion spam emails sent in 2004 at $22 billion. This includes the cost of sending spam and the value of time spent dealing with spam. Since 4.7 million adults purchased a product or service advertised in spam emails, they would have to value their purchases at over $4600 per person in order for the benefits of spam to outweigh the cost.

use it; only the FTC, state authorities and ISPs can. Second, it preempts similar state laws, some of which were in place and were stronger and arguably better. Lastly, Congress, for whatever political reason, furnished spammers with an enormous break by incorporating an opt-out provision which implies consent to receive spam unless the recipient states otherwise. Of course, the task of opting out of thousands of spam messages can be, if anything, daunting. In addition, the act of opting out verifies that the email address is valid and active, which almost certainly leads to even more spam in the future. Apparently, Congress, with perhaps the exception of well-heeled lobbyists, does not normally listen as well as a judge might to those who have something valuable to offer.

2. The Common Law in Action: Employer Liability to Third-Party Victims on the Internet

To demonstrate the common law's importance for settling ongoing disputes that arise in cyberspace we present two quite recent cases, one from New Jersey, *Doe v. XYC Co*. [13] and the other, *Delfino v. Agilent Technologies, Inc*. [14] from California. These cases reveal the increasing importance of managing employees properly in their use of the Internet and both concern well-settled common law principles that have been applied to resolve disputes occurring in the physical realm for centuries.

2.1 A Study in Failure to Protect Third Parties: The XYC Corp.

Most ISP supervisors are aware they possess legal *rights* to monitor employee Internet use. Laws, such as the federal Electronic Communications Protection Act and others, generally allow employers great leeway to monitor [15]. Yet, as mentioned, statutes can only go so far, often lacking the flexibility to react to unique but important disputes that may arise both in the physical and cyber realm. One such development arose in a 2005 New Jersey decision, *Doe v XYC Corp*. This case created a legal *duty* to monitor workers properly and reasonably in order to protect third parties. The case, the first of its kind, has subsequently sent shockwaves as well as garnered support throughout both the legal and business communities [16].

The compelling facts in *Doe* portend just how far-reaching the case and the duty it created may become. XYC's problems began when employees lodged complaints concerning a fellow employee, Doe, who was apparently accessing and viewing pornographic sites on his company computer. To verify the information, the company's Senior Network Administrator (SNA) checked Doe's computer logs

revealing sites with highly suspicious names such as "bestiality" and "necrophilia." Sensing an obvious legal and moral problem looming, the SNA confronted Doe and ordered him to stop visiting "inappropriate sites." Doe, however, continued to disobey the order. Sensing his lack of cooperation, the SNA and Doe's immediate supervisor continued to probe and found evidence of similar sites. In response the SNA went directly to the Director of the Network and PC Services (the Director) requesting an investigation. At this point, the Director made a significant mistake. Instead of investigating the allegations, she "admonished" the SNA and told him never to access any employee's Internet activity in the future. In fact, the SNA was told that violations of this policy could result in his job loss. Doe, unaware of his supervisor's latest discovery, continued to access the illicit sites. Ironically, the company also had a second Internet use policy that allowed the accessing and review of its employees' sites if it was business related.

Doe's suspicious behavior continued to agitate his coworkers. Some caught him, for example, shielding his computer and quickly minimizing images, as well as leaving provocative pictures inadvertently on his screen. Finally, in reaction to the latest developments and in apparent defiance of the Director's injunction, Doe's immediate supervisor entered his cubicle while he was at lunch and clicked on his "Web sites visited." He discovered a number of probable child pornography sites including one with the title "Teenflirts.org: The Original Non-Nude Teen Index." Thereafter, the supervisor, with permission from his superiors (who were not involved with the Director) told Doe to quit his unlawful Internet activities. He agreed to this second demand, yet defiantly continued accessing the sites.

Doe's behavior continued for approximately 2 years after the initial complaints by coworkers. Eventually, company supervisors notified the police, who found nude photographs of Doe's own 10-year-old stepdaughter in the company dumpster. These were the same pictures that he had sent out as "payment" for access onto the child pornography sites. The discovery of the pictures formed the basis for a search warrant of his office and computer in which an additional 70 downloaded pictures were discovered, including more pictures of his stepdaughter.

Doe's ex-wife subsequently sued XYC for failing to investigate and protect her daughter. XYC prevailed at the trial level in a motion for summary judgment, but the appellate court overruled it stating: "...[the] defendant had a duty to report Employee's activities to the proper authorities and to take effective internal action to stop those activities." The court further maintained that: "Defendant was under a duty to exercise reasonable care to stop Employee's activities, specifically his viewing of child pornography which by its very nature has been deemed by the state and federal lawmakers to constitute a threat to 'others'..." The court's use of the phrases "duty to report" and "duty to exercise reasonable care" are particularly significant.

The case was subsequently sent back to the trial court to determine if the victim had suffered psychological harm.[7]

2.2 *Delfino v. Agilent Technologies*: A Case of Competence

In contrast to XYC's botched ISP management, Agilent Technologies performed competently thereby avoiding the kinds of legal problems XYC suffered. Perhaps more importantly for ISP managers, Agilent's prudent crisis management may serve as a valuable common law precedent to counter the heightened legal responsibilities the *Doe* case may have created.

In *Delfino*, the plaintiffs Michelangelo Delfino and Mary Day received a series of threatening messages, as well as postings on a message board, from a source with the screen name of "Crack_smoking_jesus." In fact, an Agilent employee named Cameron Moore was sending the messages to apparently harass and intimidate the plaintiffs due to litigation pending against him instigated partly by the plaintiffs. Ultimately, it was discovered that some of Moore's threats had been sent from work. For this reason, the plaintiffs also sued Agilent for, among other actions, negligent supervision and retention of Moore.

Agilent first learned of the threats against the plaintiffs when an FBI special agent requested information on an IP address that originated from Agilent. Agilent's IT personnel quickly agreed to cooperate with the FBI and succeeded in tracing the threats to Moore. When the Agilent ISP personnel confronted him with the information, Moore apologized, but contended that no threats had gone through Agilent's computer systems. He was told to agree *in writing* to never engage in this kind of activity again. Agilent management then gave Moore a "stern warning" but acknowledged that they had no proof that any of the threatening emails had gone through its system. Moore was also reminded that the company's Standards of Business Conduct does not allow employees to use company systems for personal reasons.

After several more months of investigation, the FBI told Agilent that it was about to arrest Moore. Agilent's management inquired whether the arrest was related to Moore's use of Agilent systems. The FBI assured them that it was not. Still, Agilent did not put the matter to rest but continued to stay on top of the investigation. It asked the FBI for its arrest affidavit and continued to interrogate Moore. During the latest line of questioning, Moore admitted that he had sent, while at work, emails that "weren't nice and could be interpreted as threats." After the admission, Moore

[7] In a 8 May 2006 interview in the Lawyers Weekly USA with the lead counsel who represented the plaintiff's ex-wife and child, it was revealed that the case was privately settled.

was put on administrative leave and several days later terminated for "misuse of Agilent's assets."

The facts in the Doe case are a good example of how a company can become liable through the fault, in this case negligence, of its supervisory personnel for failing to intervene competently once they become aware of a potential legal problem. The Delfino case, on the other hand, demonstrates that when a company's management quickly and adeptly reacts to a problem, they can avoid liability. The following discussion delves into the law regarding the supervision of employees and how the court resolved both cases.

2.3 Employer Liability Without Fault—The Doctrine of Respondeat Superior

In both cases, it is apparent that the employees were *not* engaged in activities that were within the normal scope of their workplace duties. And because they were not, neither XYC nor Agilent was found liable under the theory *respondeat superior* (see Endnote). Still, an understanding of this doctrine is important to understand how companies can be found liable for their employees' wrongful acts. Under *respondeat superior*, employees' wrongful acts or torts impute liability onto their employers [3]. Simply put, liability is assigned to the employer (usually a business entity such as a corporation or a limited liability company) even if it (through its management and supervisory personnel) did not approve or consent to the employee's particular act. Still, the employee's acts must, as mentioned, occur within the scope of the employee's workplace duties. For ISP and other management personnel, the doctrine is particularly worrisome since their companies become *strictly or vicariously liable* once these acts are committed.[8] The Restatement (Second) of Agency Section 228 provides the generally accepted definition of what constitutes the scope of employment.

The conduct of the employee is within the scope of employment if:

1. It is of the kind the employee is employed to perform.
2. It occurs within authorized space and time limits.
3. Some or all of it is done to serve the employer.
4. If the employees use force against each other [7].

[8] Strict liability is liability without fault. Under strict liability, the plaintiff only needs to prove that the tort happened and that the defendant was responsible. Vicarious liability is liability for the wrongful acts of another. Negligence is behavior that falls short of what a reasonable person would do to protect others from foreseeable harm. Under negligence, the plaintiff must prove that the defendant owed a duty of care to the plaintiff, that the defendant breached that duty, that the breach of duty was the proximate cause of harm to the plaintiff, and that the plaintiff suffered injuries to his person or property. See generally West's Legal Dictionary.

Although the doctrine has existed for decades, it remains controversial since the employer is liable even after its management personnel exercises due diligence in selecting and supervising an employee who subsequently commits an illegal act.

In contrast to the foregoing, in which employees are engaged in the course and scope of their jobs, actions in which the employer would not be liable under this doctrine are those that are "purely motivated by personal interests or are outrageous in nature. . ." Exceptions to these may occur when the "employee harms another because of the opportunity that the job offers" [19]. This class of employer liability could arise due to an employee's negligence or even when he intentionally harms another such as when an employee defrauds a third party on the job to enrich himself [3].

2.4 Negligent Supervision and Retention of Employees

Since the errant employees in both the *Doe* and *Delfino* cases were not engaged in activities that fell within the scope of their jobs, the pertinent legal issue was whether their employers had negligently supervised and retained these employees. This means that even if employees on the job are not acting within the scope of employment or are not furthering their workplace duties, the employers themselves can be negligent for the hiring, supervision and retention of dangerous or careless employees. This means that an employer will *not* be subject to strict or vicarious liability like they would be under the doctrine of *respondeat superior*[3], but directly could still be liable for negligence. This theory is pertinent since, as it will be discussed below, XYC was found to be negligent in its supervision/retention of Doe, while Agilent was not in respect to how it managed Moore.

2.5 Intentional Harm on the Internet

Employers may also incur liability when their employees engage in intentionally harmful acts at work. If an employee intentionally injures another's person or property, the employer can be liable if it is "reasonably connected with the employment as to be within its 'scope'" [3]. An exception to this occurs if the employees' motives, for example, are "purely personal," that is, are "unprovoked, highly unusual, and quite outrageous" [3]. Even under this scenario a company's liability can attach if management knew or should have known that the employee would act in such a personal or outrageous way. An example of this occurs when an employee, such as a bouncer, possesses known dangerous and aggressive behaviors, and then injures someone while on the job.

Both Doe and Moore engaged in intentional acts that hurt others outside their companies. Although the case, as mentioned, was remanded to the trial court to determine what injuries his stepdaughter may have suffered, Doe's intentional transmission of "kiddie porn" is considered under the law to constitute a threat to others.[9] Moore was accused of a tort known as the intentional infliction of emotional distress, among other acts.

2.6 Cybertorts

Cybertorts are torts committed in cyberspace [17]. The legal environment surrounding cybertorts is complicated and evolving. For example, in the first part of this chapter, we explored the torts of trespass to chattels and nuisance and how they pertain to the Internet. Now the legal duty imposed on employers, after *Doe* in particular, may significantly expand the legal landscape for these and other kinds of torts committed in the workplace.

In 1996, the law surrounding ISP liability for cybertorts was greatly clarified when Title V of the Telecommunications Act of 1996, better known as the Communications Decency Act (CDA) was passed. Under the CDA, Section 230 Congress shielded commercial ISPs from civil liability should they fail to remove or block tortious activities as long as the ISP does not actually have input in the creation of the offensive material. The law also protects ISPs that attempt to block and screen offensive material under the so-called "Good Samaritan" exception. Initially the law was passed to protect ISPs from defamation but has since been expanded to include virtually all tort liability. With ISP immunity, which includes other intermediaries such as Web sites and online information content providers, successful prosecution of cybertort activities has generally been thwarted, most often because the victims are unable to locate and sue the victimizers. Still, even if they can be located, they are usually not "deep pocketed" corporate defendants who are heavily insured and able to pay off large judgments [11]. Thus, cases like *Doe* and *Delfino*, in which economically viable corporations are being sued, will likely increase.[10]

[9] The duty to report child pornography once it becomes known is required under federal law at 42 USC Section 13032(b) and imposes sanctions on ISPs at 42 USC Section 13032(b)(4).

[10] It is noteworthy that XYC Company did not assert that it had immunity as an ISP under the broad protections of the Communications Decency Act, 47 USC Section 230. The court in Delfino, on the other hand, did argue for CDA immunity successfully. Not all employers, however, own and operate their ISP's and so may not be protected by this provision. Moreover, immunity can be lost in certain circumstances. For example, in a potentially influential case from the 9th Circuit, *Fair Housing Council of San Fernando Valley v. Roommates.Com, LLC*, 2008 WL 879293 (9th Cir., 3 April 2008), an ISP lost its immunity under CDA Section 230 when it became a non-neutral "information content provider." *Roommates.com* contains facts different from both the *Doe* and *Delfino* cases discussed in this chapter. Moreover, as stated above, there is an affirmative duty to report child pornography once an ISP operator becomes aware of it.

The law concerning employer liability is complex and a full discussion is beyond the scope of this chapter. However, the main legal issues can be summarized with the aid of Fig. 1. The first question is whether the individual is an independent contractor or an employee (Node A in Fig. 1). Employers are not strictly liable for the acts of independent contractors, but may be liable for negligent selection or retention of the contractor (Node B). If the individual is an employee, the next question is whether the employee is acting within the scope of employment (Node C). If the employee is acting within the scope of employment, as described in the Restatement (Second) of Agency, then the employer is strictly liable. If the employee is not acting within the scope of employment, other issues become important (Node D). The employer may be liable for negligent supervision and retention of the employee (Node E), if the employee intentionally harms a third party (Node F) or if the employee commits cybertorts (Node G).

3. Why Doe Lost and Delfino Won—A Case of Risk Management

Even though both XYC Co. and Agilent were sued under the same cause of action—negligent supervision and retention—the differing outcomes are easy to understand.

3.1 Why the Court Said *XYC* Was Liable

The *Doe* case illustrates the harm that can be caused by highly imprudent behavior both in the way a company generally manages its workers, and its failure to effectively respond to trouble.[11] One of XYC's biggest mistakes was the confusion caused by having two computer use policies. One policy was well-distributed and specifically stated that emails were the company's property and should not be considered confidential. The policy also stated that anyone aware of the "misuse of the Internet for other than business reasons was to report it to Personnel" [14].

[11] How influential the *Doe* case will be on future common law courts is highly speculative. Although it is important to point out that, while a company like XYC may not be liable for the contents posted on its ISP if it qualifies under the CDA immunity, once an employer does learn of employee wrongdoing, at least under the reasoning of *Doe*, it takes on a legal duty to supervise and retain its employee in a non-negligent manner. One case, besides *Delfino*, has also found employers not liable for the wrongful acts of employees on the Internet. In that case, *Booker v. GTE.Net LLC*, 214 F. Supp.2d 746 (E.D. Ky. 2002), the employer GTE.Net, much like Agilent Company, was considerably more careful in how it handled the wrongful acts of its employees than the XYC Company.

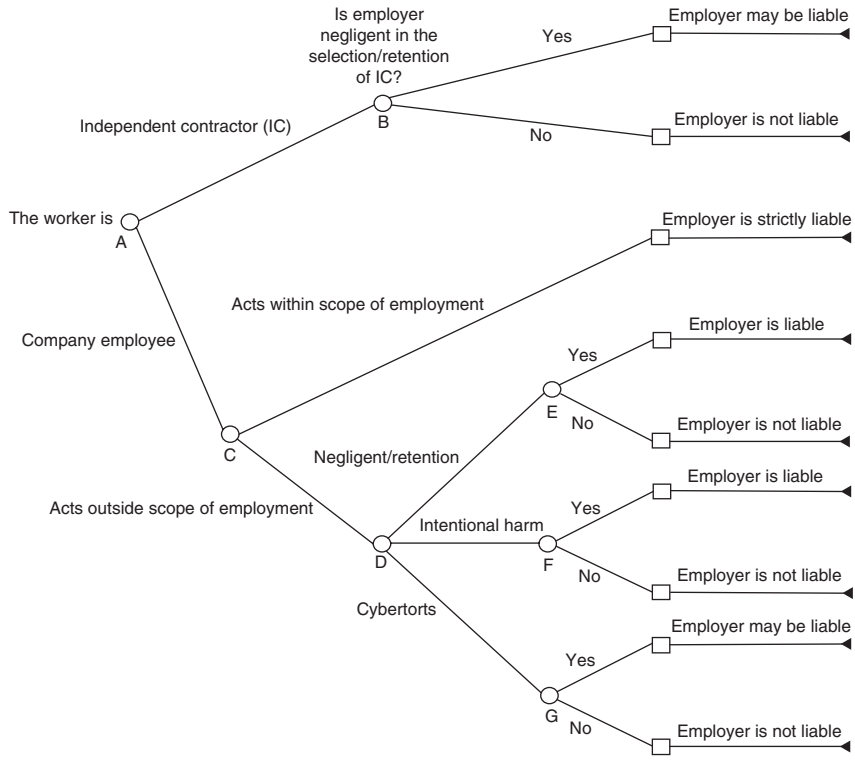

FIG. 1. Employer liability for the actions of its employees on the Internet.

Yet, at the same time it had another company policy, communicated by email, prohibiting the monitoring of employee computer usage.

The inconsistency prompted the court to rule that the first policy, in which XYC reserved rights, conflicted with the privacy rights it conferred to their employees under the second policy. In effect the court decided that the former policy negated the latter stating that "[d]efendant [XYC Co.] recognized its right to monitor employee Web site activity and emails by promulgating and distributing a policy to that effect during the relevant time period" [13]. Moreover, the court explained, XYC produced its own duty to monitor but then failed to carry it out properly. Consequently, once management had notice of Doe's dangerous actions, it had a "...duty to investigate the employee's activities and to take prompt and effective action to stop the unauthorized activity, lest it result in harm to innocent third parties" [13].

The *Delfino* case clearly demonstrates how a prudent policy that is clear and expeditiously enforced can save a company from civil liability. *Delfino* is significant, not only as a positive Internet management model for ISP personnel, but also because it suggests that a company, like Agilent, can be shielded as a "provider or user of an interactive computer service" under CDA Section 230. What is important to note is the court's statement that "even if the [CDA] immunity did not apply" Agilent still would not have been liable for the torts the plaintiffs alleged. These actions help us to better understand and apply the lessons the case offers. It also may benefit other ISP supervisors in the future who may not presently enjoy this special immunity, since it is currently a binding precedent only in a portion of California.

3.2 Why the Court Said Agilent Should Not Be Liable

The *Delfino* court gave three reasons for exonerating Agilent for the negligent supervisor/retention of Moore. First, the court stated that Agilent owed *no* duty of care (a required element in proving negligence) to the plaintiffs. Although there are a number of factors a court looks at to determine whether a duty exists, several of the factors were considered of particular importance for Agilent. One was that Agilent, despite its careful procedures, had no prior notice that Moore was harassing the plaintiffs. Foreseeability of harm is very important in creating a legal duty. XYC, on the other hand, experienced ample opportunities to foresee the kind of harm Doe might inflict on someone. Moreover, the court explained, Agilent should shoulder no "moral blame," another factor, since it had promulgated a clear, consistent policy for discovering and thus preventing this sort of activity. A *tougher* policy meant to prevent harm might have even, according to the court, resulted in a "chilling effect" and "extreme employer oversight of employee's [Internet] activities." This language suggests that Agilent did what it was supposed to do, but not to excessive lengths. The court additionally argued that imposing "a duty to the world for all acts of its employees" even when some are not business related, would be too burdensome. The court concluded by maintaining that such a risk, one that is an "unknown malicious act of an employee bearing no relationship to his job," is not likely to be insurable. Courts, it noted, have been very reluctant to impose an uninsurable duty on employers [14].

4. Conclusion

The common law has been called upon for centuries to settle some of the most troublesome problems confronting individuals, businesses, and government. Now, some of these problems have morphed from those occurring in physical space to those creating havoc in cyberspace.

These "cyberevils" range from such cybertorts as defamation, "cyberassment" (including Webjacking, spoofing, cybersquatting, denial of service attacks or email bombs, sending viruses, cyberbullying, sexual harassment, etc.) [18], intentional infliction of mental distress, as well as spamming, trespass to chattels (personal property) mentioned in the first part of this chapter. Unfortunately, all of these will continue to occur. Then again, the common law will likely be up to the task of addressing the problems as they arise and helping victims to be compensated for losses they may incur.

The *Doe* case, discussed in the second part of this chapter, portends the potential for large-scale harm that can befall ISP personnel who fail to manage prudently while also giving guidance to those who follow Agilent's example. Both cases were cases of first impression, that is, there were no previous cases or precedents for the courts to follow. Given the general paucity of Internet cases and the doctrine of *stare decisis*, they will likely be influential on courts in the future.

In the end, management personnel must be aware of the law and what it can offer both as a sword and as a shield in combating the cyber enemies who may impair property and harm people. Those, for example, who chose to imitate Agilent's management approach can feel confident relying on that legal outcome, while companies, like XYC, which retain malicious employees who harm others outside the company and possess ineffectual management policies and personnel, may become vulnerable to lawsuits and expensive judgments. These two cases, both the product of the common law, will likely influence the cyberworld of tomorrow.

Endnote Legal Definitions*

Conversion: The unauthorized exercise of dominion or control over someone's personal property (chattel).

Negligence: Behavior that falls below what the average reasonable person would do to protect others from foreseeable risks of harm. The plaintiff must prove that the defendant owed a duty of care to the plaintiff, that the defendant breached that duty, that the breach of duty was the cause of harm to the plaintiff, and that the plaintiff suffered injuries to his person or property.

Nuisance: An unreasonable interference with the use and enjoyment of land.

Respondeat superior: The liability of an employer for the wrongful acts of his employee.

Stare decisis: To abide by or adhere to decided cases.

Strict liability: Liability without fault.

Tort: A civil wrong (other than a breach of contract) that has caused harm to person or property.

Trespass to chattels: Tort with the following elements: (a) personal property-Chattel, (b) plaintiff is in possession of the chattel or is entitled to immediate possession, (c) intent to dispossess or to intermeddle with the chattel, (d) dispossession, impairment, or deprivation of use for a substantial time, (e) causation of the dispossession, impairment, or deprivation.

Vicarious liability: Liability for the wrongful acts of another.

*W. Statsky, West's Legal Thesaurus/Dictionary, West Publishing Company, St. Paul, MN, 1985 (ISBN 0-314-85305-7).

Reference Books on Internet/Cyberlaw

I. Ballon, E-Commerce and Internet Law: Treatise with Forms, Glasser Legal-Works, Little Falls, NJ, 2001 (ISBN 1-88807-94-5).

G.B. Delta, and J.H. Matsuura, Law of the Internet, Aspen Publishers, Inc., New York, NY, 2001 (ISBN 0735522197).

M.A. Lemley, P.S. Menell, R.P. Merges, P. Samuelson, Software and Internet Law, third ed., Aspen Publishers, Inc., New York, NY, 2006 (ISBN 9780735558649).

D.G. Post, P.S. Berman, P. Bellia, Bellia, Berman and Post's Cyberlaw, West Publishing Company, St. Paul, MN, 2002 (ISBN-13: 9780314166876).

D.M. Powers, The Internet Legal Guide: Everything You Need to Know When Doing Business Online, John Wiley & Sons, New York, NY, 2001 (ISBN 0471164232).

D.W. Quinto, D. Desai, Law of Internet Disputes, Aspen Publishers, Inc., New York, NY, 2001 (ISBN 0735525927).

R.A. Spinello, Regulating Cyberspace: The Policies and Technologies of Control, Quorum Books, Westport, CT, 2002 (ISBN 1-56720-445-7).

Acknowledgments

The authors wish to express their appreciation to Diane Crawford and the *Communications of the ACM* for allowing us to use research and articles we have published in that journal as the basis for this chapter.

References

[1] R. LaPorta, F. Lopez-de-Silanes, A. Scheifer, R. Vishny, Law and finance, J. Polit. Econ. 105 (1998) 1113.

[2] S. Stolberg, A. Liptak, Roberts fields questions on privacy and precedents, The New York Times. com, 2005.

[3] W. Keeton, D. Dobbs, R. Keeton, D. Owen, Prosser and Keeton on Torts, fifth ed., West Publishing Company, St. Paul, MN, 1984 (ISBN 0-314-74880-6).

[4] T. Loomis, Internet trespass: companies turn to an old tort for a new reason, New York Law J. 227 (2000) 5.

[5] Thrifty-Tel, Inc. v. Bezenek, 46 Cal. App. 4th 1559, 1996.

[6] CompuServe, Inc. v. Cyber Promotions, Inc., 962 F. Supp. 1015 (ND Ohio, 1997).

[7] American Law Institute, Restatement (Second) of Agency, American Law Institute Publishers, St. Paul, MN, 2005 (ISBN 0-314-96119-4).
[8] eBay v. Bidder's Edge, Inc., 100 F. Supp. 2d 1058 (ND Cal., 2000).
[9] Intel Corp. v. Hamidi, 71 P.3d 296 (Cal. 4th Cir., 2003).
[10] S. Kam, Intel v. Hamidi: trespass to chattels and a doctrine of cyber-nuisance, Berkeley Technol. Law J. 19 (2004) 427.
[11] Rockbridge Associates, National Technology Readiness Survey, Center for Excellence in Service, R.H. Smith School of Business, University of Maryland (available at http://www.rhsmith.umd.edu/ces/ntrs.html; last checked 1 May 2008), 2004.
[12] R. Aalberts, P. Poon, P. Thistle, Trespass, nuisance and spam: eleventh century common law meets the Internet, Commun. ACM 50 (2007) 40.
[13] Doe v. XYC Co., 382 N.J. Super. 122, 887 A.2d 1156, 2005.
[14] Delfino v. Agilent Technologies, Inc., 145 Cal App. 4th 790, 52 Cal. Rptr. 3rd 376 (Cal. App. 6th Dist., 2006).
[15] L. Sotto, et al., Workplace privacy in the U.S.: what every employer should know, Practicing Law Inst. 861 (2006) 201.
[16] H. Gunnarsson, Must employers try to stop employees "unauthorized activity"? Illinois Bar J. 94 (2006) 172.
[17] M.L. Rustad, T.H. Koenig, Rebooting cybertort law, Wash. Law Rev. 80 (2005) 335–361.
[18] C.E. Smith, Intentional infliction of emotional distress: an old arrow targets the new head of the hate hydra, Denver Univ. Law Rev. 80 (2002) 1–58.
[19] L. Papa, S. Bass, How employees can protect themselves from liability for employees' misuse of computer Internet, and email systems in the workplace, Boston Univ. J. Sci. Technol. Law 10 (2004) 110.

Author Index

Numbers in *italics* indicate the pages on which complete references are given.

C

D

L

M

N

S

T

Subject Index

C

P

Contents of Volumes in This Series

Volume 42

Volume 43

Volume 44

Volume 45

Volume 46

Volume 47

Volume 50

Volume 51

Volume 52

Volume 53

Volume 54

Volume 55

Volume 56

Volume 57

Volume 58

Volume 59

Volume 60

Volume 61

Volume 62

Volume 63

Volume 64

Volume 65

Volume 66

Volume 67

Volume 68

Volume 69

Volume 70

Volume 71

Volume 72

Volume 73

Volume 74

Volume 75

Volume 76