Sudipta Kundu
Sorin Lerner
Rajesh Gupta

High-Level Verification

Methods and Tools for Verification of System-level Designs

Springer

High-Level Verification

Sudipta Kundu · Sorin Lerner · Rajesh K. Gupta

High-Level Verification

Methods and Tools for Verification of System-Level Designs

With Chapter 6 contributed by Malay K. Ganai
and Chapter 8 contributed by Zachary Tatlock

Springer

Sudipta Kundu
Synopsys Inc.
NW Thorncroft Dr. 2174
97124 Hillsboro
USA
sudiptakundu@gmail.com

Rajesh K. Gupta
Department of Computer Science
and Engineering
University of California, San Diego
Gillman Drive 9500
92093-0404 La Jolla
USA
rgupta@ucsd.edu

Sorin Lerner
Department of Computer Science
and Engineering
University of California, San Diego
Gillman Drive 9500
92093-0404 La Jolla
USA
lerner@ucsd.edu

ISBN 978-1-4419-9358-8 e-ISBN 978-1-4419-9359-5
DOI 10.1007/978-1-4419-9359-5
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011928550

Preface

Given the growing size and heterogeneity of Systems on Chip (SOC), the design process from initial specification to chip fabrication has become increasingly complex. The growing complexity provides incentive for designers to use high-level languages such as C, SystemC, and SystemVerilog for system-level design. While a major goal of these high-level languages is to enable verification at a higher level of abstraction, allowing early exploration of system-level designs, the focus so far has been on traditional testing techniques such as random testing and scenario-based testing.

This book focuses on the rapidly growing area of high-level verification. We envision a design methodology that relies upon advances in synthesis techniques as well as on incremental refinement of the design process. These refinements can be done manually or through elaboration tools. In this book, we discuss verification of specific properties in designs written using high-level languages as well as checking that the refined implementations are equivalent to their high-level specifications. The novelty of each of these techniques is that they use a combination of formal techniques to do scalable verification of system designs completely automatically.

The verification techniques fall into two categories: (a) methods for verifying properties of high-level designs and (b) methods for verifying that the translation from high-level design to a low-level Register Transfer Language (RTL) design preserves semantics. Taken together, these two parts guarantee that properties verified in the high-level design are preserved through the translation to low-level RTL. By performing verification on the high-level design, where the design description is smaller in size and the design intent information is easier to extract, and then checking that all refinement steps are correct, we describe a hardware development methodology that provides strong and expressive guarantees that are difficult to achieve by directly analyzing the low-level RTL code.

We expect the reader to gain appreciation and knowledge of the recent advances in raising the abstraction for design verification tasks. While the complexity of the problem is not lost on a typical reader, this book would ultimately present a positive outlook on the engineering solutions to the problem that the reader can use in practice.

Acknowledgments

Acknowledgment is not a mere formality but a genuine opportunity to express the sincere thanks to all those; without whose active support and encouragement this book would not have been successful.

This book would not have been possible without contribution from Dr. Malay K. Ganai and Mr. Zachary Tatlock. Dr. Ganai shared his expertise on bounded model checking in Chap. 6 and Mr. Tatlock shared his knowledge on once-and-for-all verification techniques in Chap. 8.

Significant part of this book is based on Dr. Sudipta Kundu's doctoral dissertation work. To this end, we express our thanks to Prof. Ranjit Jhala, Prof. Ingolf Krueger, Prof. Bill Lin, Dr. Yuvraj Agarwal, Federic Doucet, Ross Tate, and Pat Rondon and to all the faculty members, staffs, and graduate students of the Department of Computer Science and Engineering for their continuous help and support.

Our sincere thanks to Chao Wang, Nishant Sinha, Aarti Gupta, and all other members of the System LSI and Software Verification group at NEC Laboratories America for uncountably many interesting discussion and explanation of key concepts in Verification.

We thank Prof. Alan J. Hu for his valuable insights and comments on the initial version of the translation validation part of this book.

We also thank the members of the MESL and Progsys group for their ceaseless effort, constant encouragement, and also for the endless good times during the making of this book.

We also agree that words are not enough to express our indebtedness and gratitude toward our families, to whom we owe every success and achievements of our life. Their constant support and encouragement under all odds has brought us where we stand today.

Contents

Acronyms

ATP	Automated Theorem Prover
BDD	Binary Decision Diagram
BMC	Bounded Model Checking
CCFG	Concurrent Control Flow Graph
CDFG	Control Data Flow Graph
CEGAR	Counter Example Guided Abstraction Refinement
CFG	Control Flow Graph
CSG	Conflict Sub-Graph
CSP	Communicating Sequential Processes
CSSA	Concurrent Static Single Assignment
CTL	Computation Tree Logic
FIFO	First In First Out
FSM	Finite State Machine
FSMD	Finite State Machine with Datapath
GALS	Globally Asynchronous Locally Synchronous
HDL	Hardware Description Language
HLD	High-Level Design
HLS	High-Level Synthesis
HLV	High-Level Verification
HTG	Hierarchical Task Graph
ICs	Integrated Circuits
LLS	Language of Labeled Segments
LTL	Linear Temporal Logic
MAT	Mutually Atomic Transaction
MC	Model Checking
OSCI	Open SystemC Initiative
PEC	Parameterized Equivalence Checking
POR	Partial-Order Reduction
RTL	Register Transfer Level
SAT	SATisfiability
SMC	Symbolic Model Checking
SMT	Satisfiability Modulo Theory
TLM	Transaction Level Modeling
TV	Translation Validation

Chapter 1
Introduction

The quantitative changes brought about by Moore's law in design of integrated circuits (ICs) affect not only the scale of the designs, but also the scale of the process to design and validate such chips. While designer productivity has grown at an impressive rate over the past few decades, the rate of improvement has not kept pace with chip capacity growth leading to the well known design-productivity-gap [105]. The problem of reducing the *design-productivity-gap* is crucial in not only handling the complexity of the design, but also combating the increased fragility of the composed system consisting of heterogeneous components. Unlike software programs, integrated circuits are not repairable. The development costs are so high that multiple design spins are ruled out, a design must be correct in the one and often the only one design iteration to implementation.

High-Level Synthesis (HLS) [61, 62, 82, 84, 136, 147, 148, 203] is often seen as a solution to bridge the *design-productivity-gap*. HLS is the process of generating the Register Transfer Level (RTL) design consisting of a data path and a control unit from the behavioral description of a digital system, expressed in languages like C, C++ and Java. The synthesis process consists of several inter dependent subtasks such as: specification, compilation, scheduling, allocation, binding and control generation. HLS is an area that has been widely explored and relatively mature implementations of various HLS algorithm have started to emerge [82, 84, 136, 203]. This shift in design paradigm enables designers to avoid many low-level design issues early in the design process. It also enables early design space exploration, and faster functional verification time. However, for verification of high-level designs, the focus so far has been on traditional testing techniques such as random testing and scenario-based testing. Over the last few decades we have seen many unfortunate examples of hardware bugs (like Pentium FDIV bug, Killer poke, and Cyrix coma bug) that have eluded testing techniques. Recently, many techniques inspired from formal methods have emerged as an alternative to ensure the correctness of these high-level designs, overcoming some of the limitations of traditional testing techniques.

The new techniques and methodology for verification and validation at higher level of abstraction are collectively called *high-level verification* techniques. We divide the process of high-level verification into two parts. The first part deals with verifying properties of high-level designs. The methods for verifying high-level

S. Kundu et al., *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, DOI 10.1007/978-1-4419-9359-5_1,

designs allow designers to check for certain properties such as functional behavior, absence of deadlocks and assertion violations in their designs. Once the properties are checked, the designers refine their design to low-level RTL manually or using a HLS tool.

HLS tools are large and complex software systems, and as such they are prone to logical and implementation errors. Errors in these tools may lead to the synthesis of RTL designs with bugs in them. As a result, the second part deals with verifying that the translation from high-level design to low-level RTL preserves semantics. Taken together, these two parts guarantee that properties satisfied by the high-level design are preserved through the translation to low-level RTL.

Unfortunately, despite significant amount of work in the area of formal verification we are far from being able to prove automatically that a given design always functions as intended, or a given synthesis tool always produces target programs that are semantically equivalent to their source versions. Yet, there have been recent advances: using practical applications of SAT solvers [74, 151], automated theorem proving [75, 164, 165], and model checking [23, 29, 196], where researchers are able to prove that the designs satisfy important properties. Also, in many cases they are able to guarantee the functional equivalence between the initial behavioral description and the RTL output of the HLS process.

We argue that despite these advances in verification, the potential for their impact on chip design largely remains unrealized. For the same reason, we argue that high-level synthesis has not made significant impact on practice, that is, they have not advanced together. Indeed, we view synthesis and verification as two sides of the design methodology coin, both equally important and need to advance together to make a practical impact on the design process. A chip design process is inherently one of elaboration, one of filling in design details to architectural and micro-architectural scaffolding. Thus, an effective design methodology incrementally refines a design as it moves through the design process. These refinements can be done manually or through elaboration tools. The chapters in this book address verification of specific properties in high-level languages as well as checking that the refined implementations are equivalent to their high-level specifications. While various literature and our experience shows that no single technique (including the ones developed specifically for high-level verification) can be universally useful, in general an intelligent combination of a number of these techniques driven by well considered heuristics is likely to prove parts of a design or tool correct, and also in many cases find bugs in them. The key contribution of this book is that it explores and describes a combination of formal techniques to do scalable verification of system designs completely automatically.

1.1 Overview of High-Level Verification

The HLS process consists of performing stepwise transformations from a behavioral specification into a structural implementation (RTL). The main benefit of HLS is that it provides faster time to RTL and faster verification time. Figure 1.1 shows

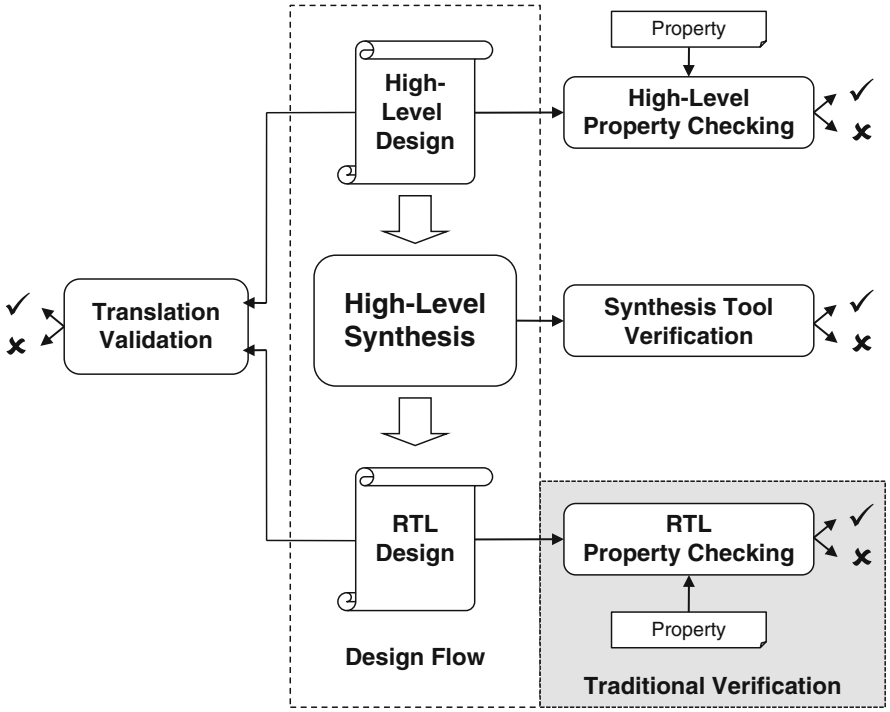


Fig. 1.1 Overview of high-level verification

the various components involved in high-level verification and how they interact. The design flow from high-level specification to RTL is shown along with various verification tasks. These tasks can be broadly classified as follows:

1. High-level property checking
2. Translation validation
3. Synthesis tool verification
4. RTL property checking

Traditionally, designers start their verification efforts directly for RTL designs. However, with the popularity of HLS, these efforts are moving more toward their high-level counterparts. This is particularly interesting because it allows faster (sometimes by three orders of magnitude [201]) functional verification time, when compared to a more detailed low-level RTL implementation. Furthermore, it enables more elaborate design space exploration, which in turn leads to better quality of design. Since RTL property checking techniques is a widely explored subject [81, 110, 142], here we focus only on the first three verification tasks.

The first category of methods, *high-level property checking*, allow various properties to be verified on the high-level designs. Once the important properties that the high-level components need to satisfy have been checked, various other

techniques are used in order to prove that the translation from high-level design to low-level RTL is correct, thereby also guaranteeing that the important properties of the components are preserved.

The second category *translation validation* include techniques that try to show, for each translation that the HLS tool performs, that the output program produced by the tool has the same behavior as the original program. Although this approach does not guarantee that the HLS tool is bug free, it does guarantee that any errors in translation will be caught when the tool runs, preventing such errors from propagating any further in the hardware fabrication process.

The third category *synthesis tool verification* consists of techniques whose goal is to prove automatically that a given optimizing HLS tool itself is correct. Although, these techniques have same goal as translation validation, i.e., to guarantee that a given HLS tool produces correct result, these techniques are different because they can prove the correctness of parts of the HLS tool *once and for all*, before they are ever run.

In this book, we have explored techniques for each one of the three areas outlined above, namely high-level property checking, translation validation, and synthesis tool verification. In the following section we briefly describe the various techniques from each of these areas that we will cover in details in the later chapters, thereby outlining the connections and trade-offs between them.

1.2 Overview of Techniques Covered in this Book

As mentioned in the previous section the approaches described in this book falls into three categories. The key insight behind these approaches is that by performing verification on the high-level design, where the design description is smaller in size and the design intent information is easier to extract, and then checking that all refinement steps are correct, these approaches expand current hardware development methodology to provide strong and expressive guarantees that are difficult to achieve by directly analyzing the low-level RTL code. In the following sections we briefly discuss each of these techniques that are developed specifically for high-level verification.

1.2.1 High-Level Property Checking

Starting with a high-level design, we will first discuss model checking techniques to verify that the design satisfies a given property such as the absence of deadlocks or assertion violations. Model checking in its pure form suffers from the well-known state explosion problem. To cope with this problem, some systems give up completeness of the search and focus on the bug finding capabilities of model checking. This line of thought lead to execution-based model checking approach, which for a given test input and depth, systematically explores all possible behaviors of the design (due to asynchronous concurrency). The most striking benefit of execution-based

model checking approach is that it can analyze feature-rich programming languages like C++, as it sidesteps the need to formally represent the semantics of the programming language as a transition relation. Another key aspect of this approach is the idea of *stateless* search, meaning it stores no state representations in memory but only information about which transitions have been executed so far. Although stateless search reduces the storage requirements, a significant challenge for this approach is how to handle the exponential number of paths in the program. To address this, one can use dynamic *partial-order-reduction* (POR) techniques to avoid generation of two paths that have the same effect on the design's behavior. Intuitively, POR techniques exploit the independence between parallel threads to search a reduced set of paths and still remain provably sufficient for detecting deadlocks and assertion violations.

In Chap. 5 we describe an approach on execution-based model checking. This approach is implemented in Satya, a *query-based* model checking tool that combines static and dynamic POR techniques along with high-level semantics of SystemC to intelligently explore all possible behaviors of a SystemC design. During exploration the runtime overhead is reduced (without significant loss of precision) by computing the dependency information statically. To illustrate its value we describe an example, where Satya was able to automatically find an assertion violation in the FIFO benchmark (distributed as a part of the OSCI repository), which may not have been found by simulation.

Another approach for model checking is to use symbolic algorithms that manipulate sets of states instead of individual states. These algorithms avoid ever building the complete state graph for the system; instead, they represent the graph implicitly using a formula in propositional logic. *Bounded Model Checking* (BMC) is one such algorithm that unrolls the control flow graph (loop) for a fixed number of steps (say k) and checks whether a property violation can occur in k or fewer steps. This typically involves encoding the bounded model as an instance of Satisfiability (SAT) problem. This problem is then solved using a SAT or SMT (Satisfiability Modulo Theory) solver. A key challenge for BMC is to generate efficient verification conditions that can be easily solved using the appropriate solver.

Chapter 6 (contributed by Dr. Malay Ganai) discusses and compares the state-of-the-art BMC techniques for concurrent programs. In particular, it compares the synchronous and asynchronous modeling styles used in formulating the decision problems, and also the sizes of the corresponding formulas. For synchronous model it discusses the partial-order based BMC technique as presented in [205] and for asynchronous model it presents two approaches namely CSSA-based (Concurrent Static Single Assignment) [174, 204] approach and token-based [67] approach.

1.2.2 Translation Validation

Once the important properties of the high-level components have been verified, the translation from the high-level design to low-level RTL still needs to be proven correct, thereby guaranteeing that the important properties of the components are

preserved. One approach to prove that the translation from high-level design to low-level RTL is correct is to show – for each translation that the HLS tool performs – the output program produced by the tool has the same behavior as the original program.

In Chap. 7, we describe a translation validation algorithm that uses a *bisimulation relation* approach to automatically prove the equivalence between two concurrent systems. We discuss the implementation of the above algorithm in a system called Surya. Furthermore, we describe our experience to use it for validating the synthesis process of Spark [84], a parallelizing HLS framework. Surya validates all the code transformation phases (except for parsing, binding and code generation) of Spark against the initial behavioral description. The experiments with Surya showed that with only a fraction of the development cost of Spark, it can validate the translations performed by Spark, and it even uncovered two previously unknown bugs that eluded testing and long-term use.

1.2.3 Synthesis Tool Verification

In order to make a practical impact on improving the level at which designs are done it is important that both synthesis and verification methods are targeted at high-level design. Practically, this means verifying not only the design but also the process that produces such designs. While validation of manual parts of the design process is out of scope here, we focus on methods that guarantee that the (high-level) synthesis tool itself is correct. This approach to validation of both process and product are at the core of our incremental refinement methodology that builds upon advances in translation validation and synthesis tool verification. Chapter 8 (contributed by Zachary Tatlock) discuss an approach that verifies the correctness of critical parts of a synthesis tool. Because some of the most error prone parts of an HLS tool are its optimizations, this approach proves the correctness of optimizations using *Parameterized Equivalence Checking* (PEC) [120]. Moreover, the PEC approach is not limited to only HLS tools; it can be used for any domain that transforms an input program using semantics-preserving optimizations, such as optimizers, compilers, and assemblers.

The PEC technique is a generalization of translation validation that proves the equivalence of *parameterized programs*. A parameterized program is a partially specified program that can represent multiple concrete programs. For example, a parameterized program may contain a section of code whose only known property is that it does not modify certain variables. To highlight the power of PEC, a domain-specific language is designed for implementing complex optimizations using many-to-many rewrite rules. This language is used to implement a variety of optimizations including software pipelining, loop unrolling, and loop unswitching. The PEC implementation was able to automatically verify that all the optimizations implemented in this language preserve program behavior.

1.3 Contributions of the Book

The primary contribution of this book is to explore various formal techniques that can be used for high-level verification. We believe by performing verification on the high-level design, and then checking that all refinement steps are correct, the domain of high-level verification can provide strong and expressive guarantees that would have been difficult to achieve by directly analyzing the low-level RTL code. The goal is to move the functional verification tasks earlier in the design phase, thereby allowing faster verification time and possibly quicker time to market.

To systematically explore the domain of high-level verification, we classified the various verification tasks into three main parts, namely high-level property checking, translation validation, and synthesis tool verification. We describe approaches for each of the above mentioned verification tasks. The novelty of these approaches is that they combine a number of formal techniques along with well considered heuristics to do *scalable* verification of high-level designs completely *automatically*.

We discuss the implementation of some of these high-level verification approaches into prototype tools. We also describe the complemented methodology to use these tools. The tools are: Satya for high-level property checking, Surya for translation validation, and PEC for synthesis tool verification. These tools use state-of-the-art techniques from areas such as model checking, theorem proving, satisfiability modulo theories, static analysis and compiler correctness. Apart from these tools, this book also covers a comparison of recent BMC techniques for concurrent programs.

The techniques presented here exploits structures specific to high-level designs, thereby in many cases simplifying the algorithms and improving their performance. For example, Satya exploits SystemC specific semantics to efficiently explore a reduced set of possible executions, and Surya relies on structure preserving transformations that are predominantly used in HLS.

The prototype tools enables experimentation with large "real-world" designs and tools. The key characteristics of the high-level verification approaches covered are as follows:

Scalable: Most of the techniques discussed here are modular as they work on one entity at a time. For example, Surya works on one procedure at a time, and PEC works on one transformation at a time. Furthermore, in the cases where the entire design have to analyzed, various reduction techniques like partial-order reduction, context bounding, and program structure based reduction (e.g. lock-unlock, fork-join, wait-notify, etc.) is applied. While these software engineering decisions and reductions theoretically limits the scope of the verification tasks on a given design, it is rarely an issue in practice as it follows the designer's and tool developer's programming abstractions.

Practical: The prototype tools are practical enough to be applied to industrial strength designs and tools. For instance, Satya was used to check an industrial benchmark namely the TAC platform [149], and Surya is being used to validate the synthesis process of Spark [84], a state-of-the-art academic HLS framework.

Useful: The tools are able to automatically guarantee the correctness of various properties and transformations. Apart from correctness guarantee, these tools are also quite useful for finding bugs. For example, Satya was able to automatically find an assertion violation in the FIFO benchmark (distributed as a part of the OSCI repository), and Surya was able to uncover two previously unknown bugs in the Spark HLS framework.

1.4 Book Organization

The organization of this book is shown in Fig. 1.2. Chapter 2 presents a brief overview of the three different parts of high-level verification on which the approaches are applied. More specifically, we present a brief introduction of high-level designs, RTL designs, and high-level synthesis. We also introduce in this chapter the concepts of model checking, partial-order reduction and a representation of concurrent programs, which we use in the rest of this book.

In Chap. 3, we present a detailed discussion of the related works in the area of high-level verification. In particular, we divide the related works in to three main areas, namely high-level property checking, translation validation, and synthesis tool

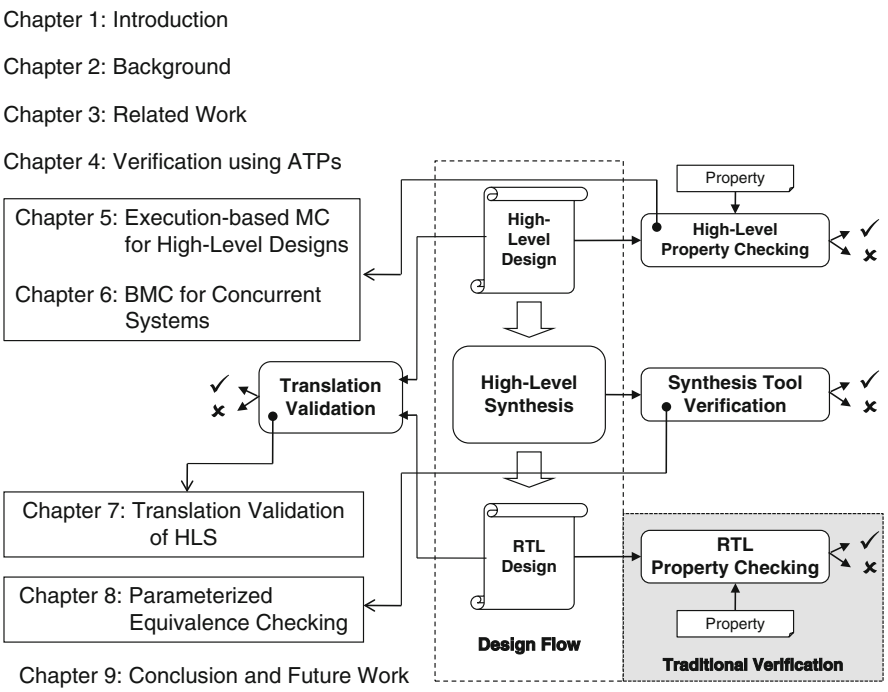


Fig. 1.2 Book organization

verification. We discuss the various tools and techniques explored in these areas. Apart from these related works, in each of the following chapters we again point the reader to further readings that are specific to the chapter.

The next chapter (Chap. 4) describes in details two core verification techniques that uses Automated Theorem Provers (ATP). It first provides a brief introduction to Satisfiability Modulo Theories (SMT) and then discusses two formal techniques, namely Hoare Logic and Weakest Preconditions, which are extensively used in high-level verification.

Chapters 5 and 6 discuss two high-level property checking techniques. In Chap. 5, we present an execution-based model checking approach for the high-level language SystemC. This approach starts with a design written in SystemC, and then intelligently explore a subset of the possible executions till a certain depth to verify that the design satisfies a given property such as absence of deadlocks or assertion violations. Chapter 6, on the other hand discuss symbolic analysis approaches for concurrent programs. It compares state-of-the-art synchronous and asynchronous BMC approaches for performance and scalability.

Chapter 7 discusses a translation validation approach that proves the translation from high-level design to the scheduled design is correct. We describe in detail an algorithm that uses a bisimulation relation approach to automatically prove the equivalence between two concurrent programs. In this chapter, we also report our efforts to validate the synthesis process of Spark, a parallelizing HLS framework.

In Chap. 8 we describe another approach that also proves the result of HLS process is correct. This approach called *Parameterized Equivalence Checking* falls in the synthesis tool verification category. This technique generalizes the translation validation technique of Chap. 7 to prove that the optimizations performed by a HLS tool is correct once and for all. We describe the details of the algorithm and experiments.

Finally, Chap. 9 wraps-up the book with a conclusion and a discussion of future work on high-level verification.

Chapter 2
Background

We envision a design methodology that is built around advances in high-level design and verification to improve the quality and time to design microelectronic systems. In this chapter, we will present a brief overview of the three different parts of high-level verification as shown in Fig. 1.1 on which the verification algorithms are applied. We first present in Sect. 2.1 and in Sect. 2.2 a description of high-level designs and RTL designs respectively. We then in Sect. 2.3 give a brief introduction of high-level synthesis. In the next two sections we introduce the concept of model checking, and our program representation scheme that is used throughout the book.

2.1 High-Level Design

A high-level design is a behavioral or algorithmic specification of a system. This specification typically is written in a high-level language such as behavioral VHDL, C and C++ (and variants). The main reason to use a high-level language is to be able to describe both the hardware and software components of a design, and to allow large system designs to be modeled uniformly. The enormous flexibility in describing computations in a high-level language enables a designer to capture design description at multiple levels of abstraction from behavioral descriptions to transaction level modeling (TLM) [27, 78, 192]. Intuitively, high-level design gives an abstract view of the system. When describing behaviors in a high-level language, designers often use programming constructs such as conditionals and loops for programming convenience often with no notion of how these constructs may affect synthesis results. For example, SystemC [78] TLM supports new synchronization procedures such as wait-notify, which makes current techniques for synthesis inapplicable. One of the key aspects of high-level design is the ability to provide a golden reference of the system in an early phase of the development process.

In this book, we use various high-level languages as input for the verification techniques. In Chap. 5 we will discuss a property checking approach for the SystemC language. In Chap. 6 we describe various BMC techniques that work on concurrent C programs. Also, the translation validation approach described in Chap. 7 uses two high-level languages namely, C and CSP.

S. Kundu et al., *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, DOI 10.1007/978-1-4419-9359-5_2,

2.2 RTL Design

At the current state of practice, RTL designs are generally considered low-level designs consisting of structural implementation details. The RTL describes the exact behavior of the digital circuits on the chip, as well as the interconnections to inputs and outputs. The structural implementation usually consists of a data path, a controller and memory elements. Figure 2.2 shows the controller and data path for a RTL design. The data path consists of component instances such as ALU, multiplier, registers, and multiplexers selected from a RTL component library. The controller is a finite-state machine (FSM) describing an ordering of the operations in the data path. A tiny functional error in the RTL design can sometimes make the entire chip inoperable. Furthermore, writing RTL designs are often tedious, complex, and error-prone as such it is desirable to have a good high level design and then use incremental refinement process to generate the final RTL.

In this book, we do not consider RTL property verification. However, for the purpose of translation validation we want to check the equivalence between a pair of high-level design and RTL design. Unfortunately, checking equivalence between the high-level design and RTL design is a hard problem. As such in Chap. 7 we describe an approach that validates the equivalence between a high-level design written in C and the scheduled design (described in the next section), which is an intermediate low-level design.

2.3 High-Level Synthesis

HLS can be seen as stepwise transformation of a high-level design into a RTL design as shown in Fig. 2.1. Figure 2.2 shows the RTL design for the example in Fig. 2.1. Different HLS tool produces different RTL design for the same high-level input as it minimizes various metrics like area, power and timing. HLS starts by capturing the behavioral description in an intermediate representation, usually a control data flow graph (CDFG). Thereafter the HLS problem has usually been solved by dividing the problem into several sub-tasks [84]. Typically the sub-tasks are:

1. *Allocation*: This task consists of determining the number of resources that need to be allocated to synthesize the hardware circuit (not shown in the figure). Typically, designers can specify an allocation in terms of the number of resources of each resource type. Resources consist of functional units (like adders and multipliers), registers and interconnection components (such as multiplexers and buses). The allocated resources constitute the *resource library* for the design.

   In our example (Figs. 2.1 and 2.2), the resource library contains a multiplier, an ALU, 2 multiplexers, 4 registers, 2 buses and a memory component (not all components are shown in the figure). Usually, a designer chooses these components based on several design constraints like area, performance and power.

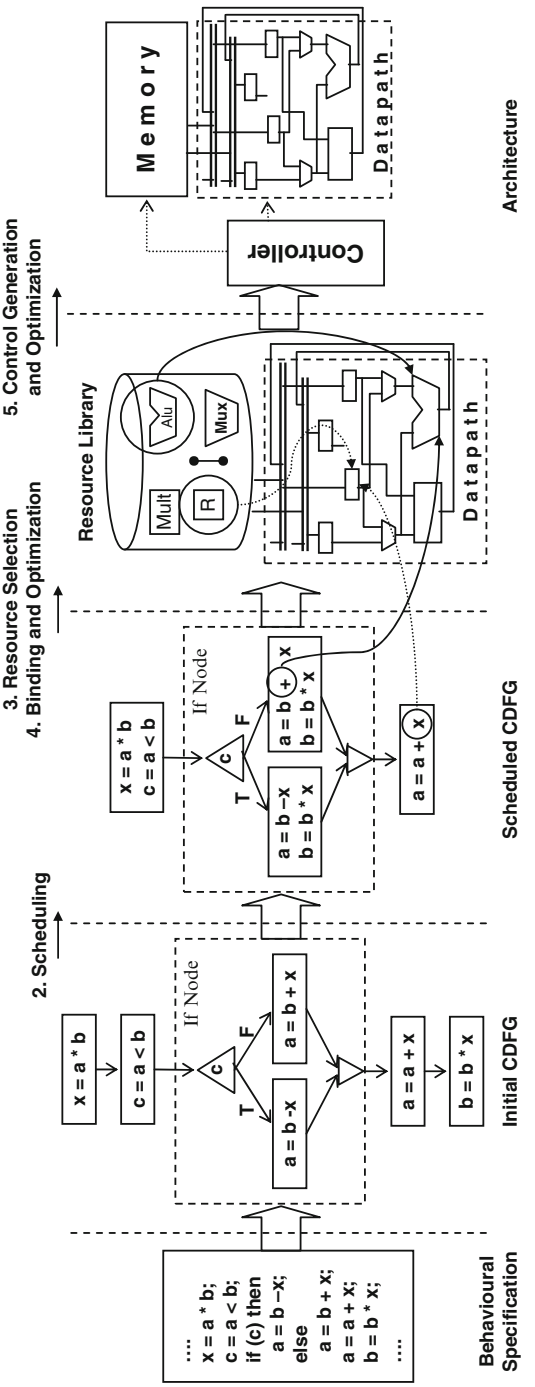


Fig. 2.1 Stepwise transformation during high-level synthesis process

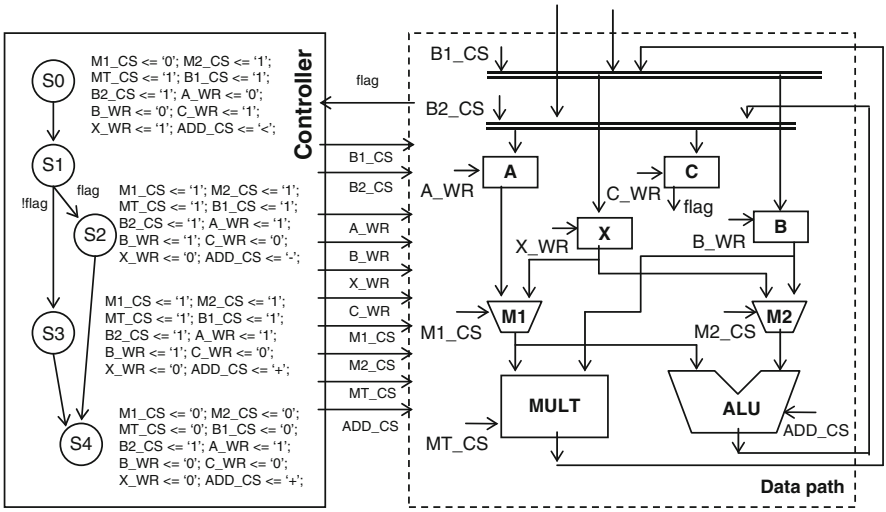


Fig. 2.2 The controller and the data path for the example in Fig. 2.1

2. *Scheduling*: This step determines the time step or the clock cycle in which each operation of the design executes. The ordering between the "scheduled" operations is constrained by the data (and possibly control) dependencies between the operations. Scheduling is often done under constraints on the number of resources.

   Resource constraint scheduling algorithms are highly dependent on the resource allocation task. For example, since in the allocation step the designer chose a multiplier and an ALU, we were able to schedule both the statements ($x = a * b$ and $c = a < b$) in one cycle (C_1). In contrast, if only an ALU is chosen then both the operations have to be scheduled in different cycles.

   Schedulers also do code motions to enhance *concurrency* and hence improving resource utilization. In our example, the statement $b = b * x$ has been moved inside the 'if block' by the scheduler to schedule it in cycle C_2 instead of cycle C_3.
3. *Resource selection*: This task determines the resource type from the resource library that an operation executes on. The need for this task arises because there are several resources of different types (and different area and timing) that an operation may execute on. For example, an addition may execute on an adder, an ALU, or a multiply-accumulate unit. Resource selection must make a judicious choice between different resources such that a metric like area or timing is minimized.

   In the example, all multiplication operations are done using the multiplier and the addition and comparison operations are executed in the ALU (see Figs. 2.1 and 2.2).
4. *Binding and Optimization*: This task determines the mapping between the operations, variables and data (and control) transfers in the design graph and the specific resources in the resource library. Hence, operations are mapped

to functional units, variables to registers and data/control transfers to interconnection components. Optimizations deals with the minimization of the physical components in the synthesized design.

In our case, the variables $a, b, c, and\ x$ in the behavioral description are bound to the registers $A, B, C, and\ X$ respectively. Figure 2.1 shows the binding of the variable a to the register A and the operation $+$ to the ALU. Notice that multiple operations can be bound to the same resource. The complete binding of the operations and the data/control transfers to functional units and interconnection components are shown in Fig. 2.2.

5. *Control Generation and Optimization*: Control synthesis generates a control unit (usually FSM) that implements the schedule. This control unit generates control signals that control the flow of data through the data path (i.e. through the multiplexers). Control optimization deals with minimizing metrics such as area and power.

   Operations in the scheduled CDFG are replaced by the concrete values of control signals going to the data path. Thus, for example the concurrent operations ($x = a * b\ and\ c = a < b$) are replaced by the following signals.

$$\begin{array}{l} \text{M1\_CS} <= \text{`0'},\ \ \text{M2\_CS} <= \text{`1'},\ \text{MT\_CS} <= \text{`1'}, \\ \text{B1\_CS} <= \text{`1'},\ \ \text{B2\_CS} <= \text{`1'},\ \ \text{A\_WR} <= \text{`0'}, \\ \text{B\_WR} <= \text{`0'},\ \ \text{C\_WR} <= \text{`1'},\ \ \text{X\_WR} <= \text{`1'}, \\ \text{ADD\_CS} <= \text{`}<\text{'} \end{array}$$

   According to these control signals, the contents of the registers A and B are simultaneously fed as inputs to the $MULT$ and the ALU units and then the result of $MULT$ and ALU is saved in the registers X and C respectively.

HLS is an area that has been widely explored and relatively mature implementations of various HLS algorithm have started to emerge [84, 136, 203]. These tools are usually very large and complex piece of software, as such their implementation are prone to errors. In Chap. 7 we discuss an approach that validates parts of a parallelizing HLS tool called Spark [84]. Furthermore, in Chap. 8 we describe a technique that can once-and-for-all prove the correctness of important parts of a HLS tool.

2.4 Model Checking

Does the design satisfy a given property (specification)? This is one of the key question that every hardware designer have to answer. Fortunately, in the last two decades, researchers have made tremendous progress in developing tools and techniques for verifying properties and design. Although, it is hard to prove if a design satisfy a given specification that is written in English, it is possible for specifications that are formulated using unambiguous formal language. Model Checking [101] is

the most successful approach in this direction. Given a model of a system, it tests *automatically* whether this model satisfy a given specification (Fig. 2.3). Typically, the model of the (hardware) system is represented using a Finite State Machine (FSM) and the specification is formulated in temporal logic [169] formula. The FSM is a directed graph consisting of nodes (or vertices) and edges.

2.4.1 Simple Elevator Example

This section describes how we can use model checking to verify properties of a simple two floor elevator example, which is shown in Fig. 2.4 using a C-style language. The program has a main function Elevator (lines 5–11) and a helper function Move (lines 12–19). The global variables of the program are shown in lines 1–4. The program uses an *enum* variable *currFloor* to keep track in which floor the elevator is currently located. It also use a boolean array *doorOpen* to indicate if the door of the elevator is open in the corresponding floor and a boolean variable *moving* to represent if the elevator is currently moving or is stopped. The Elevator function in an infinite loop sets the *doorOpen* variable for the current floor to *false* and the *moving* variable to *true* and then calls the helper function Move. The Move function depending on the current floor where the elevator is present changes the *currFloor*

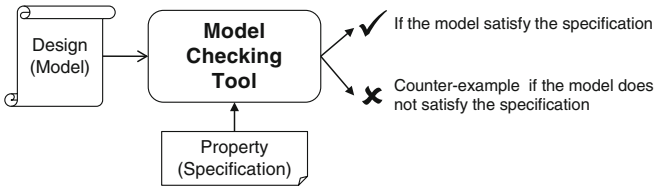


Fig. 2.3 The model checking approach

```
1.  enum Floor = {1,2}
2.  Floor currFloor
3.  bool doorOpen[2]
4.  bool moving

5.  function Elevator()
6.      while (true)
7.          wait 1 min
8.          doorOpen[currFloor − 1] = false
9.          moving = true
10.         Move()
11.     return

12. function Move()
13.     if (currFloor == 1)
14.         ++currFloor
15.     elseif (currFloor == 2)
16.         −−currFloor
17.     moving = false
18.     doorOpen[currFloor − 1] = true
19.     return
```

Fig. 2.4 Simple elevator example

variable up or down and then sets the state of the *doorOpen* of that floor to open and the state of the elevator to stop. Although this example may appear trivial, it can be easily extended to one with many floors, fire emergency state, and priority of floors, thereby increasing the complexity of the elevator controller.

A *data state* for this example is defined by a tuple of values for each of the three global variables. For example, one particular data state is

$$\langle currFloor = 1, doorOpen[2] = (true, false), moving = false \rangle$$

For brevity we use the notation $\langle 1, (T, F), F \rangle$ to represent the above data state. An assignment statement in Fig. 2.4 defines how the data state of the program changes. The data-state space of our example along with their possible transitions is shown in Fig. 2.5. Not all data states are valid initial data state. For example, let us assume that when the system starts the elevator is not moving and both the doors are closed. For this assumption, the possible reachable data states are shown within the dotted line in Fig. 2.5. A *program state* is defined as a data state along with the valuation of another variable the program counter (or program location) *pc*. The variable *pc* defines the possible locations of the program. Informally, the different values of *pc* is given in terms of the line number of the program. For our example, the variable *pc* can take values in the range [5–19] and it's initial value is 5.

An important concept in model checking is that of an *execution sequence* (also referred as execution path). Starting from a given initial program state an

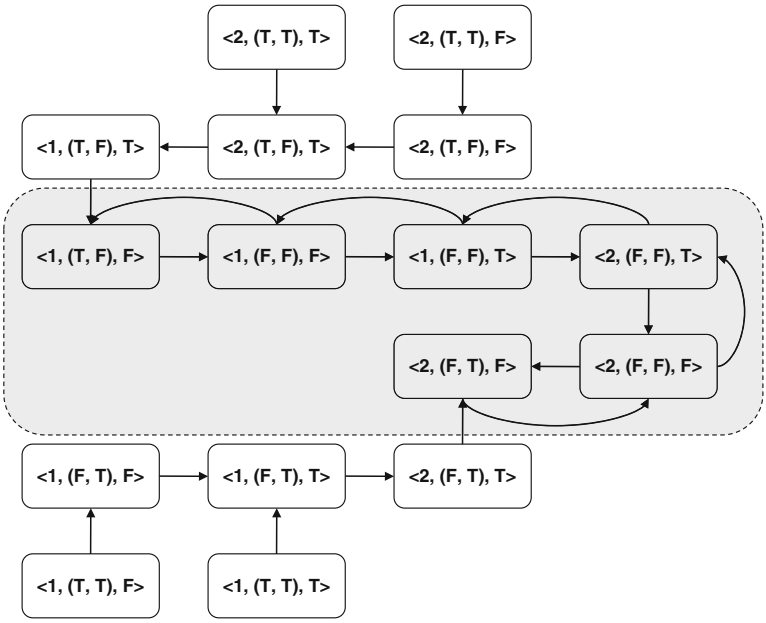


Fig. 2.5 A FSM of the data-state space for the simple elevator example

execution sequence is informally defined as the possibly infinite sequence of states obtained by following the transitions in the FSM. Note that for concurrent programs this concept of execution sequence is generalized into an *execution tree* as from any state there are many possible next states. In general, there are many possible execution sequences for a given program. The goal of model checking is to check whether or not the execution sequences satisfies a user-given property specification.

2.4.2 Property Specification

A property is usually expressed using a formula specified using boolean propositional logic. For programs we also need to specify properties involving execution paths. Some form of temporal logic is usually suitable for specifying such properties. Computation Tree Logic (CTL) is a branching time temporal logic that has been extensively used by model checking tools for this purpose.

In CTL formulas are composed of *path quantifiers* and *temporal operators*. There are two path quantifiers – **A** (for all paths) and **E** (for some path) – for specifying properties of (all or some) paths beginning from a state. There are five temporal operators – **X** (next state), **F** (for some state), **G** (for all states), **U** (until), and **R** (release) – that describes properties of (all or some) states of a path. These quantifiers and operators are summarized as follows:

Path Quantifiers:

A ϕ ϕ holds on *all* paths starting from the current state.
E ϕ There *exists* at least one path starting from the current state where ϕ holds.

Temporal Operators:

X ϕ ϕ holds at the *next* state.
G ϕ ϕ holds on *all* states for the entire subsequent path.
F ϕ ϕ *eventually* holds on some state on the subsequent path.
ϕ **U** ψ ϕ holds *until* at some state ψ holds and eventually ψ holds.
ϕ **R** ψ ψ holds until and including the state where ϕ holds, however, ϕ is not required to hold eventually.

Some examples of properties are as follows:

- $\mathbf{EF}(moving \wedge currFloor == 1)$: It is possible to get to a state where the elevator is moving and it is at the first floor.
- $\mathbf{AG}(\mathbf{EF}(\neg doorOpen[0]))$: From any state it is possible to get to a state where the door of the first floor is closed.
- $\mathbf{AG}(moving \Rightarrow \mathbf{AF}(\neg moving))$: If the elevator is moving then it will be eventually stopped.

2.4.3 Reachability Algorithm

Given a temporal logic formula ϕ and a model of the program in some form of FSM $\mathscr{M}$, a simple model-checking algorithm essentially reduces to a graph reachability algorithm that examines every reachable node of $\mathscr{M}$ to check if it satisfies the formula ϕ.

For example, consider a property $\mathbf{EF}f$ where f is a boolean propositional formula not involving any temporal quantifiers or operators. Let the FSM $\mathscr{M}$ be a tuple $\mathscr{M} = (\mathscr{N}, \mathscr{I}, \mathscr{E})$, where $\mathscr{N}$ is a set of nodes, $\mathscr{I} \subseteq \mathscr{N}$ is a set of initial nodes, and $\mathscr{E} \subseteq \mathscr{N} \times \mathscr{N}$ is a set of edges. The nodes represent program states, and the edges represent possible transitions which may alter the state. The property $\mathbf{EF}f$ is satisfied in $\mathscr{M}$ if there exists some state in $\mathscr{M}$ that is reachable from the initial states and the formula f is true in that state.

Figure 2.6 shows the algorithm CheckEF, a procedure that traverses the graph to check if the formula f holds in any state of the graph. The algorithm maintains a set *reach* of reachable states and a set *worklist* of states that are found to be reachable but whose successors may not have been explored. Initially, the set *reach* is empty, and the set *worklist* contains all the initial states. The main loop of the algorithm starts by picking a state from the *worklist*. If the formula f holds in that state then we know that the property $\mathbf{EF}f$ is satisfied in $\mathscr{M}$, and the algorithm ends. Otherwise, if the state has not been visited before, the successors of the state are added to the *worklist*. The process is repeated until all reachable states have been explored, which happens when the *worklist* becomes empty. At this point, *reach* contains exactly the set of reachable states and the formula f does not hold in any of these states. Hence, the property $\mathbf{EF}f$ is not satisfied in $\mathscr{M}$. The loop in the algorithm terminates for all programs with finite reachable states.

In practice, model-checking algorithms are far more complex, they use intelligent heuristics and efficient data structures to prune the state space to avoid checking those parts where the property is guaranteed to be true. Recent model checkers are also able to verify properties of state spaces of size as large as 10^{20} states [23]. Pioneering work on model checking was done by E.M. Clarke, E.A. Emerson and A.P. Sistla [32, 33, 52] and by J.P. Queille and J. Sifakis [173]. E.M. Clarke,

```
1.  procedure CheckEF(f)
2.      let reach := ∅
3.      let worklist := {ι | ι ∈ ℐ}
4.      while worklist not empty do
5.          let σ := worklist.Remove
6.          if f holds in σ then
7.              "EFf is satisfied in ℳ"
8.              return
9.          if σ ∉ reach then
10.             reach := reach ∪ {σ}
11.             worklist := worklist ∪ {σ' | (σ, σ') ∈ ℰ}
12.     "EFf is not satisfied in ℳ"
13.     return
```

Fig. 2.6 Algorithm for checking if $\mathbf{EF}f$ is satisfied in $\mathscr{M}$

E.A. Emerson, and J. Sifakis shared the 2007 Turing Award for their work on model checking. In Chap. 3 we will cover some of the advances in model checking and provide pointers for them. In Chaps. 5 and 6 we present two model-checking algorithms that are related to high-level verification.

2.5 Concurrent Programs

A concurrent program usually consists of a set of processes or components that interact with each other. The execution semantics of a concurrent program can be broadly divided into two types: *asynchronous* and *synchronous*. In asynchronous (or interleaved) mode of execution only one atomic statement of a process is executed at any time. In synchronous mode of execution an atomic statement from each of the processes is execute at the same time. For example, consider a simple program of two processes P_1 and P_2 interacting with shared variables y and z. The processes have only one statement each as shown below.

Process P_1	Process P_2
$y := z$	$z := y$

In asynchronous mode of execution only one process is executed at any time. Therefore, at the end of the execution either both shared variables y and z will have the old value of z (if P_1 is executed first) or both variables will have the old value of y (if P_2 is executed first). On the other hand, in synchronous mode of execution the values of y and z will be swapped (exchanged).

Apart from the mode of execution, two concurrent programs can also differ based on their mode of communications. Concurrent programs can again be *synchronous* (blocking) or *asynchronous* (non-blocking) based on their mode of communication. In synchronous mode, communication between processes is unbuffered, and processes wait (or block) until the data between them has been transferred. In asynchronous mode, communication between processes is buffered and the processes do not wait after sending or before receiving data.

2.5.1 Representation of Concurrent Programs

Unless otherwise specified all of the programs we consider are Globally Asynchronous Locally Synchronous (GALS). This means that the programs consists of components that are asynchronous (interleaved), however within a component there can be statements that are synchronous (i.e. multiple statements can be executed at the same time). We visually represent a concurrent program using an internal Concurrent Control Flow Graph (CCFG) representation. Concurrent behavior of programs can be modeled as synchronous or asynchronous. The CCFG of a simple

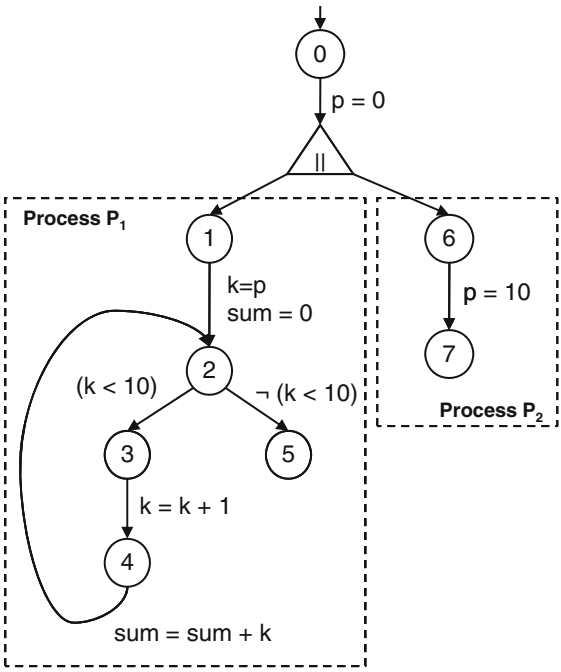


Fig. 2.7 Our Concurrent Control Flow Graph (CCFG) representation

example is shown in Fig. 2.7. In general, we omit the details of the actual code, because the CCFG representation is complete. This example consist of two processes P_1 and P_2. We use a node with the symbol $||$ to denote the asynchronous parallel composition of child processes (components). Statements on the same edge are executed at the same time (i.e. synchronously). The example computes the sum from 1 to 10 in the variable sum if the process P_1 is executed before P_2, otherwise sum is 0.

2.5.2 Partial-Order Reduction

In this section, we provide an overview of the partial-order reduction (POR) technique, which is used in almost all approaches discussed in this book. Verification of asynchronous concurrent programs is hard due to various possible interleavings between the processes. The verification model is typically obtained by composing individual process models using interleaving semantics. Thus, in the worst case, the global state space can be the product state space of the individual processes. To combat this state explosion problem, most methods employ POR techniques to restrict the state-traversal to only a representative subset of all interleavings, thereby, avoiding exploring the redundant interleaving among independent transitions [70].

POR in its most simple form exploit the commutativity of concurrently executed transitions, which result in the same state when executed in different orders.

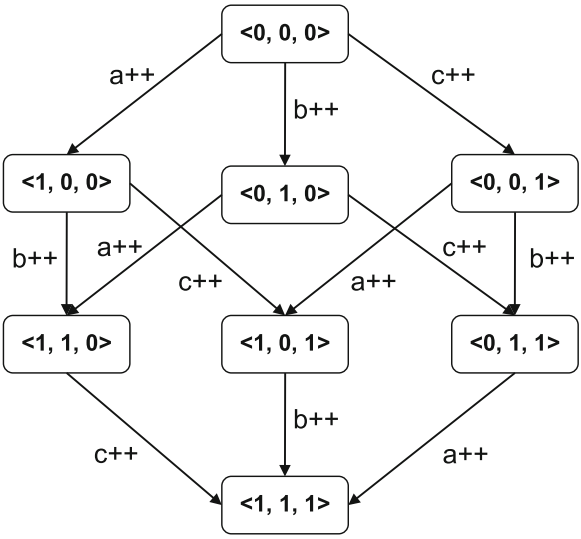


Fig. 2.8 Possible data states due to executing three concurrent statements

For example, consider a simple program of three processes P_1, P_2, and P_3. The processes have only one statement each as shown here.

Process P_1	Process P_2	Process P_3
$a++$	$b++$	$c++$

Let the initial data state of the program be $\langle a = 0, b = 0, c = 0\rangle$ (for brevity we use the notation $\langle 0,0,0\rangle$). Figure 2.8 shows the various possible data states that can be reached from the initial state due to interleavings. The three statements in this example are independent, i.e., their order of execution does not change the final state. Now, if the property we are considering is only dependent on the final state, then during exploration using POR we only need to traverse one of these interleavings.

POR techniques are extensively used by *software model checkers* for reducing the size of the state space of concurrent system [71, 94]. Other state space reduction techniques, such as slicing [88, 184] and abstraction [9], are orthogonal and can be used in conjunction with POR. The POR techniques can be divided in two main categories: *static* [70] and *dynamic* [57].

The main static POR techniques are *persistent/stubborn* sets and *sleep* sets [70]. Intuitively, the persistent/stubborn set techniques compute a provably sufficient subset of the enabled transitions in each visited states such that if a selective search is done using only the transitions from these subsets the detection of all the deadlocks and safety property violations is guaranteed. All these algorithms infer the persistent sets from the static structure (code) of the system being verified. On the other hand, the sleep set techniques exploits independences between the transitions in the persistent sets to reduce interleavings. Both these techniques are orthogonal and can be

applied simultaneously [70]. In contrast, the dynamic POR technique evaluates the dependency relation dynamically between the enabled and executed transitions for a given execution. In particular, it starts by executing the program until completion, and then infers the persistent sets dynamically by collecting information about how threads have communicated during this specific execution trace. This technique is particularly important as we will see later that most of the approaches discussed in this book uses some form of POR.

2.6 Summary

In this chapter, we presented a brief overview of the three main concepts related to our formulation of high-level verification namely, high-level design, RTL design, and HLS. We first presented a description of high-level design and RTL design in Sects. 2.1 and 2.2. We also mentioned the different high-level languages and RTL representation that we use in this book. We then briefly discussed in Sect. 2.3 the various steps of HLS using a simple example. We also provided a brief introduction to model checking in Sect. 2.4. Finally, in Sect. 2.5, we introduced the concurrent program representation and describe an important technique called partial-order reduction that we use throughout this book.

Chapter 3
Related Work

Each one of the three areas of high-level verification outlined in Chap. 1, namely high-level property checking, translation validation, and synthesis tool verification, have been explored in a wide variety of research efforts. In this chapter, we discuss various techniques from each of these areas that are directly relevant to this book.

3.1 High-Level Property Checking

The high-level designs written using languages like C, SystemC, SystemVerilog are mostly software programs with support for specialized hardware data types and other hardware features like synchronous concurrency, synchronization, and timing [83]. Thus, many efforts to use software verification tools to verify these designs have been explored. Model checking is the most prevalent automatic verification technique for software and hardware. It is a technique for verifying that a hardware or software system satisfy a given property (specification). These properties, which are usually expressed in temporal logic, typically encode deadlock and safety properties (e.g. assertion violations). In this section, we survey several software model checking techniques grouped as *explicit* and *symbolic* techniques.

3.1.1 Explicit Model Checking

In explicit state enumeration model checking, the reachable states of a design are generated using an exhaustive search algorithm. This technique explicitly stores the entire state space in memory and checks if certain error states are reachable. For finite state system this technique is both sound (i.e. whenever model checking cannot reach a given error state, it is guaranteed to not reach that error state ever in real execution) and complete (i.e. whenever model checking finds an error, it is guaranteed to be an error in real execution). However, as the size of the finite state spaces grow larger and larger, this technique suffers from the well known *state explosion* problem. To address the state explosion problem, researchers use techniques

S. Kundu et al., *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, DOI 10.1007/978-1-4419-9359-5_3,

to construct the state space *on-the-fly* [94] during the search, rather than generating all the states and transitions before the search. In addition, they use *bit-state hashing* [94], in which the hash value of the reachable state is stored, instead of the state itself. Due to possible hash collision the bit-state hashing technique is unsound. Other techniques include *partial-order-reduction* [70], *symmetry reduction* [35, 53] and *compositional techniques* [34].

Intuitively, the partial-order reduction technique exploits the independence between parallel threads to compute a provably sufficient subset of the enabled transitions in each visited states such that if a selective search is done using only the transitions from these subsets the detection of all the deadlocks and safety property violations is guaranteed. Symmetry reduction on the other hand exploits symmetries in the program, and explores one element from each symmetry class. Compositional techniques decompose the original verification problem to related smaller problems such that the result of the original problem can be obtained by combining the smaller ones.

The most popular finite state explicit model checker for concurrent programs are SPIN [94] and MURPHI [46]. Both tools have been successfully used for verification of sequential circuits and protocols.

Moreover, in order to achieve scalability some systems give up completeness of the search and focus on the bug finding capabilities of model checking. For instance, one can bound the depth of the search and/or bound the number of context switches [156]. This line of thought also leads to the execution-based model checking approach. These methods are typically used for improving the coverage of a test. Traditionally, in testing the user writes a test bench and runs it. Typically, the operating system scheduler executes only one fixed schedule out of the many possible behaviors. However, the scheduler of the execution based model checker systematically explore all possible behaviors of the program for a given test input and depth. The most striking benefit of this approach is the ease of implementing it, as it sidesteps the need to formally represent the semantics of the programming language as a transition relation. Another key aspect of this method is the idea of *stateless* [71] search, i.e., it stores no state representations in memory but only information about which transitions have been executed so far. Although stateless search reduces the storage requirements, a significant challenge for this approach is how to handle the exponential number of paths in the program. Here again various reduction techniques like symmetry, partial-order [57], and abstraction have been explored.

Verisoft [71] is the first tool in this domain, and it explores arbitrary code written in full fledged programming language like C or C++. It does so by modifying the OS scheduler and systematically exploring all possible interleavings. Java PathFinder [202] is a tool for Java programs that uses the virtual machine rather than OS scheduler to explore the different behaviors. CMC [155] is another tool for C programs that improves the efficiency of the search by storing a hash of each visited state. Dynamic validation using execution-style model checking is also well adapted for validating SystemC designs [89].

3.1.2 Symbolic Model Checking

The above reduction techniques like partial-order, address the state explosion problem for asynchronous concurrent systems (by reducing the number of interleavings that need to be explored). However, they are not so effective in the case of synchronous concurrent systems, which do not involve interleaving. Symbolic model checking techniques, on the other hand, are quite effective for both synchronous and asynchronous concurrent systems. Furthermore, the reduction techniques discussed in Sect. 3.1.1 including partial-order reduction are orthogonal and can be used in conjunction with symbolic techniques.

Symbolic algorithms manipulate sets of states, instead of individual states. These algorithms avoid ever building the complete state graph for the system; instead, they represent the graph implicitly using a formula in propositional logic. They can also represent infinite states using a single formula. For example the predicate $(x > 1 \wedge y > 1)$ denotes the set of all states in which the value of the variables x and y are both greater than 1. The first major step toward symbolic representation is the use of *Binary Decision Diagrams* (BDD) [22]. BDDs are a canonical form representation for boolean formulas, and are particularly important for finite state programs, as these programs can be represented using boolean variables. BDDs are used in symbolic model checker like SMV [144] and have been instrumental in verifying hardware designs with very large state spaces [23].

As in explicit model checking, one sometimes trades off completeness for bug finding capabilities of symbolic model checking. *Bounded Model Checking* (BMC) [15] is one such algorithm that unroll the control flow graph (loop) for a fixed number of steps (say k), and check whether a property violation can occur in k or fewer steps. This typically involves encoding the restricted model as an instance of Satisfiability (SAT) problem. This problem is then solved using a SAT [151] or SMT (Satisfiability Modulo Theory) [152] solver. BMC tools like CBMC [31] and FSoft-BMC [97] use iterative deepening depth-first search so that the above process can be repeated with larger and larger values of k until all possible violations have been ruled out.

Another area that has recently received lot of attention is *abstract model checking*, which trades off precision for efficiency. Abstraction [39, 41] attempts to prove properties of a program by first simplifying it. Next, the reachability analysis is performed on the simplified (or abstract) domain, which usually satisfies some, but not all the properties of the original (or concrete) program. Generally, one requires the abstract domain and its semantics to be sound (i.e. the properties proved in the abstract semantics implies properties in the concrete semantics). However, typically, the abstraction is not complete (i.e. not all true properties in the concrete semantics are true in the abstract semantics). An example of abstraction is, to only consider boolean variables and the control flow of a program and ignore the values of non boolean variables. Although, such an abstraction may appear coarse, it is sometimes sufficient to prove properties like mutual exclusion.

The polyhedral abstract domain has been successfully used to check for array bounds violations [42]. Another interesting domain, *predicate abstraction*

[9, 43, 76, 122] is parameterized by a fixed finite set $\mathscr{B} = \{B_1, B_2, \ldots, B_k\}$ of first-order formulas (predicates) over the program variables, and consists of the lattice of Boolean formulas over $\mathscr{B}$ ordered by implication. A cube over $\mathscr{B}$ is a conjunction of possibly negated predicates from $\mathscr{B}$. The domain of predicate abstraction is the set of all cubes, and one cube is computed at each program point.

The goal of predicate abstraction is to compute a set of predicates from $\mathscr{B}$ at every program point. Thus, given a set of predicates $\mathscr{B}$, a program statement s, and a cube over $\mathscr{B}$ flowing into the statement, it computes the cube over $\mathscr{B}$ that flows out of the statement. For example, consider the set of predicates $\mathscr{B} = \{B_1, B_2\}$, where $B_1 \equiv (a = b)$ and $B_2 \equiv (a = b + 1)$. Given the cube $B_1 \wedge \neg B_2$ and the statement $a := b + 1$, then predicate abstraction would compute that the cube $\neg B_1 \wedge B_2$ should be propagated after the statement.

A problem with abstract model checking is that although the abstraction simulates the concrete program, when the abstraction does not satisfy a property, it does not mean that this property actually fails in the concrete program. When a property fails, the model checker produces a *counterexample*. A counterexample can be *genuine*, i.e., can be reproduced on the concrete program, or *spurious*, i.e., does not correspond to a real computation but arises due to the imprecision in the analysis. Counterexamples are checked against the real state space to make sure they are genuine. In the case when it is spurious, methods have been developed to automatically refine the abstract domain and get a more precise analysis which rules out the current counterexample and possibly many others, without losing soundness. This iterative strategy is called *Counter Example Guided Abstraction Refinement* (CEGAR).

SLAM [9] is a popular CEGAR based model checker for C programs. It was used successfully within Microsoft for device driver verification [8] and has been developed into a commercial product (Static Driver Verifier, SDV). BLAST [14] is also a CEGAR based model checker that uses *lazy abstraction* [90]. The main idea of Blast is the observation that the computationally intensive steps of abstraction and refinement can be optimized by a tighter integration which would allow it to reuse the work performed in one iteration toward subsequent iterations. Lazy abstraction tightly couples abstraction and refinement by constructing the abstract model on-the-fly, and locally refining the model on-demand. MAGIC [28] is another CEGAR based compositional model checking framework for concurrent C programs. Using MAGIC, the problem of verifying a large implementation can be naturally decomposed into the verification of a number of smaller, more manageable fragments. These fragments can be verified separately, enabling MAGIC to scale up to industrial size programs.

Advances in model checking and related techniques in the past several decades have allowed researchers to verify increasingly ambitious properties of software programs including device drivers, operating systems code, and large commercial applications. They have also enabled the verification of large hardware components like microprocessors. Although this is a significant step forward toward reducing the design-productivity-gap, state-of-the-art verification techniques are still far away from proving full correctness of programs.

3.2 Translation Validation

Once the design has been checked to satisfy certain properties using techniques discussed in Sect. 3.1, the next step is to make sure that those properties are preserved through the synthesis process. In this section we discuss a category of methods called translation validation which guarantee the preservation of safety properties through the synthesis process. Translation validation techniques are employed during synthesis to check that each transformation performed by the HLS tool preserves the semantics of the initial design. The initial design is called specification and the transformed design is called implementation. The validation step check for either *refinement* or *equivalence*. Typically, the implementation is said to be a refinement of the specification if the set of execution traces of the implementation is a subset of the set of execution traces of the specification. They are equivalent when the two sets are equal. In this section, we discuss different techniques for translation validation. Depending upon the core approach these techniques are primarily based on, they are divided into three categories: relational approach, model checking, and theorem proving.

3.2.1 Relational Approach

Relational approaches [18, 47, 106, 111] are used to check the correctness of the synthesis process by establishing a functional equivalence between the Control-Data Flow Graphs (CDFG) of the program, before and after each step of HLS. The equivalence is defined on some predefined *observable events* that are preserved across the transformations. Intuitively, the idea is to show that there exists a *simulation relation* R that matches a given program state in the implementation with the corresponding state in the specification. This simulation relation guarantees that for each execution sequence of observable events in the implementation, a related and equivalent execution sequence exists in the specification. The relation $R \subseteq State_1 \times State_2$ operates over the program states $State_1$ of the specification and the program states $State_2$ of the implementation. If $Start_1$ is the set of start states of the specification, $Start_2$ is the set of start states of the implementation, and $\sigma \rightarrow^e \sigma'$ denotes state σ stepping to state σ' with observable event e, then the following conditions summarize the requirements for a correct refinement:

$$
\begin{aligned}
&\forall \sigma_2 \in Start_2 \,.\, \exists \sigma_1 \in Start_1 \,.\, R(\sigma_1, \sigma_2) \\
&\forall \sigma_1 \in State_1, \sigma_2 \in State_2, \sigma_2' \in State_2 \,. \\
&\quad \sigma_2 \rightarrow^e \sigma_2' \wedge R(\sigma_1, \sigma_2) \Rightarrow \\
&\qquad \exists \sigma_1' \in State_1 \,.\, \sigma_1 \rightarrow^e \sigma_1' \wedge R(\sigma_1', \sigma_2')
\end{aligned}
$$

These conditions respectively state that (1) for each starting state in the implementation, there must be a related state in the specification; and (2) if the specification

and the implementation are in a pair of related states, and the implementation can proceed to produce observable events e, then the specification must also be able to proceed, producing the same events e, and the two resulting states must be related. The above conditions are the base case and the inductive case of a proof by induction showing that the implementation is a refinement of the specification.

One example of using the relational approach is Karfa et al.'s technique [106] for establishing the equivalence between the initial Finite State Machine with Datapath (FSMD) and a scheduled FSMD. The technique introduces cut-points in the original and transformed FSMD automatons, which allows computations through the original and transformed FSMD to be seen as the concatenation of paths from cut-points to cut-points. The technique then establishes the equivalence by exploiting the structural similarities between related cut-points using weakest pre-condition.

Another example of the relational approach can be found in Dushina et al.'s proposed method [47] for checking the functional equivalence between a scheduled abstract FSM and the corresponding RTL after binding. The method establishes the equivalence transition by transition. In particular, for each transition in the RTL controller, it performs a symbolic execution of the associated RTL data path. The symbolic execution results are then syntactically compared with the data operations specified in the equivalent transition of the abstract FSM.

In general, relational approaches work well when the transformations preserve most of the program's control flow structure. Such transformations are called *structure-preserving* [212] transformations. Unfortunately, relational approaches tend to be ineffective in the face of non structure-preserving transformations like loop unrolling, loop tiling and loop reordering. Despite these limitations, relational approaches are very useful in practice: as with only a fraction of the development cost of an HLS tool, they can uncover bugs that elude testing.

3.2.2 Model Checking

Techniques involving model checking [6, 17] are used for verifying register-transfer logic against its scheduled behavior. The key idea is to partition the equivalence checking task into two simpler subtasks, verifying the validity of register sharing/binding, and verifying correct synthesis of the RTL interconnect and control. The success of these methods can be attributed to the observation that the state space explosion in most designs is caused by the data path registers rather than the number of control states.

The following algorithm outlines the method of verifying the validity of register sharing.

- Identify paths in the scheduled graph along which potential conflicts can occur. During this step no interpretation of the data path is done. If no conflict is identified then verification successfully terminates.
- Otherwise for each violation the set of all conflict paths is summarized in a reduced Conflict Sub-Graph (CSG).

- The reduced CSG is then checked to find out if the conflict was benign. If a conflict is detected during the checking, then a logically possible path with incorrect register binding has been detected. In this case the appropriate path is shown to the user as a counterexample.
- Else, if it ends without any conflict detected, all possible conflict paths are logically impossible and the verification algorithm successfully terminates.

Ashar et al. [6] analyzed potential conflicts by means of structural methods, and then the reduced CSG is checked for satisfiability by the VIS model checker [196]. Whereas Blank [17] identifies possible conflicts using a symbolic model checker [23]. The result of the analysis is summarized in a reduced internal representation called Language of Labeled Segments (LLS) [91], which is then checked by symbolic simulation [150]. However, symbolic simulation allows reasoning for a defined finite number of steps. Thus, loops in the program cannot be verified for an arbitrary number of iterations.

Ashar et al. also presented an algorithm to verify the correct synthesis of the RTL interconnect and control [6]. This part of equivalence checking is done state-by-state, i.e., for each state in the schedule, the computations performed in that state are shown to be equivalent to those performed in the RTL implementation for the same state. The equivalence is shown using symbolic simulation.

3.2.3 Theorem Proving

Although most of the translation validation approaches discussed so far use theorem provers in some way, the theorem prover is not at the center of the approach. The Correctness Condition Generator [138], on the other hand, is primarily based on a theorem proving technique. This approach assumes that the synthesis tool can identify the binding relation between specification variables and registers in the RTL design, and between the states in the behavior and the corresponding states in the RTL design. A correctness condition generator is tightly integrated with the high-level synthesis tool to automatically generate (1) formal specifications of the behavior and the RTL design including the data path and the controller, (2) the correctness lemmas that establish equivalence between the synthesized RTL design and its behavioral specification, and (3) proof scripts for these lemmas that can be submitted to a higher-order logic proof checker without further human interaction. The tight integration of the synthesis process with the theorem prover allows the theorem prover to gather information about what kinds of transformations were performed, and therefore better reason about them.

3.3 Synthesis Tool Verification

Another attractive way of proving that an HLS tool produces correct RTL is to verify the correctness of HLS tool once and for all, before it is ever run once.

One can categorize such techniques into three broad classes: (1) *formal assertions*, which can be used to guarantee the correctness of the synthesis tool, (2) *transformational synthesis tools*, which are correct by construction, and (3) *witness generators*, which recreate the steps that an existing HLS tool has performed using formally verified transformations.

3.3.1 Formal Assertions

Narasimhan et al. proposed a Formal Assertions approach [157–159] to building a verified high-level synthesis system, called *Asserta*. The approach works under the following premise: If each stage in the system, like scheduling, register optimization, interconnect optimization etc. can be verified to perform correct transformations on the input specification, then by compositionality, we can assert that the resulting RTL design is equivalent to its input specification. This technique has the following four main steps.

1. *Characterization*: A base specification model is identified for each synthesis task. The base specification model is usually a tight set of correctness properties that completely characterizes the synthesis task.
2. *Formalization*: The base specification model is then formalized as a collection of theorems in a higher order logic theorem proving environment, which form the base formal assertions. An algorithm is also chosen to realize the corresponding synthesis task and is described in the same formal environment.
3. *Verification*: The formal description of the algorithm is verified against the base theorems. Inconsistencies in the base model are identified during the verification exercise. Furthermore, the model is enhanced with several additional formal assertions derived during verification. The formal assertions now represent the invariants in the algorithm.
4. *Formal Assertions Embedding*: In the next step a software implementation of the algorithm that was formally verified in the previous stage is developed. The much enhanced and formally verified set of formal assertions is then embedded within this software implementation as program assertions.

During synthesis, the implementation of each task is continually evaluated against its specification model specified by these assertions and any design error during synthesis can be detected.

Asserta [157] is a high-level synthesis system developed to show the effectiveness of assertion-based verification techniques to generating first-time correct RTL designs. The synthesis engine has three main stages, namely scheduling, register optimization and interconnect generation. The proof effort was conducted using Prototype Verification System (PVS) [164], a higher order logic theorem prover.

Since the main tasks of Asserta have been verified, it can be used with an increased degree of confidence. This approach is also not affected by the state space or complexities of any synthesized RTL design. However, the correctness of the

system depends on the completeness and correctness of the base assertions. Another concern is that during the formal assertions embedding step, due to difference in the expressive power of logic and software program, the translation process often could get quite complicated and finally, the correctness of the method hinges on this translation process. It is also hard to generate a tight base specification for all the steps of the synthesis process. Thus, although Asserta is a first step toward achieving correct synthesis, verifying large synthesis programs is quite tedious and complex.

3.3.2 Transformational Synthesis Tools

The basic idea of this method is to determine a set of transformations, which when applied to an initial specification, transform the source into the required implementation. This transformations are then embedded in a theorem prover to prove their correctness. The correctness of the HLS system thus follows from a 'correct by construction' argument.

Transformational synthesis is an area that has been widely explored [51, 87, 98, 126, 176, 185]. Various tools have been developed in the recent past, which mainly differ in the expressiveness of their input language, the theorem prover used and the type of transformations allowed. Sharp et al. [185] developed the T-Ruby design system, where the Ruby language is used for specifying VLSI circuits and the theorem prover Isabelle [165] is used to formalize the correctness-preserving transformations. DDD [98] is another system, which is a based on functional algebra. Both systems uses hardware specific calculus to describe a design. The following are few systems based on behavioral transformations. Veritas [87] is a theorem prover based on an extension of typed higher order logic, which provides an interactive environment for converting the specification into an implementation. Larsson [126] presented a transformational approach to digital system design based on the HOL proof system [75]. Hash [51] is another system based on the theorem prover HOL [75]. McFarland [141] investigated the correctness of behavioral transformations using behavior expressions. Rajan [176] on the other hand, used the PVS [164] theorem proving system to specify and verify behavioral transformations.

However, unlike the formal assertion technique presented in Sect. 3.3.1, techniques based on transformational synthesis reason only about the specification of the transformations, not their software implementations, which is where many of the bugs arise.

3.3.3 Witness Generator

The main idea behind witness generator techniques [54, 146, 175] is to use a set of behavior-preserving elementary transformations for validating an existing non transformational synthesis system by discovering and to some extent isolating software errors.

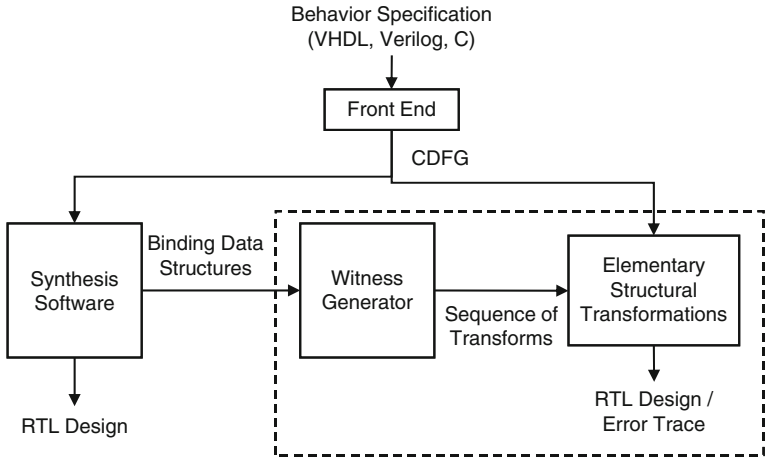


Fig. 3.1 Using a witness generator system to validate synthesis tools

Figure 3.1 shows an overview of the witness generator approach. The source program is first converted into a CDFG, after which point the CDFG passes through a regular unverified HLS process. Following the regular HLS process, the CDFG is also passed to a transformational system that consists of a set of elementary structural transformations, all of which have been formally verified given a set of preconditions. These transformations are sequenced together by the witness generator, whose goal is to find a sequence of elementary transformations which when applied to the initial design, achieve the same RTL outcome. For this technique to be broadly applicable, the set of elementary transformations must collectively capture a wide variety of synthesis algorithms.

The task of the witness generator is facilitated by the following information, which is provided by the synthesis tool:

- The outcome of a synthesis task can be captured by a simple data structure (binding data structure) such that any algorithm for this task can record its outcome in this data structure. For example, the outcome of any scheduling algorithm can be recorded as a schedule table which records the mapping between operations to control steps and the outcome of any register allocation algorithm can be recorded as mapping from variables to registers.
- It is possible to generate a sequence of elementary transformations to perform the same task by examining this data structure, without any knowledge of the synthesis algorithm used to perform the task.

When a precondition fails during the execution of the sequence of transformations identified by the witness generator, the sequence applied so far forms a counter-example that can be presented to the user.

Radhakrishnan et al. [175] identified a set of six elementary transformations which were sufficient to emulate the effect of many existing high-level synthesis

algorithms. Each of these transformations is mechanically proved in PVS [164] to preserve the computational behavior.

Eveking et al. [54] uses a similar approach to verify the correctness of various scheduling algorithms. They represented the initial CDFG using the LLS [91] internal language. After that, the process of equivalence verification consists of a number of computationally equivalent LLS transformation steps which *assimilate* the original design to the scheduled design.

Recently, Mendías et al. [146] used equational specification to describe behaviors and/or structures, in a elaborate formal framework called *Fresh*. In Fresh, seven formal derivation rules, classified into structural and behavioral were used to transform the initial design to the RTL design.

The above systems essentially recreate, within a formal framework, each of the design decisions taken by an external (and potentially incorrect) HLS algorithm. The latest HLS tools are complex and use a variety of transformations to optimize the synthesis result for metrics like area, performance and power. As a result, it is becoming increasingly difficult to find a small set of correct transformations that can recreate all the design decisions taken by external HLS tools.

3.4 Summary

In this chapter we discussed various state-of-the-art related work in the area of high-level verification. The last decade witnessed great improvements in formal methods and HLS. Recently, many commercial formal verification tools for system-level designs such as Hector [112], SLEC [193], SCADE Design Verifier [194], and Statemate [38] have become available. However, their adoption is in the early stages and believe that we will see more of such tools in the coming years.

Acknowledgments This chapter in part, has been published as:

"High-Level Verification" by Sudipta Kundu, Sorin Lerner and Rajesh Gupta in *IPSJ Transactions on System LSI Design Methodology* [118].

Chapter 4
Verification Using Automated Theorem Provers

An automated theorem prover is a tool that determines, automatically or semi-automatically, the validity of formulas in a particular logic. Automated theorem provers have been put to use in many applications. They have been used to solve open problems in mathematics, such as Robbin's problem in boolean algebra [140], which was open since the 1930s, and various open problems about quasi-groups [188]. They have also been used to prove interesting properties about real-world systems, properties that would have been hard, difficult or tedious to prove by hand. For example, automated theorem provers have been used to verify micro-processors [19, 20, 164], communication protocols [20], concurrent algorithms [20], and various properties of software systems [9, 58, 100, 130, 132, 164, 168].

From the perspective of this book, we are interested in this later use of automated theorem provers: verification of systems. Within this space, there are two broad categories of automated theorem provers. First, there are *interactive theorem provers*. Examples of such theorem provers include Coq [13], NuPrl [36], ACL2 [20], and Twelf [168]. Such theorem provers are essentially proof assistants that allow the programmer to state and prove theorems interactively. The mode and level of human interaction varies by theorem prover. For example, in the Coq theorem prover, the proof is built up in full detail by the programmer using a scripting language. In ACL2, the programmer provides hints in the form of annotations and in the form of helper lemmas that the theorem prover should attempt to prove first. Such interactive theorem provers have been used to establish the correctness of software systems, some notable examples including an entire compiler and an entire database management system. Such proof efforts require an inordinate amount of manual work, since they require formalizing every last detail in full formal logic. However, precisely because of this level of detail, once the verification is done, it provides a very high-level of assurance.

At the other end of the automation spectrum, one finds *fully automated theorem provers*, which are the theorem provers we will be interested in for this book. Such theorem provers simply take a formula, typically in some restricted logic, and return either *valid* or *invalid*. If *valid* is returned, then we know that the original formula is a tautology. If *invalid* is returned, then we learn different things depending on the completeness of the theorem prover: if the theorem prover is complete, we learn that the formula is *not* a tautology; otherwise we learn nothing – the

S. Kundu et al., *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, DOI 10.1007/978-1-4419-9359-5_4,

formula may actually be valid, but the theorem prover was simply not able to prove it. Although such fully automated theorem provers cannot prove theorems whose proofs require real insight, they are very good at quickly grinding through the large and tedious case analysis required in proofs about programs. As a result, fully automated theorem provers are very specialized tools, and as such, they are typically combined with other tools that process the system being verified in various ways to generate the actual queries to the automated theorem prover. For example, SLAM [9] repeatedly sends queries to an automated theorem prover to perform predicate abstraction within a larger outer loop of counter-example guided abstraction refinement. Cobalt [131], Rhodium [133] and PEC [120] attempts to verify the correctness of compiler optimizations by querying a theorem prover with various logical formulas generated from optimizations written in a domain-specific language.

Within fully automated theorem provers, the kinds of formulas that typically arise in program verification are first-order logic formulas that make use of various *theories*, for example the theory of integers, reals, arrays, lists, or bit-vectors. Such formulas are known to be instances of a problem called *Satisfiability Modulo Theories*, or SMT. As a result, so-called *SMT solvers* are the most common kind of fully automated theorem prover used in program verification. Two prominent examples of such SMT solvers include SIMPLIFY [44] and Z3 [152]. For the remaining of this chapter, we first give an overview of SMT solvers (Sect. 4.1), then cover two of the main verification techniques used in combination with SMT solvers, namely Hoare logic (Sect. 4.2) and weakest preconditions (Sect. 4.3), and finally, we cover several additional complexities that arise in realistic programs (Sect. 4.4).

4.1 Satisfiability Modulo Theories

Formulas that typically arise during system verification typically involve several theories. SMT solvers are meant to determine the validity of precisely such formulas. An SMT solver typically has a decision procedure for each theory that it handles. A decision procedure for a theory is simply an algorithm for determining the validity of logical formulas in that theory. Having individual decision procedures, however, is not enough, since the formulas such as the one above span *multiple* theories. An SMT solver must therefore also have an approach for *combining* individual decision procedures. The predominant approach for doing this is the Nelson–Oppen approach [161], used both in SIMPLIFY and Z3. In this approach, the individual theories simply communicate with each other by propagating equalities between variables.

In addition to combining theories, SMT solvers must also have a way to handle quantifiers, namely $\forall$ and $\exists$. The main challenge in doing so is to figure out when and how to instantiate universally quantified assumptions. One of the common techniques for doing this is a heuristic called *matching*. Suppose for example that

$\forall x_1 \ldots x_n.P$ is known to hold. The goal then is to find substitutions θ giving values to $x_1 \ldots x_n$ such that $\theta(P)$ will be useful in the proof. The general idea in matching is to pick a term t from P called a trigger, and to instantiate P with a substitution θ if $\theta(t)$ is a term that the prover is likely to require information about. Each prover has its own way of deciding when it wants information about a term $\theta(t)$. In SIMPLIFY and Z3, the theorem prover checks to see if $\theta(t)$ is represented in a data structure called the E-graph, which stores all the currently know equalities. The intuition of why matching is a good heuristic is that, since P contains the term t, $\theta(P)$ will provide information about $\theta(t)$, and this is likely to be useful since the prover wants information about $\theta(t)$. The choice of a trigger is obviously important. Various heuristic can be used to determine a trigger automatically, for example by picking the smallest term, or set of terms, that cover all the quantified variables. Many SMT solvers also supports user-defined triggers.

We have now seen an overview of SMT solvers. However, a system developer does not typically interact with the SMT solver directly. Instead, an automated tool usually sits between the developer and the SMT solver. Such tools process the system being verified to generate queries to the SMT solver. In doing so, these tools make use of variety of formal techniques to guarantee that the formulas established by the SMT solver really do imply that the system being verified has the properties it is meant to have. We will next cover two of these formal techniques: Hoare Logic and Weakest Preconditions.

4.2 Hoare Logic

Hoare logic (sometimes also called Floyd–Hoare logic) is a formal system introduced by C. A. R. Hoare in 1969 [92], following ideas published earlier by Robert Floyd in 1967 [60]. Hoare logic can be used to reason formally about the correctness of programs by specifying precisely what conditions on entry to a program guarantee what conditions on exit.

Hoare logic makes use of *assertions* to specify conditions at the entry and exit points of a program. An assertion is simply a predicate over a program state. Examples of assertions include: $\mathtt{x} > 5$, $\mathtt{x} = \mathtt{y}$, or $\mathtt{x} = \mathtt{y} + \mathtt{z}$. Each one of these assertions evaluates to either true or false in a program state. For example, in a simple programming language with no heap, the program state simply indicates what value each variables has. Consider the program state where $\mathtt{x}$ has the value 1, $\mathtt{y}$ has the value 2, and $\mathtt{z}$ has the value 3; in this program state, $\mathtt{x} > 5$ does not hold, $\mathtt{x} = \mathtt{y}$ does not hold, and $\mathtt{z} = \mathtt{x} + \mathtt{y}$ holds.

The main judgment of Hoare logic is a *Hoare triple* $\{P\}\ S\ \{Q\}$, where P and Q are assertions as explained above, and S is a program. The Hoare triple $\{P\}\ S\ \{Q\}$ states that if S starts in a state satisfying P, and it terminates, then it terminates in

a state satisfying Q. The assertion P is called the *precondition*, and Q is called the *postcondition*. Some examples of Hoare triples include:

$$\{true\}\ \mathtt{x} := 5\ \{\mathtt{x} = 5\}$$
$$\{\mathtt{x} = 0\}\ \mathtt{x} := 5\ \{\mathtt{x} = 5\}$$
$$\{\mathtt{x} = 5\}\ \mathtt{z} := 0\ \{\mathtt{x} = 5\}$$
$$\{\mathtt{x} = 1\}\ \mathtt{y} := \mathtt{x} + 1\ \{\mathtt{y} = 2\}$$
$$\{\mathtt{x} \leq 7\}\ \mathtt{x} := \mathtt{x} + 1\ \{\mathtt{x} \leq 8\}$$
$$\{\mathtt{x} = 1 \wedge y \leq 9\}\ \mathtt{x} := \mathtt{x} + 1;\ \mathtt{y} := \mathtt{y} - 2\ \{\mathtt{x} = 2 \wedge y \leq 7\}$$
$$\{\mathtt{x} = \mathtt{y} + \mathtt{z}\}\ \mathtt{x} := \mathtt{x} + 1;\ \mathtt{y} := \mathtt{y} - 2\ \{\mathtt{x} = \mathtt{y} + \mathtt{z} + 3\}$$

One thing to notice about the above examples is that one statement, for example $\mathtt{x} := 5$, can satisfy many Hoare triples. The relation between all the Hoare triples for a given statement is captured by what is known as the *rule of consequence*:

$$\text{if } P' \Rightarrow P \text{ and } \{P\}\ S\ \{Q\} \text{ and } Q \Rightarrow Q' \text{ then } \{P'\}\ S\ \{Q'\} \tag{4.1}$$

This rule states that if we have $\{P\}\ S\ \{Q\}$, then we can always replace P with something that implies P and Q with something that Q implies.

In general, this is just one of many rules that as a whole form Hoare logic. Whereas the above rule of consequence is geared towards relating all the Hoare triples for an *arbitrary* statement S, the other rules in Hoare logic are geared towards defining Hoare triples for the *specific* kinds of statements in a language (for example assignment, conditionals, loops). However, we will actually not go into the details of the other rules of Hoare logic here, because we will instead use a reformulation of Hoare logic based on the notion of *weakest precondition*.

4.3 Weakest Preconditions

Weakest preconditions, which were introduced by Edsger W. Dijkstra in 1975 [45], provide an alternate view of Hoare Logic. Even though Hoare Logic provides rules for showing that a Hoare triple holds, it does not by itself define an algorithm for automatically combining these rules to show that a particular Hoare triple holds. Weakest preconditions, on the other hand, do precisely this: they provide a mechanism for automatically finding a valid way to combine Hoare rules to establish a Haore triple.

However, we're not quite ready yet to define weakest preconditions. Before doing that, we must first define the notion of *stronger* and *weaker* predicates. In particular, if $P \Rightarrow Q$, then we say that P is *stronger* than Q, and Q is *weaker* than P. In essence, P imposes more restrictions on the state, and Q imposes less restrictions. The strongest predicate is *false* (since *false* implies anything) and the weakest predicate is *true* (since any predicate implies *true*).

Having seen what weaker/stronger predicates are, we can now define weakest preconditions. In particular, the *weakest precondition* of a predicate Q with respect to a program S, denoted by $\mathsf{wp}(S, Q)$, is the weakest predicate P such that $\{P\}\ S\ \{Q\}$. More specifically, $\mathsf{wp}(S, Q) = P$ if and only if the following two conditions hold:

$$\{P\}\ S\ \{Q\} \tag{4.2}$$

$$\text{for all } P' \text{ such that } \{P'\}\ S\ \{Q\},\ P' \Rightarrow P \tag{4.3}$$

Immediately from this definition of wp, we see that:

$$\{P'\}\ S\ \{Q\} \text{ if and only if } P' \Rightarrow \mathsf{wp}(S, Q) \tag{4.4}$$

Property (4.4) is at the core of many verification algorithms. Before seeing how this property is used, let's see why it holds. In particular, we'll show the left-to-right direction and then the right-to-left direction of the "if and only if" in (4.4). In both of these directions, let $P = \mathsf{wp}(S, Q)$. In the left-to-right direction, we assume $\{P'\}\ S\ \{Q\}$ and we need to show $P' \Rightarrow P$. From part two of the wp definition, namely condition (4.3), and from $\{P'\}\ S\ \{Q\}$, we immediately get $P' \Rightarrow P$. In the right-to-left direction, we assume $P' \Rightarrow P$ and we need to show $\{P'\}\ S\ \{Q\}$. From part one of the wp definition, namely condition (4.2), we have $\{P\}\ S\ \{Q\}$. Using the rule of consequence (4.1), combined with $P' \Rightarrow P$, we then get $\{P'\}\ S\ \{Q\}$.

To see how property (4.4) can be used for verification, let's assume for now that we have an automated way of computing wp, and that we have an SMT solver of the kind described in Sect. 4.1. Then property (4.4) gives us a way of performing automated program verification: to establish $\{P\}\ S\ \{Q\}$, all we need to do is compute $\mathsf{wp}(S, Q)$, and then ask the SMT solver to show $P \Rightarrow \mathsf{wp}(S, Q)$.

This verification approach sounds simple, elegant, and in the end quite appealing. However, in presenting the approach, we made one huge assumption, which is that wp can be computed. We need to carefully revisit this assumption. After all, the definition of wp given so far is *descriptive*, in that it tells us the properties that wp must satisfy, but it is not *prescriptive*, in that it does not tell us how to compute $\mathsf{wp}(S, Q)$. So, can $\mathsf{wp}(S, Q)$ be computed? Unfortunately, in the general case, the answer is no – computing $\mathsf{wp}(S, Q)$ in general is undecidable. However, it turns out that the main source of undecidability lies in looping constructs, and although loops are important, for the sake of simplicity, we will first present the case without loops, and then discuss loops later.

If we ignore loops, then $\mathsf{wp}(S, Q)$ becomes computable using very simple syntactic rules. Each statement kind in the programming language has an associated rule. We cover here the most important statement kinds: assignment, sequences of statements, and conditionals.

Skip. The simplest weakest precondition rule is for the no-op statement `skip`:

$$\mathsf{wp}(\texttt{skip}, Q) = Q$$

Assignment. Assignment, on the other hand, is more complicated. The weakest precondition for assignments is given by:

$$\mathsf{wp}(X := E, Q) = Q[X \mapsto E]$$

In the above, X is a variable, E is a pure expression with no side-effects, and $Q[X \mapsto E]$ stands for the predicate Q with every occurrence of the variable X replaced with the expression E. To see why this rule works, note that whenever the postcondition Q refers to X, it is referring to the value of X *after* the assignment. Thus, the weakest precondition *before* the assignment is that Q must hold, but on the value of X *after* the assignment. The problem here is that before the assignment, any references to X refer, not surprisingly, to the value of X before the assignment. So to make Q in the precondition refer to the value of X *after* the assignment, we simply replace X with E, its value after the assignment.

Here are several examples of computing the weakest precondition for assignments:

$$\begin{aligned}
\mathsf{wp}(\mathtt{x} := 5, \mathtt{x} = 5) &= 5 = 5 \\
\mathsf{wp}(\mathtt{z} := 0, \mathtt{x} = 5) &= \mathtt{x} = 5 \\
\mathsf{wp}(\mathtt{y} := \mathtt{x} + 1, \mathtt{y} = 2) &= \mathtt{x} + 1 = 2 \\
\mathsf{wp}(\mathtt{x} := \mathtt{x} + 1, \mathtt{x} \leq 8) &= \mathtt{x} + 1 \leq 8 \\
\mathsf{wp}(\mathtt{z} := \mathtt{x} + \mathtt{y}, \mathtt{z} \leq 8) &= \mathtt{x} + \mathtt{y} \leq 8
\end{aligned}$$

Sequencing. The weakest precondition for sequences of statements is given by:

$$\mathsf{wp}(S_1; S_2, Q) = \mathsf{wp}(S_1, \mathsf{wp}(S_2, Q))$$

Essentially, we first compute the weakest precondition with respect to S_2, and then with respect to S_1. Here are several examples:

$$\begin{aligned}
\mathsf{wp}(\mathtt{x} := \mathtt{x} + 1;\ \mathtt{y} := \mathtt{y} - 2,\ \ \mathtt{x} = 2 \wedge y \leq 7) &= \mathtt{x} + 1 = 2 \wedge y - 2 \leq 7 \\
&= \mathtt{x} = 1 \wedge y \leq 9 \\
\mathsf{wp}(\mathtt{x} := \mathtt{x} + 1;\ \mathtt{y} := \mathtt{y} - 2,\ \ \mathtt{x} = \mathtt{y} + \mathtt{z} + 3) &= \mathtt{x} + 1 = \mathtt{y} - 2 + \mathtt{z} + 3 \\
&= \mathtt{x} = \mathtt{y} + \mathtt{z}
\end{aligned}$$

Conditionals. The weakest precondition for conditionals is given by:

$$\mathsf{wp}(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, Q) = (B \Rightarrow \mathsf{wp}(S_1, Q)) \wedge (\neg B \Rightarrow \mathsf{wp}(S_2, Q))$$

The intuition is that if B holds, then S_1 executes and so $\mathsf{wp}(S_1, Q)$ must hold; if B does not hold, then S_2 executes and $\mathsf{wp}(S_2, Q)$ must hold. For example:

$$
\begin{aligned}
&\mathsf{wp}(\textbf{if } \mathtt{a} < 0 \textbf{ then } \mathtt{a} := -\mathtt{a} \textbf{ else } \mathtt{skip}, \mathtt{a} = 5) \\
&= (\mathtt{a} < 0 \Rightarrow \mathsf{wp}(\mathtt{a} := -\mathtt{a}, \mathtt{a} = 5)) \wedge (\neg(\mathtt{a} < 0) \Rightarrow \mathsf{wp}(\mathtt{skip}, \mathtt{a} = 5)) \\
&= (\mathtt{a} < 0 \Rightarrow -\mathtt{a} = 5) \wedge (\neg(\mathtt{a} < 0) \Rightarrow \mathtt{a} = 5) \\
&= (\mathtt{a} < 0 \Rightarrow \mathtt{a} = -5) \wedge (\neg(\mathtt{a} < 0) \Rightarrow \mathtt{a} = 5)
\end{aligned}
$$

This is equivalent to $(\mathtt{a} < 0 \Rightarrow \mathtt{a} = -5) \wedge (\mathtt{a} \geq 0 \Rightarrow \mathtt{a} = 5)$, which in turn, is equivalent to $\mathtt{a} = -5 \vee \mathtt{a} = 5$.

Recap Example. We now show an example that puts together all the statement kinds we've seen so far, and shows how we can use weakest preconditions and an SMT solver to perform program verification.

Say we want to establish the following Hoare triple:

$$
\begin{aligned}
&\{\mathit{true}\} \\
&\quad S_1 : \ \mathtt{a} := \mathtt{x}; \\
&\quad S_2 : \ \textbf{if } \mathtt{a} < 0 \textbf{ then } \mathtt{a} := -\mathtt{a} \textbf{ else } \mathtt{skip}; \\
&\quad S_3 : \ \textbf{if } \mathtt{z} > 0 \textbf{ then } \mathtt{z} := \mathtt{z} - 1 \textbf{ else } \mathtt{skip} \\
&\{\mathtt{a} \geq 0\}
\end{aligned}
$$

The above code computes the absolute value of `x` and stores the result in `a`. It also decrements `z`. The Hoare triple states that after this computation, the value in `a` (which is the absolute value of `x`) must be positive, no matter what the original value of `x` is.

First, we start by systematically computing the weakest precondition with respect to the above program, using the rules we have already seen:

$$
\begin{aligned}
&\mathsf{wp}(S_1; S_2; S_3, \mathtt{a} \geq 0) \\
&= \mathsf{wp}(S_1, \mathsf{wp}(S_2; S_3, \mathtt{a} \geq 0)) \\
&= \mathsf{wp}(S_1, \mathsf{wp}(S_2, \mathsf{wp}(S_3, \mathtt{a} \geq 0))) \\
&= \mathsf{wp}(S_1, \mathsf{wp}(S_2, (\mathtt{z} > 0 \Rightarrow \mathtt{a} \geq 0) \wedge (\neg(\mathtt{z} > 0) \Rightarrow \mathtt{a} \geq 0))) \\
&= \mathsf{wp}\left(S_1, \begin{bmatrix} (\mathtt{a} < 0 \Rightarrow ((\mathtt{z} > 0 \Rightarrow -\mathtt{a} \geq 0) \wedge (\neg(\mathtt{z} > 0) \Rightarrow -\mathtt{a} \geq 0))) \wedge \\ (\neg(\mathtt{a} < 0) \Rightarrow ((\mathtt{z} > 0 \Rightarrow \mathtt{a} \geq 0) \wedge (\neg(\mathtt{z} > 0) \Rightarrow \mathtt{a} \geq 0))) \end{bmatrix}\right) \\
&= \begin{bmatrix} (\mathtt{x} < 0 \Rightarrow ((\mathtt{z} > 0 \Rightarrow -\mathtt{x} \geq 0) \wedge (\neg(\mathtt{z} > 0) \Rightarrow -\mathtt{x} \geq 0))) \wedge \\ (\neg(\mathtt{x} < 0) \Rightarrow ((\mathtt{z} > 0 \Rightarrow \mathtt{x} \geq 0) \wedge (\neg(\mathtt{z} > 0) \Rightarrow \mathtt{x} \geq 0))) \end{bmatrix}
\end{aligned}
$$

Then, recall that to establish the Hoare triple, our verification strategy is to make use of (4.4). Thus, we ask an SMT solver to show that the precondition implies the weakest precondition, namely:

$$\mathit{true} \Rightarrow \mathsf{wp}(S_1; S_2; S_3, \mathtt{a} \geq 0)$$

The above condition, which is sent to the SMT solver to discharge, is in general called a *verification condition*. That is to say, a verification condition is a condition that, if established, will guarantee the property we are aiming to show about our program. The process of computing the verification condition is typically called *verification condition generation*, sometimes abbreviated as *VCGen*.

In our case, let's take the above verification condition, and convert it into the input format of an SMT solver. For example, in SIMPLIFY's input format, the verification condition would look like:

```
(IMPLIES TRUE
   (AND (IMPLIES (< x 0)
                 (AND (IMPLIES (> z 0) (>= (- 0 x) 0))
                      (IMPLIES (<= z 0) (>= (- 0 x) 0))))
        (IMPLIES (>= x 0)
                 (AND (IMPLIES (> z 0) (>= x 0))
                      (IMPLIES (<= z 0) (>= x 0))))))
```

The above query is actually a rather simple SMT query, and if we send it to any of the prominent SMT solvers, for example Z3 [152] or SIMPLIFY [44], we would get back *valid*, indicating that the condition holds. Thus, using simple syntactic techniques for computing the weakest precondition, combined with an SMT solver, we are able to automatically establish the above Hoare triple. This verification technique of using weakest preconditions combined with an SMT solver is in fact at the core of several important program verification tools such as ESCJava [58] and Boogie [11], and several translation validation tools [116, 117] (which will be covered later in the book).

4.4 Additional Complexities for Realistic Programs

4.4.1 Path-Based Weakest Precondition

In some cases, including many of the techniques described in this book, it is easier to perform weakest preconditions along the paths of the control flow graph (CFG), rather on the syntactic structure of the program. In general, given a point in the CFG, the weakest precondition at that point is the conjunction of all the weakest preconditions along the CFG paths that start at that point. We will make this clearer in just a moment with an example, but before going through an example we must first introduce a new statement kind that is necessary for path-based weakest preconditions.

In particular, to model assumptions along a path (due to conditionals for example), we introduce a special statement **assume** B, where B is a boolean condition. This statement encodes the fact that during verification we are allowed to assume

that B holds at the point where the assume statement is found. An assume statement is useful in verification when B is somehow known to hold from somewhere outside of the verification framework, and we want to allow the verification to take advantage of this. The most common form of assume comes from conditionals: once we are in the true side of a conditional, we can assume that the branch condition holds; similarly, on the false side of a conditional, we can assume that the negation of the branch condition holds. Thus, if we look at a conditional statement **if** B **then** S_1 **else** S_2, and we convert this to a CFG with assume statements, there would be two paths through this statement: (1) **assume** $B; S_1$ and (2) **assume** $\neg B; S_2$.

As with any other statement kind, assume statements also have a weakest precondition rule. In particular:

$$\mathsf{wp}(\textbf{assume } B,\ Q) = B \Rightarrow Q$$

This encodes precisely the idea that we are allowed to assume B when trying to establish the property Q after the statement **assume** B.

Now that we have seen what assume statements are and how to compute their weakest preconditions, let's return to path-based weakest preconditions. We had stated previously that the weakest precondition at a point in the CFG is the conjunction of all the weakest preconditions along the CFG paths that start at that point. As an example, let's try to derive the weakest precondition rule for conditionals that we have already seen using a path-based approach.

So consider the conditional statement **if** B **then** S_1 **else** S_2. As we saw before, there are two paths through this conditional, and so using the path-based approach to weakest preconditions, we would take the conjunction of the weakest precondition along the two paths:

$$\begin{aligned}
&\mathsf{wp}(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, Q) \\
&\quad = \mathsf{wp}(\textbf{assume } B; S_1, Q) \wedge \mathsf{wp}(\textbf{assume } \neg B; S_2, Q) \\
&\quad = \mathsf{wp}(\textbf{assume } B, \mathsf{wp}(S_1, Q)) \wedge \mathsf{wp}(\textbf{assume } \neg B, \mathsf{wp}(S_2, Q)) \\
&\quad = B \Rightarrow \mathsf{wp}(S_1, Q) \wedge \neg B \Rightarrow \mathsf{wp}(S_2, Q)
\end{aligned}$$

This coincides precisely with the rule we had previously seen for conditionals. Although we only show one example here, the path-based approach combined with assume statements generalizes to many different kinds of common control-flow, all of which insert assumptions along various paths, for example switch statements, pattern matching, and object oriented dynamic dispatch. In additional, assume statements are useful in many other settings, and so one typically has to have the infrastructure to deal with them anyway. For these reasons, many of the techniques presented in this book in fact use the path-based approach to weakest preconditions.

4.4.2 Pointers

The examples we have seen so far used a very simple language with variables, constants and primitive operations like $+$ and $-$. Realistic languages have additional features such as pointers which complicate the computation of the weakest precondition.

Consider for example $\mathsf{wp}(\mathtt{x} := *\mathtt{y}+1, \mathtt{x}=5)$. Here we assume for the moment that pointers such as $\mathtt{y}$ can point only to variables. The regular assignment rules tells us to replace $\mathtt{x}$ with $*\mathtt{y}+1$ in the postcondition, which gives us $*\mathtt{y}+1=5$ (in turn this is equivalent to $*\mathtt{y}=4$). It turns out that this is in fact the correct weakest precondition, and so we may be tempted to think that the regular rules work even in the face of pointers.

However, unfortunately, this is not the case. Note how in the previous example we started with a predicate $x=5$ that did not contain a pointer, but we finished with a predicate $*\mathtt{y}=4$ that does. So let's see what happens when we *start* with a predicate that contains a pointer. For example, consider $\mathsf{wp}(\mathtt{x} := *\mathtt{y}+1, \mathtt{x}=*\mathtt{y}+1)$. The regular assignment rule tells us to replace $\mathtt{x}$ with $*\mathtt{y}+1$ in the postcondition, which gives us $*\mathtt{y}+1=*\mathtt{y}+1$, which in turn is equivalent to *true*. In other words, no matter what the starting state is, the postcondition holds. But this is completely wrong: if $\mathtt{y}$ happens to point to $\mathtt{x}$, then the postcondition is $\mathtt{x}=\mathtt{x}+1$, which can *never* be true ($\mathtt{x}$ can never be equal to itself plus one). So at the very least, the precondition should state that $\mathtt{y}$ cannot point to $\mathtt{x}$. In fact, in this example, this is precisely the weakest precondition: $\mathtt{y} \neq \&\mathtt{x}$.

To see why this is the weakest precondition, we can do a case analysis on whether $\mathtt{y}$ points to $\mathtt{x}$ or to some other variable (say $\mathtt{z}$), and then in each case do the weakest precondition using the regular rules. In particular:

$$
\begin{aligned}
&\mathsf{wp}(\mathtt{x} := *\mathtt{y}+1, \mathtt{x}=*\mathtt{y}+1) \\
&= \begin{bmatrix} (\mathtt{y}=\&\mathtt{x} \Rightarrow \mathsf{wp}(\mathtt{x} := \mathtt{x}+1, \mathtt{x}=\mathtt{x}+1)) \wedge \\ (\mathtt{y}=\&\mathtt{z} \Rightarrow \mathsf{wp}(\mathtt{x} := \mathtt{z}+1, \mathtt{x}=\mathtt{z}+1)) \end{bmatrix} \\
&= \begin{bmatrix} (\mathtt{y}=\&\mathtt{x} \Rightarrow \mathtt{x}+1=\mathtt{x}+1+1) \wedge \\ (\mathtt{y}=\&\mathtt{z} \Rightarrow \mathtt{z}+1=\mathtt{z}+1) \end{bmatrix} \\
&= (\mathtt{y}=\&\mathtt{x} \Rightarrow \mathit{false}) \wedge (\mathtt{y}=\&\mathtt{z} \Rightarrow \mathit{true}) \\
&= \mathtt{y} \neq \&\mathtt{x}
\end{aligned}
$$

This example suggests one approach for handling pointers, which is to perform case analysis on all the possible aliasing scenarios. Assuming we are computing $\mathsf{wp}(S,Q)$, let X be the set of variables used in S or Q, and P be the set of variables that are *dereferenced* in S or Q (that is to say the variables p such that $*p$ occurs in S or Q). Note that P is a subset of X. Starting with the original set X, we add to X one fresh variable that is different from all other variables in X, representing the case where a pointer points to none of the variables in the statement or postcondition.

We now have one case for each possible assignment of variables in P to the address of variables in X. Note that there are $|P|^{|X|}$ cases, since there are $|P|$ pointers, each of which can point to $|X|$ variables. For each case, the aliasing becomes clear, and so the weakest precondition is computable as before. The results for all the cases are combined in the same way as in the example above, using implication ($\Rightarrow$) and conjunction ($\wedge$).

Although this approach is feasible, it is complex to implement, it does not generalize easily to other kinds of aliasing such as arrays and dynamically allocated heaps, and it leads to a large number of cases in the weakest precondition formula. For example, if the statement is $*\mathtt{x} := *\mathtt{y} + *\mathtt{z}$, then $|P| = 3$, $|X| = 4$, and so there are $3^4 = 81$ cases.

There is another approach to dealing with aliasing that is much simpler and avoids the above drawbacks. The key insight in this approach is that by making the program data state *explicit*, rather than *implicit*, we can get a simpler formulation of the weakest precondition computations. Indeed, up until now, when we said that a predicate holds (for example $\mathtt{x} = 5$), we left it implicit in what program data state it holds. If the predicate was used as a *postcondition*, then it would hold in the state *after* the statement; if it was used as a *precondition*, then it would hold in the state *before* the statement. We never gave formal names to the program data states before and after the statement, and they never appeared in the formalism. Alternatively, one can make the program data state explicit in the formalism, by remarking that a predicate is simply a unary relation on states, or equivalently a function from states to booleans. We use σ for a program data state, and so a predicate Q is a function that takes a program data state σ and returns the boolean $Q(\sigma)$ stating whether Q holds in that state.

In addition to the change from implicit states to explicit states, we also make use of a function *step*, which implements the forward semantics of statements in the programming language. In particular, given a program data state σ and a program S, $step(S,\sigma)$ returns the program data state that results from executing S starting in state σ.

With this new definition of program data states, and with the new *step* function, we can express the weakest precondition computation as follows:

$$\mathsf{wp}(S,Q)(\sigma) = Q(step(S,\sigma)) \tag{4.5}$$

This basically states that for the weakest precondition $\mathsf{wp}(S,Q)$ to hold on a state σ right before executing S, the postcondition Q must hold on the state resulting from executing S starting in σ – in other words, the weakest precondition holds on precisely those states that make the postcondition hold after executing S.

As an example, we show how to use this new formulation of weakest preconditions to compute $\mathsf{wp}(*\mathtt{x} := \mathtt{y} + 1, \mathtt{y} = 5)$. Because the program data state is now explicit, the predicate $\mathtt{y} = 5$ must be re-expressed as a predicate Q over a state σ. We do this using the *map* theory from SMT solvers. The map theory represents maps from indices to values. It has two operators, *select* and *store*. The term $select(m,i)$ represents the value in map m at index i. The term $store(m,i,v)$ represents a new

map that is identical to m, except that index i is updated to contain v. There are two axioms that govern the theory of maps:

$$i = j \Rightarrow \mathit{select}(\mathit{store}(m,i,v),j) = v$$
$$i \neq j \Rightarrow \mathit{select}(\mathit{store}(m,i,v),j) = \mathit{select}(m,j)$$

The first axiom, a shorter version of which is $\mathit{select}(\mathit{store}(m,i,v),i) = v$, states that if we store a value at a given index, then reading at the same index will return the original value. The second axiom states that if we store a value at a given index, then reading at a *different* index returns the value from the original map, before the store. The map theory is commonly used theory in formal verification, and SMT solvers have become very effective at reasoning about maps.

Returning to our example, we represent a program data state σ as a map from program variables to values, and we use *select*/*store* to model variable reads/writes. For example, the predicate $\mathtt{y} = 5$ would be expressed as $Q(\sigma) = [\mathit{select}(\sigma,\mathtt{y}) = 5]$. Since for simplicity we assume that our programs only have variables and no dynamically allocated memory, a pointer value is simply the name of a variable. For example, a use of $*\mathtt{y}$ would be expressed as $\mathit{select}(\sigma,\mathit{select}(\sigma,\mathtt{y}))$.

We now need to understand more carefully what *step* does. In general, *step* just implements an operational semantics for the language, by simply defining how to run statements in the language. For example, for the assignment $*\mathtt{x} := \mathtt{y}+1$, *step* needs to first read the value of $\mathtt{x}$ – this value, call it v, is the variable in σ where the result of $\mathtt{y}+1$ should be stored. The *step* function should then read $\mathtt{y}$ from σ, add 1 to this value, and then store the result in σ at v. This can be formalized using *select* and *store* as follows:

$$\mathit{step}(*\mathtt{x} := \mathtt{y}+1,\sigma) = \mathit{store}(\sigma,\mathit{select}(\sigma,\mathtt{x}),\mathit{select}(\sigma,\mathtt{y})+1)$$

Here we only show the definition of *step* for the assignment $*\mathtt{x} := \mathtt{y}+1$, but in general *step* is straightforward to define for other cases, even in the presence of pointers, arrays and heaps – *step* just formalizes our common intuition of how statements in the language run.

We are now ready to apply (4.5) to compute the weakest precondition for our example:

$$\begin{aligned}
&\mathsf{wp}(*\mathtt{x} := \mathtt{y}+1, Q)(\sigma) \quad \text{where } Q(\sigma) \;=\; [\mathit{select}(\sigma,\mathtt{y}) = 5] \\
&\quad = Q(\mathit{step}(*\mathtt{x} := \mathtt{y}+1,\sigma)) \\
&\quad = \mathit{select}(\mathit{step}(*\mathtt{x} := \mathtt{y}+1,\sigma),\mathtt{y}) = 5 \\
&\quad = \mathit{select}(\mathit{store}(\sigma,\mathit{select}(\sigma,\mathtt{x}),\mathit{select}(\sigma,\mathtt{y})+1),\mathtt{y}) = 5
\end{aligned}$$

4.4.3 Loops

So far, we have dealt with loop-less code. The Hoare logic rule for loops is:

$$\text{if } \{P \wedge B\}\ S\ \{P\} \text{ then } \{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end}\ \{\neg B \wedge P\} \tag{4.6}$$

In the above rule, P is a *loop invariant*, which is a predicate that holds at each iteration of a loop. Indeed, $\{P \wedge B\}\ S\ \{P\}$ essentially guarantees that the body S of the loop **while** B **do** S **end** preserves P: if P holds before executing S, it will also hold after executing S, which means it will hold at the beginning of the next iteration. The addition of B in $\{P \wedge B\}\ S\ \{P\}$ accounts for the fact that when we are about to execute S, we just tested the loop condition B and it must have evaluated to true (note that we could alternatively have used an assume statement, as such: $\{P\}$ **assume** $B; S\ \{P\}$).

Rule (4.6) states that if P is a loop invariant, meaning that $\{P \wedge B\}\ S\ \{P\}$ holds, then if P holds going into the loop, P will be preserved throughout the loop and hold after the loop, giving us $\{P\}$ **while** B **do** S **end** $\{\neg B \wedge P\}$. The addition of $\neg B$ is justified by the fact that once the loops exits, we know that B must have just evaluated to false.

The Hoare rule for loops is very simple, but computing weakest preconditions for loops can be challenging. Consider for example:

$$\mathsf{wp}(\textbf{while}\ \mathtt{x} > 0\ \textbf{do}\ \mathtt{x} := \mathtt{x} - 1; \mathtt{y} := \mathtt{y} - 1\ \textbf{end}, y = 0)$$

According to the definition of wp, we therefore want to compute the weakest P such that:

$$\{P\}\ \textbf{while}\ \mathtt{x} > 0\ \textbf{do}\ \mathtt{x} := \mathtt{x} - 1; \mathtt{y} := \mathtt{y} - 1\ \textbf{end}\ \{y = 0\}$$

According to rule (4.6), we may be tempted to try $y = 0$ as the loop invariant. Unfortunately $y = 0$ is simply not a loop invariant for this loop. It turns out that the correct loop invariant in this case is $x = y \wedge x \geq 0$, and this is also the weakest precondition P we are seeking. Indeed, note that $x = y \wedge x \geq 0$ is preserved through the loop body:

$$\{x = y \wedge x \geq 0 \wedge x > 0\}\ \mathtt{x} := \mathtt{x} - 1; \mathtt{y} := \mathtt{y} - 1\ \{x = y \wedge x \geq 0\}$$

As a result, rule (4.6) gives us:

$$\{x = y \wedge x \geq 0\}\ \textbf{while}\ \mathtt{x} > 0\ \textbf{do}\ \mathtt{x} := \mathtt{x} - 1; \mathtt{y} := \mathtt{y} - 1\ \textbf{end}\ \{\neg(x > 0) \wedge x = y \wedge x \geq 0\}$$

Now note that $\neg(x > 0) \wedge x = y \wedge x \geq 0$ is equivalent to $x \leq 0 \wedge x = y \wedge x \geq 0$, which implies $y = 0$. Thus the rule of consequence (4.1) gives us:

$$\{x = y \wedge x \geq 0\}\ \textbf{while}\ \mathtt{x} > 0\ \textbf{do}\ \mathtt{x} := \mathtt{x} - 1; \mathtt{y} := \mathtt{y} - 1\ \textbf{end}\ \{y = 0\}$$

The important lesson of this example is that to find the weakest precondition, we had to find the right loop invariant, and unfortunately the post-condition $y = 0$ did *not* directly provide us this loop invariant. Instead, we had to *strengthen* the postcondition $y = 0$ to something which, combined with the knowledge that loop exited, would imply $y = 0$. In our case, this strengthening lead us to $x = y \wedge x \geq 0$, but in general, if the language we are dealing with is Turing complete, then it is undecidable to automatically find the weakest loop invariant that is strong enough to establish a given postcondition. For this reason, computing the *weakest* precondition through loops is undecidable.

Now, for verification, we actually don't always need to find the *weakest* precondition. Often, it is enough to just find *some* precondition. Unfortunately, it is even undecidable to just determine whether there exists *some* invariant that is strong enough to establish the postcondition. So in the end, we are left with designing *heuristics* for finding loop invariants, where we use "heuristics" to indicate that these are not algorithms. However, do not be fooled, these techniques often use very principled approaches, founded in well-established theory, including abstract interpretation [39, 40], predicate abstraction [43], and SMT solvers. In fact, there has been a huge amount of work in finding loop invariants, and many verification problems in the end boil down to finding good invariants. Not surprisingly then, the techniques described in Chaps. 7 and 8 will in part cover some simple techniques for finding such loop invariants.

Chapter 5
Execution-Based Model Checking for High-Level Designs

In this chapter, we present an high-level property checking approach. We begin with a general description of verification of concurrent programs, and then describe it for a high-level language called SystemC [78]. In this approach, we start with a design written in SystemC, and then use *model checking* techniques to verify that the design satisfies a given property such as the absence of deadlocks or assertion violations.

5.1 Verification of Concurrent Programs

Verification of multi-threaded concurrent programs is hard due to complex and unexpected interleaving between the threads. In general, the problem of verifying two-threaded programs (with unbounded stacks) is undecidable [177]. Therefore for practical reasons, the verification techniques often trades off completeness or precision or sometimes both, to address the scalability of the problem, thereby focusing only on their bug finding capabilities. The verification model is typically obtained by composing individual thread models using interleaving semantics, and model checkers are applied to systematically explore the global state space. Due to the potentially large number of interleavings of transitions from different threads, the global state space can be, in the worst case, the product state space of individual thread state space. To combat the state explosion problem, most methods employ partial-order reduction (POR) techniques to restrict the state-traversal to only a representative subset of all interleavings, thereby, avoiding exploring the redundant interleaving among independent transitions [70]. Explicit model checkers [46, 57, 71, 94] explore the states and transitions of concurrent system by explicit enumeration, while symbolic model checkers [3, 67, 103, 174, 205] manipulate sets of states, instead of individual states. Symbolic algorithms avoid explicitly building the complete state graph for the system; instead, they represent the state space implicitly using a formula in decidable subset of first-order logic. This formula is then solved using a SAT or SMT solver. In this chapter, we discuss an explicit model checking technique for SystemC designs, and in the next chapter we focus on a symbolic approach based on BMC.

S. Kundu et al., *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, DOI 10.1007/978-1-4419-9359-5_5,

5.2 Overview of SystemC

SystemC is a system description language that enables a designer to write designs at various levels of abstraction. These are particularly useful in behavioral/algorithmic and transaction level modeling (TLM) [27, 78, 192]. The idea of SystemC TLM is to provide a golden reference of the system in an early phase of the development process and allow fast simulation. This design abstraction supports new synchronization procedures such as wait-notify, which make current techniques for RTL validation inapplicable. SystemC is a set of library routines and macros implemented in C++, which makes it possible to simulate concurrent processes, each described by ordinary C++ syntax. Instantiated in the SystemC framework, the processes described in this manner may communicate in a simulated real-time environment, using shared variables, events and signals. SystemC is both a description language and a simulation kernel. The code written will compile together with the library's simulation kernel to give an executable that behaves like the described model when it is run. In Sects. 5.5 and 5.6 we will discuss some more language features of SystemC.

5.3 Problem Statement

Simulation has so far been the "workhorse" for validating SystemC designs. As pointed out in [201], adapting software formal verification techniques to SystemC has been a formidable task, mainly due to its object-oriented nature and its support for both synchronous and asynchronous semantics of concurrency along with a notion of time. Furthermore, SystemC allows features such as co-operative multitasking, delayed/immediate notification, wait-to-wait atomicity, blocking and non-blocking variable updates. In the absence of accepted formal semantics, SystemC models and methods attempt to speed up simulation. However, simulation can not guarantee completeness without being exhaustive, hence the need for formal verification techniques to improve system level simulation coverage.

5.4 Overview of Execution-Based MC for SystemC Designs

The goal of the Execution-based MC approach for SystemC designs (EMC-SC) [115] is to devise methods to check all possible execution traces of a SystemC description. It focuses on using formal verification techniques developed for software to extend dynamic validation of SystemC TLM designs. EMC-SC use an execution-based MC approach, which for a given test input and depth, systematically explores all possible behaviors of the design (due to asynchronous concurrency). The most striking benefit of this approach is that it can analyze feature-rich programming languages like C++, as it sidesteps the need to formally represent the semantics of

the programming language as a transition relation. Another key aspect of this approach is the idea of *stateless* [71] search, meaning it stores no state representations in memory but only information about which transitions have been executed so far. Although stateless search reduces the storage requirements, a significant challenge for this approach is how to handle the exponential number of paths in the program. In what follows, EMC-SC assumes the representative inputs are already provided, possibly using techniques presented in [77] and the execution terminates. Thus, we focus our discussion mainly on detecting deadlocks, write-conflicts and safety property violations such as assertion violations. Note that termination can be guaranteed in SystemC by bounding the execution length during simulation.

To cope with the exponential number of paths, EMC-SC use a combination of static and dynamic POR techniques. In particular, it first uses static analysis techniques to compute if two atomic blocks are *independent*, meaning that their execution does not interfere with each other, and changing their order of execution will not alter their combined effect. Next, it starts by executing one random trace of the program until completion, and then dynamically compute backtracking points along the trace that identify alternative transitions that need to be explored because they may lead to different final states. However, unlike dynamic techniques [57, 89] EMC-SC use the information obtained by static analysis in a *query-based* approach, rather than dynamically collecting the information and analyzing it during runtime. Using static information trades off precision for performance. This is particularly suitable for this approach because for most SystemC designs the dependency relation can be found quite precisely by using static analysis only. Intuitively, this approach infers the persistent sets dynamically using information obtained by static analysis (see Sect. 2.5.2 for an introduction to POR).

Moreover, EMC-SC adapts the POR techniques for SystemC specific semantics, thereby improving the efficiency of the algorithms. Adaptations are needed because in SystemC: processes are co-operatively multitasking; supports the concept of δ-cycle, which reduces the analysis of backtracking points immensely; supports signal variables that do not change values until an update phase; synchronization is done using events instead of locks; and enabled processes cannot be disabled by another one. This synchronous semantics of SystemC reduces the size of persistent set and consequently reduces the analysis of backtracking points immensely.

5.5 SystemC Example

Let us start by examining the salient features of SystemC using a simple producer–consumer example shown in Fig. 5.1. Though this example is simple, it highlights the kind of synchronization you will normally expect in a high-level-design language. As mentioned, SystemC is essentially a C++ library that provides macros and APIs to model hardware and software systems. A SystemC program is a set of interconnected modules communicating through channels using *transactions*, events and shared variables collectively called *communication objects*. A module comprises of

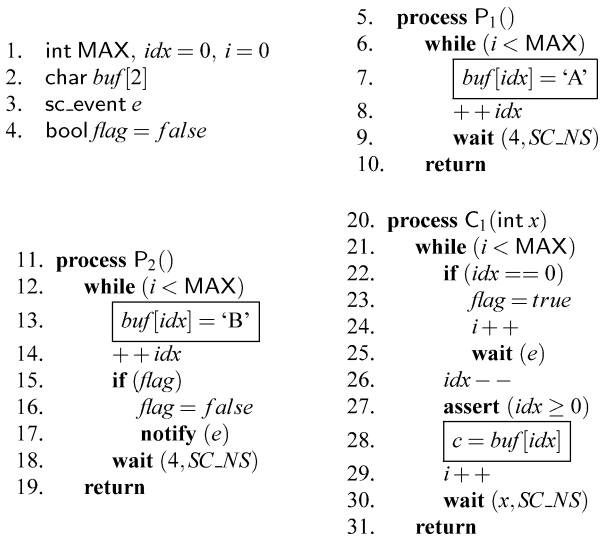


Fig. 5.1 Simple producer–consumer example

a set of ports, variables, processes and methods. Processes are small pieces of code that run concurrently with other processes and are managed by a *non-preemptive scheduler*. The semantics of concurrency is *cooperatively multitasking*: a type of multitasking in which the process currently executing must offer control to other processes. As such, a wait-to-wait block in a process is atomic. The processes exchange data between themselves using shared variables (signals and non-signals). During the execution of a SystemC design, all signal values are stable until all processes reach the waiting state. When all processes are waiting, signals are updated with the new values (see Update Phase in Sect. 5.6). In contrast, the non-signal variables are standard C++ variables which are updated immediately during execution.

For clarity the syntactic details of SystemC are not shown in Fig. 5.1. It has three processes namely P_1 (lines 5–10), P_2 (lines 11–19) and C_1 (lines 20–31). The global variables of the program are shown in lines 1–4. The program uses a shared array *buf* as a buffer, and an integer *idx*, which indicates the total number of elements in the buffer. The producer P_1 in a loop writes to the buffer and then synchronizes by waiting (or blocking) (line 9) on time for 4 nanoseconds (*SC_NS*). Similarly, producer P_2 writes to the buffer and if *flag* is set then notifies the event *e* and then synchronizes using time. The consumer C_1 on the other hand, waits (or blocks) (line 25) on the event *e* when the buffer is empty, until the notify (line 17) on *e* is invoked in the P_2 process. If there are elements in the buffer then C_1 consumes it and synchronizes on time like the other processes. For synchronization SystemC uses wait-notify on events and time. In what follows, we will use this example to guide our discussion.

5.6 SystemC Simulation Kernel

The illusion of concurrency is provided to the user by a simulation kernel implementing a discrete event scheduler. Simulation involves the execution of a discrete event scheduler, which in turn triggers or resumes the execution of processes within the application. The functionality of the scheduler (as per IEEE 1666 Language Reference Manual for SystemC [95]) can be summarized as follows:

1. *Initialization Phase*: Initialize every eligible method and thread process instance in the object hierarchy to the set of runnable processes.
2. *Evaluation Phase*: From the set of runnable processes, select a process instance in an unspecified order and execute it non-preemptively. This can, in turn, notify other events, which can result in new processes being ready to run. Continue this step until there are processes ready to run.
3. *Update Phase*: Update values of the signal variables for all processes in step 2, that requested for it.
4. δ*-Notification Phase*: Trigger all pending δ-delayed notifications, which in turn can wake up new processes and make them ready to run. If, at the end of this phase, the set of runnable processes is non-empty, go back to the evaluation phase (Step 2).
5. *Timed-Notification Phase* (τ): If there are pending timed notification, advance simulation time to the earliest deadline. Determine the set of runnable processes that are ready to run at this time and go to step 2. Otherwise, end simulation.

To simulate synchronous concurrent reactions on a sequential computer SystemC supports the concept of δ-cycle. A δ-cycle is an event cycle (consisting of evaluate, update and δ-notification phase) that occurs in 0 simulation time.

5.6.1 Nondeterminism

For a given input, a SystemC program can produce different output behavior due to nondeterministic scheduling. The election algorithm of the SystemC simulator makes the output of a program deterministic. Thus even exhaustive simulation of a SystemC program cannot guarantee correctness due to its inability to produce all possible behaviors. To illustrate this let us consider the processes P_1 and C_1 from the example in Sect. 5.5 with $\mathsf{MAX} = 2$ and $x = 4$ (line 20). It has the following 4 possible executions, where τ denotes a time elapse:

- $\mathsf{P}_1\mathsf{C}_1\tau\mathsf{P}_1\mathsf{C}_1\tau\mathsf{P}_1\mathsf{C}_1$ and $\mathsf{P}_1\mathsf{C}_1\tau\mathsf{P}_1\mathsf{C}_1\tau\mathsf{C}_1\mathsf{P}_1$ leads to a successful termination of the program with 2 A's being produced and consumed.
- $\mathsf{P}_1\mathsf{C}_1\tau\mathsf{C}_1\mathsf{P}_1$ leads to a deadlock situation. Since the process C_1 is waiting for the event *e* (line 25) and the process P_1 has terminated.
- $\mathsf{C}_1(\mathsf{P}_1\tau)^*$ leads to an array bound violation as the process C_1 waits for the event *e* and the process P_1 goes on producing in the array *buf*.

In general, a simulator will execute only one of the 4 possible executions. For instance with the reference OSCI simulation kernel [95], only the first execution will be scheduled and the other buggy executions will be ignored. Thus, it is important to test all possible execution of a SystemC design.

Now let us consider the same example with all 3 processes and $\mathsf{MAX} = 8$ and $x = 2$ (line 20). It has 3701 possible executions. A naive algorithm will try to explore all possible executions one by one and will face scalability issues. A technique which for most practical cases allows to cope with this scalability problem is POR. In the following sections we discuss EMC-SC, an approach that adapts POR techniques for SystemC, while exploring all possible behaviors of the SystemC design. For our example, this approach will explore only 767 executions and still remain *provably sufficient for detecting deadlocks and assertion violations*.

5.7 State Transition System

In this section, we describe some standard definitions used in the context of POR [57, 70, 89], which have been adapted here for SystemC.

We represent the behavior of a concurrent program using a state transition system. Semantically a concurrent program in our case consists of a set of processes (threads). The control structure of a program is represented in terms of *generalized program locations*. A generalized program location represents a point of control in the program (possibly concurrent). A generalized program location can either be a node identifier, or it can be a pair of two generalized program locations, representing the state of two processes running in parallel. For instance, a generalized program location for the example in Fig. 5.5 is $((9,18),30)$, which means the control of the program is at location 9 and 18 in processes P_1 and P_2 respectively and at location 30 in C_1.

Let $\mathscr{L}$ denote the finite set of generalized program locations, *VAR* denote the set of variables and *VAL* denote the domain of values. We define a program state to be a function $VAR \times \mathscr{L} \rightarrow VAL$, assigning values to variables and program locations. A transition then describes how the system moves from one state to a subsequent state. In SystemC there are three types of transitions:

1. *Immediate-transition* change the state by executing a finite sequence of operations of a chosen process followed by a *wait* operation or termination of the same process.
2. δ*-transition* change the state by updating all the signals, and by triggering all the δ-delayed notification that were requested in the current δ-cycle.
3. A *time-transition* change the system state by updating the simulation time.

Let $\mathscr{T}$ denote the set of all transitions of the system and Σ the set of all program states. An i^{th} transition of process P is denoted by P^i. For $t = P^i \in \mathscr{T}$ we denote, $\mathsf{Process}(t)$ as the process P.

Definition 1 (Runnable). A transition $t \in \mathscr{T}$ is runnable in state $\sigma \in \Sigma$, written $t \in \mathit{runnable}(\sigma)$ if it can be executed in σ.

If $t \in \mathit{runnable}(\sigma)$, then we say the execution of t from σ produces a successor state σ', written $\sigma \xrightarrow{t} \sigma'$. We write $\sigma \xRightarrow{w} \sigma'$ to mean that the execution of the finite sequence $w \in \mathscr{T}^*$ leads from σ to σ'. A state σ, where $\mathit{runnable}(\sigma) = \emptyset$ is called a deadlock, or a terminating state. Formally a state transition system is defined as follows.

Definition 2 (State Transition System). A state transition system is a tuple $\mathscr{M} = (\Sigma, \sigma_0, \Delta)$, where σ_0 is the initial state of the system and $\Delta \subseteq \Sigma \times \Sigma$ is the transition relation defined by

$$(\sigma, \sigma') \in \Delta \quad \text{iff} \quad \exists\, t \in \mathscr{T} : \sigma \xrightarrow{t} \sigma'$$

Definition 3 (Co-Runnable). Two transitions $t_1, t_2 \in \mathscr{T}$ are *co-runnable*, written $\mathsf{CoRunnable}(t_1, t_2)$ if $\exists\, \sigma \in \Sigma$ such that both $t_1, t_2 \in \mathit{runnable}(\sigma)$.

Note that in SystemC two transitions of the same process cannot be co-runnable. We define an execution of the program as a trace of the system.

Definition 4 (Trace). A trace $\phi \in \mathscr{T}^*$ of $\mathscr{M}$ is a finite (possibly empty) sequence of transitions $t_0, \ldots, t_n$ where there exists states $\sigma_0, \ldots \sigma_{n+1}$ such that σ_0 is the initial state of $\mathscr{M}$ and $\sigma_0 \xrightarrow{t_0} \sigma_1 \cdots \xrightarrow{t_n} \sigma_{n+1}$.

For a given trace $\phi = t_0, \ldots, t_n$:

- ϕ_i represents the transition t_i.
- $\phi_{0\ldots i}$ denotes the trace $t_0, \ldots, t_i$.
- $\mathsf{Pre}(\phi, i)$ denotes the state σ_i and $\mathsf{Post}(\phi, i)$ denotes the state σ_{i+1}.

The goal of EMC-SC is to explore all possible behavior of the system $\mathscr{M}$. However, $\mathscr{M}$ typically contains many traces that are simply different execution order of uninteracting transitions that leads to the same final state. This observation has been exploited by POR techniques to explore a subset of the possible traces [70]. The following definition states the condition when two transitions are independent, meaning that they result in the same state when executed in different orders.

Definition 5 (Independence Relation). A relation $\mathscr{I} \subseteq \mathscr{T} \times \mathscr{T}$ is an independence relation of $\mathscr{M}$ if $\mathscr{I}$ is symmetric and irreflexive and the following conditions hold for each $\sigma \in \Sigma$ and for each $(t_1, t_2) \in \mathscr{I}$:

1. if $t_1, t_2 \in \mathit{runnable}(\sigma) \ \wedge \ \sigma \xrightarrow{t_1} \sigma'$ then $t_2 \in \mathit{runnable}(\sigma')$
2. if $t_1, t_2 \in \mathit{runnable}(\sigma)$, then there is a unique state σ' such that $\sigma \xRightarrow{t_1 t_2} \sigma' \ \wedge$ $\sigma \xRightarrow{t_2 t_1} \sigma'$

Transitions $t_1, t_2 \in \mathscr{T}$ are *independent* in $\mathscr{M}$ if $(t_1, t_2) \in \mathscr{I}$. Thus, a pair of independent transitions cannot make each other runnable when executed and runnable independent transition commute. The complementary dependence relation $\mathscr{D}$ is given by $(\mathscr{T} \times \mathscr{T}) - \mathscr{I}$.

Two traces are said to be *equivalent* if they can be obtained from each other by successively permuting adjacent independent transitions. Thus, given a valid independence relation, traces can be grouped together into *equivalence classes*. For a given trace, we define a *happens-before* relation between its transitions as follows:

Definition 6 (Happens-before). Let $\phi = t_0 \ldots t_n$ be a trace in $\mathscr{M}$. A happens-before relation $\prec_\phi$ is the smallest relation on $\{0 \ldots n\}$ such that:

1. if $i \leq j$ and $(\phi_i, \phi_j) \in \mathscr{D}$ then $i \prec_\phi j$.
2. $\prec_\phi$ is transitively closed.

This happens-before relation is a partial order relation also known as *Mazurkiewicz's traces* [70]. The correspondence between equivalence classes of traces and the partial order relation $\prec_\phi$ is that the equivalence class containing the trace ϕ is the set of all linearizations of the partial order. The EMC-SC algorithm described here use a variant of the above happens-before relation which is defined as follows: for a given trace $\phi = t_0 \ldots t_n$ in $\mathscr{M}$ and $i \in \{0 \ldots n\}$, i happens-before process P, written, $i \prec_\phi P$ if either

1. $\mathsf{Process}(\phi_i) = P$ or
2. $\exists k \in \{i+1, \ldots, n\}$ such that $i \prec_\phi k \ \wedge \ \mathsf{Process}(\phi_k) = P$.

5.8 The EMC-SC Approach

In EMC-SC [115] approach, the partial-order information of *runnable* processes is obtained statically by identifying the dependent transitions. A transition in SystemC is an atomic block, which in turn is a non-preemptive sequence of operations between *wait* to *wait*. Note, due to branching within an atomic block, such blocks may not be derived statically. An atomic execution is *dependent* on another atomic execution if it is enabled or disabled by the other or there exists *read-write* conflicts on the shared variable accesses in these blocks. EMC-SC first derives wait-notify control skeleton of the SystemC design, and then enumerates all possible atomic blocks. It then performs dependency analysis on the set of atomic blocks, and represents the information symbolically. These static information are used later, while exploring the different executions of the design. In particular, this approach *queries* to check if a given pair of atomic blocks (corresponding to the runnable processes) need to be interleaved. If not, it does not consider that interleaving of runnable processes. Another alternative is to compute the partial-order information dynamically while exploring the different executions of the design [89]. The main difference between these two approaches is that the first trades off precision for runtime performance and the later trades of performance for precision. In the following sections we describe the EMC-SC algorithm in more details.

5.8.1 Static Analysis

The goal of EMC-SC is to execute only one trace from each equivalence class for a given dependence relation. Thus, the first step is to compute this dependence relation. As discussed earlier EMC-SC uses static analysis techniques to compute if two transitions are dependent. Intuitively, two transitions are dependent if they operate on some shared communication objects. In particular, the following rules are used to compute the dependence relation $\mathscr{D}$, i.e. $\forall\, t_1, t_2 \in \mathscr{T}, (t_1, t_2) \in \mathscr{D}$ if any of the following holds:

1. A write on a shared *non-signal* variable v in transition t_1 and a read or a write on the same variable v in the other transition t_2.
2. A write on a shared *signal* variable s in transition t_1 and a write on the same variable s in transition t_2.
3. A wait on an *event* e in transition t_1 and an immediate notification on the same event e in transition t_2.

Note here that the order in which the statements occur within a transition does not matter. For each transition $t \in \mathscr{T}$, EMC-SC maintains four sets – read and write sets for shared non-signal variables and shared signal variables (written, $R_{t,ns}, W_{t,ns}, R_{t,s}, W_{t,s}$ respectively). Thus, rule 1 can be re-written as,

$$(W_{t_1,ns} \cap R_{t_2,ns}) \cup (W_{t_1,ns} \cap W_{t_2,ns}) \neq \emptyset$$

And rule 2 can be re-written as,

$$W_{t_1,s} \cap W_{t_2,s} \neq \emptyset$$

In the rules mentioned above, in general, two transitions with write operations on a shared variable are dependent. But to exercise more independency special cases of write operations (called *symmetric write*) can be considered as being independent (applying Definition 5). For instance, two constant addition or constant multiplication with the same variable can be considered as being independent. Moreover, static slicing techniques can be used to remove irrelevant operations to further extract more independency between the transitions [88]. Intuitively, if a statement does not influence the property that is being checked then that statement can be removed in the sliced program.

To illustrate the above rules, consider the example from Fig. 5.1. Consider the *wait* to *wait* atomic transition consisting of the lines (6–9) in process P_1 and the transition consisting of the lines (12–18) in process P_2. In general, these two transitions are dependent because they both write to the variable *buf* and *idx*. However, if the property that we are checking is the assertion in line 27 then we can get a sliced program by removing the statements inside the boxes, while still remaining correct for detecting the assertion violation. Now, if we consider only the rules 1, 2 and 3 from above then the two transitions are still dependent in the sliced program because they both write to the variable *idx*. But, notice that both the writes to the

variable *idx* are symmetric (increment). Thus, we have that the two transitions are independent if the property that we are checking is only the assertion ($idx \geq 0$) at line 27.

5.8.2 The Explore Algorithm

The Explore algorithm shown in lines 5–19 of Fig. 5.2, explores a reduced set of possible executions of a SystemC design. This algorithm is *stateless* [71], i.e., it stores no state representations in memory but only information about which transitions and traces have been executed so far. Although, this approach will be slower than an algorithm that maintains full state information, it requires considerably less amount of memory, especially when the design has a large number of variables. It explores each *non-equivalent* trace of the system by re-executing the design from its initial state.

The algorithm maintains a sequence *sched* of type *Schedule*. A *Schedule* is a sequence of *SchedulerStates*. Each *SchedulerState* σ is a 3-tuple (*Runnable*, *Todo*, *Sleep*) where, *Runnable* is a sequence of *Transitions* that are runnable in state σ, *Todo* is a set of *Transitions* that needs to be explored from σ, and *Sleep* is the set of *Transitions* that are no longer needed to be explored from σ.

The algorithm also uses a function Simulate (not shown here) that takes as input a prefix schedule and then executes it according to the trace corresponding to the schedule. Once the prefix trace ends, it randomly chooses a runnable transition that is not in the *Sleep* set of the current state and executes it. The function continues the

```
1.  type Runnable := list of Transition
2.  type TSet := set of Transition
3.  type SchedulerState := Runnable × TSet × TSet
4.  type Schedule := sequence of SchedulerState

5.  function Explore()
6.      let sched := Simulate(∅)
7.      let depth := sched.Size − 1
8.      while depth ≥ 0 do
9.          let φ := sched.Trace
10.         let σ := sched.At(depth)
11.         Analyze(φ, depth)
12.         if ∃t ∈ σ.Todo \ σ.Sleep then
13.             σ.Runnable.Add(0, σ.Runnable.Remove(t))
14.             let newSched := sched.Copy(0, depth)
15.             sched := Simulate(newSched)
16.             depth := sched.Size − 1
17.         else
18.             depth := depth − 1
19.     return
```

Fig. 5.2 The Explore algorithm

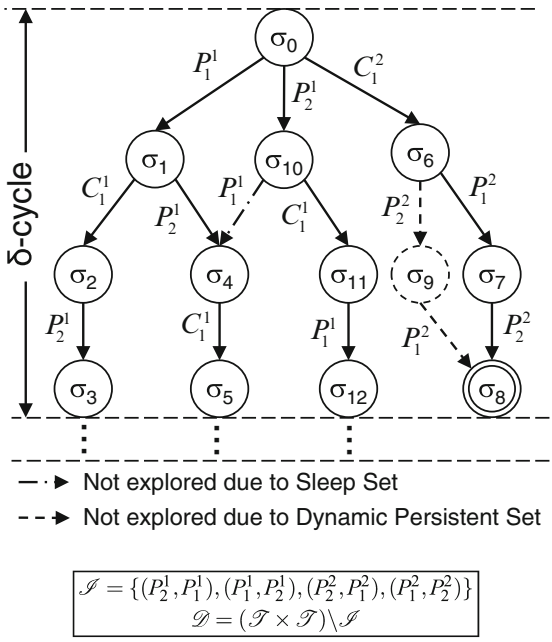
above step till completion of the simulation and returns the schedule for the current execution. The Simulate function computes the *Sleep* set for each scheduler state in the same way as explained in VeriSoft [70,71]. The use of *Sleep* sets in the algorithm help to further reduce the explored transitions.

The Explore function starts by executing a random schedule (as the prefix trace is $\emptyset$) and returns the schedule in *sched* (line 6). Next the algorithm traverses the execution-tree bottom up and *depth* maintains the position in the tree such that the sub-tree below *depth* has been fully explored. Note that by traversing the execution-tree bottom-up the Explore algorithm needs to maintain very little state information. While the algorithm has not traversed the entire execution-tree, let $sched = \sigma_0, \ldots \sigma_{depth}, \ldots \sigma_n$ and $\phi = t_0, \ldots t_i, \ldots t_{n-1}$ (line 9) is the trace corresponding to *sched* such that $\phi_i = \sigma_i.\mathsf{Runnable.At}(0)$ and $\sigma = \sigma_{depth}$ (line 10). Using the computed trace ϕ, Explore then finds out the transitions that can be dependent with the transition ϕ_{depth} using the function Analyze (line 11) and adds those in the *Todo* set of the corresponding state. Next, if there exists any transition $t \in Todo \backslash Sleep$ in the state σ (line 12), then the Explore function swap the transition t with the first element of *Runnable* in state σ (line 13), copies the prefix schedule (line 14) and simulate it using the Simulate function (line 15). Otherwise, it has explored all required transitions in the sub tree below *depth* and now will explore all the transitions in the sub tree below $depth - 1$ (line 18).

The Analyze function shown in lines 20–32 of Fig. 5.3 takes as argument a trace ϕ and an integer *depth*. Next, it finds the start of the δ-cycle to which ϕ_{depth} belongs (line 21). Then, for each transition ϕ_i such that $i < depth$ and belongs to the same delta cycle (line 22), it checks if ϕ_i and ϕ_{depth} are dependent using a query function and may be co-runnable (line 23). If true, then it computes the state $\sigma = \mathsf{Pre}(\phi, i)$ and p as the process to which the transition ϕ_{depth} belongs. Next, if there exists a transition of p that is runnable at σ (line 26) then it adds that transition to the *Todo* set of σ (line 27). Else, if there exists $j > i$ such that $j \prec_{t_{0 \cdots depth}} p$ (see Definition 6) and the runnable set of σ contains a transition that belongs to the process to which ϕ_j belongs (line 28) then it adds that transition to the *Todo* set of σ (line 29). Otherwise, it adds all runnable transitions to the *Todo* set of σ (line 31).

```
20. function Analyze(φ : 𝒯*, depth : int)
21.     let start := StartOfDeltaCycle(φ_depth)
22.     for each i | start ≤ i < depth do
23.         if Query(φ_i, φ_depth) = Dependent and CoRunnable(φ_i, φ_depth)) then
24.             let σ := Pre(φ, i)
25.             let p := Process(φ_depth)
26.             if Runnable(σ, p) then
27.                 σ.Todo := σ.Todo ∪ {Transition(σ, p)}
28.             elseif ∃ j > i | Runnable(σ, Process(φ_j)) and j ≺_{φ_{0···depth}} p then
29.                 σ.Todo := σ.Todo ∪ {Transition(σ, Process(φ_j))}
30.             else
31.                 σ.Todo := σ.Runnable
32.     return
```

Fig. 5.3 The Analyze function



$$\mathscr{I} = \{(P_2^1, P_1^1), (P_1^1, P_2^1), (P_2^2, P_1^2), (P_1^2, P_2^2)\}$$
$$\mathscr{D} = (\mathscr{T} \times \mathscr{T}) \backslash \mathscr{I}$$

Fig. 5.4 A partial execution-tree showing only the first δ-cycle

To review the EMC-SC approach, consider the example from Fig. 5.1 with all 3 processes and MAX $= 1$ and $x = 2$ (line 20). A partial execution-tree for this example consisting of only the first δ-cycle is shown in Fig. 5.4. The jth transition of the process *Proc* is given by $Proc^j$. In particular, Fig. 5.4 shows the following *wait* to *wait* atomic transitions P_1^1 (lines 6–9), P_1^2 (lines 6, 10), P_2^1 (lines 12–18), P_2^2 (lines 12, 19), C_1^1 (21, 22, 26–30) and C_1^2 (21–25). Using static analysis (as explained in Sect. 5.8.1), the following independence relation is obtained.

$$\mathscr{I} = \{(P_2^1, P_1^1), (P_1^1, P_2^1), (P_2^2, P_1^2), (P_1^2, P_2^2)\}.$$

If two transitions t_1 and t_2 are such that $(t_1, t_2) \notin \mathscr{I}$, then t_1 and t_2 are dependent. Note that slicing of *buf* and symmetric writes on *idx* is used to determine the independency relation.

For a given data-input, let ϕ be a *trace* $(P_1^1, C_1^1, P_2^1, \ldots)$ and its corresponding state sequence be $(\sigma_0, \sigma_1, \sigma_2, \sigma_3, \ldots)$. Using the trace ϕ, we present an overview of the algorithm to explore all possible behaviors of a design for a given data-input. The state σ_0 is the initial state with three runnable processes, i.e., P_1, P_2, C_1.

The Explore algorithm examines the current trace bottom up and restrict its analysis for adding backtracking points within a δ-cycle. Intuitively, for every state σ_i, it checks if the transition ϕ_i, which is executed from state σ_i is dependent on any other transition ϕ_j for $j < i$, i.e., in its prefix trace, that belongs to the same δ-cycle (see Analyze function (lines 20–32)). If true, then it finds the runnable transition t_k in the pre-state σ_j of ϕ_j (see Definition 4), which has a causal order with ϕ_i and

adds t_k to the backtracking set of σ_j. For example, when the algorithm examines the state σ_2, it adds P_2^1 to the backtracking set of σ_1 (since, P_2^1 and C_1^1 are dependent). Next, when it analyzes the state σ_1 the algorithm adds C_1^2 to the backtracking set of σ_0 and then explores the trace $\psi = (P_1^1, P_2^1, C_1^1, \ldots)$ (as P_2^1 was in σ_1's backtracking set). Next, it analyzes the new trace ψ in a similar fashion. The algorithm continues in this way, till it reach state σ_7, at this point P_2^1 is added to the backtracking set of σ_0. The transition and state shown in dashed line is not explored. The state σ_8 is a deadlock state. Note that the transition P_1^1 is not explored in the state σ_{10} because it is in the *Sleep* set of σ_{10} (as P_1^1 and P_2^1 are independent). The EMC-SC algorithm explores only 4 different traces out of the 8 possible traces for this example.

5.9 The Satya Tool

In this section, we discuss the implementation of the EMC-SC algorithm in a prototype tool called Satya. The implementation of Satya consists of 2 main modules – a static analyzer and a verification module. The Satya software tool is over 18,000 lines of C++ code and uses the EDG C++ front-end parser [50] and the OSCI SystemC simulator [95]. Of those, about 17,500 lines are the intermediate representation (IR) and utility functions needed by the static analyzer and the verification module (explore and query engine) is only about 800 lines.

Figure 5.5 presents an overview of the Satya tool. It takes a SystemC design as input – currently with the restriction of no dynamic casting and no dynamic process creation. After parsing the design description, it captures the EDG intermediate language into its own IR that consists of basic blocks encapsulated in Hierarchical Task Graphs (HTGs) [69]. The choice of HTG over other IR's like CDFG have certain benefits as HTG maintains the hierarchical structuring of the design such as if-then-else blocks and for- and while-loops which are used during static analysis. The static

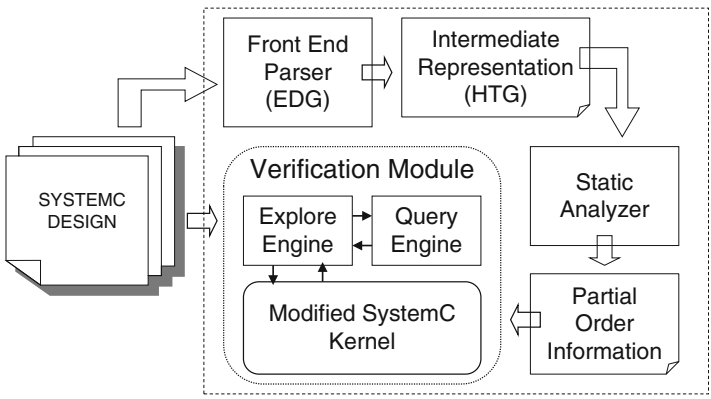


Fig. 5.5 The Satya prototype tool

analyzer work on the HTGs in a modular way to generate the dependency relation, which is then used by the query engine. The implementation of the explore engine follows closely the algorithm described in Sect. 5.8.2.

The SystemC design is compiled with the verification module which contains a modified OSCI's SystemC kernel. The modified kernel implements the Simulate function of the Explore algorithm (Fig. 5.2). It takes as input a prefix schedule and executes it till completion such that the prefix of the executed trace is same as the trace corresponding to the input prefix schedule. The modifications are still in compliance with the SystemC specification [95]. In particular, the modifications are to replace the election algorithm of the scheduler by one, which takes as input a prefix trace that acts as a golden reference for that run and execute it till completion.

5.10 Experiments and Results

In this section, we report the results of using Satya on two benchmark designs. To compare the performance improvement due to POR, the experiments have two settings – one which uses the EMC-SC algorithm (POR) and the other which explores all possible execution traces (no-POR). The following metric for performance improvement (PI) due to POR is computed for the benchmarks. Let for a given set of traces, τ_i denote the time for executing the ith trace. Also, let for a given design, n be the total number of possible traces, g be the total number of reduced traces explored by Satya and ε be the overhead due to the query. Then,

$$PI = \left(1 - \frac{\sum_{i=1}^{g} \tau_i + \varepsilon}{\sum_{i=1}^{n} \tau_i}\right) \times 100\,\%$$

5.10.1 FIFO Benchmark

The first benchmark is a FIFO channel example obtained from the OSCI's example repository [95]. The example has an hierarchical FIFO channel. To use the FIFO channel it uses a producer–consumer scenario. The example works fine when executed in one producer and one consumer scenario. However, if we use two producers writing to the channel and one consumer reading from that channel then we have an assertion violation. Moreover, since this bug is not visible in every trace of the example, simulation may not find it. Satya was able to find the bug and consequently we changed the code to correct it. The following results are measured on the corrected example. The example has 3 processes executing concurrently. The total number of possible traces is directly proportional to the number of elements produced by the producers. To quantify the scalability of the tool, we report in Table 5.1 for different number of elements produced by the two producers, the time required using POR and the number of reduced traces explored by Satya, along with the total number of possible traces and the time required without POR.

Table 5.1 Results for the FIFO benchmark

Elements produced	Reduced #traces g	Time (POR) $\sum_{i=1}^{g} \tau_i + \varepsilon$ secs	Total #traces n	Time (no-POR) $\sum_{i=1}^{n} \tau_i$ secs	PI %
14	6	00.0	8	00.0	30.4
28	42	00.3	80	00.5	43.5
44	318	02.3	992	06.3	63.5
62	2514	19.0	13376	93.6	79.7

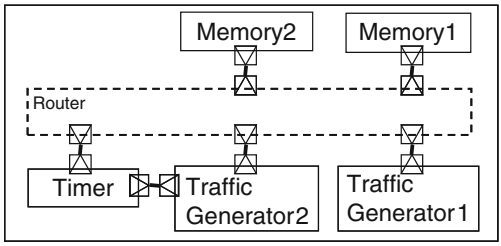


Fig. 5.6 SystemC TAC benchmark

5.10.2 TAC Benchmark

The second benchmark is the industrial Transaction Accurate Communication (TAC) example [149] developed by ST Microelectronics, which includes a platform composed of the following 6 modules: two traffic generators, two memories, a timer and a router to connect them as shown in Fig. 5.6. These modules are based on the TAC protocol built on top of OSCI's TLM standard. This benchmark is over 12,000 lines of SystemC code and consists of 349 functions. The example can be executed for certain number of transactions. A transaction is a read or write by the masters, namely the two traffic generators. When executed for 80,000 transactions, there are 12,032 total possible traces. It took 89.47 min to explore all these traces. In contrast, while checking for deadlocks in the program, we did a static slicing of the router and then Satya found that all these traces are equivalent to one trace, which is executed in 1.3 s. It is interesting that all the different traces of the program is equivalent to one trace for this benchmark. This is because the way this benchmark is written the traffic generator1 writes only to memory1 and traffic generator2 writes only to memory2, and as such there are no conflict between them. Furthermore, note that for this example simulation has same coverage as Satya, however simulation cannot provide the correctness guarantee that is provided by Satya.

5.11 Further Reading

In this section we mention some of the recent works that are related to this chapter. These are also very interesting further readings.

Partial-Order Reduction: POR techniques are extensively used by *software model checkers* for reducing the size of the state space of concurrent system at the implementation level [71, 94]. Other state space reduction techniques, such as slicing [88, 184] and abstraction [9], are orthogonal and can be used in conjunction with POR. The POR techniques can be divided in two broad categories, namely *static* [70] and *dynamic* [57]. The main static POR techniques are *persistent/stubborn* sets and *sleep* sets [70]. In contrast, the dynamic POR technique evaluates the dependency relation dynamically between the enabled and executed transitions for a given execution.

SystemC Verification: Recent work on SystemC focuses mainly on improving simulation performance [145] and generating representative inputs for the design [77] and formalizing the semantics of SystemC [86, 114, 153, 186]. Another approach that uses dynamic POR techniques for automatically generating all valid scheduling of the design [89] is also quite an interesting read.

5.12 Summary

In this chapter, we have discussed EMC-SC, a scalable approach for testing SystemC designs. This approach combines static and dynamic POR techniques to reduce the number of interleavings required to expose all behaviors of a SystemC design. Furthermore, the algorithm presented here exploits SystemC specific semantics to reduce the number of backtracking points, and thereby improving the efficiency of the approach. We also discussed the implementation of the EMC-SC algorithm in a query-based tool called Satya. The experiments performed using Satya shows the efficacy of this approach and also found bugs that may not have been found using a simulator.

Acknowledgments This chapter in part, has been published as:

"Partial Order Reduction for Scalable Testing of SystemC TLM Designs", by Sudipta Kundu, Malay Ganai and Rajesh Gupta in *DAC 08: Proceedings of the 45th annual conference on Design Automation* [115].

Chapter 6
Bounded Model Checking for Concurrent Systems: Synchronous Vs. Asynchronous

Malay K. Ganai[1]

Concurrent systems are hard to verify due to complex and unintended asynchronous interactions. Exploring the state space of such a system is a daunting task. Model checking techniques that use symbolic search and partial-order reduction are gaining popularity. In this chapter, we focus primarily on bounded model checking (BMC) approaches that use decision procedures to search for bounded length counter-examples to safety properties such as data races and assertion violations in multi-threaded concurrent systems. In particular, we contrast several state-of-the-art approaches based on the synchronous and asynchronous modeling styles used in formulating the decision problems, and the sizes of the corresponding formulas.

6.1 Introduction

The growth of cheap and ubiquitous multi-processor systems and concurrent library support are making concurrent programming very attractive. However, verification of multi-threaded concurrent systems remains a daunting task especially due to complex and unexpected interactions between asynchronous threads. Exposing concurrency related bugs – such as atomicity violations and data races – require not only bug-triggering inputs but also bug-triggering thread interleavings. Exploring the global state space for every interleaving on every data input is often practically impossible. In general, the problem of verifying two-threaded programs (with unbounded stacks) is undecidable [177].

In practice, verification efforts often use *incomplete* methods, or *imprecise* models, or sometimes both, to address the scalability problem. The verification model is typically obtained by composing individual thread models using interleaving semantics, and model checkers are applied to systematically explore the global state space. Such exhaustive exploration often encounters the unavoidable state explosion, which is mainly due to: (a) non-deterministic choice of thread interleaving, (b) non-deterministic choice of data values as provided explicitly in the program.

[1] NEC Labs America, 4 Independence Way, Princeton, NJ 08540 USA. (malay@nec-labs.com)

S. Kundu et al., *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, DOI 10.1007/978-1-4419-9359-5_6,

To combat the state explosion problem due to thread interleaving, most model checking methods employ partial-order reduction (POR) techniques to restrict the traversal to only a representative subset of all interleavings, thereby, avoiding the redundant interleavings among independent transitions [70, 166, 200]. Verification of concurrent software predominately employs explicit model checkers such as [4, 57, 71, 80, 94] that explore the states and transitions of a concurrent system by explicit enumeration. These methods were successfully applied to verify programs with multiple threads by exploiting POR efficiently, but were severely limited when applied to systems with a significant amount of data. To overcome the state explosion due to a large set of data values, employing symbolic data structures and methods [16, 143] can lead to more efficient reasoning on variables with large domains than performing explicit enumeration. In hardware domain, such methods have been successfully applied to verify systems that are clearly out-of-reach of any explicit-state model checkers with state space that goes beyond 10^{20} states. Influenced by such successes, various symbolic methods [3,37,64,67,79,103,104,129,174,205] were proposed for verifying concurrent software. However, combining classical POR methods with symbolic algorithms turns out to be non-trivial, especially due to the fact that symbolic methods implicitly manipulate large set of states, wherein capturing and identifying independent transitions for a set of states is much harder than for each individual state.

In this chapter, we focus primarily on symbolic techniques that combine POR with symbolic manipulation of data for efficient state space search. Specifically, we discuss bounded model checking (BMC) approaches that use decision procedures to search for bounded length counter-examples to safety properties such data races and assertions. BMC [16] has been successfully applied to verify real-world designs. Strengths of BMC are manifold: First, expensive existential quantification used in BDD-based symbolic model checking [143] is avoided. Second, reachable states are not stored (symbolically or explicitly), avoiding the blow-up of intermediate state representation. Third, modern SAT solvers are able to search through the relevant paths of the program even though the paths get longer with the each BMC unrolling [63].

BMC is a model checking technique where the falsification of a given LTL property ϕ is checked at a given sequential depth. Typically, it consists of the following steps: unrolling of the design for k time frames, translating the BMC instance into a decision problem ψ such that ψ is satisfiable *iff* the property ϕ has a counter-example of depth (less than or) equal to k, and using a decision procedure to check if ψ is satisfiable. In Satisfiability Modulo Theory (SMT)-based BMC [5], ψ is a quantifier-free formula in a decidable subset of first order logic, and it is checked by an SMT solver. With the growing use of high-level design abstraction to capture today's complex design features, the focus of verification techniques has been shifting towards using SMT solvers [21, 48, 162]. SMT-based BMC [5, 66] is a potentially more scalable alternative method to SAT-based [63] or BDD-based methods [143].

We focus on using SMT-based symbolic approaches to generate efficient formulas to check for bounded length witness traces. Based on how verification models are built, symbolic approaches can be broadly classified into two categories: *synchronous* (i.e., with scheduler) and *asynchronous* (i.e., without scheduler).

6.1.1 Synchronous Models

In this category of symbolic approaches [3,37,79,103,104,129,205], a synchronous model of a concurrent program is constructed with a scheduler as shown in Fig. 6.1. Such a model is constructed based on interleaving (operational) semantics, where at most one thread transition is scheduled to execute at a time. The scheduler is then constrained – by guard strengthening – to explore only a subset of interleavings. Verification using BMC comprises unrolling such a model for a certain depth, and generating SAT/SMT formula with the property constraints.

To guarantee correctness (i.e., cover all necessary interleavings), the scheduler must allow context-switch between accesses that are *conflicting*, i.e., accesses whose relative execution order can produce different global system states. One determines conservatively which pair-wise locations require context switches, using persistent [70]/ample [101] set computations. One can further use lock-set and/or lock-acquisition history analysis [59, 103, 135, 191], and conditional dependency [72, 205] to reduce the set of interleavings need to be explored (i.e., remove redundant interleavings).

Even with the above-mentioned state reduction methods, the scalability problem remains. To overcome that, some researchers have employed sound abstraction [4] with bounded number of context switches [172] (i.e., under-approximation), while

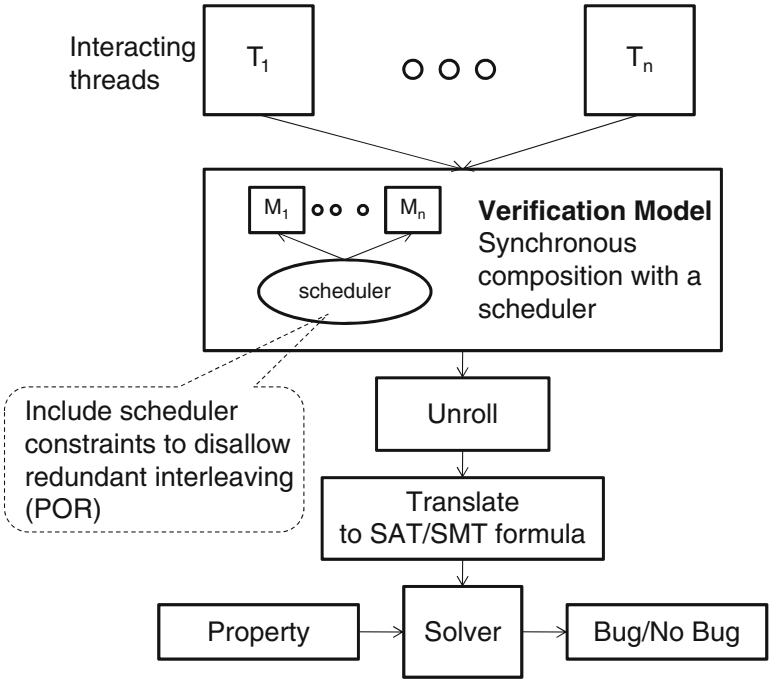


Fig. 6.1 Synchronous modeling and model checking concurrent system

others have used finite-state model abstractions [37], combined with proof-guided method to discover the context switches [79].

In another approach [205], an optimal reduction in interleaved state space is achieved for two threaded system, which was extended for a multi-threaded system in [104]. Note, these approaches achieve state space reduction at the expense of increased BMC formula size.

6.1.2 Asynchronous Models

In the synchronous modeling-based state-reduction approaches, the focus has been more on the reduction of state space, and not so much on the reduction of model size. The overhead of adding static constraints to the formula seems to abate the potential-benefit of less state-space search. Many of the constraints are actually never used, resulting in wasted efforts.

There is a paradigm shift in model checking approaches [64, 67, 174, 205] where the focus is now on generating efficient verification conditions without constructing a synchronous models, and that can be solved easily by the decision procedures. The concurrency semantics used in these modeling are based on sequential consistency [124]. In this semantics, the observer has a view of only the local history of the individual threads where the operations respect the program order. Further, all the memory operations exhibit a common *total order* that respect the *program order* and has the *read value property*, i.e., the read of a variable returns the last write on the same variable in that total order. In the presence of synchronization primitives such as locks/unlocks, the concurrency semantics also respects the mutual exclusion of operations that are guarded by matching locks. Sequential consistency is the most commonly used concurrency semantics for software development due to ease of programming, especially to obtain correctly synchronized threads.

Asynchronous modeling paradigm has advantages over synchronous modeling, and have been shown to suit better for SAT/SMT encoding. To that effect, the symbolic approaches such as CSSA-based (Concurrent Static Single Assignment) [174, 204] and token-based [67] generate verification conditions directly without constructing a synchronous model of concurrent programs, i.e., without using a scheduler as shown in Fig. 6.2. The concurrency constraints that maintain sequentially consistency are included in the verification conditions for a bounded depth analysis.

Specifically, in the CSSA-based approach [174, 204], read-value constraints are added between each read and write accesses (on a shared variable), combined with happens-before constraints ordering other writes (on the same variable) relative to the pair. Context-bounding [172] are also added to reduce the interleavings to be explored in the verification conditions.

In the token-based approach [67], a single-token system of decoupled threads is constructed first, and then token-passing and memory consistency constraints are added between each pair of accesses that are shared in the multi-threaded

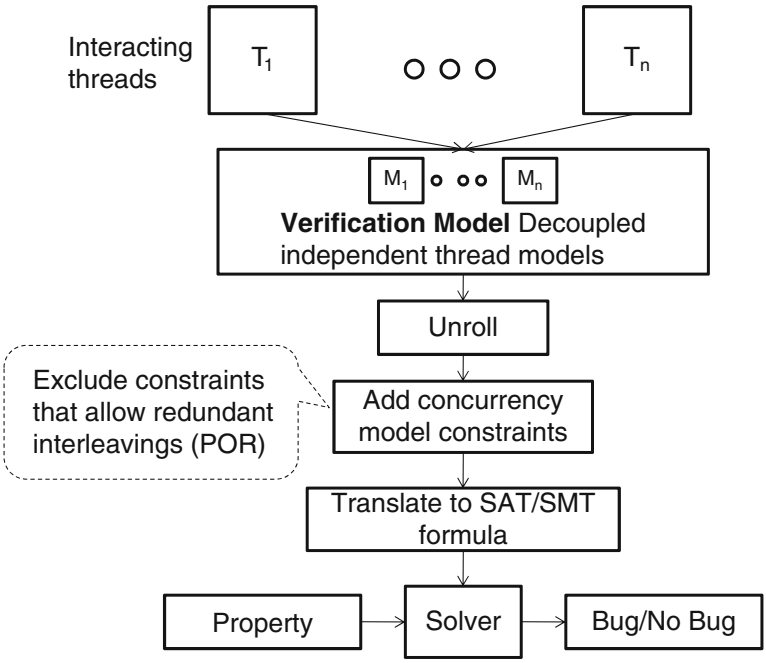


Fig. 6.2 Asynchronous modeling and model checking concurrent system

system. The constraints ensures a total order in the token passing events so that the synchronization of the *localized* (shared) variables takes place at each such event. Such a token-based system guarantees completeness, i.e., only allows traces that are sequentially consistent [125], and adequacy i.e., captures all the interleavings present in the original multi-threaded system. For effective realization, the constraints are added lazily and incrementally at each BMC unrolling depth, and thereby, reduced verification conditions are generated with a guarantee of completeness and adequacy. For further reduction of the size of the verification conditions, the approach uses lockset analysis to reduce the pair-wise constraints between the accesses that are provably unreachable (such as by static analysis).

In [64], a state-reduction based on partial-order technique has been exploited in the token-based modeling approach [67] to exclude the concurrency constraints that allow redundant interleavings, and thereby, reduce the search space and the size of the formula.

6.1.3 Outline

In Sect. 6.2, we provide the operational and non-operational concurrency semantics, and modeling based on such semantics. This is followed by an overview of BMC in

Sect. 6.3 and concurrent modeling in Sect. 6.4. In the following sections, we contrast three representative approaches based on aforementioned concurrency semantics. Specifically, in Sects. 6.5, 6.6, we discuss modeling, BMC encoding, and the formula size in synchronous modeling-based approach that uses optimal POR [205]. In Sect. 6.7, we discuss the essence of asynchronous modeling. In Sect. 6.8, we discuss a CSSA-based (asynchronous) BMC encoding, and formula size in asynchronous modeling approaches [174, 205]. In Sect. 6.9, we discuss token-based modeling [67] approach for BMC, combined with partial order reduction [64]. We contrast these approaches in Sect. 6.10. We provide pointers for further reading in Sect. 6.11, followed by our summary discussion in Sect. 6.12.

6.2 Concurrent System

A multi-threaded concurrent program P comprises a set of threads and a set of shared variables, some of which, such as locks, are used for synchronization. Let M_i $(1 \leq i \leq n)$ denote a thread model represented by a control and data flow graph of the sequential program it executes. Let V_i be a set of local variables in M_i and $\mathscr{V}$ be a set of (global) shared variables. Let C_i be a set of control states in M_i. Let $\mathscr{S}$ be the set of global states of the system, where a state $s \in \mathscr{S}$ is a valuation of all local and global variables of the system. Let VL_i denote a set of tuple values for local data state variables in V_i, and VG denote a set of tuple values for shared variables in $\mathscr{V}$. A global state s of P is a tuple $\langle s_1, \ldots, s_n, v \rangle \in \mathscr{S} = (C_1 \times VL_1) \cdots \times (C_n \times VL_n) \times VG$ where $s_i \in C_i \times VL_i$ is a local state, and $v \in VG$ denotes the values of the shared variables. Note, s_i denotes the local state tuple $\langle c_i, x_i \rangle$ where $c_i \in C_i$ represents the local control state, and $x_i \in VL_i$ represents the local data state.

6.2.1 *Interleaving (Operational) Semantics*

A thread transition t is a 4-tuple $\langle c, g, u, c' \rangle$ that corresponds to a thread M_i, where $c, c' \in C_i$ represent the control states of M_i, g is an enabling condition (or *guard*) defined on $V_i \cup \mathscr{V}$, and u is a set of update assignments of the form $v := exp$ where variable v and variables in expression exp belong to the set $V_i \cup \mathscr{V}$. We use operator $next(v)$ to denote the next state update of variable v.

Let pc_i denote a thread program counter of thread M_i. For a given transition $t = \langle c, g, u, c' \rangle$, and a state $s \in \mathscr{S}$, if g evaluates to true in s, and $pc_i = c$, we say that t is *enabled* in s. Let $enabled(s)$ denote the set of all enabled transitions in s. We assume each thread model is deterministic, i.e., at most one local transition of a thread can be enabled.

The interleaving semantics of concurrent system is a model in which precisely one local transition is scheduled to execute from a state.[2] Formally, a global transition system for P is an interleaved composition of the individual thread models, where a global transition consists of firing of a local transition $t \in enabled(s)$ from state s to reach a next state s', denoted as $s \xrightarrow{t} s'$.

A *schedule* of the concurrent program P is an interleaving sequence of thread transitions $\rho = t_1 \cdots t_k$. An event e occurs when a unique transition t is fired, which we refer as the *generator* for that event, and denote it as $t = gen(P,e)$. A *run* (or concrete execution trace) $\sigma = e_1 \cdots e_k$ of a concurrent program P is an ordered sequence of events, where each event e_i corresponds to firing of a unique transition $t_i = gen(P,e_i)$.[3] We illustrate the differences between schedules and runs in Sect. 6.4.

Let $begin(t)$ and $end(t)$ denote the beginning and the ending control states of $t = \langle c,g,u,c' \rangle$, respectively. Let $tid(t)$ denote the corresponding thread of the transition t. We assume each transition t is atomic, i.e., uninterruptible, and has at most one shared memory access. Let T_i denote the set of all transitions of M_i, and $\mathscr{T} = \bigcup_i T_i$ be the set of all transitions.

A *transaction* is an uninterrupted sequence of transitions of a particular thread. For a transaction $tr = t_1 \cdots t_m$, we use $|tr|$ to denote its length, and $tr[i]$ to denote the ith transition where $i \in \{1,\ldots,|tr|\}$. We define $begin(tr)$ and $end(tr)$ as $begin(tr[1])$ and $end(tr[|tr|])$, respectively. In the sequel, we use the notion of *transaction* to denote an uninterrupted sequence of transitions of a thread as *observed* in a system execution.

We say a transaction (of a thread) is *atomic* w.r.t. a schedule, if the corresponding sequence of transitions are executed uninterrupted, i.e., without an interleaving of another thread in-between. For a given set of schedules, if a transaction is atomic w.r.t. all the schedules in the set, we refer to it as an *independent transaction* w.r.t. the set.[4]

Given a run σ for a program P we say e *happens-before* e', denoted as $e \prec_\sigma e'$ if $i < j$, where $\sigma[i] = e$ and $\sigma[j] = e'$, with $\sigma[i]$ denoting the ith access event in σ. Let $t = gen(P,e)$ and $t' = gen(P,e')$. We say $t \prec_\sigma t'$ iff $e \prec_\sigma e'$. For some σ, if $e \prec_\sigma e'$ and $tid(t) = tid(t')$, we say $e \prec_{po} e'$ and $t \prec_{po} t'$, i.e., the events and the transitions are in thread program order.

---

[2] Step semantics [102], in contrast to interleaving semantics, allows more than one local transitions of different thread models to be scheduled as long as no reachable state is lost. It is an optimization step to reduce the set of interleavings to explore.

[3] Firing of generator transitions may correspond to instances of the same thread transition when it is fired in a thread loop. Note that during BMC unrolling, each thread transition in a loop is instantiated uniquely. In the sequel, we use a thread transition to refer to such an instance.

[4] We compare the notion of atomicity used here, vis-a-vis previous works [56, 182, 206]. Here the atomicity of transactions corresponds to the observation of the system, which may not correspond to the user intended atomicity of the transactions. Previous work assume that the atomic transactions are system specification that should always be enforced, whereas here atomic (or rather independent) transactions is inferred from the given system under test, and are used to reduce the search space of symbolic analysis.

A *data race* corresponds to a global state where two different threads can access the same shared variable simultaneously, and at least one of them is a write.

6.2.2 Axiomatic (Non-Operational) Semantics

Concurrency semantics for shared memory multi-thread concurrent program can also be expressed in an axiomatic style [209] based on the execution effect of such a system on the underlying shared memory. The most commonly used semantics in software development is sequential consistency.

A schedule is *sequentially consistent* [125] iff it satisfies the following rules:

Program Order Rule	Transitions of the same thread are in a thread program order.
Read Value Rule	Each shared variable read access gets the last data written at the same memory address location.
Total Order Rule	Transitions corresponding to shared variable accesses across all threads are in a total order.
Mutual Exclusion Rule	Synchronization semantics is maintained. For example, the same locks are not acquired in a row without a corresponding release in between.

Intuitively, such a semantics allows analyzing a concurrent system without a scheduler. Many approaches have used this basic idea to overcome state explosion problem.

6.2.3 Partial Order

Definition 7 (Dependency Relation (D)). Given a set $\mathscr{T}$ of transitions, we say a pair of transitions $(t,t') \in \mathscr{T} \times \mathscr{T}$ is dependent, i.e. $(t,t') \in D$ iff one of the following conditions hold: (a) $t \prec_{po} t'$, (b) (t,t') is conflicting, i.e., accesses are on the same global variable, may be co-enabled (i.e., simultaneously enabled), and at least one of them is a write access.

If $(t,t') \notin D$, we say the pair is *independent*. The dependency relation in general, is hard to obtain; however, one can obtain such relation conservatively using static analysis [70], which may result in a larger dependency set than actual.

One can also define independency relation equivalently in an operational semantics. The transitions t,t' are said to be independent for all state $s \in \mathscr{S}$ iff $t,t' \in enabled(s)$, and there is a unique state $s' \in \mathscr{S}$ such that $s \xrightarrow{t \cdot t'} s'$ and $s \xrightarrow{t' \cdot t} s'$. In other words, independent transitions can neither disable or enable each other, and enabled independent transitions commute. To obtain a more precise independency relation, one can define such a relation with respect to a particular state $s \in \mathscr{S}$

(as opposed to all states) i.e., $\langle t,t',s\rangle$ [72, 107]. Such a relation is referred as *conditionally* independent. An independent relation can also be defined [205] over a set of states as represented by a predicate G over local and global variables, i.e., $\langle t,t',G\rangle$. In other words, t,t' are independent for states in which the predicate G holds. Such a relation is referred as *guarded independent* relation.

Definition 8 (Equivalency Relation ($\simeq$)). We say two schedules $\rho_1 = t_1 \cdots t_i \cdot t_{i+1} \cdots t_n$ and $\rho_2 = t_1 \cdots t_{i+1} \cdot t_i \cdots t_n$ are equivalent (Mazurkiewicz's trace theory [139]), denoted as $\rho_1 \simeq \rho_2$, if $(t_i,t_{i+1}) \notin D$.

An equivalent class of schedules can be obtained by iteratively swapping the consecutive independent transitions in a given schedule. Final values of both local and shared variables remains unchanged when two equivalent schedules are executed. Note, if $(t,t') \in D$, all equivalent schedules agree on either $t \prec t'$ or $t' \prec t$, but not both.

Definition 9 (Partial Order). A *partial order* is a relation $R \subseteq \mathscr{T} \times \mathscr{T}$ on a set of transition $\mathscr{T}$, that is reflexive, antisymmetric, and transitive.

A partial order is also a *total order* if, for all $t,t' \in \mathscr{T}$, either $(t,t') \in R$, or $(t',t) \in R$. *Partial order*-based reduction (POR) methods [70] avoid exploring all possible interleavings of shared accesses by exploiting the commutativity of the independent transitions. Thus, instead of exploring all interleavings that realize these partial orders it is adequate to explore just the representative interleaving of each equivalence class.

6.3 Bounded Model Checking

Let s^i denote a state and $T(s^i,s^{i+1})$ denote the state transition relation of a concurrent system. A path is a finite sequence $\pi^{0,k} = (s^0,\ldots,s^k)$ satisfying the following predicate:

$$T^{0,k} \stackrel{def}{=} \bigwedge_{0\le i<k} T(s^i,s^{i+1}) \tag{6.1}$$

where $T^{0,0} \stackrel{def}{=} true$. A path has length k if it makes k transitions. In bounded model checking, whether an LTL property ϕ can be falsified in k execution steps from some initial state ψ is formulated as a satisfiability problem [16]:

$$BMC^k(\psi,\phi) \stackrel{def}{=} \psi(s^0) \wedge T^{0,k} \wedge \neg\phi(s^k) \tag{6.2}$$

where $\phi(s^k)$ means that ϕ holds in state s^k, and $\psi(s^0)$ means ψ holds in state s^0. Given a predetermined bound d, BMC iteratively checks the satisfiability of BMC^k for $0 \le k \le d$ using a SAT/SMT solver.

6.4 Concurrent System: Model

We contrast synchronous and asynchronous modeling of a concurrent system informally, where we motivate our readers with an example. We use that example to guide the rest of our discussion.

Consider a system P comprising interacting threads M_a and M_b with local variables a_i and b_i, respectively, and shared variables X, Y, Z, L. This is shown in Fig. 6.3a where threads are synchronized with *Lock/Unlock*. Thread M_b is created and destroyed using fork-join primitives. The concurrent control flow graph (CCFG) of the program P is shown in Fig. 6.3b. A thread transition $(2a, true, a_1 = Z, 3a)$ (also represented as $2a \overset{a_1=Z}{\rightarrow} 3a$) can be viewed as a generator of access event $R(Z)_a$ corresponding to the read access of the shared variable Z. Each node in CCFG denotes a thread control state (and the corresponding thread location), and each edge represents one of the following: thread transition, a context switch, a fork, and a join. To not clutter up the figure, we do not show edges that correspond to possible context switches (30 in total).

Figure 6.4 is the lattice representing the complete interleaving space of the program. Each node in the lattice denotes a global control state, shown as a pair of

a

Thread M_a

0. $X = Y = Z = [0..2]$;
F. **fork**(M_b);
1a. $Lock(L)$
2a. $a_1 = Z$;
3a. $X = a_1 + 1$;
4a. $Unlock(L)$;
5a. $Y = a_1$;
Ja. **join**(M_b);
7. **assert**($Y \leq 5$);

Thread M_b

1b. $b_1 = Y$;
2b. $Lock(L)$
3b. $b_2 = X$;
4b. $Z = b_1 + 1$
5b. $Unlock(L)$
6b. $Y = b_1 + b_2$
Jb. /* join M_a */

b

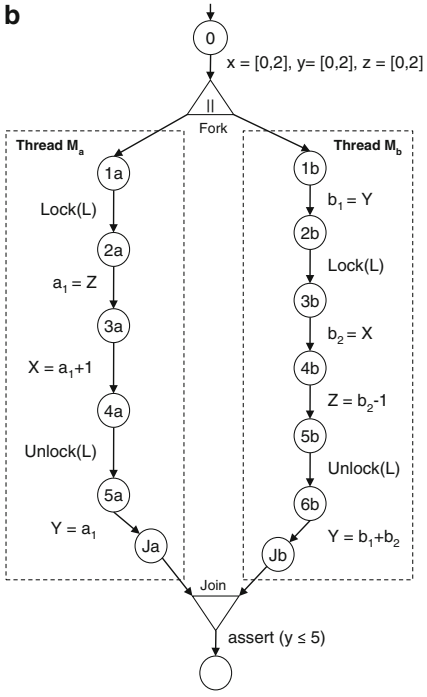


Fig. 6.3 (**a**) Concurrent system P with threads M_a, M_b and local variables a_i, b_i respectively, communicating with shared variable X, Y, Z, L, (**b**) CCFG of the program P

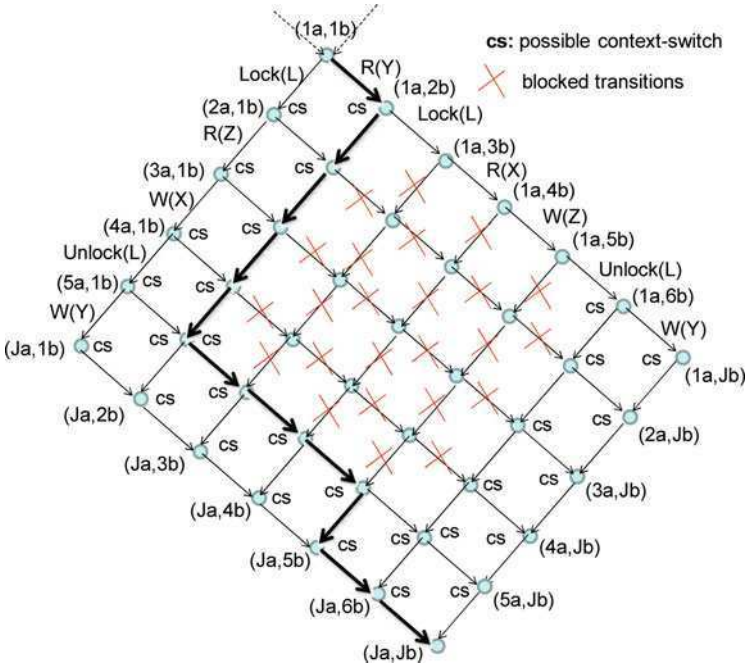


Fig. 6.4 Lattice and a run σ highlighted in bold

the thread local control states. An edge denotes a shared write/read access of global variable, labeled with $W(.)/R(.)$ or *Lock(.)/Unlock(.)*. Note, some interleavings are not feasible due to Lock/Unlock, which we crossed out $(\times)$ in the figure. We also labeled all possible context switches with `cs`. The highlighted interleaving corresponds to a concrete execution (run) of program P

$$\sigma = R(Y)_b \cdot Lock(L)_a \cdots Unlock(L)_a \cdot Lock(L)_b \cdots W(Z)_b \cdot W(Y)_a \cdot Unlock(L)_b \cdot W(Y)_b$$

where the suffices a, b denote the corresponding thread accesses.

The corresponding schedule ρ of the run σ is

$$\rho = \left(1b \overset{b_1=Y}{\rightarrow} 2b\right)\left(1a \overset{Lock(L)}{\rightarrow} 2a\right)\cdots\left(4a \overset{Unlock(L)}{\rightarrow} 5a\right)\left(2b \overset{Lock(L)}{\rightarrow} 3b\right)\cdots\left(6b \overset{Y=b_1+b_2}{\rightarrow} Jb\right)$$

In the next two sections, we discuss the synchronous modeling and model checking in detail.

6.5 Synchronous Modeling

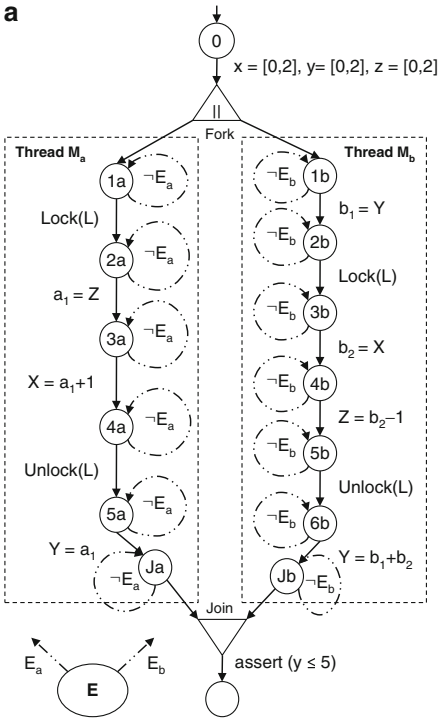
One obtains a synchronous execution model for P by defining a scheduling function $E : \mathscr{M} \times \mathscr{S} \mapsto \{0,1\}$ such that $t_i \in T_i$ is said to be *executed* at global state s, *iff* $t_i \in enabled(s)$ and $E(M_i, s) = 1$. Note that in the interleaving semantics, at most one

enabled transition will be executed at a global state s. In this synchronous execution model, each thread local state s_i (with shared access) has a *wait-cycle*, i.e., a self-loop to allow all possible interleavings. To model such a scheduler, one can add a variable *sel* whose domain is the set of thread indices $\{1,2,\ldots,n\}$. A enabled transition $t \in T_i$ is executed only when $sel = i$. The transition relation of such a synchronous model is computed as follows:

$$T(s,s') = \bigvee_{i=1}^{n} ((sel = i) \wedge T_i(s,s')) \tag{6.3}$$

A synchronous model for the concurrent system with threads models M_a and M_b, is shown in Fig. 6.5a with a scheduler E. It is obtained by inserting a wait-cycle, i.e., a self-loop at each control state of model M_a and associating the edge with a Boolean guard $\neg E_a$ such that $E_a = 1$ *iff* $E(M_a,s) = 1$ (and similarly, for model M_b). To understand the need for such wait-cycles, consider a global state s with thread control states at $1a$ and $1b$, respectively. To explore both the interleaving $Lock(L)_a \cdot R(Y)_b$ and $R(Y)_b \cdot Lock(L)_a$ from s, each thread needs to wait when the other makes the transition.

a



b

Set of reachable control states pairs with independent transitions I
$\langle 1a,1b\rangle, \langle 2a,1b\rangle, \langle 3a,1b\rangle, \langle 4a,1b\rangle$ $\langle 5a,1b\rangle, \langle Ja,1b\rangle, \langle 1a,2b\rangle, \langle 2a,2b\rangle$ $\langle 3a,2b\rangle, \langle 5a,2b\rangle, \langle Ja,2b\rangle, \langle 1a,3b\rangle$ $\langle 5a,3b\rangle, \langle Ja,3b\rangle, \langle 1a,4b\rangle, \langle 5a,4b\rangle$ $\langle Ja,4b\rangle, \langle 5a,5b\rangle, \langle Ja,5b\rangle, \langle 1a,6b\rangle$ $\langle 2a,6b\rangle, \langle 3a,6b\rangle, \langle 4a,6b\rangle, \langle Ja,6b\rangle$ $\langle 1a,Jb\rangle, \langle 2a,Jb\rangle, \langle 3a,Jb\rangle, \langle 4a,Jb\rangle$ $\langle Ja,Jb\rangle$
POR constraints on scheduler E: $\forall_{\langle i,j\rangle \in I} PC_a = i \wedge PC_b = j \rightarrow \neg(E_b \wedge next(E_a))$

Fig. 6.5 (**a**) Synchronous model of P with scheduler E (**b**) POR constraints on the scheduler E [205]

Symbolic model checking techniques such as [63,143] can be directly applied on such synchronous models. However, without any partial order reduction, *sel* takes arbitrary values at every step. This corresponds to exploring all interleavings, leading to state explosion. Partial order reduction can be implemented to alleviate that situation by adding constraints to the scheduler so that it disallows redundant interleavings. As shown in Fig. 6.5b, for a given set I of reachable control state pairs (obtained conservatively), the scheduler is constrained with the following set of constraints:

$$\forall_{\langle i,j\rangle \in I}\ (PC_a = i \wedge PC_b = j) \rightarrow \neg(E_b \wedge next(E_a))$$

where $next(E_a)$ corresponds to next state value of E_a. Such a set of constraints allow the SAT/SMT formula to capture only representative interleavings, as claimed in Theorem 6.1 (provided in the next section).

6.6 BMC on Synchronous Models

We now discuss the essential steps in the partial-order based bounded model checking for two threads, as presented in [205]. We skip the discussion of the general approach [104] for more than two threads.

Let $V = \mathscr{V} \cup \bigcup_i V_i$, where $\mathscr{V}$ denote the global variables, and V_i denote the local variables in M_i. For every local (global) program variable, a state variable is introduced in $V_i(\mathscr{V})$. For each thread model M_i, a program counter variable PC_i is introduced to track the local control state. To model the non-determinism in the scheduler, a *sel* variable is introduced whose domain is the set of thread indices $\{1,\ldots,n\}$. A Boolean predicate $E_j \stackrel{def}{=} (sel = j)$ is defined such that a transition in T_j is executed when E_j is true.

At every time frame, fresh copies of the set of state variables are added. Let $v^i \in V^i$ denote the copy of $v \in V$ at the ith time frame. All interleavings of k-length can be represented as k unrolling of the transition relation (6.3).

$$T^{0,k}(s^0, s^{k+1}) = \bigwedge_i^k POR(V^i) \wedge \left(\bigvee_j^n (E_j^k \wedge T_j(s^k, s^{k+1}) \vee \neg E_j^k \wedge (V_j^{i+1} = V_j^i)) \right) \tag{6.4}$$

When E_j^k is not true, then variables in V_j do not change values. This corresponds to a wait-cycle, as shown in Fig. 6.5. Without any partial order reduction, E_j^k can be $true$ at any step for any thread. Such an unrestricted choice captures all interleavings.

For $t_1, t_2 \in enabled(s^i)$, let $en_{t_1}(V^i)$ and $en_{t_2}(V^i)$ denote the enabling condition for transitions t_1 and t_2 at ith time frame, and $\langle t_1, t_2, G\rangle$ denote the guarded independent relation where G denote the corresponding state predicate. Let R_G denote the set of such tuples of guarded independent pair-wise transitions.

A partial-order reduction can be achieved by adding the following constraints for each $\langle t_1, t_2, G\rangle \in R_G$

$$POR(V^i) \stackrel{def}{=} \bigvee_{\langle t_1,t_2,G\rangle \in R_G} (en_{t_1}(V^i) \wedge en_{t_2}(V^i) \wedge G(V^i)) \rightarrow \neg(E^i_{tid(t_2)} \wedge E^{i+1}_{tid(t_1)}) \quad (6.5)$$

where $tid(t_1) < tid(t_2)$. Such a constraint would forbid any schedule with transition order $t_2 \cdot t_1$. By giving priority to lower thread index, such constraints would allow all and only representative interleavings as stated in the following theorem. Such a scheme was referred as *peep-hole* partial order reduction.

Theorem 6.1 (Wang 08). *For two-threaded system, the peephole partial order reduction removes all and only redundant interleavings.*

For the running example, the total number of allowed context-switching by peep-hole partial reduction is 15. These are shown as `cs` in Fig. 6.6. The highlighted interleaving $Lock(L)_a \cdots Unlock(L)_a \cdot R(Y)_b \cdot W(Y)_a \cdot Lock(L)_b \cdots W(Y)_b$, is the representative interleaving equivalent to the interleaving highlighted in Fig. 6.4. For more than two threads, peephole does not guarantee removal of all redundant interleavings. However, an approach proposed in [104] guarantees the removal of all and only redundant interleavings.

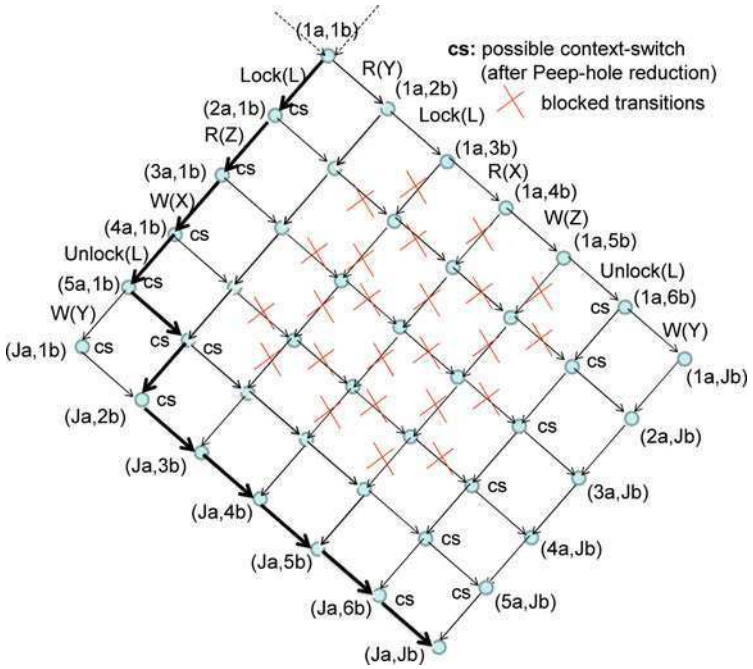


Fig. 6.6 Possible context switches after Peep-hole POR [205]

6.6.1 BMC Formula Sizes

For n threads each with m transitions, the size of the added scheduler constraints is $O(n^2 \cdot m^2)$. For an unrolling of depth d, the size of thread transition constraints is $O(n \cdot m \cdot d)$ and the size of scheduler constraints is $O(n^2 \cdot m^2 \cdot d)$.

A witness trace of length $n \cdot k$ of interleaved transitions, where the number of transitions per thread is k, would require a BMC unrolling of depth $n \cdot k$ in the worst case. Thus, the size of BMC formula would be $O(n^3 \cdot k^3)$.

In the next three sections, we discuss the essence of asynchronous modeling, followed by two state-of-the-art BMC approaches based on such models.

6.7 Asynchronous Modeling

Multi-thread concurrent systems are inherently asynchronous. Instead of using operational semantics, one can use axiomatic (non-operational) semantics to model the system, where only sequential consistent schedules are modeled. The basic modeling steps are two folds:

- The individual thread models are first decoupled, wherein the global shared variables are localized by renaming, and every shared read access is modeled using symbolic input values. Note, the thread models so obtained are *independent* as the transition relation of each thread model depends only local variables and symbolic input values.
- Due to non-deterministic reads, the individual threads model, so obtained, have additional behaviors due to unconstrained symbolic input values. To eliminate that, concurrency constraints are added to capture inter- and intra-thread dependencies. These constraints are added on-the-fly during BMC unrolling. The added concurrency constraints (over the symbolic input values) maintains the sequential consistency semantics, i.e., the *read value* and *total order* properties. As the thread models are decoupled, they can be unrolled at different depths during BMC unrolling (unlike synchronous modeling approaches). The constraints due to the transitions relation of each thread model ensures that memory accesses within the thread follow the program order.

In the next two sections we discuss two such approaches, namely (a) CSSA-based modeling, and (b) token-based modeling. The former approach, as presented in [174, 205], uses concurrent static single assignment (CSSA) form, inspired by [128]. Such a translation is shown in Fig. 6.7. The latter approach, as presented in [67], uses decoupled CFG of the thread models as shown in Fig. 6.8, and combines partial-order reduction technique to obtain reduction in state space as well as formula size. Note that these models do not have a scheduler or self-loop.

Thread M_a

0. $X_{11} = Y_{11} = Z_{11} = [0..2], L_{11} = 0;$
F. `/* fork(M_b); */`
1a. $assume(L_{12} = 0), L_{13} = 1;$ `/* L12 ← ND() */` `/* π(L11,L21,L23,L25) */`
2a. $a_{11} = Z_{12};$ `/* Z12 ← ND() */` `/* π(Z11,Z21,Z22) */`
3a. $X_{12} = a_{11} + 1;$
4a. $assume(L_{14} = 1), L_{15} = 0;$ `/* L14 ← ND() */` `/* π(L13,L21,L23,L25) */`
5a. $Y_{12} = a_{11};$
Ja. `/* join(M_b); */`
7. **assert**$(Y_{13} \leq 5);$ `/* Y13 ← ND() */` `/* π(Y12,Y21,Y23) */`

Thread M_b

0. $X_{21} = Y_{21} = Z_{21} = [0..2], L_{21} = 0;$
1b. $b_{11} = Y_{22};$ `/* Y22 ← ND() */` `/* π(Y11,Y12,Y21) */`
2b. $assume(L_{22} = 0), L_{23} = 1;$ `/* L22 ← ND() */` `/* π(L11,L13,L15,L21) */`
3b. $b_{21} = X_{22};$ `/* X22 ← ND() */` `/* π(X11,X12,X21) */`
4b. $Z_{22} = b_{11} + 1;$
5b. $assume(L_{24} = 1), L_{25} = 0;$ `/* L24 ← ND() */` `/* π(L11,L13,L15,L23) */`
6b. $Y_{23} = b_{11} + b_{21};$
Jb. `/* join(M_a); */`

Fig. 6.7 CSSA form of the concurrent program P

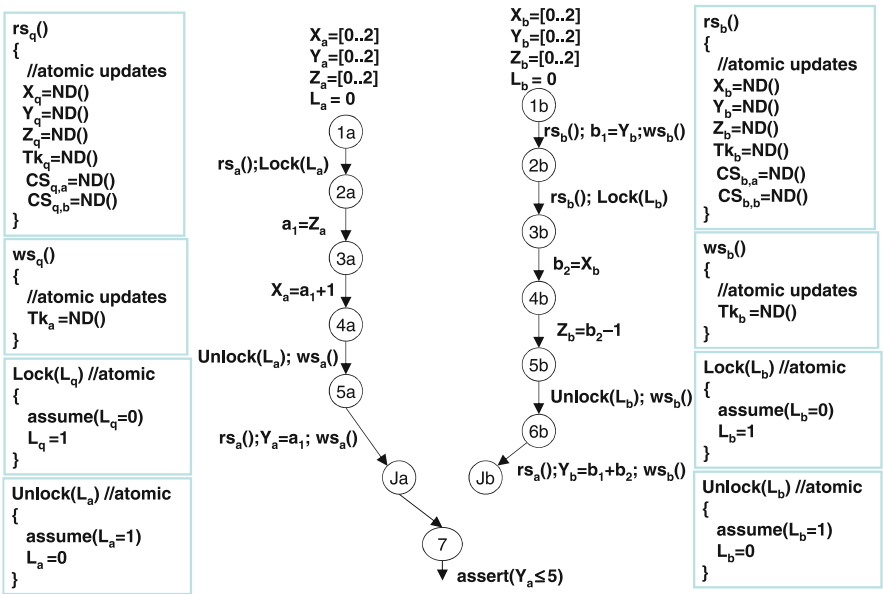


Fig. 6.8 Token-based asynchronous modeling of the concurrent program P

6.8 BMC on Asynchronous Models: CSSA-Based Approach

The approach discussed here is a CSSA-based encoding presented in [174, 204]. Both of these approaches transforms the concurrent program directly into a CSSA form, where the program is unrolled for a bounded depth. The constraints are generated directly from the CSSA, which are then given to a decision procedure to solve. The transition relation constraints comprise two components:

$$T^{0,k} \stackrel{def}{=} \Omega_{TP} \wedge \Omega_{CC} \tag{6.6}$$

where Ω_{TP} encodes the thread programs in CSSA forms, and Ω_{CC} encodes the concurrency constraints.

6.8.1 Thread Program Constraints: Ω_{TP}

We give the basic description of a CSSA form (more details can be found in [128, 204]). A variable v is said to be *defined* when the variable is updated (i.e., appears in left-hand-side of assignment), and is said to be *used* when the variable appears in an update or guarded expression (i.e., appears in a right-hand-side of assignment or in a condition). In CSSA form, each variable is defined exactly once.

A statement in a thread program is executed when all conditionals that lead to it are evaluated true. In CSSA (similar to SSA for sequential program), guard variables are introduced that are associated with a conjunction of all the conditions that lead to the statement.

Each use of shared variable $v \in \mathcal{V}$ is preceded immediately by a non-deterministic function `ND()`. This modeling allows mapping multiple definitions due to thread interleavings.

A CSSA form of the running example is shown in Fig. 6.7. Each local and shared variables have unique names. Each use of local variable is replaced with the most recent definition. Lock/Unlock are modeled using `assume` and define.

The thread program constraints comprise translation of the CSSA form into a quantifier-free first-order logic formula. The constraints also capture the program order of local thread events (i.e., order of use and definitions).

6.8.2 Concurrency Constraints: Ω_{CC}

In the above model, the newly introduced non-deterministic functions (`ND()`) add additional behaviors. To eliminate that a $\pi(v_1, \ldots, v_l)$-function is introduced to constraint the non-deterministic function, where each $v_i, 1 \leq i \leq l$ is the most recent definition of v in some thread.

The semantics of π-function that is that it returns exactly one parameter so as to satisfy *read value* property in a total ordered interleaving. Let t be transition where v is used. Let v' represent the corresponding new variable introduced in CSSA encoding. Then v' is mapped with v_i iff event t_i, which defines v_i, is executed (happens) before event t, and any event t_j that defines $v_j, 1 \leq j \leq l$, $j \neq i$ is executed before the definition t_i or after the use of t. Thus, for each $v' \leftarrow \pi(v_1 \cdots v_l)$ the following constraints are added

$$\Omega_{CC} = \bigwedge_{v' \leftarrow \pi(v_1 \cdots v_l)} \left(\bigvee_{i=1}^{l} (v' = v_i) \wedge path(t_i) \wedge HB(t_i, t) \wedge \bigwedge_{j=1, j \neq i}^{l} (HB(t_j, t_i) \vee HB(t, t_j)) \right) \tag{6.7}$$

where $path(t)$ is the enabling path condition such that t is executed iff $path(t)$ is true, and $HB(t, t')$ is an order constraint to denote that event t is executed before t'.

The constraints corresponding to π-function are added on-the-fly during BMC unrolling. Note that with BMC unrolling, the set of definitions of variable increases, and hence the π-function constraints changes with unrolling depth.

6.8.3 BMC Formula Sizes

For m transitions in the unrolled BMC instances, the CSSA-based encoding produces a formula size of $O(m^3)$ in the worst case when there are $O(m)$ read/write accesses on a single shared variable.

A witness trace of length $n \cdot k$ of interleaved transitions, where the number of transitions per thread is k, would require a BMC unrolling of depth k. Thus, the size of BMC formula would be $O(n^3 \cdot k^3)$, as the number of transitions would be $O(n \cdot k)$.

In the next section, we discuss another asynchronous modeling approach that would generate formula of size quadratic in the number of events. Further, such an approach also benefits from partial-order reduction techniques.

6.9 BMC on Asynchronous Models: Token-Based Approach

The main idea of token-passing model (TPM) is to introduce a single Boolean token *TK* and a clock vector *CV* in a model, and then manipulate the passing of the token between inter-thread control states to capture all necessary interleavings in the given system. The semantics of the token assertion by a thread is that all the last writes to shared memory are visible to the thread. The clock vector records the number of times the token *TK* is passed. Unlike a synchronous model, TPM does not have a scheduler in the model. The total ordering is achieved by updating the clock vector when the token is passed. Following theorem provides the basis for generating verification model based on token-based approach.

Theorem 6.2 (Ganai, 2008 [67]). *The token-based model is both complete, i.e., it allows only sequentially consistent traces, and sound, i.e., captures all necessary interleaving, for a bounded unrolling of threads. Further, the size of pair-wise constraints added grow quadratically (in the worse case) with the unrolling depth.*

The main goal of the token-based approach is to generate verification conditions that capture necessary interleaving for some bounded unrolling of the threads, aimed at detecting reachability properties such as data races and assertion violations. These verification conditions together with the property constraints are encoded and solved by an SMT solver. A satisfiable result is typically accompanied by a trace – comprising data input valuations, and a total-ordered thread interleaving – that is witness to the reachability property. On the other hand, an unsatisfiable result is followed by these steps (a)–(c): (a) increase unroll depths of the threads, (b) generate verification conditions for increased depths, and (c) invoke SMT solver on these conditions. Typically, the search process (i.e., to find witnesses) is terminated when a resource – such as time, memory or bound depth – reaches its limit. For effective implementation, these verifications constraints are added on-the-fly, lazily and incrementally at each unrolled depth. The transition relation constraints comprise two components:

$$T^{0,k} \stackrel{def}{=} \Omega_{TP} \wedge \Omega_{CC} \tag{6.8}$$

where Ω_{TP} corresponds to transition relation of independent thread models and Ω_{CC} corresponds to concurrency constraints.

The token-based approach has been recently augmented with a partial-order reduction technique [64] to remove *all* the redundant interleavings but to keep the necessary ones for a given unroll bound. Such a POR technique is also targeted to reduce the size of verification conditions (as opposed to increase in the verification formula size in the case of synchronous modeling). Specifically, the size of the formula Ω_{CC} is reduced by identifying redundant context switches.

The verification model is obtained in three phases: In the *first* phase, for a given set of unrolled threads, a partial order technique referred as *Mutually Atomic Transaction* analysis (MAT, in short) [64] is used to identify a subset of possible context switches such that all and only representative schedules are permissible. Using such analysis, a set of so-called *independent transaction* is obtained. Recall, an independent transaction is atomic with respect to a set of schedules (Sect. 6.2).

In the *second* phase, the goal is to obtain abstract and decoupled thread models. Each thread is decoupled from the other threads by localizing all the shared variables. Each model is then abstracted by allowing renamed (i.e., localized) variables to take non-deterministic values at every shared access. To achieve that, each independent transaction is instrumented with a pre-access atomic transition referred as *read_sync* access (denoted as `rs`) and a post-access atomic transition referred as *write_sync* access (denoted as `ws`). In *read_sync* access, all localized shared variables get non-deterministic values. We show such a token-passing model in the Fig. 6.8. Note, the transition (update) relation for each localized shared variable depends only on other local variables, thereby, making the model independent

(i.e., decoupled). However, due to non-deterministic read values, the model have additional behaviors, hence, it is an abstract model. The constraints corresponding to independent thread models capture the program order property.

In the *third* phase, the goal is to remove the imprecision caused due to abstraction. In this phase, the constraints are added to restrict the introduced non-determinism and to capture the necessary interleavings. More specifically, for each pair of inter-thread control states (identified in MAT analysis) *token-passing constraints* are added to allow passing of the token from one thread to another, giving a total order in the shared accesses. Furthermore, these constraints allow the values of the localized shared variables to synchronize between threads. These constraints satisfy the *read value* and *total order* property.

6.9.1 MAT-Based Partial Order Reduction

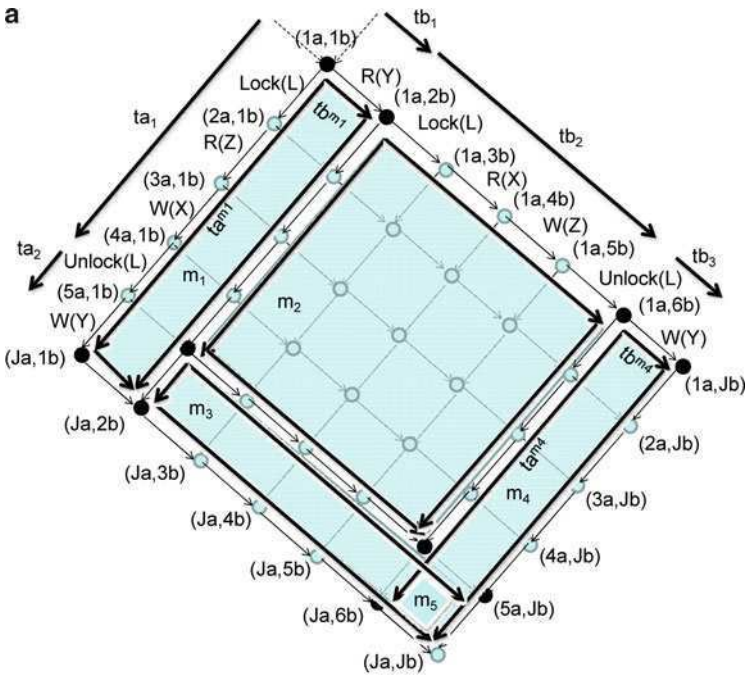
Consider a pair (ta^{m_1}, tb^{m_1}), shown as the shaded rectangle m_1 in Fig. 6.9a, where $ta^{m_1} \equiv Lock(L)_a \cdot R(Z)_a \cdots W(Y)_a$ and $tb^{m_1} \equiv R(Y)_b$ are transactions of threads M_a and M_b, respectively. For the ease of readability, we use an event to imply the corresponding generator transition.

Note that from the control state pair $(1a, 1b)$, the pair $(Ja, 2b)$ can be reached by one of the two representative interleavings $ta^{m_1} \cdot tb^{m_1}$ and $tb^{m_1} \cdot ta^{m_1}$. Such a transaction pair (ta^{m_1}, tb^{m_1}) is *atomic pair-wise* as one avoids interleaving them *in-between*, and hence, referred as *Mutually Atomic Transaction* (MAT) [64]. Note that in a MAT only the last transition pair have shared accesses on the same variable, maybe co-enabled, and at least one of them being write. Other MATs $m_2 \cdots m_5$ are similar. In general, transactions associated with different MATs are not mutually atomic. For example, ta^{m_1} in m_1 is not mutually atomic with tb^{m_3} in m_3, where $tb^{m_3} \equiv Lock(L)_b \cdots W(Y)_b$.

The basic idea of MAT-based partial order reduction [64] is to restrict context switching only between the two transactions of a MAT. A context switch can only occur from the ending of a transaction to the beginning of the other transaction in the same MAT. Such a restriction reduces the set of necessary thread interleavings. For a given MAT $\alpha = (f_i \cdots l_i, f_j \cdots l_j)$, we define a set $TP(\alpha)$ of possible context switches as ordered pairs, i.e., $TP(\alpha) = \{(end(l_i), begin(f_j)), (end(l_j), begin(f_i))\}$. Note that there are exactly two context switches for any given MAT.

Let TP[5] denote a set of possible context switches. For a given unrolled thread CFGs, we say the set TP is *adequate* iff for every feasible thread schedules of the unrolled CFGs there is an equivalent schedule that can be obtained by choosing context switching only between the pairs in TP. Given a set $\mathscr{MAT}$ of MATs, we define $TP(\mathscr{MAT}) = \bigcup_{\alpha \in \mathscr{MAT}} TP(\alpha)$. A set $\mathscr{MAT}$ is called *adequate* iff

[5] In the token-passing model, as we see later, TP is exactly the set such that the token passes from a to b iff $(a,b) \in TP$.



b

#I	p∈Q	$\mathcal{MAT}_{ab}$	Q
			(1a,1b)
1	(1a,1b)	m_1:(1a⇒Ja,1b⇒2b)	**(1a,2b)**
2	(1a,2b)	m2:(1a⇒5a,2b⇒6b)	**(5a,2b)(1a,6b) (5a,6b)**
3	(5a,2b)	m3:(5a⇒Ja,2b⇒Jb)	
4	(1a,6b)	m4:(1a⇒Ja,6b⇒Jb)	
5	(5a,6b)	m5:(5a⇒Ja,6b⇒Jb)	

Context-Switching Pairs(TP($\mathcal{MAT}_{ab}$))={(2b,1a)(Ja,1b) (6b,1a) (5a,2b), (Ja,2b)(Jb,5a),(Ja,6b)(Jb,1a)}

Fig. 6.9 (**a**) MATs $\{m_1, \ldots, m_5\}$, (**b**) A run of GenMAT

$TP(\mathcal{MAT})$ is adequate. For a given unrolled CFGs, one can use an algorithm `GenMAT` [64] (not shown) to obtain an adequate set of $\mathcal{MAT}$ that allows only representative thread schedules, as claimed in the following theorem.

Theorem 6.3 (Ganai, 2009). *`GenMAT` generates a set of MATs that captures all (i.e., adequate) and only (i.e., optimal) representative thread schedules. Further, its running cost is $O(n^2 \cdot k^2)$, where n is number of threads, and k is the maximum number of shared accesses in a thread.*

The GenMAT algorithm on the running example proceeds as follows. It starts with the pair $(1a, 1b)$, and identifies two MAT candidates: $(1a \cdots Ja, 1b \cdot 2b)$ and $(1a \cdot 2a, 1b \cdots 6b)$. By giving M_b higher priority over M_a, it selects a MAT uniquely from the MAT candidates. The choice of M_b over M_a is arbitrary but fixed throughout the MAT computation, which is required for the optimality result. After selecting MAT m_1, it inserts in a queue Q, three control state pairs $(1a, 2b), (Ja, 2b), (Ja, 1b)$ corresponding to the *begin* and the *end* pairs of the transactions in m_1. These correspond to the three corners of the rectangle m_1. In the next step, it pops out the pair $(1a, 2b) \in Q$, selects MAT m_2 using the same priority rule, and inserts three more pairs $(5a, 2b), (5a, 6b), (1a, 6b)$ in Q. Note that MAT $(1a \cdots 5a, 2b \cdot 3b)$ is ignored as the interleaving $2b \cdot 3b \cdot 1a \cdots 5a$ is infeasible [65]. Note that if there is no transition from a control state such as Ja, no MAT is generated from $(Ja, 2b)$. The algorithm terminates when all the pairs in the queue (denoted as • in Fig. 6.9a) are processed.

We present the run of GenMAT in Fig. 6.9b. The table columns provide each iteration step (#I), the pair $p \in Q$ selected, the chosen $\mathscr{MAT}_{ab}$, and the new pairs added in Q (shown in bold).

Note that the order of pair insertion in the queue can be arbitrary, but the same pair is never inserted more than once. For the running example, a set $\mathscr{MAT}_{ab} = \{m_1, \cdots m_5\}$ of five MATs is generated. Each MAT is shown as a rectangle in Fig. 6.9a. The total number of context switches allowed by the set, i.e., $TP(\mathscr{MAT}_{ab})$ is 8. These context switches are marked as cs in Fig. 6.10.

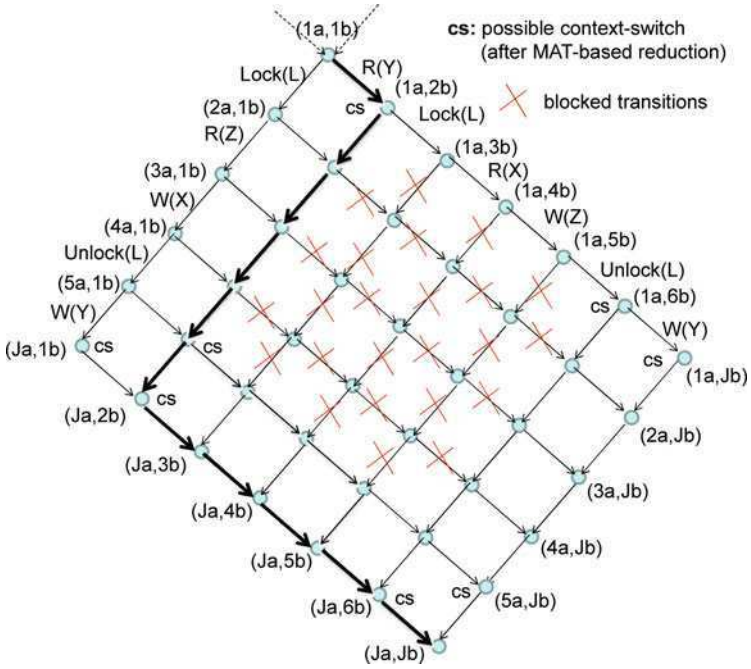


Fig. 6.10 Possible context switches (denoted as cs) after MAT-based POR [64]

The highlighted interleaving (shown in Fig. 6.4) is equivalent to the representative interleaving $tb^{m_1} \cdot ta^{m_1} \cdot tb^{m_3}$, highlighted in Fig. 6.10. One can verify (the optimality) that this is the only representative schedule (of this equivalence class) permissible by the set $TP(\mathscr{MAT}_{ab})$.

6.9.2 Independent Modeling

Given a set of MATs, we obtain a set of independent transactions of a thread M_i, denoted as AT_i, by splitting the pair-wise atomic transactions of the thread M_i as needed into multiple transactions such that a context switching (under MAT-based reduction) can occur either to the beginning or from the end of such transactions.

For the running example, the sets of independent transactions corresponding to $\mathscr{MAT}_{ab}$ are $AT_a = \{1a \cdots 5a, 5a \cdot Ja\}$ and $AT_b = \{1b \cdot 2b, 2b \cdots 6b, 6b \cdot Jb\}$. These are shown as outlines of the lattice in Fig. 6.9b.

Using independent transactions, the independent thread models, denoted as LM_i, are obtained from corresponding thread model M_i as follows:

- *Token*: A global Boolean variable, a token TK, is introduced to signify that the thread with the token can execute a shared access operation and commit its current shared state to be *visible* to the future transitions. Initially, only one thread, chosen non-deterministically, is allowed to assert TK. Later, this token is *passed*, from one thread to another, i.e., de-asserted in one thread and asserted by the other thread, respectively.
- *Logical clock*: To obtain a total ordering on token *passing* events, a clock vector $CV = \langle CS_1 \cdots CS_n \rangle$ is introduced to represent the logical clock [124]. These variables are initialized to 0. Whenever a token TK is acquired by a thread LM_i, CS_i is incremented by 1 in LM_i. The variable CS_i keeps track of the number of occurrences of token passing events wherein thread LM_i acquires the token from another thread LM_j, $j \neq i$.
- *Localization*: For each thread, the global variables are *localized* by renaming. We use the corresponding thread subscript to denote such variables. For example, TK_i represents the localized variable TK in thread i. Similarly, CS_{ji} represent the localized variable CS_j in thread i. For each thread model LM_i, a program counter PC_i is introduced to track the thread local control state. A Boolean predicate $B_{c_i} \stackrel{def}{=} (PC_i = c_i)$ encodes that current PC_i is c_i.
- *Atomic procedures*: For every independent transaction $c \stackrel{t \cdots t'}{\rightarrow} c'$, two atomic thread-specific procedures i.e., `read_sync` (rs) and `write_sync` (ws) are added before and after the transaction, respectively. It is denoted as $c \stackrel{rs}{\rightarrow} c_r \stackrel{t \cdots t'}{\rightarrow} c_w \stackrel{ws}{\rightarrow} c'$, where c_r and c_w represent the control state post `read_sync` procedure and control state pre `write_sync` procedure, respectively. In `read_sync` procedure, each localized shared variable gets a non-deterministic value, (`ND`), while in the `write_sync` procedure TK gets an `ND` value.

- *Synchronization primitives*: Operations `Lock(L)` and `Unlock(L)` are modeled as atomic operations $\{\texttt{assume}(L = 0)$; L=1$\}$ and $\{\texttt{assume}(L = 1)$; L=0$\}$, respectively. To maintain synchronization semantics, only wait-free execution are considered where the acquisition of the same lock twice is disallowed in a row without an intermediate unlock. Note, this consideration is sufficient to find all data races.

6.9.3 Concurrency Constraints

Given independent abstract models, obtained as above, the concurrency constraints are added *incrementally*, and *on-the-fly* to each BMC instance, in addition to the transition constraints of the individual thread models and property constraints. The concurrency constraints capture *inter-* and *intra*-thread dependencies due to interleavings, and thereby, eliminate additional behaviors in the models up to a bounded depth.

The pair-wise constraints are added only between the inter-thread control states as identified by the set of possible context-switches $TP(\mathscr{MAT})$. For an independent transaction, $c \xrightarrow{rs} c_r \xrightarrow{t\cdots t'} c_w \xrightarrow{ws} c'$, the control state c is designated to receive a token, and the control state c_w is designated to send a token. We use NTR_i to represent the set of control states of LM_i that can receive token, and NTS_i to represent the set of control states of LM_i that can send token.

The concurrency constraints Ω_{CC} comprise of four components:

$$\Omega_{CC} = \Omega_{TPM} \wedge \Omega_{NTR} \wedge \Omega_{NTS} \wedge \Omega_{ST} \tag{6.9}$$

where Ω_{TPM} corresponds to token-passing constraints, Ω_{NTR} corresponds to constraints when no token is received, Ω_{NTS} corresponds to constraints when no token is sent, and Ω_{ST} corresponds to single token constraint allowing only one thread, chosen non-deterministically, to have token.

For every token-passing pair of control states $(c_j, c_i) \in TP(\mathscr{MAT})$, corresponding to thread models LM_j, LM_i, following token-passing modeling constraint is added.

$$\begin{aligned}\Omega_{TPM} = \bigvee_{(c_j,c_i)\in TP(\mathscr{MAT})} & \overbrace{B_{c_i} \wedge B_{c_j} \wedge \neg TK_i \wedge TK_j \wedge CS_{ii} = CS_{ij}}^{\text{Token-passing Enabling Constraints}} \\ & \rightarrow \bigwedge_{v\in\mathscr{V}} (next(v_i) = v_j) \wedge next(TK_i) \wedge \neg next(TK_j) \\ & \bigwedge_{q=1,q\neq i}^{n} next(CS_{qi}) = CS_{qj} \wedge \ (next(CS_{ii}) = CS_{ii} + 1)\end{aligned} \tag{6.10}$$

Note, c_j refers to the control state before `write_sync` access, and v_i, v_j refer to the localized global variable v in thread model LM_i, LM_j, respectively.

The token passing condition is enabled when (a) the *read_sync* access is enabled at state c_i in thread model LM_i which does not hold the token, (b) the *write_sync* access is enabled at state c_j in thread model LM_j which holds the token, and (c) the thread model LM_j has the latest value of clock variable of LM_i, and both threads agree on that. If the token is passed, then each localized shared variable of LM_i gets the current state value of the corresponding localized shared variable of LM_j. The next state of token in LM_i is constraint to true, while it is constraint to false in LM_j. The next state value of clock variable of LM_i is incremented by 1, while the remaining clock variables in LM_i are synchronized with the respective values in LM_j. Note that these constraints ensure that the token passing will be enabled for at most one pair of *read_sync* and *write_sync* accesses.

When no token is received, following constraints are added to ensure that the next state values of each localized shared variable remain unchanged, and the token remains asserted.

$$\Omega_{NTR} = \bigvee_{i=1}^{n} \bigvee_{c \in NTR_i} (B_c \wedge TK_i) \rightarrow \left(next(TK_i) \wedge \bigwedge_{v \in \mathcal{V}} (next(v_i) = v_i) \wedge \bigwedge_{j=1}^{n} next(CS_{ji}) = CS_{ji} \right) \tag{6.11}$$

When no token is sent, following constraints are added to ensure that the token remains asserted.

$$\Omega_{NTS} = \bigvee_{i=1}^{n} \bigvee_{c \in NTS_i} (B_c \wedge TK_i) \rightarrow next(TK_i) \tag{6.12}$$

Single token constraint is added to ensure that initially exactly one thread model has the token, i.e.,

$$\Omega_{ST} = \left(\bigvee_{i=1}^{n} TK_i^0 \right) \wedge \left(\bigwedge_{i \neq j} TK_i^0 \rightarrow \neg TK_j^0 \right) \tag{6.13}$$

where TK_i^0 is the initial token value of a thread i.
Combining Theorems 6.2 and 6.3, we have

Theorem 6.4. *The token-based model with MAT analysis allows only and all sequentially consistent representative interleavings for a bounded unrolling of threads. Further, the size of pair-wise constraints added grow quadratically (in the worse case) with the unrolling depth.*

6.9.4 BMC Formula Sizes

For m transitions per unrolled thread, and n threads, token-based approach adds concurrency constraints of size $O(n^2 \cdot m^2)$ and thread transition constraints of size $O(n \cdot m)$.

For a given BMC depth k and n concurrent threads, the token-based approach guarantees finding a witness trace (if it exists), i.e., a sequence of global interleaved transitions, of length $\leq n \cdot k$, where the number of local thread transitions is at most k. In contrast, in a synchronous modeling approach using BMC [205], an unrolling depth of $n \cdot k$ is needed for such a guarantee. Thus, there is a reduction of memory use by a factor of $n \cdot k$ using token-based approach over synchronous modeling approach.

6.10 Comparison Summary

The comparisons between synchronous and asynchronous modeling based BMC approaches are summarized in Fig. 6.11. One can relate synchronous modeling with interleaving (operational) semantics, where each thread model wait in a self-loop for its turn to execute. In contrast, one can relate asynchronous modeling with axiomatic (non-operational) semantics, where the constraints are added between the thread models, which are otherwise decoupled, to maintain the sequential consistency. In the former approach, the scheduler can be constrained to remove redundant interleaving during POR, but this is often at the increased cost in the formula size. In the latter approach, the constraints are added to allow necessary interleavings. Further, one can reduce the constraints added using partial order reduction techniques. The latter approach often leads to smaller formula size compared to the former. We also compare the various BMC approaches discussed, in Fig. 6.12.

Synchronous Modeling	Asynchronous Modeling
With scheduler: execution model	Without scheduler: axiomatic model
Models are coupled	Models are de-coupled
Constraints are added to remove redundant interleavings	Constraints are added to allow necessary interleavings
Focuses on reducing interleaved space using added constraints	Focuses on reducing formula (and interleaving space)
BMC unrolling depth for each thread same	BMC unrolling can be different

Fig. 6.11 Comparing synchronous and asynchronous modeling

Features	Peep-hole [205]	CSSA-encoding [174, 204]	Token-based Model [67, 64]
Modeling	Synchronous	Asynchronous	Asynchronous
POR	Yes (optimal)	No	Yes (Optimal)
Size	cubic	cubic	quadratic

Fig. 6.12 Comparing state-of-the-art BMC approaches for concurrent verification

6.11 Further Reading

In this section, we discuss relationship between the presented work and related state-of-the-art approaches. We present in Fig. 6.13 a tree-diagram showing the relationship between the approaches.

Model checkers such as SPIN [94], Verisoft [71], Zing [4] explore states and transitions of the concurrent system using explicit enumeration. They use state reduction techniques based on partial order methods [70, 166, 200] and transactions-based methods [59, 135, 190, 191]. These methods explore only a subset of transitions (such as persistent set [70], stubborn set [200]), and sleep set [57]) from a given global state. One can obtain a persistent set using conservative static analysis. Since static analysis does not provide precise dependency relation (i.e., hard to obtain in practice), a more practical way would be to obtain the set dynamically [57]. One can also use a sleep set [70] to eliminate redundant interleaving not eliminated by persistent set. Additionally, one can use conditional dependency relation to declare two transitions being dependent with respect to a given state [72]. In previous works, researchers have also used lockset-based transactions to cut down interleaving between access points that are provably unreachable [59, 103, 135, 190, 191]. Some of these methods also exploit the high level program semantics based on transactions and synchronization to reduce the set of representative interleavings.

Symbolic model checkers such as BDD-based SMV [143], and SAT-based BMC [63] use symbolic representation and traversal of state space, and have been shown to be effective for verifying synchronous hardware designs. There have been some efforts [3, 103, 129, 174] to combine symbolic model checking with the above mentioned state-reduction methods for verifying concurrent software using interleaving semantics. To improve the scalability of the method, some researchers have employed sound abstraction [4] with bounded number of context switches [172], while some others have used finite-state model [79, 174] or Boolean program abstractions with bounded depth analysis [37]. This is also combined with a bounded number of context switches known *a priori* [174] or a proof-guided method to discover them [79].

There have been parallel efforts [2, 24, 208] to detect bugs for weaker memory models. As shown in [209], one can check these models using axiomatic memory style specifications combined with constraint solvers. Note, though these methods support various memory models, they check for bugs using given test programs.

6.12 Summary

We presented an overview of the state-of-the-art bounded model checking techniques for concurrent programs. These techniques can be classified into two main categories: synchronous modeling and asynchronous modeling. The synchronous modeling approach combines two advanced techniques, i.e., symbolic model checking for synchronous models and partial-order reduction. However, as the concurrent

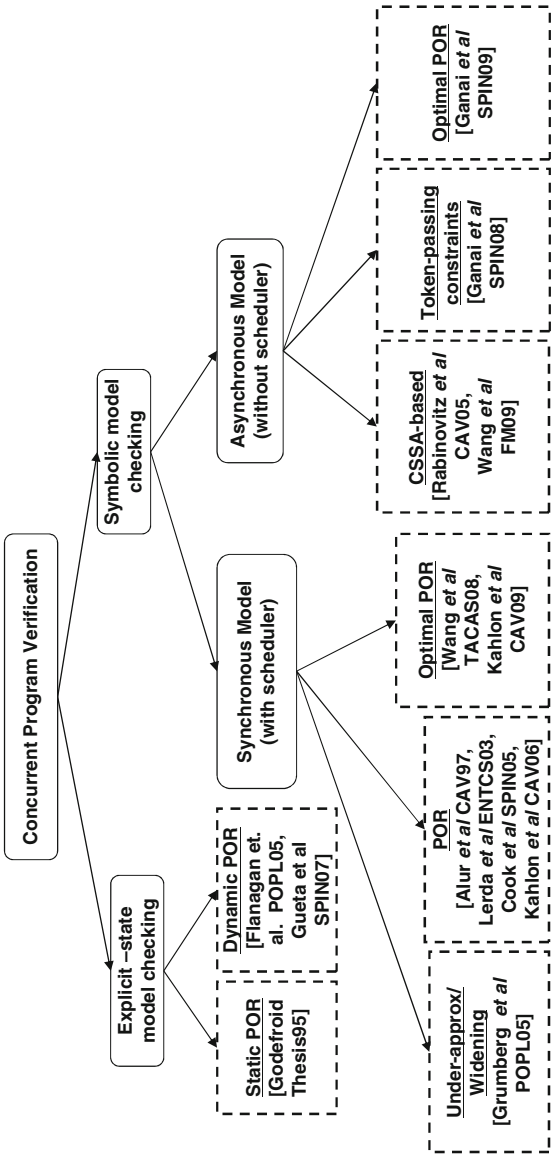


Fig. 6.13 Model checking concurrent system

system are inherently asynchronous, such synchronous modeling based approach has inherent limitations. Addition of constraints to prune search space works in principle, but in practice it often leads to lower performance of the decision procedures. To overcome that, researchers have been actively pursuing techniques that optimize model checking without modeling the system synchronously. The asynchronous modeling techniques focus primarily on generating smaller size verification conditions that can be solved easily by the decision procedures.

Acknowledgements This chapter is based on the following publications:

"Bounded Model Checking of Concurrent Programs", by I. Rabinovitz and O. Grumberg in *CAV 2005: Proceedings of 17th International Conference of Computer-Aided Verification* [174].

"Peephole Partial Order Reduction", by C. Wang, Z. Yang, V. Kahlon, and A. Gupta in *TACAS 2008: Proceedings of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* [205].

"Efficient Modeling of Concurrent Systems in BMC", by M. K. Ganai and A. Gupta in *SPIN 2008: Proceedings of 15th International SPIN Workshop on Model Checking of Software* [67].

"Reduction of Verification Conditions for Concurrent System using Mutually Atomic Transactions", by M. K. Ganai and S. Kundu in *SPIN 2009: Proceedings of 16th International SPIN Workshop on Model Checking of Software* [64].

"Symbolic predictive analysis for concurrent programs" by C. Wang, S. Kundu, M. Ganai, and A. Gupta in *FM 2009: Proceedings of 16th International Symposium on Formal Methods* [204].

Chapter 7
Translation Validation of High-Level Synthesis[1]

Once the important properties of the high-level components have been verified possibly using techniques presented in Chaps. 5 and 6, the translation from the high-level design to low-level RTL still needs to be proven correct, thereby also guaranteeing that the important properties of the components are preserved. In this chapter we will discuss an approach that proves that the translation from high-level design to a scheduled design is correct, for each translation that the HLS tool performs. In the next chapter we will describe another approach that will allow us to write part of these tools in a provably correct manner.

7.1 Overview of Translation Validation

HLS tools are large and complex software systems, often with hundreds of thousands of lines of code, and as with any software of this scale, they are prone to logical and implementation errors. Apart from applying a monolithic tool, HLS process is characterized by significant user intervention from recoding to directing the synthesis goals. Consequently, the HLS process, even with automated HLS tools, is error prone and may lead to the synthesis of RTL designs with bugs in them, which often have expensive ramifications if they go undetected until after fabrication or large-scale production. Hence, correctness of the HLS process (manual or automatic) has always been an important concern.

In general, proving that the HLS process *always* produces target RTL designs that are semantically equivalent or refinement to their source versions is usually very hard. However, even if one cannot prove the HLS process correct once and for all, one can try to show, for each translation that HLS performs, that the output program produced by these steps has the same behavior as the original program. Although this approach does not guarantee that the HLS process is bug free, it does guarantee that any errors in translation will be caught when the particular steps of HLS are

[1]

S. Kundu et al., *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, DOI 10.1007/978-1-4419-9359-5_7,

performed, preventing such errors from propagating any further in the hardware fabrication process. This approach to verification, called *translation validation*, has previously been applied with success in the context of optimizing compilers [73, 160, 170, 178, 212].

7.2 Overview of the TV-HLS Approach

During the HLS process, an engineer starts with a high-level description of the design, usually called a specification, which is then refined into progressively more concrete implementations. Checking correctness of these refinement steps has many benefits, including finding bugs in the translation process, while at the same time guaranteeing that properties checked at higher-levels in the design are preserved through the refinement process, without having to recheck them at lower levels. For example, if one checks that a given specification satisfies a safety property, and that an implementation is a correct trace refinement of the specification, then the implementation will also satisfy the safety property. In this chapter, we discuss an approach TV-HLS [116, 117, 119] that describes how translation validation can effectively be implemented in a HLS process. The novelty of this approach comes from the fact that it can account for concurrency which is inherent in hardware design. The TV-HLS approach deals with this concurrency using standard techniques for computing weakest preconditions and strongest postconditions of parallel programs [30].

The TV-HLS algorithm uses a simulation relation approach to prove refinement. The algorithm consists of two components. The first component is given a relation, and checks that this relation satisfies the properties required for it to be a correct refinement simulation relation. The second component automatically infers a correct simulation relation just from the specification and the implementation programs. In particular, the inference algorithm automatically establishes a relation that states what points in the implementation program are related to what points in the specification program. This relation guarantees that for each execution sequence in the implementation, an equivalent execution sequence exists in the specification. Apart from refinement checking, TV-HLS also generalize both the checking and inference algorithms to prove equivalence between the specification and the implementation programs using a bisimulation relation approach.

Toward the end of this chapter we discuss an implementation of this algorithms in a validating system called Surya. We then report the results of using Surya to check the correctness of a variety of refinements of infinite state concurrent systems represented using Communicating Sequential Processes (CSP) [93] programs. Next, we discuss the results of using Surya to validate a HLS tool. In particular, Surya validates all the phases (except for parsing, binding and code generation) of the Spark HLS tool [84] against the initial behavioral description. With over 4,000 downloads, and over 100 active members in the user community, Spark is a widely used tool. Although commercial HLS tools exists, these tools are not available for academic experimentation – Spark represents the state of the art in the academic community.

The TV-HLS verification approach is modular as it works on one procedure at a time. Furthermore, for the experiments on Spark, Surya took on average 6 s to run per procedure, showing that translation validation of HLS transformations can be fast enough to be practical. Finally, in running Surya, two failed validation runs have directed us to discover two previously unknown bugs in the Spark tool. These bugs cause Spark to generate *incorrect* RTL for a given high-level program. This demonstrates that translation validation of the HLS process can catch bugs that even testing and long-term use may not uncover.

7.3 Illustrative Example

At the heart of a HLS process is a model of a system consisting of concurrent pieces of functionality, often expressed as sequential program-like behavior, along with synchronous or asynchronous interactions [127, 183]. CSP is a calculus for describing such concurrent systems as a set of processes that communicate synchronously over explicitly named channels. In this chapter we describe the TV-HLS algorithm using CSP-style concurrent programs. While CSP presents a good model for a large number of hardware models described using Hardware Description Languages (HDLs), we note that the core algorithms of the TV-HLS approach does not depend on the choice of the input language. For example, in the experiment section, the TV-HLS approach is used for programs that may include arrays, and function calls that are generally not part of CSP programs.

We start out by describing the salient features of CSP required for understanding the examples in this chapter. A CSP program is a set of (possibly mutually recursive) process definitions. An asynchronous parallel composition of two processes P and Q is written as $(P \mid\mid Q)$. Asynchronous parallel processes in this version of CSP (and Hoare's original version [93]) can only communicate through messages on channels. Although there are no explicit shared variables, these can easily be simulated using a process that stores the value of the shared variable, and that services reads and writes to the variable using messages. $c?v$ denotes reading a value from a channel c into a variable v and $c!v$ denotes writing a variable v to a channel c. Reads and writes are synchronous. Channels can be visible or hidden. Visible channels are externally observable, and these are the channels whose behavior is preserved when checking for correctness. Furthermore, the TV-HLS algorithms also allow simple C-style control instructions and synchronous parallel composition. By allowing both asynchronous (inherent in CSP) and synchronous semantics of concurrency the TV-HLS approach support system designs which are *Globally Asynchronous Locally Synchronous* (GALS).

We now present a simple example that illustrates the approach (Fig. 7.1). For now ignore the dashed lines in the figure. The specification is a sequential process X shown in Fig. 7.1a using our internal Concurrent Control Flow Graph (CCFG) representation after tail recursion elimination has been performed. We omit the details of the actual CSP code, because the CCFG representation is complete, and we

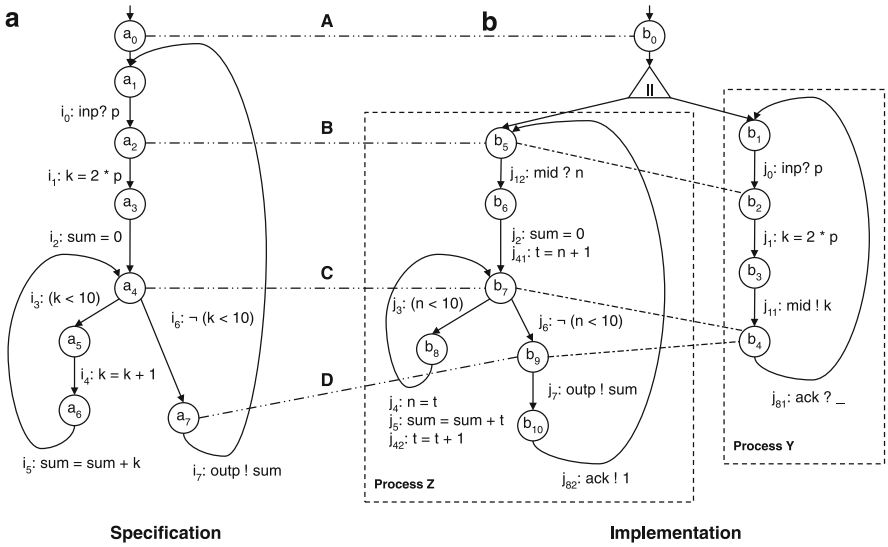


Fig. 7.1 CCFGs of our running example along with a simulation relation

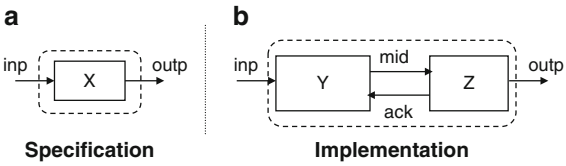


Fig. 7.2 Communication diagrams of our running example

believe the CSP code only makes the example harder to follow. This process is continually reading values from an input channel called inp into a variable p and then computes the sum from $(2 \times p + 1)$ to 10 using a loop. Finally, it writes the sum out to a channel named outp. In refinement based hardware development, the designer often starts with such a high-level description of a sequential design, refining the details of the implementation later on.

An implementation (Fig. 7.1b) may use two separate parallel processes (components) Y and Z, communicating via a hidden channel mid and an acknowledgment channel ack as shown in Fig. 7.2b. Like its specification it also takes a value from the inp channel into a variable p and outputs the sum from $(2 \times p + 1)$ to 10 in the outp channel. However, now it does so in 2 steps, first the process Y multiplies p by 2 and sends it to the component Z then process Z computes the sum and writes it to the outp channel. One additional subtlety of this example is that, in order for the refinement to be correct, an additional channel needs to be added for sending an acknowledgment token (in this case the value 1) back to the process Y, so that a new value isn't read from the inp channel until the current value has been written out to the outp channel. The value read from the ack channel is not used, and so we use an "_" for the variable being read. Instructions on the same transition edge are executed in *parallel* (synchronously).

Apart from the architectural differences, the loop-structure in the implementation is different from the one in the specification in several ways. First, a loop-shifting transformation has moved the operation i_4 from the beginning of the loop body to the end of the loop body (j_{42}), while also placing a copy of the operation in the loop header (j_{41}) using the temporary variable t. The effect of this loop-shifting transformation is a form of software pipelining [123]. Note that without this pipelining transformation it would not have been possible to schedule the operation i_4 and i_5 together due to the data dependence between them. In addition to loop-shifting, a copy propagation of instruction j_4 to j_5 and j_{42} are also performed. This ability to make large scale code transformations via parallelizing code transformations as shown here is an important aspect of parallelizing HLS implemented in SPARK. Even without HLS tools, similar source-level transformations are often done manually by the designer to optimize the generated code as a part of high-level design process.

7.3.1 Translation Validation Approach

The TV-HLS [116, 117, 119] translation validation approach consists of two parts, which theoretically are independent, but for practical reasons, one part subsumes the other as explained below. The first part is a *checking algorithm* that, given a relation, determines whether or not it satisfies the properties required for it to be a valid simulation relation. The second part is an *inference algorithm* that infers a relation given two programs, one of which is a specification, and the other is an implementation. To check that one program is a refinement to another, one therefore runs the inference algorithm to infer a relation, and then one uses the checking algorithm to verify that the resulting relation is indeed the required relation. Because the inference algorithm does a similar kind of exploration as the checking algorithm, this leads to duplicate work. To reduce this work, the inference algorithm has been made to also perform checking, with only a small amount of additional work. This avoids having the checking algorithm duplicate the exploration work done by the inference algorithm. The checking algorithm is nonetheless useful by itself, in case the inference algorithm is not capable of finding an appropriate relation, and the relation is manually provided by the system designer.

7.3.2 Simulation Relation

The goal of the simulation relation in this approach is to guarantee that the specification and the implementation interact in the same way with any surrounding environment that they would be placed in. The simulation relation guarantees that the set of execution sequences of visible instructions in the implementation is a subset of the set of execution sequences in the specification. In what follows, let us

Table 7.1 A simulation relation for our running example

	(gl_1, gl_2, ϕ)
A.	$(a_0, b_0, true)$
B.	$(a_2, (b_2, b_5), \mathsf{p_s} = \mathsf{p_i})$
C.	$(a_4, (b_4, b_7), \mathsf{k_s} = \mathsf{n_i} \wedge \mathsf{sum_s} = \mathsf{sum_i} \wedge (\mathsf{k_s} + 1) = \mathsf{t_i})$
D.	$(a_7, (b_4, b_9), \mathsf{sum_s} = \mathsf{sum_i})$

consider the visible instructions to be read and write operations to visible channels. However, in Sect. 7.8.2, we will use visible instructions to be function calls and return statements.

The simulation relation (defined formally in Sect. 7.5) consists of a set of entries of the form (gl_1, gl_2, ϕ), where gl_1 and gl_2 are program locations in the specification and implementation respectively, and ϕ is a predicate over variables of the specification and implementation. The pair (gl_1, gl_2) captures how the control state of the specification is related to the control state of the implementation, whereas ϕ captures how the data is related. For our running example, the entries in the simulation relation are labeled A through D in Fig. 7.1, and each entry has a predicate associated with it as shown in Table 7.1.

The first entry 'A' in the simulation relation relates the start location of the specification and the implementation. For this entry, the relevant data invariant is *true*, as we have no information about the states of the programs in those locations. The second entry 'B' shows the specification just as it finishes reading a value from the inp channel. The corresponding control state of the implementation has the Y process in the same state, just as it finishes reading from the inp channel and the other process Z is at the top of its loop. We use subscript s to denote variables in the specification and subscript i for variables in the implementation. For this entry, the relevant data invariant is $\mathsf{p_s} = \mathsf{p_i}$, which states that the value of p in the specification is equal to the value of p in the implementation. This is because both the specification and the implementation have stored in p the same value from the surrounding environment. In the Sect. 7.3.4, we explain in further detail how this algorithm models the environment as a set of separate processes that are running in parallel with the specification and the implementation. For now we hide these additional processes for clarity of exposition.

The next entry 'C' in the simulation relation relates the loop head (a_4) in the specification with the loop head (b_7) of the Z process in the implementation. This entry represent two loops that run in synchrony, one loop being in the specification and the other being in the implementation. The invariant can be seen as a loop invariant across the specification and the implementation, which guarantee that the two loops produce the same effect on the visible instructions. The data part of this entry guarantee that the two loops are in fact synchronized. Nominally, we need at least one entry in the simulation relation that "cuts through" every loop pair, in the same way that there must be at least one invariant through each loop when reasoning about a single sequential program.

The last entry 'D' in the simulation relation relates the location a_7 in the specification with the location (b_4, b_9) of the implementation. The relevant invariant

for this entry is $\mathsf{sum_s} = \mathsf{sum_i}$, since the specification is about to write $\mathsf{sum_s}$ to the externally visible outp channel and the implementation is about to write $\mathsf{sum_i}$ to the same channel (our correctness criterion).

Simultaneous execution from the last entry 'D' can reach back to 'B', establishing the invariant $\mathsf{p_s} = \mathsf{p_i}$, since by the time execution reaches the second entry again, both the specification and the implementation would have read the next value from the environment (details of how the algorithm establishes that the two next values read from the environment processes are equal is explained in the Sect. 7.3.4).

7.3.3 Checking Algorithm

The entries in the simulation relation must satisfy some simple local requirements (which are made precise in Sect. 7.5). Intuitively, for any entry (gl_1, gl_2, ϕ) in the simulation relation, if the specification and implementation start executing in parallel at control locations gl_1 and gl_2 in states where ϕ holds, and they reach another simulation entry (gl_1', gl_2', ϕ'), then ϕ' must hold in the resulting states.

Given a simulation relation, the checking algorithm checks each entry in the relation individually. For each entry (gl_1, gl_2, ϕ), it finds all other entries that are reachable from (gl_1, gl_2), without going through any intermediate entries. For each such entry (gl_1', gl_2', ψ), it then checks using a theorem prover that if (1) ϕ holds at gl_1 and gl_2, (2) the specification executes from gl_1 to gl_1' and (3) the implementation executes from gl_2 to gl_2', then ψ will hold at gl_1' and gl_2'.

For our example, the traces in the implementation and the specification from B to C and the traces from C to itself are shown in Fig. 7.3a and b respectively. The communication events have been transformed into assignments and the original communication events are shown in parenthesis.

For the B - C path shown in Fig. 7.3a, the checking algorithm uses a theorem prover to validate that if $\mathsf{p_s} = \mathsf{p_i}$ holds before the two traces, then $(\mathsf{k_s} = \mathsf{n_i} \wedge \mathsf{sum_s} = \mathsf{sum_i} \wedge (\mathsf{k_s} + 1) = \mathsf{t_i})$ holds after the traces have been executed. Recall in Chap. 4 we discussed how an automated theorem prover can be used to verify the above condition. Similarly, the theorem prover checks the traces from C to C shown in Fig. 7.3b and also all the other entries in the simulation relation. If there were multiple paths from an entry, then the algorithm checks all of them.

7.3.4 Inference Algorithm

The inference algorithm starts by finding the pairs of locations in the implementation and the specification that need to be related in the simulation. In the given example, the algorithm first adds (a_0, b_0) as a pair of interest, which is the entry location of both programs. Then it moves forward simultaneously in the implementation and the specification until it reaches a branch or an operation (read or write) on a visible

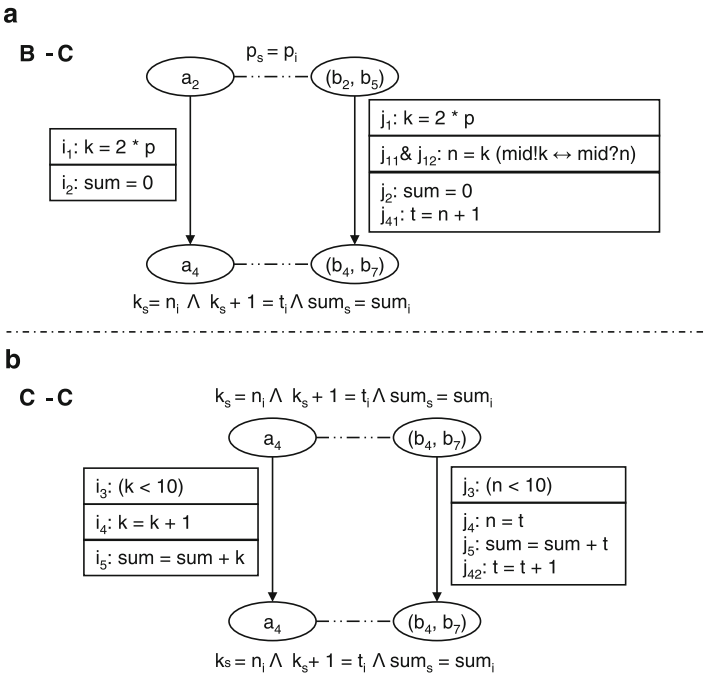


Fig. 7.3 Checking the simulation relation. (**a**) Traces from B to C (**b**) Traces from C to C

Table 7.2 Iterations for computing the simulation relation

	(gl_1, gl_2)	1st iteration	2nd iteration	3rd iteration (ϕ)
A.	(a_0, b_0)	*true*	*true*	*true*
B.	$(a_2, (b_2, b_5))$	$p_s = p_i$	$p_s = p_i$	$p_s = p_i$
C.	$(a_4, (b_4, b_7))$	$k_s = n_i$	$k_s = n_i \wedge sum_s = sum_i \wedge (k_s + 1) = t_i$	$k_s = n_i \wedge sum_s = sum_i \wedge (k_s + 1) = t_i$
D.	$(a_7, (b_4, b_9))$	$sum_s = sum_i$	$sum_s = sum_i$	$sum_s = sum_i$

channel. In the example from Fig. 7.1, the algorithm finds that there is a branch, an input and an output event that must be matched (the specification events inp?p and outp!sum should match, respectively, with the implementation events inp?p and outp!sum). This amounts to computing the first column of Table 7.2. While finding these pairs of locations, the inference algorithm also does two more things. First, it correlates the branch in the specification and the implementation (details of how branch correlations are established is explained in Sect. 7.6). Next, it finds the local conditions that must hold for the visible events to match. For events that output to externally visible channels, the local condition states that the written values in the specification and the implementation must be the same. For example, the local condition for the output event is $sum_s = sum_i$.

For events that read from externally visible channels, the local condition states that the specification and the implementation are reading from the same point

in the conceptual stream of input values. To achieve this, the algorithms use an environment process that models each externally visible input channel c as an unbounded array values of input values, with an index variable i stating which value in the array should be read next. This environment process runs an infinite loop that continually outputs values[i] to c and increments i. Assuming that i and j are the index variables from the environment processes that model an externally visible channel c in the specification and the implementation, respectively, then the local condition for matching events c?a (in the specification) and c?b (in the implementation) would then be $i_{\mathsf{s}} = j_{\mathsf{i}}$. The equality between the index variables implies that the values being read are the same, and since this fact is always true, the algorithms directly add this to the generated local condition, producing $i_{\mathsf{s}} = j_{\mathsf{i}} \wedge a_{\mathsf{s}} = b_{\mathsf{i}}$.

Once the related pairs of locations have been collected the inference algorithm defines, for each pair of locations (gl_1, gl_2), a constraint variable $\psi_{(gl_1,gl_2)}$ to represent the state-relating formula that will be computed in the simulation relation for that pair. It then defines a set of constraints over these variables to ensure that the would-be simulation relation is a simulation.

There are two kinds of constraints. First, for each pair of locations (gl_1, gl_2) that are related, we want $\psi_{(gl_1,gl_2)}$ to imply that the local condition at those locations hold. For example, $\psi_{(a_7,(b_4,b_9))}$ should imply $\mathsf{sum}_{\mathsf{s}} = \mathsf{sum}_{\mathsf{i}}$, so that the output values are the same. Such constraints guarantee that the computed simulation relation is strong enough to show that the visible instructions behave the same way in the specification and the implementation. A second kind of constraint is used to state the relationship between one pair of related locations and other pairs of related locations. For example, if starting at (gl_1, gl_2) in states satisfying $\psi_{(gl_1,gl_2)}$, the specification and implementation can execute in parallel to reach another related pair of locations (gl'_1, gl'_2), then $\psi_{(gl'_1,gl'_2)}$ must hold in the resulting states. As shown later in Sect. 7.6, such constraints can be stated over the variables $\psi_{(gl_1,gl_2)}$ and $\psi_{(gl'_1,gl'_2)}$ using the weakest precondition operator (wp). This second kind of constraint guarantees that the computed simulation relation is in fact a simulation.

Once the constraints are generated, next the algorithm solves them using an iterative algorithm that starts with all constraint variables set to *true* and then iteratively strengthens the constraint variables until a theorem prover is able to show that all constraints are satisfied. Although in general this constraint-solving algorithm is not guaranteed to terminate, in practice it can quickly find the required simulation relation.

The constraint solving for our example is shown in Table 7.2. The algorithm first initializes the constraint variables with the local conditions that are required for the visible instructions to be equivalent (second column of Table 7.2). In particular, it initializes A with *true* as both the programs are at their entry locations, B with $\mathsf{p}_{\mathsf{s}} = \mathsf{p}_{\mathsf{i}}$ for input matching, C with $\mathsf{k}_{\mathsf{s}} = \mathsf{n}_{\mathsf{i}}$ for branch correlation, and D with $\mathsf{sum}_{\mathsf{s}} = \mathsf{sum}_{\mathsf{i}}$ for output matching. The algorithm next chooses any entry from the table, say C and finds the entries that can reach it (i.e. C and B). Consider the synchronized loop from C to C shown in Fig. 7.4a. The algorithm computes the weakest precondition of the formula at the bottom ($\mathsf{k}_{\mathsf{s}} = \mathsf{n}_{\mathsf{i}}$) over the instructions in the implementation and in the specification, which happens to be

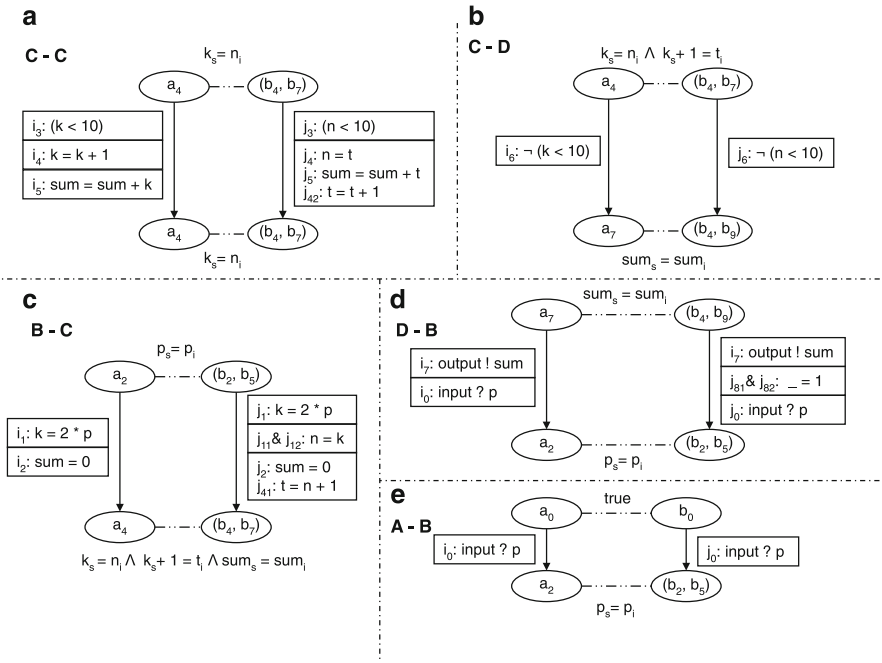


Fig. 7.4 Steps of the 2nd iteration for computing the simulation relation

$\delta = [(k_s < 10) \Rightarrow (n_i < 10) \Rightarrow (k_s + 1) = t_i]$ (recall from Sect. 4.3, how to compute weakest precondition). Next, it asks a theorem prover if the condition at the top, i.e., $k_s = n_i$ implies δ. Since it does not, the algorithm strengthens the constraint variable at the top with $(k_s + 1) = t_i$ which is a stronger condition than δ. A similar pass through Fig. 7.4b strengthens the constraint variable at C with $(sum_s = sum_i)$. For the other paths B - C, D - B, and A - B shown in Fig. 7.4 the theorem prover is able to validate the implication, and as such it does not need to strengthen. The constraint solving continues in this manner until a fixpoint is reached.

7.4 Definition of Refinement

We now present a formal description of the approach that builds upon the illustration discussed in the previous section. The TV-HLS approach verifies each procedure in the specification against the corresponding procedure in the implementation. Semantically a concurrent program in the TV-HLS approach consists of a set of processes. For simplicity, let us assume that the programs are single-entry-single-exit programs.

In this chapter, we represent each process in the specification and the implementation using a *transition diagram* that describes the control structure of the process in terms of *generalized program locations* and *program transitions*. A generalized program location represents a point of control in the (possibly

concurrent) program. A generalized program location is either a node identifier, or a pair of two generalized program locations, representing the state of two processes running in parallel. A transition describes how the program state changes from one program location to another. We represent these transitions by instructions.

Let $\mathscr{L}$ denote the finite set of generalized program locations, *VAR* denote the set of variables and *VAL* denote the domain of values.

More formally, we define a *data state* to be a function $VAR \to VAL$ assigning values to variables. We denote by Σ the set of all data states. We define an instruction to be a pair (c, f) where $c : \Sigma \to \mathscr{B}$ is a predicate and $f : \Sigma \to \Sigma$ is a state transformation function. The predicate c is the condition under which the state transformation function f can happen. For instance, in Fig. 7.1 the instruction i_3 has $c(\sigma) = (\sigma(\mathsf{k}) < 10)$ and $f(\sigma) = \sigma$, whereas the instruction i_2 has $c(\sigma) = \mathit{true}$ and $f(\sigma) = \sigma[\mathsf{sum} \mapsto 0]$. Finally a transition diagram is defined as follows.

Definition 10 (Transition Diagram). A transition diagram π is a tuple $(\mathscr{L}, \mathscr{I}, \to, \iota, \varepsilon)$, where $\mathscr{I}$ is a finite set of instructions, $\to \subseteq \mathscr{L} \times \mathscr{I} \times \mathscr{L}$ is a finite set of triples (gl, i, gl') called transitions, $\iota \in \mathscr{L}$ is the entry location, and $\varepsilon \in \mathscr{L}$ is the exit location. We write $gl \xrightarrow{i} gl'$ to denote $(gl, i, gl') \in \to$.

Definition 11 (Configuration). Given a transition diagram $\pi = (\mathscr{L}, \mathscr{I}, \to, \iota, \varepsilon)$, we define a configuration to be a pair $\langle gl, \sigma \rangle$, where $gl \in \mathscr{L}$ and $\sigma \in \Sigma$.

Definition 12 (Semantic Step). Given a transition diagram $\pi = (\mathscr{L}, \mathscr{I}, \to, \iota, \varepsilon)$, two configurations $\langle gl, \sigma \rangle$ and $\langle gl', \sigma' \rangle$, and an instruction $i \in \mathscr{I}$, the semantic step relation is defined as follows:

$$\langle gl, \sigma \rangle \overset{i}{\rightsquigarrow} \langle gl', \sigma' \rangle \quad \mathit{iff} \quad gl \xrightarrow{i} gl' \ \wedge \ i = (c, f) \ \wedge \ c(\sigma) = \mathit{true} \ \wedge \ \sigma' = f(\sigma).$$

Definition 13 (Execution Sequence). For a given transition diagram $\pi = (\mathscr{L}, \mathscr{I}, \to, \iota, \varepsilon)$, an execution sequence η starting in configuration $\langle gl_0, \sigma_0 \rangle$ is a sequence of configurations such that:

$$\langle gl_0, \sigma_0 \rangle \overset{i_1}{\rightsquigarrow} \langle gl_1, \sigma_1 \rangle \overset{i_2}{\rightsquigarrow} \cdots \overset{i_n}{\rightsquigarrow} \langle gl_n, \sigma_n \rangle$$

We denote by $\mathscr{N}$ the set of all execution sequences. We use the shorthand notation $\eta\langle \pi, gl_0, \sigma_0 \rangle$ to represent an execution sequence η starting in configuration $\langle gl_0, \sigma_0 \rangle$ in π.

We define ϑ to be the set of *visible instructions*. These are the instructions whose semantics are preserved between the specification and implementation. Because the TV-HLS algorithm is parameterized by the set ϑ of visible instructions, we can apply this approach to various settings. For example, in this discussion we consider visible instructions to be input and output to visible channels. In Sect. 7.8.2, however we define visible instructions to be function calls and return statements. For $v_1, v_2 \in \vartheta$, we write $\langle v_1, \sigma_1 \rangle \equiv \langle v_2, \sigma_2 \rangle$ to represent that v_1 in state σ_1 is equivalent to v_2 in states σ_2. In the case of channels, two visible instructions are equivalent iff they both are inputs, or both outputs on the same channel and their values are the same. In the case of function calls and returns, we say that two function calls are equivalent iff the

state of globals, the arguments and the address of the called function are the same. Furthermore, we say that two returns are equivalent iff the returned value and the state of the globals are the same. This concept of equivalence for visible instruction can be extended to execution sequences as follows.

Definition 14 (Equivalence of Execution Sequences). Two execution sequences $\eta_1 \in \mathscr{N}$ and $\eta_2 \in \mathscr{N}$ are said to be equivalent, written $\eta_1 \equiv \eta_2$, if the two sequences contain visible instructions that are pairwise equivalent.

Definition 15 (Refinement of Transition Diagrams). Given two transition diagrams $\pi_1 = (\mathscr{L}_1, \mathscr{I}_1, \rightarrow_1, \iota_1, \varepsilon_1)$ and $\pi_2 = (\mathscr{L}_2, \mathscr{I}_2, \rightarrow_2, \iota_2, \varepsilon_2)$, we define π_1 to be a refinement of π_2 (written $\pi_1 \sqsubseteq \pi_2$) iff for every $\sigma_1 \in \Sigma$ and $\eta_1\langle\pi_1, \iota_1, \sigma_1\rangle \in \mathscr{N}$ there exists $\sigma_2 \in \Sigma$ and $\eta_2\langle\pi_2, \iota_2, \sigma_2\rangle \in \mathscr{N}$ such that $\eta_1 \equiv \eta_2$.

7.5 Simulation Relation

A *verification relation* between two transition diagrams π_1 and π_2 is a set of triples (gl_1, gl_2, ϕ), where $gl_1 \in \mathscr{L}_1$, $gl_2 \in \mathscr{L}_2$ and ϕ is a predicate over the variables live at locations gl_1 and gl_2. Let the set of such predicates be denoted by $\Phi \stackrel{def}{=} \Sigma \times \Sigma \rightarrow \mathscr{B}$. We write $\phi(\sigma_1, \sigma_2) = true$ to indicate that ϕ is satisfied in $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$.

Simulation relations are verification relations with a few additional properties. To define these properties, we make use of a *cumulative semantic step* relation $\rightsquigarrow^+$, which works like $\rightsquigarrow$, except that it can take multiple steps at once, and it accumulates the steps taken into an execution sequence.

Definition 16 (Cumulative Semantic Step). Given configurations $\langle gl_0, \sigma_0\rangle$ and $\langle gl_n, \sigma_n\rangle$, and an execution sequence η that contains at least one transition, we define $\rightsquigarrow^+$ as follows:

$$\langle gl_0, \sigma_0\rangle \stackrel{\eta}{\rightsquigarrow^+} \langle gl_n, \sigma_n\rangle \quad \textit{iff} \quad \eta = \langle gl_0, \sigma_0\rangle \stackrel{i_1}{\rightsquigarrow} \cdots \stackrel{i_n}{\rightsquigarrow} \langle gl_n, \sigma_n\rangle$$

Definition 17 (Simulation Relation). A simulation relation R for two transition diagrams $\pi_1 = (\mathscr{L}_1, \mathscr{I}_1, \rightarrow_1, \iota_1, \varepsilon_1)$ and $\pi_2 = (\mathscr{L}_2, \mathscr{I}_2, \rightarrow_2, \iota_2, \varepsilon_2)$ is a verification relation such that:

$$\begin{aligned}
&R(\iota_1, \iota_2, true)\\
&\quad \forall gl_2, gl_2' \in \mathscr{L}_2, gl_1 \in \mathscr{L}_1, \sigma_1, \sigma_2, \sigma_2' \in \Sigma, \phi \in \Phi, \eta_2 \in \mathscr{N}.\\
&\quad \left[\begin{array}{l} \langle gl_2, \sigma_2\rangle \stackrel{\eta_2}{\rightsquigarrow_2^+} \langle gl_2', \sigma_2'\rangle \wedge \\ \qquad R(gl_1, gl_2, \phi) \wedge \phi(\sigma_1, \sigma_2) = true \end{array}\right]\\
&\qquad \Rightarrow \exists gl_1' \in \mathscr{L}_1, \sigma_1' \in \Sigma, \phi' \in \Phi, \eta_1 \in \mathscr{N}.\\
&\qquad \left[\begin{array}{l} \langle gl_1, \sigma_1\rangle \stackrel{\eta_1}{\rightsquigarrow_1^+} \langle gl_1', \sigma_1'\rangle \wedge \\ R(gl_1', gl_2', \phi') \wedge \phi'(\sigma_1', \sigma_2') = true \wedge \eta_1 \equiv \eta_2 \end{array}\right]
\end{aligned}$$

Intuitively, these conditions respectively state that (1) the entry location of π_1 must be related to the entry location of π_2; and (2) if π_1 and π_2 are in a pair of related configurations, and π_2 can proceed one or more steps producing an execution sequence η_2, then π_1 must also be able to proceed one or more steps, producing a sequence η_1 that is equivalent to η_2, and the two resulting configurations must be related.

The following lemma and theorem connect the above relation with our definition of refinement for transition diagrams (Definition 15).

Lemma 7.1 (Refinement). *If R is a simulation relation for π_1, π_2, then for each element $(gl_1, gl_2, \psi) \in R$, $\sigma_2 \in \Sigma$, and $\eta_2\langle \pi_2, gl_2, \sigma_2 \rangle \in \mathcal{N}$, there exists $\sigma_1 \in \Sigma$, and $\eta_1\langle \pi_1, gl_1, \sigma_1 \rangle \in \mathcal{N}$ such that $\eta_1 \equiv \eta_2 \wedge \psi(\sigma_1, \sigma_2) = true$.*

Theorem 7.1 (Refinement). *If there exists a simulation relation for π_1, π_2, then $\pi_2 \sqsubseteq \pi_1$.*

The conditions from Definition 17 are used as the base case and the inductive case of a proof by induction showing that π_2 is a refinement of π_1. Thus, a simulation relation is a witness that π_2 is a refinement of π_1.

7.6 The Translation Validation Algorithm

The TV-HLS translation validation algorithm consists of two parts, checking and inference. To show that a transition diagram is a refinement of another transition diagram, it shows there exist a simulation relation between them. In the following sections we describe the algorithms for computing a simulation relation.

Given a transition diagram π and a set of locations S, we define the *skipping transition* relation $\hookrightarrow$, which is a version of $\longrightarrow$ that skips over all locations not in S. This transition allows us to focus our attention on only those locations that are in S.

Definition 18 (Skipping Transition). Let $\pi = (\mathscr{L}, \mathscr{I}, \rightarrow, \iota, \varepsilon)$ be a transition diagram, $gl, gl' \in S$, and $w \in \mathscr{I}^*$, where $w = i_0 \cdots i_n$. We define the skipping transition relation $\hookrightarrow$ for π as follows:

$$gl \overset{(w,S)}{\hookrightarrow}_{\pi} gl' \quad \textit{iff} \quad \exists gl_1, \ldots, gl_n \in (\mathscr{L} - S).\ gl \xrightarrow{i_0} gl_1 \cdots gl_n \xrightarrow{i_n} gl'$$

Throughout the rest of this chapter, we assume that $\pi_1 = (\mathscr{L}_1, \mathscr{I}_1, \rightarrow_1, \iota_1, \varepsilon_1)$ represents the procedure in the specification, and $\pi_2 = (\mathscr{L}_2, \mathscr{I}_2, \rightarrow_2, \iota_2, \varepsilon_2)$ represents the corresponding procedure in the implementation. Thus, the goal here is to show that π_2 is a refinement of π_1 (i.e. $\pi_2 \sqsubseteq \pi_1$).

7.6.1 Checking Algorithm

In this section, we present the details of the checking algorithm that checks whether a verification relation is indeed a correct simulation relation. We let

$\mathscr{R} \subseteq \mathscr{L}_1 \times \mathscr{L}_2 \times \Phi$ to be the verification relation that needs to be checked. We then define two sets of locations $\mathscr{P}_1$ and $\mathscr{P}_2$, which are of interest to the algorithm.

$$\mathscr{P}_1 = \{gl_1 \mid \exists gl_2, \phi.\ (gl_1, gl_2, \phi) \in \mathscr{R}\}$$
$$\mathscr{P}_2 = \{gl_2 \mid \exists gl_1, \phi.\ (gl_1, gl_2, \phi) \in \mathscr{R}\}$$

To focus our attention on only those locations in $\mathscr{P}_1$ and $\mathscr{P}_2$, we use the skipping transition relation $\hookrightarrow$. In this section, we use the shorthand notation $gl_1 \overset{w_1}{\hookrightarrow}_1 gl_1'$ for $gl_1 \overset{(w_1, \mathscr{P}_1)}{\hookrightarrow}_{\pi_1} gl_1'$, and $gl_2 \overset{w_2}{\hookrightarrow}_2 gl_2'$ for $gl_2 \overset{(w_2, \mathscr{P}_2)}{\hookrightarrow}_{\pi_2} gl_2'$.

Given an entry in $\mathscr{R}$, we then define the *next transition* relation $\longrightarrow\!\!\!\!\rightarrow$, which traverses the two transition diagrams π_1 and π_2 simultaneously to the next entries reachable from it.

Definition 19 (Next Transition). Given $(gl_1, gl_2, \phi) \in \mathscr{R}$, $(gl_1', gl_2', \psi) \in \mathscr{R}$, $w_1 \in \mathscr{I}_1^*$ and $w_2 \in \mathscr{I}_2^*$, we define $\longrightarrow\!\!\!\!\rightarrow$ as follows:

$$(gl_1, gl_2, \phi) \overset{(w_1, w_2)}{\longrightarrow\!\!\!\!\rightarrow} (gl_1', gl_2', \psi) \quad \textit{iff} \quad gl_1 \overset{w_1}{\hookrightarrow}_1 gl_1' \wedge gl_2 \overset{w_2}{\hookrightarrow}_2 gl_2'$$

For the verification relation $\mathscr{R}$ to be a simulation relation we require it to satisfy certain conditions. In particular, we want the conditions to make sure that the entry locations are related, and the exit locations are related. Furthermore, the conditions should make sure that for every path in the implementation there is a corresponding path in the specification (our refinement criterion). These conditions are made precise by the following definition of *well-formed relation*. If the relation $\mathscr{R}$ is *not* well-formed, then the checking algorithm immediately rejects the verification relation $\mathscr{R}$.

Definition 20 (Well-Formed Relation). We define the relation $\mathscr{R}$ to be well formed if the following holds:

1. $(\iota_1, \iota_2, \mathit{true}) \in \mathscr{R}$
2. $\exists \phi \in \Phi.\ (\varepsilon_1, \varepsilon_2, \phi) \in \mathscr{R}$
3. $\forall (gl_1, gl_2, \phi) \in \mathscr{R}, gl_2' \in \mathscr{P}_2, w_2 \in \mathscr{I}_2^*$
   $gl_2 \overset{w_2}{\hookrightarrow}_2 gl_2' \implies \exists gl_1' \in \mathscr{P}_1, \psi \in \Phi, w_1 \in \mathscr{I}_1^*.\ gl_1 \overset{w_1}{\hookrightarrow}_1 gl_1' \wedge (gl_1', gl_2', \psi) \in \mathscr{R}$

The checking algorithm is shown in Fig. 7.5. The CheckRelation procedure takes as input a well-formed relation $\mathscr{R}$, and verifies each entry in the verification relation individually. For each possible entry (line 2), the algorithm iterates through all the next transitions as shown in line 3. In doing this search, infeasible paths are pruned out on line 4.

The IsInfeasible function (lines 9–10), checks using an automated theorem prover (ATP) whether or not it is in fact feasible for the specification to follow trace w_1 and the implementation to follow w_2. The trace combination is infeasible if the strongest

```
1.  function CheckRelation(R)
2.      for each (gl1, gl2, φ) ∈ R do
3.          for each (gl1, gl2, φ) ⇒^(w1,w2) (gl1', gl2', ψ) do
4.              if ¬IsInfeasible(w1, w2, φ) then
5.                  if ¬WellPaired(w1, w2, φ) then
6.                      Error("Traces are not well paired")
7.                  if ATP(φ ⇒ wp(w1, wp(w2, ψ))) ≠ Valid then
8.                      Error("Cannot verify relation entry")

9.  function IsInfeasible(w1 ∈ I1*, w2 ∈ I2*, φ ∈ Φ) : B
10.     return ATP(¬sp(w1, sp(w2, φ))) = Valid
```

Fig. 7.5 Algorithm for checking a simulation relation

postconditions (computed using the sp function) with respect to w_2 and then with respect to w_1 is inconsistent. This takes care of pruning within a single program, but also across the specification and the implementation. For a given formula ϕ and trace w, the strongest postcondition $\mathsf{sp}(w, \phi)$ is the strongest formula ψ such that if the instructions in the trace w are executed in sequence starting in a program state satisfying ϕ, then ψ will hold in the resulting program state. The sp computation itself is standard, except for the handling of communication events, which are simulated as assignments. When computing sp with respect to one sequence, the algorithm treats all variables from the other sequence as constants. As a result, the order in which it processes the two sequences does not matter.

Once the algorithm has identified that the two sequences w_1 and w_2 may be a feasible combination, it then checks that they are well paired using the WellPaired predicate (lines 5–6). The WellPaired predicate (not shown here) checks that there is at most one visible instruction in the sequences w_1 and w_2. It also checks that the visible instructions are equivalent.

Next, for well paired sequences, it checks that if we start at states that satisfy the predicate ϕ and execute w_1 in π_1 and w_2 in π_2 then the resulting states should satisfy the predicate ψ. To do this it first computes the weakest precondition of ψ with respect to the two traces, and then asks an ATP to show ϕ implies the weakest precondition (line 7). For a given formula ψ and trace w, the weakest precondition $\mathsf{wp}(w, \psi)$ is the weakest formula ϕ such that executing the trace w in a state satisfying ϕ leads to a state satisfying ψ. Here again, the wp computation itself is standard, and the order in which it processes the two traces does not matter. Recall from Chap. 4 how weakest precondition and Hoare logic can be used for verification using ATP. If at the end of the algorithm there is no error then the verification relation is indeed a simulation relation.

There are additional optimizations which are performed in the TV-HLS algorithm. These optimizations are not explicitly shown in the algorithm from Fig. 7.5. However, these are important in improving the efficiency of the refinement checking process. For example, when exploring the control state (both in the checking and in the inference algorithm), the algorithm performs a simple partial order reduction [167] that is very effective in reducing the size of the control state space: if

two communication events happen in parallel, but they do not depend on each other, and they do not involve externally visible channels, then it only considers one ordering of the two events.

7.6.2 Inference Algorithm

Since there can be many possible paths through a loop, writing simulation relations by hand can be tedious, time consuming and error prone. We therefore want methods for generating (inferring) these relations automatically, not just checking them. This will in turn also be able to automate the validation process entirely. Nevertheless, the checking algorithm is useful by itself, in case the inference algorithm is not capable of finding an appropriate relation, and a human wants to provide the relation by hand.

Here again to focus our attention on only those locations for which the approach infers the relation entries, we define two sets of locations $\mathscr{Q}_1$ and $\mathscr{Q}_2$ for the transition diagrams π_1 and π_2 respectively. These include all locations corresponding to visible instructions and also all locations before branch statements. In this section, we do notation abuse by re-using the shorthand $gl_1 \overset{w_1}{\longhookrightarrow}_1 gl_1'$ for $gl_1 \overset{(w_1,\mathscr{Q}_1)}{\longhookrightarrow}_{\pi_1} gl_1'$, and $gl_2 \overset{w_2}{\longhookrightarrow}_2 gl_2'$ for $gl_2 \overset{(w_2,\mathscr{Q}_2)}{\longhookrightarrow}_{\pi_2} gl_2'$.

We now define a parallel transition relation $\leftarrow\!\!\!\rightarrow\!\!\!\rightarrow$ that essentially traverses the two transition diagrams (specification and implementation) in synchrony, while focusing on only those locations for which this approach infers the relation entries.

Definition 21 (Parallel Transition). Given $(gl_1, gl_2) \in \mathscr{Q}_1 \times \mathscr{Q}_2$, $(gl_1', gl_2') \in \mathscr{Q}_1 \times \mathscr{Q}_2$, $w_1 \in \mathscr{I}_1^*$ and $w_2 \in \mathscr{I}_2^*$, we define $\leftarrow\!\!\!\rightarrow\!\!\!\rightarrow$ as follows:

$$(gl_1, gl_2) \overset{(w_1,w_2)}{\leftarrow\!\!\!\rightarrow\!\!\!\rightarrow} (gl_1', gl_2') \quad \textit{iff}$$
$$gl_1 \overset{w_1}{\longhookrightarrow}_1 gl_1' \wedge gl_2 \overset{w_2}{\longhookrightarrow}_2 gl_2' \wedge \mathsf{Rel}(w_1, w_2, gl_1, gl_2) \wedge \mathsf{WellMatched}(w_1, w_2).$$

We now describe the two predicates Rel and WellMatched used in the above definition. The predicate $\mathsf{Rel} : \mathscr{I}^* \times \mathscr{I}^* \times \mathscr{Q}_1 \times \mathscr{Q}_2 \to \mathscr{B}$ is a heuristic that tries to estimate when a path in the specification is related to a path in the implementation. Consider for example the branch in the specification of Fig. 7.1 and the corresponding branch in the implementation. For any two such branches, the Rel function uses heuristics to guess a correlation between them: either they always go in the same direction, or they always go in opposite direction. Using these correlations, $\mathsf{Rel}(w_1, w_2, gl_1, gl_2)$ returns true only if the paths w_1 and w_2 follow branches in a correlated way.

The implementation of TV-HLS described in Sect. 7.8 implements the predicate Rel in such a way that it correlates branches in two ways. First, using the results of a strongest postcondition pre-pass over the specification and the implementation, Rel tries to use a theorem prover to prove that certain branches are correlated. If the

theorem prover is not able to determine a correlation, Rel uses the structure of the branch predicate and the structure of the instructions on each side of the branch to guess a correlation. For instance, in the example of Fig. 7.1, since the strongest postcondition involves the input parameter p, the theorem prover is unable to reason about it. However, because the structure of the branch predicate is not changed in the implementation, Rel can conclude that the two branches go in the same direction.

The other predicate $\mathsf{WellMatched} : \mathscr{I}^* \times \mathscr{I}^* \to \mathscr{B}$ prunes some of these pair of transitions if the sequence of instructions are not similar (well-matched). We define two sequences (w_1, w_2) of instructions are well-matched if neither of them contain a visible instruction or they each contains a single visible instruction of the same type; i.e. they are both input or both output on the same channel. Although Rel and WellMatched make guesses about the correlation of branches and visible instructions, the later constraint solving phase of the inference algorithm makes sure that these guesses are correct.

We now define the relation $\mathscr{R} \subseteq \mathscr{Q}_1 \times \mathscr{Q}_2$ of location pairs that will form the entries of our simulation relation.

Definition 22 (Pairs of Interest). The relation $\mathscr{R} \subseteq \mathscr{Q}_1 \times \mathscr{Q}_2$ is defined to be the minimal relation that satisfies the following three properties:

$$\mathscr{R}(\iota_1, \iota_2)$$
$$\mathscr{R}(\varepsilon_1, \varepsilon_2)$$
$$\mathscr{R}(gl_1, gl_2) \wedge (gl_1, gl_2) \overset{(w_1,w_2)}{\hookrightarrow\!\!\!\!\rightarrow} (gl_1', gl_2') \implies \mathscr{R}(gl_1', gl_2')$$

The set $\mathscr{R}$ defined above can easily be computed by starting with the empty set, and applying the above three rules exhaustively.

For this approach to successfully infer a simulation relation, the computed set $\mathscr{R}$ must cover every path in the implementation (our refinement criterion). This condition is made precise by the following definition of *well-formed pairs of interest*. The well-formed condition here is similar to the one described in Definition 20, except that now it is for the pairs of interest relation $\mathscr{R}$. We do not need the first two conditions here as they are satisfied by construction. Here again if the computed set $\mathscr{R}$ is *not* well-formed, then the validation approach immediately rejects the translation from specification to implementation.

Definition 23 (Well-Formed Pairs of Interest). We define the pairs of interest relation $\mathscr{R}$ to be well formed if the following holds:

$$\forall (gl_1, gl_2) \in \mathscr{R}, gl_2' \in \mathscr{Q}_2, w_2 \in \mathscr{I}_2^*$$
$$gl_2 \overset{w_2}{\hookrightarrow}_2 gl_2' \implies \exists gl_1' \in \mathscr{Q}_1, w_1 \in \mathscr{I}_1^* .\ gl_1 \overset{w_1}{\hookrightarrow}_1 gl_1' \wedge (gl_1', gl_2') \in \mathscr{R}.$$

We now describe the inference algorithm in terms of constraint solving. In particular, for each $(gl_1, gl_2) \in \mathscr{R}$ we define a constraint variable $\psi_{(gl_1,gl_2)}$ representing the predicate that will be computed for the simulation entry (gl_1, gl_2). We denote

by Ψ the set of all such constraint variables. Using these constraint variables, the final simulation relation will have the form:

$$\{(gl_1, gl_2, \psi_{(gl_1,gl_2)}) \mid \mathscr{R}(gl_1, gl_2)\}$$

To compute the predicates that the constraint variables $\psi_{(gl_1,gl_2)}$ stand for, we define a set of constraints on these variables, which are then used during the constraint solving phase. The constraints are defined as follows.

Definition 24 (Constraint). A constraint is a formula of the form $\psi_1 \Rightarrow f(\psi_2)$, where $\psi_1, \psi_2 \in \Psi$, and f is a boolean function.

Definition 25 (Set of Constraints). The set $\mathscr{C}$ of constraints is defined by:

$$\begin{aligned}&\text{For each } (gl_1, gl_2) \overset{(w_1,w_2)}{\underset{c}{\longrightarrow}} (gl_1', gl_2'):\\ &\quad \left[\psi_{(gl_1,gl_2)} \Rightarrow \mathsf{CreateSeed}(w_1, w_2)\right] \in \mathscr{C}\\ &\quad \left[\psi_{(gl_1,gl_2)} \Rightarrow \mathsf{wp}(w_1, \mathsf{wp}(w_2, \psi_{(gl_1',gl_2')}))\right] \in \mathscr{C}\end{aligned}$$

The CreateSeed function above creates for each pair of instruction sequences (w_1, w_2) a formula, which does not refer to any constraint variables. There are two cases either they are well-matched or they are branches (Definition 21). If the instructions are well-matched, then the formula returned by CreateSeed states that the visible instructions in them are equivalent as defined in Sect. 7.4; and if they are branches, then the formula states the two branches are correlated (either they both go in the same direction, or in opposite directions).

The other function wp used above computes the weakest precondition with respect to w_2 and then with respect to w_1. The weakest precondition computation is the same as the one described in Sect. 7.6.1.

Having created a set of constraints $\mathscr{C}$, the validation approach now solves these constraints using the algorithm in Fig. 7.6. The algorithm starts by setting each constraint variable to *true* (line 13) and initializing a *worklist* with the set of all

```
11. function SolveConstraints(𝒞)
12.     for each (gl1, gl2) ∈ ℛ do
13.         ψ(gl1,gl2) := true
14.     let worklist := 𝒞
15.     while worklist not empty do
16.         let [ψ(gl1,gl2) ⇒ f(ψ(gl1',gl2'))] := worklist.Remove
17.         if ATP(ψ(gl1,gl2) ⇒ f(ψ(gl1',gl2'))) ≠ Valid then
18.             if (gl1, gl2) = (t1, t2) then
19.                 Error("Start Condition not strong enough")
20.             ψ(gl1,gl2) := ψ(gl1,gl2) ∧ f(ψ(gl1',gl2'))
21.             worklist := worklist ∪ {c ∈ 𝒞 | ∃ψ, g . c = [ψ ⇒ g(ψ(gl1,gl2))]}
```

Fig. 7.6 Algorithm for solving constraints

constraints (line 14). Next, while the *worklist* is not empty, it removes a constraint from the worklist (line 16), and checks using a theorem prover if it is *Valid* (line 17). If not, then it appropriately strengthens the left-hand-side variable of the constraint (line 20) and adds to the worklist all the constraints that have this variable in the right-hand side (line 21). Finally, if at the end of the algorithm there is no error then it has inferred a simulation relation.

7.7 Equivalence of Transition Diagrams

Apart from checking refinements, we also sometimes want to check equivalence between two transition diagrams. In this section, we describe how we can generalize the TV-HLS algorithms to check for equivalence. We first define two transition diagrams to be equivalent as follows:

Definition 26 (Equivalence of Transition Diagrams). Two transition diagrams π_1 and π_2 are said to be equivalent iff $\pi_1 \sqsubseteq \pi_2$ and $\pi_2 \sqsubseteq \pi_1$.

We then define a *bisimulation relation* using the definition of simulation relation.

Definition 27 (Bisimulation Relation). A verification relation R is a bisimulation relation for π_1, π_2 iff R is a simulation relation for π_1, π_2 and $R^{-1} = \{(gl_2, gl_1, \phi) \mid R(gl_1, gl_2, \phi)\}$ is a simulation relation for π_2, π_1.

The following theorem connects the above relation with our definition of equivalence for transition diagrams.

Theorem 7.2 (Equivalence). *If there exists a bisimulation relation for π_1, π_2, then π_1 and π_2 are equivalent.*

Like simulation relation, a bisimulation relation is a witness that two transition diagrams are equivalent. Therefore, to check if the specification is equivalent to the implementation the algorithms now have to show that there exists a bisimulation relation between them. Both the checking and inference algorithms can be easily extended for this purpose with just slight modifications.

For the checking algorithm, we only have to strengthen the definition of well-formed relation (Definition 20) with this fourth condition.

$$\forall (gl_1, gl_2, \phi) \in \mathscr{R}, gl'_1 \in \mathscr{P}_1, w_1 \in \mathscr{I}_1^* \\ gl_1 \overset{w_1}{\hookrightarrow}_1 gl'_1 \implies \exists gl'_2 \in \mathscr{P}_2, \psi \in \Phi, w_2 \in \mathscr{I}_2^* .\ gl_2 \overset{w_2}{\hookrightarrow}_2 gl'_2 \wedge (gl'_1, gl'_2, \psi) \in \mathscr{R}.$$

Similarly, for the inference algorithm, we only have to strengthen the definition of well-formed pairs of interest (Definition 23) with this condition.

$$\forall (gl_1, gl_2) \in \mathscr{R}, gl'_1 \in \mathscr{Q}_1, w_1 \in \mathscr{I}_1^* \\ gl_1 \overset{w_1}{\hookrightarrow}_1 gl'_1 \implies \exists gl'_2 \in \mathscr{Q}_2, w_2 \in \mathscr{I}_2^* .\ gl_2 \overset{w_2}{\hookrightarrow}_2 gl'_2 \wedge (gl'_1, gl'_2) \in \mathscr{R}.$$

7.8 Experiments and Results

In this section, we discuss the implementation of the TV-HLS algorithm in a prototype tool called Surya that uses the Simplify ATP [44]. We have used Surya to validate programs in two different settings. First, we used it to *automatically* check refinements of various concurrent programs, written in CSP. Next, we used Surya to validate the result of the high-level synthesis framework Spark.

7.8.1 Automatic Refinement Checking of CSP Programs

For refinements, the goal is to infer a simulation relation. The visible events in this case are input and output on visible channels. We wrote a variety of CSP refinements, and checked them for correctness automatically. The refinements that we checked are shown in Table 7.3, along with the number of parallel threads, the number of instructions, the number of simulation relation entries, the number of calls to the theorem prover, and the time required to automatically check them. Apart from the theorem prover calls discussed in this chapter, Surya also use the theorem prover to reduce the size of the formulas described in the algorithms. The number of calls to the theorem prover mentioned in Table 7.3 include *all* these calls.

The first 11 refinements were inspired from examples that come with the FDR tool [137]. FDR is a state-of-the-art tool to check CSP refinements. The approach that FDR uses for checking refinement is to perform an exhaustive search of the implementation-specification combined state space. Although in its pure form this approach only works for finite state systems, there is one way in which it can be

Table 7.3 Timings for the refinement examples checked using Surya

Description	T	I	SRE	TP	Time mins
1. Simple buffer	7	29	3	14	00.00
2. Simple vending machine	2	20	9	32	00.00
3. Cyclic scheduler	6	65	157	11082	00.49
4. Student tracking system	3	63	12	115	00.01
5. 1 comm link	11	54	3	14	00.01
6. 2 parallel comm links	18	105	37	486	00.04
7. 3 parallel comm links	25	144	45	1861	00.21
8. 4 parallel comm links	32	186	124	7228	01.11
9. 5 parallel comm links	39	228	315	24348	02.32
10. 6 parallel comm links	46	270	762	74991	08.29
11. 7 parallel comm links	53	312	1785	217131	37.28
12. SystemC refinement	8	39	3	14	00.00
13. EP2 system	3	173	208	5648	01.47

T: Number of parallel Threads; *I*: Number of Instructions; *SRE*: Number of Simulation Relation Entries; *TP*: Number of Theorem Prover calls

extended to infinite systems. In particular, if an infinite state system treats all the data it manipulates as black boxes, then one can use skolemization and simply check the refinement for one possible value. Such systems are called *data-independent*, and FDR can check the refinement of these systems using the skolemization trick, even if they are infinite [180].

Unfortunately, for high-level programs, there are many refinement examples that are not finite, because they do not specify the bit-width of integers (in particular, we want the refinement to work for any integer size). Nor are the processes data-independent, as they manipulate the data during the refinement process. In particular, our example from Fig. 7.1 is neither finite nor data-independent, since both the specification and the implementation are "inspecting" the variables when manipulating them. Indeed, it would not at all be safe to simply check the refinement for any one particular value, since, if we happen to pick 0 for p, and the implementation erroneously sets the output to 4 times the input (instead of 2 times), we would not detect the error. FDR cannot check the refinement of such infinite data-dependent CSP systems, except by restricting them to a finite subset first, for example by picking a bit-width for the integers, and then doing an exhaustive search. Not only would such an approach not prove the refinement for any bit-width, but furthermore, despite many techniques that have been developed for checking larger and larger finite state spaces [23, 35, 167, 181], the state space can still grow to a point where automation is impossible. For example, we tried checking the refinement example '2 parallel comm links' from Table 7.3 in FDR using 32-bit integers as values, and the tool had to be stopped because it ran out of memory after several hours (Surya, in contrast, is able to check this example for any sized integers, not just 32-bit integers, in about 4 s).

We implemented generalizations of these 11 FDR examples to make them data-dependent and operate over infinite domains. Surya was able to check these generalized refinements that FDR would not be able to check.

The 12th refinement in the list is a hardware refinement example taken from a SystemC book [78]. This example models the refinement of an abstract FIFO communication channel to an implementation that uses a standard FIFO hardware channel, along with logic to make the hardware channel correctly implement the abstract communication channel.

In the 13th refinement from Table 7.3, we checked part of the EP2 system [1], which is a new industrial standard for electronic payments. We followed the implementation of the data part of the EP2 system found in the paper on CSP-Prover [96]. The EP2 system states how various components, including service centers, credit card holders, and terminals, interact.

In all of the above examples, we used Surya to check for trace subset refinement (see Definition 15). Since trace subset refinement preserves safety properties, we can also conclude that the implementation has all the safety properties of the specification.

7.8.2 SPARK: High-Level Synthesis Framework

Spark is a C-to-VHDL parallelizing high-level synthesis framework that employs a set of compiler, parallelizing compiler, and synthesis transformations to improve the quality of high-level synthesis results. Figure 7.7 shows an overview of the Spark HLS framework. What makes Spark an excellent candidate for experimenting is not only the easy availability of source code but also the fact that it uses a single intermediate representation, called Hierarchical Task Graphs (HTGs) [69]. Spark starts with a behavioral description in ANSI-C as input – currently with the restrictions of no pointers, no recursion, and no irregular control-flow jumps. It converts the input program into its own IR, and then applies a set of code transformations, including loop unrolling, loop fusion, common sub-expression elimination, copy propagation, dead code elimination, loop-invariant code motion, induction variable analysis, and operation strength reduction. Following these transformations, Spark performs a scheduling phase using resource allocation information provided by the user. This scheduling phase also performs a variety of transformations, including speculative code motion, dynamic renaming of variables, dynamic branch balancing, chaining of operations across conditional blocks, and scheduling on multi-cycle operations. The scheduling phase is followed by a resource binding phase and finally by a back-end code generation pass that produces RTL VHDL.

Surya in this setting takes as input the IR program that is produced by the parser, and the IR program right before resource binding (see Fig. 7.7), and verifies that the

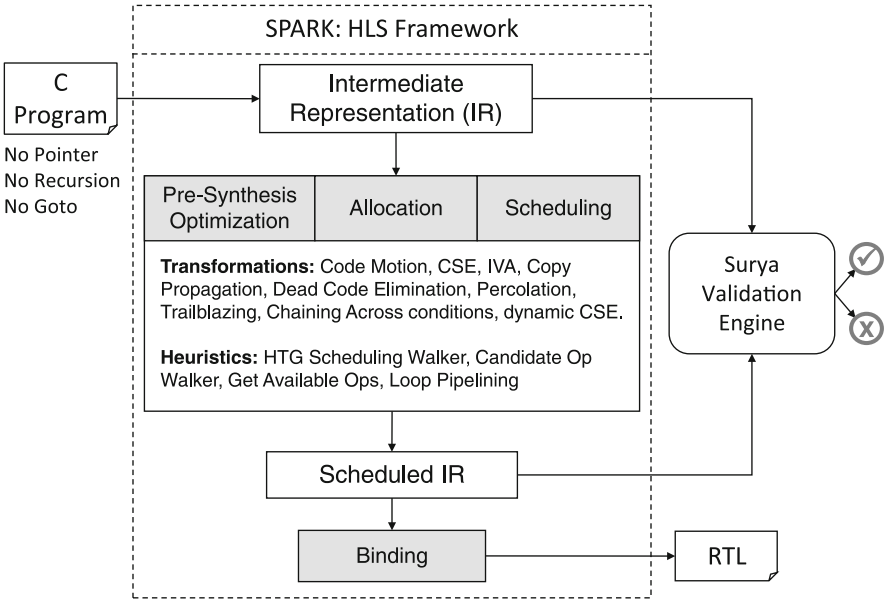


Fig. 7.7 Overview of the Spark framework along with Surya

Table 7.4 Spark benchmarks successfully checked

Benchmarks	No. of bisimulation relation entries	No. of calls to theorem prover	Time secs
1. Incrementer	6	9	00.52
2. Integer-sum	6	20	00.81
3. Array-sum	6	24	00.83
4. Diffeq	7	41	01.68
5. Waka	11	79	02.61
6. Pipelining	12	75	02.30
7. Rotor	14	71	02.57
8. Parker	26	281	05.23
9. S2r	27	570	26.73
10. Findmin8	29	787	14.86

two are equivalent by showing that there exist a bisimulation relation. It therefore validates the entire HLS process of Spark, except for parsing, resource binding and code generation. Note that Surya is around 7,500 lines of C++ code, whereas Spark's implementation excluding the parser consists of over 125,000 lines of C++ code. Thus, with around 15 times less effort compared to Spark's implementation we can build a tool that validates its synthesis process.

We tested Surya on 12 benchmarks obtained from Spark's test suite. Of these benchmarks, 10 passed and 2 failed. The benchmarks that were successfully checked are shown in Table 7.4, along with the number of bisimulation relation entries, the number of calls to the theorem prover, and the time required to check each benchmark. All these benchmarks are single threaded. For the ones that passed, Surya was able to quickly find the bisimulation relation, taking on average around 6 seconds per procedure, and a maximum of 27 s for the largest procedure (80 lines of code). Furthermore, the computed bisimulation relations were small, ranging in size from 6 to 29 entries, with an average of about 14. To infer these bisimulation relations, Surya made an average of 189 calls to the theorem prover per procedure (with a minimum of 9 and a maximum of 797). The TV-HLS approach is compositional since it works on one procedure at a time, and the above results show that this approach can handle realistically size procedures.

As mentioned previously, two benchmarks failed the validation test. Upon further analysis each of them lead us to discover previously unknown bugs in Spark. One bug occurs in a particular corner case of copy propagation for array elements. The other bug is in the implementation of the code motion algorithm in the scheduler. The fact that Surya found two previously unknown bugs in a widely-used HLS framework emphasizes the usefulness and bug-isolating capabilities of the TV-HLS approach.

In general, the TV-HLS approach work well when the transformations that are performed preserve most of the program's control flow structure. Such transformations are called *structure-preserving transformations* [212]. The only non structure-preserving transformation that Spark performs is loop unrolling, but in the above examples this transformation did not trigger.

7.9 Further Reading

The approach described in this chapter is related to translation validation [73, 116, 160, 170, 178, 212, 213], relational approaches to reasoning about programs [12, 25, 55, 99, 121], CSP refinement checking [49, 96, 137, 195], and HLS verification [6, 54, 111, 117, 158]. We now discuss each area in more detail and point the readers to few interesting readings.

Translation Validation: The inference algorithm of the TV-HLS approach was inspired by Necula's translation validation algorithm for inferring simulation relations that prove equivalence of sequential programs [160]. Necula's approach collects a set of constraints in a forward scan of the two programs, and then solves these constraints using a specialized solver and expression simplifier. However, unlike Necula's approach, the TV-HLS algorithm must take into account statements running in parallel, since hardware is inherently concurrent and one of the main tasks that HLS tools perform is to schedule statements for parallel execution.

Relational Approaches: Relational approaches are a common tool for reasoning about programs, and they have been used for a variety of verification tasks, including model checking [25, 55], translation validation [160, 170], CSP refinement checking [99] and reasoning about optimizations once and for all [12, 121].

CSP Refinement Checking: There has been a long line of work on reasoning about refinement of CSP programs. One of the most used tool for CSP refinement checking is FDR [137], which uses various intelligent techniques to exhaustively explore the state space.

Various interactive theorem provers have been extended with the ability to reason about CSP programs. As one example, Dutertre and Schneider [49] reasoned about communication protocols expressed as CSP programs using the PVS theorem prover [163]. As another example, Tej and Wolff [195] have used the Isabelle theorem prover [165] to encode the semantics of CSP programs. Isabelle has also been used by Isobe and Roggenbach to develop a tool called CSP-Prover [96] for proving properties of CSP programs. All these uses of interactive theorem provers follow a common high-level approach: the semantics of CSP is usually encoded using the native logic of the interactive theorem prover, and then a set of tactics are defined for reasoning about this semantics. Users of the system can then write proof scripts that use these tactics, along with built-in tactics from the theorem prover, to prove properties about particular CSP programs. Although these interactive theorem proving approaches have extensive formal underpinnings, they all require some amount of human intervention.

The TV-HLS approach checks one particular property of CSP programs, namely trace subset refinement. This kind of refinement only preserves safety properties. Algorithms and tools exist for checking other kinds of refinements. For example, CSP-Prover [96] can check refinements using a failures semantics that preserves liveness properties and deadlock freedom (in addition to safety properties). The FDR [137] tool can also check refinements in a failures/divergence model, which can also preserve livelock freedom.

HLS Verification: Techniques like correctness-preserving transformations [54], formal assertions [158], symbolic simulation [7], and relational approaches for functional equivalence of FSMDs [106, 111] have been used to validate the scheduling step of HLS. Another interesting work that is complementary to the approach presented in this chapter uses model checking to validate the binding step of HLS [6], which is the only internal step of Spark that Surya does not validate.

7.10 Summary

In this chapter, we have presented an automated algorithm for translation validation of the HLS process (TV-HLS). We also discussed the implementation of the TV-HLS approach in a validation system called Surya and demonstrated its effectiveness through its application in two different settings. The experiments with Spark showed that with only a fraction of the development cost of Spark, this algorithm can validate the translations performed by Spark, and it also uncovered bugs that eluded long-term use. The work presented in this chapter also solves the critical problem of handling more sophisticated datatypes than finite bit-width enumeration types associated with typical RTL code and thus enables stepwise refinement of system designs expressed using high-level languages.

Acknowledgments This chapter in part, has been published as:

"Automated Refinement Checking of Concurrent Systems", by Sudipta Kundu, Sorin Lerner and Rajesh Gupta in *ICCAD 07: Proceedings of the 2007 IEEE/ACM International Conference on Computer Aided Design* [116].

"Validating High-Level Synthesis", by Sudipta Kundu, Sorin Lerner and Rajesh Gupta in *CAV 08: Proceedings of the 20th international conference on Computer Aided Verification* [117].

"Translation Validation of High-Level Synthesis", by Sudipta Kundu, Sorin Lerner and Rajesh Gupta in *TCAD 10: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* [119].

Chapter 8
Parameterized Program Equivalence Checking

Zachary Tatlock[1]

In the previous chapter we discussed an approach to verify if two programs are equivalent, thereby proving that the translation (performed by an HLS tool) from high-level design to low-level design is correct. In this chapter, we discuss another approach that guarantees correctness of the translation from high-level design to low-level design, by proving the HLS tool itself correct. Unlike translation validation, this approach proves the correctness of an HLS tool *once and for all*, before it is ever run. In the following sections we describe in details an approach called *Parametrized Equivalence Checking* [120] (PEC) that generalizes the translation validation approach discussed in the previous chapter to automatically establish the correctness of semantics preserving transformations once and for all.

8.1 Overview of Synthesis Tool Verification

HLS tools are a fundamental component of the tool chains hardware designers rely on for system-level designs. As a result, correctness of HLS tool is crucially important. A bug in a HLS tool can in turn introduce errors in each generated RTL. Furthermore, HLS tool bugs can invalidate strong guarantees that were established on the original source program (discussed in Chaps. 5 and 6). Unfortunately, as discussed throughout this book building reliable compilers is difficult, error-prone, and requires significant manual effort.

One of the most error prone parts of a HLS tool is its optimization phase. Many optimizations require an intricate sequence of complex transformations. Often these transformations interact in unexpected ways, leading to a combinatorial explosion in the number of cases that must be considered to ensure that the optimization phase is correct.

[1] University of California San Diego, La Jolla, CA 92093 USA. (ztatlock@cs.ucsd.edu)

S. Kundu et al., *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, DOI 10.1007/978-1-4419-9359-5_8,

8.1.1 Once-And-For-All Vs. Translation Validation

Existing techniques for providing correctness guarantees for optimizations can be divided into two categories: *once and for all* and *translation validation*.

The primary advantage of once-and-for-all techniques is that they provide a very strong guarantee: optimizations are known to be correct when the tool is built, before they are run even once. In contrast, translation validation provides a weaker correctness guarantee. This is because translation validation guarantees that only a particular run of the optimization is correct. Tools that include translation validation may still contain bugs and it is unclear what a designer should do when the translation validator flags a particular translation to be incorrect.

On the other hand, translation validation techniques have a clear advantage over once-and-for-all techniques in terms of automation. Most of the techniques that provide once-and-for-all guarantees require user interaction. Those that are fully automated, for example Cobalt [131] and Rhodium [133] approaches for compilers, work by having programmers implement optimizations in a domain-specific language using flow functions and single-statement rewrite rules. Unfortunately, the set of optimizations that these techniques can prove correct has lagged behind translation validation. In particular, translation validation can already handle complex loop optimizations like skewing, splitting and interchange, which have thus far eluded automated once-and-for-all approaches. A common intuition is that once-and-for-all proofs are harder to achieve because they must show that any application of the optimization is correct, as opposed to a single instance.

8.2 Overview of the PEC Approach

In this chapter, we present a technique for proving optimizations correct called Parameterized Equivalence Checking [120] (PEC) that bridges the gap between translation validation and once-and-for-all techniques. PEC generalizes translation validation to handle parameterized programs, which are partially specified programs that can represent multiple concrete programs. For example, a parameterized program may contain a section of code whose only known property is that it does not define or use a particular variable.

The key insight of PEC is that existing translation validation techniques can be adapted to work in the broader setting of parameterized programs. This allows translation validation techniques, which have traditionally been used to prove *concrete* programs equivalent, to prove *parameterized* programs equivalent. Most importantly, because optimizations can be expressed as nothing more than parameterized transformation rules, using before and after parameterized code patterns, PEC can prove once and for all that such optimizations preserve semantics.

At the core of the PEC system is a domain-specific programming language that was especially designed for writing optimizations. Although the original motivation of PEC was *compiler* optimizations, many HLS optimizations bear a huge amount

of similarity to compiler optimizations, and so the PEC language, and the approach in general, can be fruitfully applied to the context of HLS.

The PEC language for writing optimizations is much more expressive than previous such optimization languages, like Cobalt [131] and Rhodium [133]: whereas Cobalt and Rhodium only supported local rewrites of a single statement to another, the PEC language supports many-to-many rewrite rules. Such rules are able to replace an entire set of statements, even entire loops and branches, with a completely different set of statements. Using these rules, one can express many more optimizations than in Cobalt and Rhodium.

Once optimizations are expressed in this domain-specific language, the PEC system can automatically check them for correctness using Parameterized Equivalence Checking. In particular, an automated tool parses the optimizations, and processes them using the PEC approach to generate several obligations for an automated theorem prover. The PEC approach guarantees that, if the theorem prover can discharge these obligations, then the transformations will be semantics preserving. It is important to realize that, unlike in translation validation, these obligations are generated and verified *before* the HLS tool even runs once, thus providing a *once and for all* guarantee.

8.3 Illustrative Example

In the previous chapter we discussed the validation of the Spark [84] HLS tool. In Spark it was shown that compiler optimizations and various other source-to-source transformations can be extremely useful for generating highly parallel designs. In this section we illustrate the main ideas of the PEC approach through one such compiler optimizations: loop pipelining. Loop pipelining can break dependencies inside a loop body without increasing the code size of the loop body, and thus provides more flexibility during the scheduling phase of HLS.

As an example, consider the code in Fig. 8.1a. The statements in the original loop cannot be scheduled together (to be executed in parallel) as there is a dependency between them – the instruction `b[i]+=a[i]` must wait until the instruction `a[i]+=1` finishes. Figure 8.1b shows the result of applying loop pipelining on this loop. The statements in this transformed loop does not depend on each other

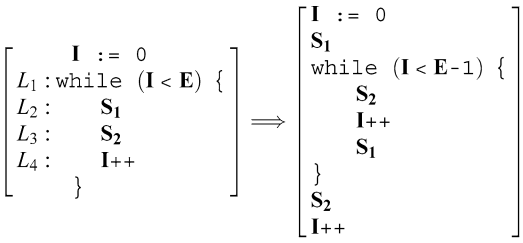
a

```
i := 0
while (i < n) {
   a[i] += 1;
   b[i] += a[i];
   i++;
}
```

b

```
i := 0
a[i] += 1;
while (i < n - 1) {
   b[i] += a[i];
   i++;
   a[i] += 1;
}
b[i] += a[i];
i++;
```

Fig. 8.1 Loop pipelining: (**a**) original code, and (**b**) optimized code



where
$DoesNotModify(\mathbf{S_1}, \mathbf{I})@L_2 \wedge DoesNotModify(\mathbf{S_2}, \mathbf{I})@L_3 \wedge$
$StrictlyPositive(\mathbf{E})@L_1 \wedge DoesNotModify(\mathbf{S_1}, \mathbf{E})@L_2 \wedge$
$DoesNotModify(\mathbf{S_2}, \mathbf{E})@L_3 \wedge DoesNotModify(\mathbf{I}{+}{+}, \mathbf{E})@L_4$

Fig. 8.2 Loop pipelining transformation

and can be scheduled together. In particular, the instruction `b[i]+=a[i]` can be scheduled with the instructions `i++;a[i]+=1`. However, to make this transformation correct, one has to add a prologue at the beginning of the transformed loop in order to setup the pipelining effect. There is also an epilogue after the loop to execute the remaining instructions.

8.3.1 Expressing Loop Pipelining

Loop pipelining can be implemented in the PEC language as shown in Fig. 8.2. The transformation simply moves some instructions (namely $\mathbf{S_1}$) from the current iteration to the next iteration. Optimizations in the PEC language are written as parameterized rewrite rules with side conditions: $P_1 \Longrightarrow P_2$ **where** ϕ, where P_1 and P_2 are parameterized programs, and ϕ is a side condition that states when the rewrite rule can safely be fired. An optimization $P_1 \Longrightarrow P_2$ **where** ϕ states that when a concrete program is found that matches the parameterized program P_1, it should be transformed to P_2 if the side condition ϕ holds.

8.3.2 Parameterized Programs

A parameterized program is a partially specified program that can represent multiple concrete programs. For example, in the original and transformed programs from Fig. 8.2, $\mathbf{S_1}$ ranges over concrete statements (including branches, loops, and sequences of statements) that are single-entry-single-exit code; $\mathbf{I}$ ranges over concrete program variables; and $\mathbf{E}$ ranges over concrete expressions. Because variables like $\mathbf{S_1}$, $\mathbf{I}$ and $\mathbf{E}$ range over the syntax of concrete programs, such variables are called *meta-variables*. To simplify exposition, rather than provide explicit types for all meta-variables, we instead use the following naming conventions: meta-variables starting with $\mathbf{S}$ range over statements, meta-variables starting with $\mathbf{E}$ range over expressions, and meta-variables starting with $\mathbf{I}$ range over variables.

Fig. 8.3 Meanings of some facts used in the PEC system

fact *StrictlyPositive*(**E**)
has meaning $eval(\sigma, \mathbf{E}) > 0$

fact *DoesNotModify*(**S**, **E**)
has meaning $eval(\sigma, \mathbf{E}) = eval(step(\sigma, \mathbf{S}), \mathbf{E})$

8.3.3 Side Conditions

The side conditions are boolean combinations of facts that must hold at certain points in the original program. For example the side condition *DoesNotModify*($\mathbf{S_1}, \mathbf{I}$) @L_2 in Fig. 8.2 states that at location L_2 in the original program, $\mathbf{S_1}$ should not modify **I**. In general, side conditions are first-order logic formulas with facts like *DoesNotModify*($\mathbf{S_1}, \mathbf{I}$)@L_1 as atomic predicates.

Each fact used in the side condition must have a semantic meaning, which is a predicate over program states. Figure 8.3 gives the semantic meanings for the two primary facts used in the PEC system. In general, meanings can be first-order logic formulas with a few special function symbols: (1) σ is a term that represents the program state at the point where the fact holds. (2) *eval* evaluates an expression in a program state and returns its value; (3) *step* executes a statement in a program state and returns the resulting program state.

The semantic meanings are used by the PEC algorithm to determine the semantic information that can be inferred from the side conditions when proving correctness. Although optimization writers must provide these meanings, there is a small number of common facts used across many different optimizations (for example *DoesNotModify*), and since these meanings only need to be written once, the effort in writing meanings is not onerous.

8.3.4 Executing Optimizations

Optimizations written in the PEC language are meant to be executed by an execution engine. When running an optimization $P_1 \Longrightarrow P_2$ **where** ϕ, the execution engine must find concrete program fragments that match P_1. Furthermore, it must perform some program analysis to determine if the facts in the side condition ϕ hold. One option for implementing these program analysis is to use a general purpose programming language. Although this provides the most flexibility, it does not guarantee that the facts in the side condition are computed correctly. Alternatively, if one wants stronger correctness guarantees, the facts in the side conditions can be computed in a way that guarantees that their semantic meanings hold, for example using the Rhodium system of Lerner et al. [133], or using Leroy's Compcert system[134]. Note that it is straightforward to see how the rewrite rule in Fig. 8.2 performs loop pipelining on the example from Fig. 8.1.

8.3.5 Proving Correctness of Loop Pipelining

The goal of PEC is to show that the loop pipelining optimization written in the PEC language is correct, once and for all, before it is even run once. To do this, one must show that the rewrite rule from Fig. 8.2 satisfies the following property: given the side conditions, the original parameterized program and the transformed parameterized program have the same behavior. We next discuss the details of the PEC approach.

8.3.6 Parameterized Equivalence Checking

In Chap. 7 we discussed Translation Validation (TV), a technique that proves concrete, fully specified programs equivalent. In the setting of this chapter, the goal is to prove *parameterized* programs equivalent. Parameterized Equivalence Checking (PEC) achieves this goal by generalizing traditional TV techniques to the setting of parameterized programs.

There are two simple observations that intuitively explain why techniques from translation validation can be generalized to parameterized programs. The first observation is that if a program fragment **S** in the original program executes in a program state σ, and the same program fragment **S** executes in the transformed program in the same state σ, then we know that the two resulting states are equal. This shows that we can reason about state equality even if we don't know what the program fragments are. The second observation is that when proving equivalence, we are usually interested in some key invariants that justify the optimization. The insight is that the semantic meaning of the side condition captures precisely when these key invariants can be propagated throughout statements that are not fully specified. For example, if the correctness of an optimization really depends on **I** not being modified in a region of code, the side condition will allow us to know this fact, and thus reason about **I** across such unknown statements.

8.3.7 Bisimulation Relation

PEC proves equivalence using *bisimulation relation* (Definition 27). Recall that bisimulation relation is defined in terms of the more basic concept of *verification relation*, which is a set of entries of the form (gl_1, gl_2, ϕ), where each entry relates a program point gl_1 in the original program with a corresponding program point gl_2 in the transformed program, and the predicate ϕ indicates how the state of the two programs are related at that point. Moreover, a bisimulation relation is simply a verification relation that satisfies the property that the predicate on any entry in the relation implies the predicate on all entries reachable from it. In the following we denote the original parameterized program as the specification and the transformed parameterized program as the implementation.

The PEC approach generalize the inference algorithm described in the previous chapter. Similar to the TV algorithm, the PEC approach works in two steps. Broadly speaking, the first step generates a set of constraints, and the second step solves these constraints. More specifically, the first step (constraint generation) starts by finding pairs of locations in the specification and the implementation that need to be related in the bisimulation and then for each pair of locations (gl_1, gl_2), it defines a constraint variable $\psi(gl_1, gl_2)$ to represent the state-relating formula that will be computed in the bisimulation relation for that pair. The final part of constraint generation is to actually generate a set of constraints over these variables that must be satisfied in order for the would-be bisimulation relation to in fact be a bisimulation. The second step then proceeds to solve these constraints using an iterative algorithm.

Figure 8.4 shows the concurrent control flow graph (CCFG) for the two parameterized programs of the running example, along with the related locations that the PEC approach generates. The related locations are shown using a dashed line (labelled A – F), and each entry has a predicate associated with it which is shown in

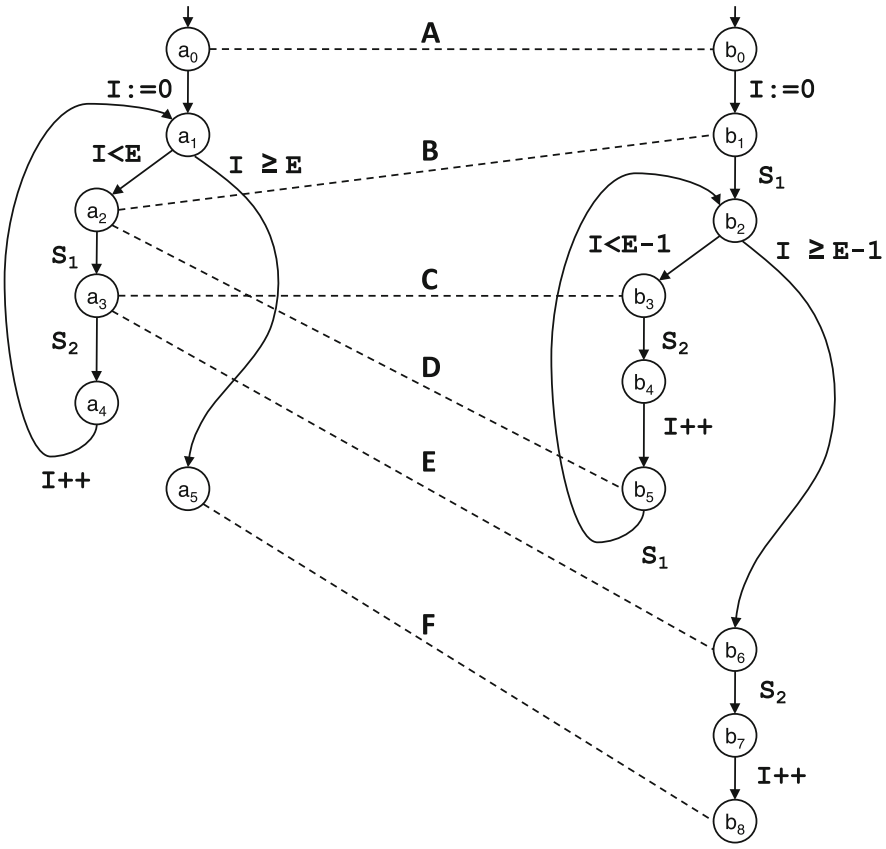


Fig. 8.4 CCFGs of running example with the related locations

Table 8.1 A bisimulation relation for the running example

(gl_1, gl_2)	ϕ
A. (a_0, b_0)	$\sigma_1 = \sigma_2$
B. (a_2, b_1)	$\sigma_1 = \sigma_2 \wedge \mathbf{I_1} < \mathbf{E_1}$
C. (a_3, b_3)	$\sigma_1 = \sigma_2 \wedge \mathbf{I_1} < \mathbf{E_1} \wedge \mathbf{I_2} < \mathbf{E_2} - 1$
D. (a_2, b_5)	$\sigma_1 = \sigma_2 \wedge \mathbf{I_1} < \mathbf{E_1} \wedge \mathbf{I_2} < \mathbf{E_2}$
E. (a_3, b_6)	$\sigma_1 = \sigma_2 \wedge \mathbf{I_1} < \mathbf{E_1} \wedge \mathbf{I_2} \geq \mathbf{E_2} - 1$
F. (a_5, b_8)	$\sigma_1 = \sigma_2$

Table 8.1. These predicates operate over the program states σ_1 and σ_2 of the specification and implementation programs. To make the notation cleaner, we use some shorthand notation. For example, $\mathbf{E}_1$ means $eval(\sigma_1, \mathbf{E})$. Using this notation, the predicate at C states that (1) the two programs states σ_1 and σ_2 are equal, (2) $\mathbf{I} < \mathbf{E}$ holds in σ_1 and (3) $\mathbf{I} < \mathbf{E} - 1$ holds in σ_2. Together the related locations and the predicates form the bisimulation relation for this example.

8.3.8 Generating Constraints

The PEC algorithm first relates the start locations of the two programs and then adds the constraint that the constraint variable at A ($\psi_{(a_0,b_0)}$) should imply the predicate $\sigma_1 = \sigma_2$; this indicates that we can assume the program states are equal at the start of the two code fragments. Similarly, it also adds the constraint $\psi_{(a_5,b_8)} \Rightarrow (\sigma_1 = \sigma_2)$ corresponding to the end locations (F); this constraint indicates that we must establish that the program states are equal after the two programs execute. To generate the locations in between, the PEC algorithm traverses both programs in parallel from the top entry. Each time a statement is reached like $\mathbf{S_1}$, and $\mathbf{S_2}$, the algorithm finds the corresponding location in the other program, and adds a relation entry between the two locations with the corresponding constraint variable implying the predicate $\sigma_1 = \sigma_2$ (since this is the only mechanism we have to preserve equivalence of arbitrary statements). Apart from these, the PEC algorithm also generates constraints such that the constraint variable for each related pair that is under a branch, implies the strongest post condition of the branch condition. These constraints lead to the various predicates relating $\mathbf{I}$ and $\mathbf{E}$ in Table 8.1. This allows entries in the bisimulation relation to encode information about what branch conditions they are under, thereby pruning some pair of paths that are simultaneously unreachable.

Recall from Sect. 7.3.4 that the constraints described above are the one that ensures the predicate at a pair of locations (gl_1, gl_2) imply that any visible instructions about to execute at (gl_1, gl_2) behave the same way. The visible instruction in this case are the parameterized statement variables like $\mathbf{S_1}$, and $\mathbf{S_2}$. The intuition behind choosing these statements as visible instructions is that since very little is known about them, one would predict they should behave in the same way in the two programs.

Apart from the above kind of constraint there is another kind of constraint, which is used to state the relationship between one pair of related locations and other pairs

of related locations. For example, if starting at (gl_1, gl_2) in states satisfying $\psi_{(gl_1,gl_2)}$, the specification and implementation can execute in parallel to reach another related pair of locations (gl'_1, gl'_2), then $\psi_{(gl'_1,gl'_2)}$ must hold in the resulting states. As shown in Sect. 7.6.2 and again here in Sect. 8.6, such constraints can be stated over the constraint variables $\psi_{(gl_1,gl_2)}$ and $\psi_{(gl'_1,gl'_2)}$ using the weakest precondition operator (wp). This second kind of constraint guarantees that the computed bisimulation relation is in fact a bisimulation. In the example from Fig. 8.4, the paths that the PEC algorithm would discover between relation entries are as follows: A to B, B to E, B to C, C to D, D to C, D to E, and E to F.

During this step the PEC algorithm also prunes infeasible paths. For example, when starting at E, it is impossible for the specification to stay in the loop – it must exit to F. The PEC algorithm can determine this from the predicate corresponding to the branch condition at E. In particular, let i and e be the original values of $\mathbf{I}$ and $\mathbf{E}$ at E (in either σ_1 or σ_2 since they are equal). The value e does not change through the loop as stated by the side conditions. If the original program chooses to stay in the loop, the $\texttt{assume}(\mathbf{I} < \mathbf{E})$ would lead to $i+1 < e$ (where the "+1" comes from the increment of $\mathbf{I}$ and the fact that $\mathbf{S_2}$ does not modify $\mathbf{I}$). This would be inconsistent with the assumption from E stating that $i \geq e-1$, and thus the path is pruned.

8.3.9 Solving Constraints

Once the constraints are generated, they can be solved using an iterative algorithm that starts with all the constraint variables set to *true* and then iteratively strengthens the constraint variables until a theorem prover is able to show that all constraints are satisfied. The PEC algorithm first initializes the constraint variables with the conditions that are required for the visible instructions to be equivalent, thereby solving all the constraints of the first kind. Then it chooses any constraint of the second kind and iteratively solves it till it reaches a fix-point.

As an example, consider the constraint corresponding to the B-C path, the PEC tool would ask a theorem prover to show that, for any σ_1 and σ_2: if (1) $\psi_{(a_2,b_1)} := (\sigma_1 = \sigma_2 \wedge eval(\sigma_1, \mathbf{I}) < eval(\sigma_1, \mathbf{E}))$ holds and (2) the original program executes $[\mathbf{S_1}]$ and (3) the transformed program executes $[\mathbf{S_1}; \texttt{assume}(\mathbf{I} < \mathbf{E} - 1)]$, then $\psi_{(a_3,b_3)} := (\sigma_1 = \sigma_2 \wedge eval(\sigma_1, \mathbf{I}) < eval(\sigma_1, \mathbf{E}) \wedge eval(\sigma_2, \mathbf{I}) < eval(\sigma_2, \mathbf{E}) - 1)$ will hold after the two statements have executed. In this case, the implication follows immediately from the assume and the fact that $\mathbf{S_1}$ produces the same program state if started in the same program state. In fact for this example, the value of the constraint variable after solving the constraints of the first kind is indeed the bisimulation relation. If for some path pair X to Y, the implication does not hold (this is not the case in Fig. 8.4), the PEC algorithm would strengthen the current value of the constraint variable at X with the weakest precondition of the current value of the constraint variable at Y. Using such iterative strengthening, the PEC algorithm tries to convert the original guessed relation into a bisimulation relation.

8.4 Parameterized Equivalence Checking

We now describe the PEC approach in more detail. Recall that the goal is to show that two parameterized programs P_1 and P_2 are equivalent under side conditions ϕ. We represent each program P as a transition diagram (Definition 10), which we denote by $\pi = (\mathscr{L}, \mathscr{I}, \rightarrow, \iota, \varepsilon)$. In particular, we assume that $\pi_1 = (\mathscr{L}_1, \mathscr{I}_1, \rightarrow_1, \iota_1, \varepsilon_1)$ is the transition diagram of the original program, and $\pi_2 = (\mathscr{L}_2, \mathscr{I}_2, \rightarrow_2, \iota_2, \varepsilon_2)$ is the transition diagram of the transformed program.

PEC uses a stronger definition of equivalence (defined below) between two parameterized program, because parts of these programs are unknown. Furthermore, the programs considered by PEC are deterministic, meaning that starting at a configuration $\langle \iota, \sigma \rangle$ there is at most one execution sequence that end in the exit location ε of the program.

Definition 28 (Final State). Given a transition diagram $\pi = (\mathscr{L}, \mathscr{I}, \rightarrow, \iota, \varepsilon)$ and a state $\sigma \in \Sigma$, we use the notation $\pi(\sigma)$ to represent the final state after executing π starting in state σ. i.e.

$$\pi(\sigma) = \sigma' \quad \text{iff} \quad \exists\, \eta.\, \langle \iota, \sigma \rangle \overset{\eta}{\rightsquigarrow}^{+} \langle \varepsilon, \sigma' \rangle$$

Definition 29 (Equivalence). Given two transition diagram π_1 and π_2, we define π_1 to be equivalent to π_2 if for any state $\sigma \in \Sigma$, we have $\pi_1(\sigma) = \pi_2(\sigma)$.

The above definition of equivalence allows PEC optimizations to be used anywhere inside of a program: by establishing program state equivalence, PEC guarantees that the remainder of the program, after the optimization, runs the same way in the original and the transformed programs. Observable events such as IO can be modeled using heap updates. For example, a call to `printf` can just append its arguments to a linked list on the heap. In this setting, the PEC approach guarantees that the order of IO events is preserved.

The instructions in a PEC transition diagram are taken from a concrete programming language of instructions, extended with meta-variables. The main components of the PEC approach do not depend on the choice of the concrete language of instructions: this language can for example include pointers, arrays, and function calls. The PEC approach does however make one exception to this rule: it assumes the existence of `assume` instructions. In particular, conditionals are modeled using `assume` instructions on the transitions that flow away from a branch location (as shown for the example in Fig. 8.4). Such `assume` instructions are also used to insert the information from side conditions into the original or transformed program as needed, so that the PEC tool can reason about the side conditions. The choice of concrete language only affects the semantics of instructions, which is entirely modularized in a function called *step* (which we have already seen). The only part of the system that knows about *step* is the theorem prover, which is given background axioms about the semantics of instructions (so that it knows for example how **I**++ updates the store). All other parts of the system treat *step* as a black box.

8.4.1 Bisimulation Relation

The PEC approach is based on using a bisimulation relation to relate the execution of the original program and the transformed program. We use the same definitions as described in Sect. 7.4, however we redefine some of them again here to adapt them in the setting of parameterized programs. The PEC system considers visible instructions to be the statement meta-variables (e.g. $\mathbf{S_1}, \mathbf{S_2} \in \vartheta$). We define two statement meta-variables to be equivalent if the variable name and the state of the program are the same. We also slightly modify the definition of simulation relation (Definition 17) to reflect the stronger definition of equivalence defined above.

Definition 30 (Simulation Relation). A simulation relation R for two transition diagrams $\pi_1 = (\mathscr{L}_1, \mathscr{I}_1, \rightarrow_1, \iota_1, \varepsilon_1)$ and $\pi_2 = (\mathscr{L}_2, \mathscr{I}_2, \rightarrow_2, \iota_2, \varepsilon_2)$ is a verification relation such that:

$$
\begin{array}{l}
R(\iota_1, \iota_2, \sigma_1 = \sigma_2) \text{ and } R(\varepsilon_1, \varepsilon_2, \sigma_1 = \sigma_2). \\
\forall gl_2, gl_2' \in \mathscr{L}_2, gl_1 \in \mathscr{L}_1, \sigma_1, \sigma_2, \sigma_2' \in \Sigma, \phi \in \Phi, \eta_2 \in \mathscr{N}. \\
\quad \left[\begin{array}{l} \langle gl_2, \sigma_2 \rangle \overset{\eta_2}{\rightsquigarrow}_2^+ \langle gl_2', \sigma_2' \rangle \wedge \\ \qquad R(gl_1, gl_2, \phi) \wedge \phi(\sigma_1, \sigma_2) = true \end{array} \right] \Rightarrow \\
\quad \exists gl_1' \in \mathscr{L}_1, \sigma_1' \in \Sigma, \phi' \in \Phi, \eta_1 \in \mathscr{N}. \\
\qquad \left[\begin{array}{l} \langle gl_1, \sigma_1 \rangle \overset{\eta_1}{\rightsquigarrow}_1^+ \langle gl_1', \sigma_1' \rangle \wedge \\ R(gl_1', gl_2', \phi') \wedge \phi'(\sigma_1', \sigma_2') = true \wedge \eta_1 \equiv \eta_2 \end{array} \right]
\end{array}
$$

The definition of bisimulation relation in this context is the same as in Definition 27. Furthermore, from Theorem 7.2 we know that if there exists a bisimulation relation between π_1 and π_2 then π_1 and π_2 are equivalent. Thus, a bisimulation relation is a witness that two transition diagrams are equivalent. The PEC approach is based on Theorem 7.2. In particular, the general approach is to try to infer a bisimulation relation to show that π_1 and π_2 are equivalent.

8.4.2 Architectural Overview

Figure 8.5 shows the pseudo-code of the PEC approach. There are three steps: the Permute module, the GenerateConstraints module and the SolveConstraints module. The Permute module runs as a pre-processor before the main bisimulation-based approach is performed. The Permute module applies a general form of the Permute theorem that has been used in translation validation of loop optimizations [73], but it does so on parameterized programs. The Permute module allow us to prove certain non-structure preserving transformations that is not possible using the bisimulation-based approach. After the Permute module has run, the

Fig. 8.5 Parameterized Equivalence Checking

function PEC(π_1, π_2, ϕ)
  let $(\pi_1', \pi_2') :=$ Permute(π_1, π_2, ϕ)
  let $\mathscr{C} :=$ GenerateConstraints(π_1', π_2', ϕ)
  SolveConstraints($\mathscr{C}$)

GenerateConstraints and SolveConstraints module implement the bisimulation approach. These modules are similar to the one presented in Sect. 7.6. In particular, the GenerateConstraints module first generates a set of pair-of-interest locations $\mathscr{R}$ from the two transition diagrams π_1 and π_2, and then generates a set of constraints $\mathscr{C}$. The SolveConstraints module next solves these constraints such that the properties from Definitions 30 and 27 hold, possibly strengthening the relation in order to guarantee property 3 of Definitions 30. The next three sections of this chapter describe each of these modules of the PEC system. We first describe the GenerateConstraints and SolveConstraints modules, which are at the heart of the PEC approach, and then move on the Permute module, which acts as a preprocessing step.

8.5 GenerateConstraints Module

To prove that two parameterized programs are equivalent the PEC approach attempts to discover a bisimulation relation between them. To do this, the GenerateConstraints module computes a set of constraints for the set of pair-of-interests, which will then be strengthened to a bisimulation relation by the SolveConstraints module.

Here again to focus our attention on only those locations for which the PEC approach infers the relation entries, we define two sets of locations $\mathscr{Q}_1$ and $\mathscr{Q}_2$ for the transition diagrams π_1 and π_2 respectively. These are the locations that immediately precede a statement meta-variable. We use the *skipping transition* relations $gl_1 \overset{(w_1,\mathscr{Q}_1)}{\hookrightarrow_{\pi_1}} gl_1'$ and $gl_2 \overset{(w_2,\mathscr{Q}_2)}{\hookrightarrow_{\pi_2}} gl_2'$ (Definition 18) that skips over all locations not in $\mathscr{Q}_1$ and $\mathscr{Q}_2$ respectively. We also use the definition of *parallel transition* relation $\hookrightarrow\!\!\!\!\rightarrow$ (Definition 21) that essentially traverses the two transition diagrams (specification and implementation) in synchrony, while focusing on only those locations that are in $\mathscr{Q}_1$ and $\mathscr{Q}_2$ respectively.

The predicate $\mathsf{Rel} : \mathscr{I}^* \times \mathscr{I}^* \times \mathscr{Q}_1 \times \mathscr{Q}_2 \to \mathscr{B}$ used in the parallel transition definition checks if the pair of paths are feasible. To do this the PEC algorithm conservatively determines if two paths are *infeasible*, and if not then we say they are feasible i.e.

$$\mathsf{Rel}(w_1, w_2, gl_1, gl_2) = \neg\mathsf{Infeasible}(w_1, w_2, gl_1, gl_2)$$

For infeasibility check, the PEC system first defines two sets of locations $\mathscr{A}_1$ and $\mathscr{A}_2$ for π_1 and π_2 respectively. These sets consists of locations that immediately precede an assume statement. We define for each pair $(gl_1, gl_2) \in \mathscr{Q}_1 \times \mathscr{Q}_2$ a variable $\mathscr{X}_{(gl_1,gl_2)}$ such that:

$$\mathscr{X}_{(gl_1,gl_2)} = Post(gl_1,\mathscr{A}_1,\pi_1) \wedge Post(gl_2,\mathscr{A}_2,\pi_2) \wedge \sigma_1 = \sigma_2 \quad \text{and}$$
$$Post(gl,\mathscr{A},\pi) = \bigvee_{\{gl' \xrightarrow{(w,\mathscr{A})}_{\pi} gl\}} \mathsf{sp}(w,true)$$

Here $\mathscr{X}_{(gl_1,gl_2)}$ computes a conservative formula over σ_1 and σ_2 that should hold when π_1 and π_2 are at locations gl_1 and gl_2 respectively. Within $\mathscr{X}$, the predicate $Post(gl,\mathscr{A},\pi)$ is the disjunction of the strongest post conditions with respect to *true* over paths w for which there exists some gl' such that $gl' \xrightarrow{(w,\mathscr{A})}_{\pi} gl$.

The PEC implementation of the Infeasible function can be succinctly represented as follows:

$$\mathsf{ATP}(\neg(\mathsf{sp}(w_1,\mathscr{X}_{(gl_1,gl_2)}) \wedge \mathsf{sp}(w_2,\mathscr{X}_{(gl_1,gl_2)}))) = Valid$$

Infeasible first computes the strongest postcondition of w_1 and w_2 with respect to the formula $\mathscr{X}_{(gl_1,gl_2)}$. If an automated theorem prover (ATP) can show that the two post-conditions are inconsistent, then the combination of those two paths is infeasible, and can be pruned. The Infeasible function performs the pruning that was intuitively described for the loop pipelining example in Sect. 8.3.5.

The other predicate WellMatched : $\mathscr{I}^* \times \mathscr{I}^* \rightarrow \mathscr{B}$ in the definition of parallel transition $\leftrightsquigarrow$ checks if the two sequences w_1 and w_2 of instructions are well-matched, i.e., neither of them contain a statement meta-variable or they each contains the same statement meta-variable. Using these definitions of Rel, WellMatched and relation $\leftrightsquigarrow$, we now use the relation $\mathscr{R} \subseteq \mathscr{Q}_1 \times \mathscr{Q}_2$ of location pairs that will form the entries of the bisimulation relation (see Definition 22). As before the set $\mathscr{R}$ can easily be computed by starting with the empty set, and applying the three rules exhaustively. The PEC algorithm then checks if the computed set $\mathscr{R}$ is a well-formed pairs of interest relation using Definition 23.

To uniformly handle the side conditions ϕ, PEC inserts the side condition assumptions into the original and transformed programs in the form of `assume` statements. An `assume` statement takes as argument a predicate over the program state σ that occurs at the point where the assume holds. To ease presentation, we make the simplifying assumption that $\phi = \phi_1@L_1 \wedge \ldots \wedge \phi_n@L_n$ (the PEC implementation handles the general case). For each side condition ϕ_i, we define $[\![\phi_i]\!]$ to be a predicate over σ that directly encodes the side condition's meaning provided by the optimization writer. Then for each $\phi_i@L_i$, the PEC system finds the location L_i in either the original or the transformed program, and insert `assume`$([\![\phi_i]\!])$ at that location.

Similar to Sect. 7.6.2, we now describe the PEC algorithm in terms of constraint solving. In particular, for each $(gl_1,gl_2) \in \mathscr{R}$ we define a constraint variable $\psi_{(gl_1,gl_2)}$ (Definition 24) representing the predicate that should be computed for the bisimulation entry (gl_1,gl_2). Using these constraint variables, the final bisimulation relation will have the form:

$$\{(gl_1,gl_2,\psi_{(gl_1,gl_2)}) \mid \mathscr{R}(gl_1,gl_2)\}$$

To compute the predicates that the constraint variables $\psi_{(gl_1,gl_2)}$ stand for, the PEC algorithm generates a set of constraints on these variables, and then solves the constraints. We use a slightly modified version of Definition 25 to compute the set of constraints $\mathscr{C}$.

Definition 31 (Set of Constraints). The set $\mathscr{C}$ of constraints is defined by:

$$\left[\psi_{(\iota_1,\iota_2)} \Rightarrow \sigma_1 = \sigma_2\right] \in \mathscr{C} \text{ and } \left[\psi_{(\varepsilon_1,\varepsilon_2)} \Rightarrow \sigma_1 = \sigma_2\right] \in \mathscr{C}$$
$$\text{For each } (gl_1, gl_2) \text{ in } \mathscr{R}\text{:} \quad \left[\psi_{(gl_1,gl_2)} \Rightarrow \mathscr{X}_{(gl_1,gl_2)}\right] \in \mathscr{C}$$
$$\text{For each } (gl_1, gl_2) \overset{(w_1,w_2)}{\dashrightarrow} (gl_1', gl_2')\text{:} \quad \left[\psi_{(gl_1,gl_2)} \Rightarrow \mathsf{pwp}(w_1, w_2, \psi_{(gl_1',gl_2')})\right] \in \mathscr{C}$$

There are three kinds of constraints. The first kind of constraints make sure that if the two parameterized programs start in equal states then they end in equal states. The next kind of constraints implies that for each pair of locations (gl_1, gl_2) the instructions about to execute at gl_1 and gl_2 are equivalent. This condition in the PEC setting is captured using the formula $\mathscr{X}_{(gl_1,gl_2)}$.

The last kind of constraints state that for each pair of paths between two entries in $\mathscr{R}$, the predicate at the beginning of the paths must imply the predicate at the end of the paths. We express this condition using the weakest precondition computation pwp, which is a parameterized version of the regular weakest precondition.

The main challenge in expressing this weakest precondition is that the traditional formulation of weakest precondition depends on the structure of the statements being processed. As a result, it is difficult to use this definition for statements like $\mathbf{S_1}$ and $\mathbf{S_2}$ in parameterized programs, because the precise structure of these statements is not known. To address this challenge, the PEC system use an alternate yet equivalent definition of weakest precondition. In particular, consider the traditional weakest precondition computation, and assume that the predicate we are computing is a function from program states to booleans. Then the traditional weakest precondition wp can be expressed as:

$$\mathsf{wp}(\mathbf{S}, \varphi)(\sigma) = \varphi(\mathit{step}(\sigma, \mathbf{S}))$$

If we assume that the program state σ is simply a free variable in the predicate φ, then wp can be expressed as:

$$\mathsf{wp}(\mathbf{S}, \varphi) = \varphi[\sigma \mapsto \mathit{step}(\sigma, \mathbf{S})]$$

This is precisely the alternate definition of weakest preconditions that was developed at the end of Sect. 4.4 to address the complexities of pointers.

Generalizing this to two parallels paths in two different programs, the predicates now have free variables σ_1 and σ_2, and we can express pwp as follows:

$$\mathsf{pwp}(w_1, w_2, \varphi) = \varphi[\sigma_1 \mapsto \mathit{step}(\sigma_1, w_1), \sigma_2 \mapsto \mathit{step}(\sigma_2, w_2)]$$

8.6 SolveConstraints Module

Once the set of constraints $\mathscr{C}$ have been generated, the SolveConstraints module tries to solve these constraints iteratively by starting with all the *constraint variables* initialized to *true*, and iteratively strengthening the constraint variables in the relation until all the constraints are satisfied. The SolveConstraints function used here is the exact function presented in Fig. 7.6. As before one subtlety is that one cannot strengthen the relation at the entry points ι_1, ι_2. If the algorithm ever tries to do this, it produces an error. Here again because SolveConstraints is trying to compute a fixed-point over the very flexible but infinite domain of boolean formulas, it may not terminate. However, the PEC experiments have shown that in practice SolveConstraints can quickly find a fix-point.

8.7 Permute Module

The main technique for PEC to prove equivalence relies on the bisimulation approach mentioned above. However, the bisimulation approach has some known limitations. In particular, bisimulation relations are not well suited for proving the correctness of non-structure preserving transformations, which are transformations that change the execution order of code across loop iterations. Previous work on translation validation has devised a technique called Permute [213] for handling such transformations on concrete programs. PEC adapts this technique to the setting of parameterized programs.

The Permute module runs as a pre-pass to the bisimulation relation approach. Permute looks for loops in the original and transformed programs that it can prove equivalent, and for the ones it can, it replaces them with a new fresh variable **S**, which will then allow the bisimulation relation part of the PEC approach to see that they are equivalent.

The Permute algorithm tries to find a general nested loop of the following form, where the symbol $\prec_L$ denotes a total order on L.

$$\begin{array}{c} \texttt{for } i_1 \in I_1 \texttt{ by } \prec_{I_1} \texttt{ do} \\ \ddots \\ \texttt{for } i_n \in I_n \texttt{ by } \prec_{I_n} \texttt{ do} \\ B(i_1, \ldots, i_n)\,; \end{array}$$

where I_j is the domain of the index variable i_j

The relation $\prec_{I_j}$ represents the order in which the index variable i_j is traversed. The above general nested loop can be represented more compactly as follows:

$$\texttt{for } \mathbf{i} \in \mathbf{I} \texttt{ by } \prec_{\mathbf{I}} \texttt{ do } B(\mathbf{i})\,;$$
$$\textbf{where} \quad \mathbf{I} = I_1 \times \cdots \times I_n \quad \textbf{and}$$
$$\mathbf{i} \prec_{\mathbf{I}} \mathbf{j} \iff \bigvee_{k=1}^{n} (i_1, \ldots, i_{k-1}) = (j_1, \ldots, j_{k-1}) \wedge i_k \prec_{I_k} j_k$$

The relation $\prec_{\mathbf{I}}$ above is the lexicographic order on $\mathbf{I}$.

The algorithm tries to find a loop structure as above in the original program and in the transformed program, and for each such pair, it tries to show that the following loop reordering transformation is correct:

$$\begin{array}{c} \texttt{for } \mathbf{i_1} \in \mathbf{I_1} \texttt{ by } \prec_{\mathbf{I_1}} \texttt{ do } B(\mathbf{i_1}) \\ \Downarrow \\ \texttt{for } \mathbf{i_2} \in \mathbf{I_2} \texttt{ by } \prec_{\mathbf{I_2}} \texttt{ do } B(F(\mathbf{i_2})) \end{array} \tag{8.1}$$

The above transformation may change the order of the index variables by changing the domain $\mathbf{I_1}$ to $\mathbf{I_2}$ and the relation $\prec_{\mathbf{I_1}}$ to $\prec_{\mathbf{I_2}}$ and also possibly changing the loop's body by applying a linear transformation from $B(\mathbf{i_1})$ to $B(F(\mathbf{i_2}))$.

To show that the above transformation is correct, we need to ensure that the transformed loop executes the same instances of the loop body in an order that preserves the body's behavior. In order to define the conditions under which this happens, we first define when two program fragments commute.

Definition 32 (Commute). We say two program fragments S_1 and S_2 commute, written $S_1 \approx S_2$, if starting from an arbitrary initial state, the resultant state of executing S_1 and then S_2 is the same as executing S_2 and then S_1.

We can now guarantee that the original and transformed loops are equivalent by requiring the following properties to hold:

1. There exists a 1-1 correspondence between $\mathbf{I_1}$ and $\mathbf{I_2}$.
2. For every $\mathbf{i_1}, \mathbf{i_2} \in \mathbf{I_1}$, if $B(\mathbf{i_1})$ executes before $B(\mathbf{i_2})$ in the original program and $B(\mathbf{i_2})$ executes before $B(\mathbf{i_1})$ in the transformed program then $B(\mathbf{i_1})$ and $B(\mathbf{i_2})$ commute, i.e. $B(\mathbf{i_1}) \approx B(\mathbf{i_2})$

The first property above can be established by showing that the linear function $F : \mathbf{I_2} \longrightarrow \mathbf{I_1}$ is a bijective function, i.e. F is one-to-one and onto. This in turn can be guaranteed by defining an inverse function $F^{-1} : \mathbf{I_1} \longrightarrow \mathbf{I_2}$. The above observations are summarized in the following Permute Theorem.

Theorem 8.1 (Permute). *A loop reording transformation of the form shown in Formula* (8.1) *preserves semantics if the following hold:*

1. $\forall \mathbf{i_2} \in \mathbf{I_2}.$ $\quad F(\mathbf{i_2}) \in \mathbf{I_1}$
2. $\forall \mathbf{i_1} \in \mathbf{I_1}.$ $\quad F^{-1}(\mathbf{i_1}) \in \mathbf{I_2}$
3. $\forall \mathbf{i_2} \in \mathbf{I_2}.$ $\quad \mathbf{i_2} = F^{-1}(F(\mathbf{i_2}))$
4. $\forall \mathbf{i_1} \in \mathbf{I_1}.$ $\quad \mathbf{i_1} = F(F^{-1}(\mathbf{i_1}))$
5. $\forall \mathbf{i_1}, \mathbf{i_1'} \in \mathbf{I_1}.$ $\quad \mathbf{i_1} \prec_{\mathbf{I_1}} \mathbf{i_1'} \wedge F^{-1}(\mathbf{i_1'}) \prec_{\mathbf{I_2}} F^{-1}(\mathbf{i_1}) \implies B(\mathbf{i_1}) \approx B(\mathbf{i_1'})$

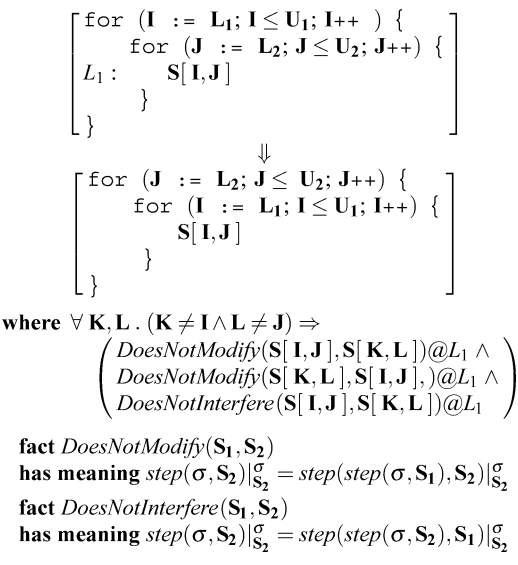


Fig. 8.6 Loop interchange example using Permute module

Theorem 8.1 was introduced and proved in previous work [171, 179, 213]. The Permute module tries to apply Theorem 8.1 by asking an automated theorem prover to discharge the preconditions of the theorem assuming the side conditions given in the transformation. As an example, consider the simple loop interchange optimization shown in Fig. 8.6. For clarity and ease of explanation, the example is simplified here to have constant bounds (L_1, U_1, L_2, U_2) instead of arbitrary expressions.

The Permute module first transforms the original and transformed programs into the canonical representations of loops. In particular, the original program is summarized as

$$\begin{aligned} &\mathbf{I_1} = \{(i,j) \mid i \in [\mathbf{L_1}, \mathbf{U_1}], j \in [\mathbf{L_2}, \mathbf{U_2}]\} \\ &\textbf{and} \quad B((i,j)) = \mathbf{S}[\,\mathbf{i},\mathbf{j}\,] \\ &\textbf{and} \quad \prec_{\mathbf{I_1}} \text{ is the lexicographic order on } \mathbf{I_1} \end{aligned}$$

and the transformed program is represented as

$$\begin{aligned} &\mathbf{I_2} = \{(i,j) \mid i \in [\mathbf{L_2}, \mathbf{U_2}], j \in [\mathbf{L_1}, \mathbf{U_1}]\} \\ &\textbf{and} \quad B((i,j)) = \mathbf{S}[\,\mathbf{j},\mathbf{i}\,] \\ &\textbf{and} \quad \prec_{\mathbf{I_2}} \text{ is the lexicographic order on } \mathbf{I_2} \end{aligned}$$

Since there is one loop in the original program and one in the transformed program, Permute tries to prove them equivalent. In order to apply the Permute Theorem, the PEC tool needs to infer the two mapping functions F and F^{-1}, and prove properties 1–4 of Theorem 8.1. Permute infers these functions automatically using a simple heuristic that runs a range analysis over the original and transformed

programs, and uses the results of the upper and lower bounds on index variables to infer F and F^{-1}. For this loop interchange optimization, the PEC tool automatically infers that the two functions are: $F((i,j)) = (j,i)$, and $F^{-1}((i,j)) = (j,i)$. The above heuristic infers the appropriate mapping functions in all the optimizations that uses the Permute module and are presented in Sect. 8.8. However, the PEC tool also provide the ability for the programmer to provide F and F^{-1} in the case where the above heuristic cannot find the appropriate functions.

The purpose of the side conditions of loop interchange is to allow the theorem prover to show property 5 of Theorem 8.1. One option for expressing the side condition is to use the *Commute* fact as shown here:

fact $Commute(\mathbf{S_1}, \mathbf{S_2})$
has meaning $step(step(\sigma, \mathbf{S_1}), \mathbf{S_2}) = step(step(\sigma, \mathbf{S_2}), \mathbf{S_1})$

This fact directly gives a predicate that is very close to property 5. However, we now need to use a heavyweight analysis when the compiler runs to establish *Commute* (for example a theorem prover, the Omega test [171], or more generally dependence analysis [154]). Another option, which is shown in Fig. 8.6 illustrates the flexibility of the PEC approach. It provides a more syntactic definition of commutativity, using two new facts: *DoesNotModify*, which holds when a statement does not modify the variables or heap locations that another may read, and *DoesNotInterfere*, which holds when a statement does not modify the variables or heap locations that another may write to. The notation $\sigma_1|_{\mathbf{S}}^{\sigma_2}$ represents the state σ_1 projected onto the variables and heap locations that $\mathbf{S}$ modifies if it executes starting in state σ_2. The benefit of using the more syntactic *DoesNotModify* and *DoesNotInterfere* facts is that they can more easily be implemented using simple Rhodium dataflow functions, which in turn can be proved correct automatically. In this way we will know that the computed facts when the compiler runs imply the semantic meanings that the PEC technique assumed when proving the correctness of loop interchange once and for all.

8.8 Experiments and Results

The PEC algorithm was implemented using Simplify theorem prover [44] to realize the ATP module from Sect. 8.5. The PEC system has been used to express and prove correct a variety of transformations, including: copy propagation, constant propagation, loop invariant code hoisting, conditional speculation, software pipelining, loop unswitching, loop unrolling, loop fusion and loop splitting.

Figure 8.7 lists a selection of optimizations that were proved correct using PEC. For each optimization the figure lists the time it took to carry out PEC and

Optimizations	Uses permute	Time (secs)	# ATP calls
Category 1			
Copy propagation	No	1	3
Constant propagation	No	1	3
Common sub-expression elim	No	1	3
Partial redundancy elimination	No	3	13
Category 2			
Loop invariant code hoisting	No	8	25
Conditional speculation	No	2	14
Speculation	No	3	12
Category 3			
Software pipelining	No	5	19
Loop unswitching	No	16	94
Loop unrolling	No	10	45
Loop peeling	No	6	40
Loop splitting	No	15	64
Loop alignment	Yes	1	5
Loop interchange	Yes	1	5
Loop reversal	Yes	1	5
Loop skewing	Yes	2	5
Loop fusion	Yes	4	10
Loop distribution	Yes	4	10

Fig. 8.7 Optimizations proven correct using PEC. *Category 1*: expressible and provable in Rhodium; *Category 2*: provable in Rhodium, but the PEC version is more general and easier to express; *Category 3*: not expressible or provable in Rhodium

the number of queries to the theorem prover. To be clear about the contribution compared to the Rhodium system for automatically proving optimizations correct, Fig. 8.7 partitions the optimizations into three categories.

Category 1: Optimizations that were also expressed and proved correct in Rhodium, and whose PEC formulation is equivalent to the Rhodium formulation.

Category 2: Optimizations that could have been expressed and proved correct in Rhodium, but the PEC versions are much more general than the Rhodium version, and also much easier to express. For example, in the case of loop invariant code hoisting, PEC can prove the correctness of hoisting loop-invariant branches or even entire loops, while the Rhodium version could only hoist loop-invariant assignments. Furthermore, these optimizations are much easier to express in the PEC formulation because of the explicit support for many-to-many rewrites. In contrast, implementing these optimizations in Rhodium would require an expert to carefully craft sequences of local statement rewrites that achieves the intended effect. For example, moving a statement in Rhodium requires inserting a duplicate copy of the statement at the target location, and then removing the original statement in a separate pass.

Category 3: Optimizations that cannot be proved correct, or even expressed, in Rhodium. The support for many-to-many rewrite rules makes it easy to express these optimizations, and the PEC technique is general enough to handle their cor-

rectness proofs. For example, in the case of software pipelining PEC is able to prove the transformation is correct even though the loop bodies of the original and transformed programs are not identical which changes the structure of the computation between loop iterations.

The trusted computing base for the PEC system includes: (1) the PEC checker, comprising 2,408 lines of OCaml code (2) the Simplify automated theorem prover, a widely used and well tested theorem prover, and (3) the execution engine that will run the optimizations. Within the execution engine, the trust can be further subdivided into two components. The first component of the execution engine must perform the syntactic pattern matching for rewrite rules, and apply rewrite rules when they fire. This part is always trusted. The second component of the execution engine must perform program analysis to check each optimization's side-conditions in a way that guarantees their semantic meaning. Here the PEC system offers a choice. These analysis can either be trusted and thus implemented inside the compiler using arbitrarily complex analysis, or untrusted and implemented using a provably safe analysis system like Rhodium.

8.9 Execution Engine

In this section we discuss the implementation of a prototype execution engine that runs optimizations checked by PEC. Although PEC can be applied to any intermediate representation for which weakest preconditions can be computed, this prototype execution engine transforms programs written in a C-like intermediate language including arrays and function calls. This prototype is able to run all the optimizations described in the previous section. Although the execution engine is a prototype, it demonstrates how optimizations can be incorporated into a compiler, and also shows that the optimizations that are checked execute as expected.

The execution engine is embodied in a function called *Apply*, which takes as input a program p, a transformation rule $[P_1 \Rightarrow P_2$ **where** $\phi]$, and a profitability heuristic ρ, and returns a transformed program. The *Apply* function first uses pattern matching to find all locations in the program p where the pattern P_1 occurs. Then for each match that is found, *Apply* evaluates the side condition ϕ to make sure that the match is valid. The current prototype checks side conditions conservatively using read/write sets. For example, to guarantee that a statement s_1 does not modify another statement s_2, it checks that $\mathit{WriteSet}(s_1) \cap \mathit{ReadSet}(s_2) = \emptyset$.

For each match that is found where the side condition holds, *Apply* builds a substitution θ that records information about the match: θ maps the free variables in P_1 to concrete fragments of p, and it also records the location where the match occurred in p. *Apply* collects the resulting substitutions θ into a set Θ, and then it calls the profitability heuristic ρ with Θ as a parameter. The role of the profitability heuristic ρ is to select from the set Θ of all substitutions that have been found (representing all the possible applications of the transformation rule) those substitutions that it wants to apply. Because all the substitutions in Θ represent correct transformations, it does

Fig. 8.8 Implementation of Loop Pipelining using *Apply*

function $LoopPipe(p) :=$
 let $p' := Apply(p, t, \rho_{lp})$
 if $(p' = p)$ **then** p' **else** $LoopPipe(p')$

not matter which subset the profitability heuristic chooses, and so the profitability heuristic can perform arbitrary computation without being trusted. The above approach to profitability heuristic uses the generate-and-test approach presented in the Cobalt system [131]. Alternatively, an execution engine could also employ the more demand-driven approach used in the Rhodium system [133], where side conditions directly refer to profitability facts, thus constraining which matches are explored.

Once the profitability heuristic has selected the set of substitutions it wants to apply, the *Apply* function performs the corresponding transformations. If the profitability heuristic returns substitutions that overlap in the program fragments they match, then the *Apply* function picks an order to apply the substitutions in, and only applies a substitution θ if no previously applied substitution has transformed elements mentioned in θ.

As an example, Fig. 8.8 shows a function *LoopPipe* that uses *Apply* to perform loop pipelining. Let t represent the loop pipelining transformation from Fig. 8.2. The *LoopPipe* function uses *Apply* to repeatedly apply t. The loop pipelining profitability heuristic ρ_{lp} is applied after t has run. The prototype implements ρ_{lp} by selecting matches that reduce the number of dependencies between instructions in loop bodies.

8.10 Further Reading

The PEC work presented in this chapter is related to long lines of work in translation validation, proving loop optimizations correct, automated correctness checking of optimizations, human-assisted correctness checking of optimizations, and languages for expressing optimizations. We now discuss each area in more detail.

Translation Validation: As has been made clear several times before, the PEC approach is heavily inspired by the work that has been done on translation validation [73, 116, 117, 160, 170, 178]. However, unlike previous translation validation approaches, the PEC equivalence checking algorithm addresses the challenge of reasoning about statements that are not fully specified. As a result, the PEC approach is a *generalization* of traditional translation validation techniques that allows optimizations to be proved correct once and for all.

Proving Loop Optimizations Correct: The Permute approach discussed in this chapter, for reasoning about loop reordering transformations by having a single canonical representation for all these transformations is similar to the translation validation work of Zuck et al. [73] and the legality check approach of Kelly et al. [109].

Automated Correctness Checking of Optimizations: As with the PEC algorithm, the Cobalt [131] and Rhodium [133] systems are able to check the correctness of optimizations once and for all. However, Cobalt and Rhodium only support rewrite rules that transform a single statement to another statement, thus limiting the kinds of optimizations they can express and prove correct. The PEC approach can handle complex many-to-many rewrite rules explicitly, allowing it to prove many more optimizations correct.

Human-Assisted Correctness Checking of Optimizations: A significant amount of work has been done on manually proving optimizations correct, including abstract interpretation [39, 41], the work on the VLISP compiler [85], Kleene algebra with tests [113], manual proofs of correctness for optimizations expressed in temporal logic [121, 189], and manual proofs of correctness based on partial equivalence relations [12]. Analyses and transformations have also been proven correct mechanically, but not automatically: the soundness proof is performed with an interactive theorem prover that requires guidance from the user. For example, Young [211] has proven a code generator correct using the Boyer–Moore theorem prover enhanced with an interactive interface [108]. As another example, Cachera et al. [26] show how to specify static analyses and prove them correct in constructive logic using the Coq proof assistant. Via the Curry–Howard isomorphism, an implementation of the static analysis algorithm can then be extracted from the proof of correctness. Leroy's Comcert project [134] has also used a similar technique to manually develop a semantics preserving, optimizing compiler for a large subset of C. The Comcert compiler provides an end-to-end correctness guarantee, and does not just focus on optimizations, as we do in our approach. Tristan et al. has also proved that certain translation validators are correct once and for all, but here again by implementing the proof manually [198, 199]. In all these cases, however, the proof requires help from the user. In contrast to these approaches, the PEC system aims to be fully automated but trusts that the side conditions are computed correctly when the compiler executes.

Languages for Expressing Optimizations: The idea of analyzing optimizations written in a specialized language was introduced by Whitfield and Soffa with the Gospel language [207]. Many other frameworks and languages have been proposed for specifying dataflow analyses and transformations, including Sharlit [197], System-Z [210], languages based on regular path queries [187], and temporal logic [121, 189]. Although these systems all provide various benefits, none of them address automated correctness checking of the specified optimizations.

8.11 Summary

In this chapter we presented Parameterized Equivalence Checking (PEC), a technique for automatically proving optimizations correct once and for all. PEC bridges the gap between translation validation and once-and-for-all techniques.

The PEC approach generalizes previous translation validation techniques to handle parameterized programs, which are partially specified programs that can represent multiple concrete programs, thereby adapting them to provide once and for all correctness proofs. Furthermore, PEC's use of expressive many-to-many rewrite rules and a robust proof technique enables PEC to automatically prove correct optimizations that have been difficult or impossible to prove in other systems.

Acknowledgments This chapter in part, has been published as:

"Proving Optimizations Correct using Parameterized Program Equivalence" by Sudipta Kundu, Zachary Tatlock, and Sorin Lerner in *PLDI 09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation* [120].

Chapter 9
Conclusions and Future Work

We have addressed the need for high-level verification methodologies that allows us to do functional verification early in the design phase and then iteratively use correct refinement steps to generate the final RTL design. We believe that by performing verification on the high-level design, where the design description is smaller in size and the design intent information is easier to extract, and then checking that all refinement steps are correct, the domain of high-level verification can provide strong and expressive guarantees that would have been difficult to achieve by directly analyzing the low-level RTL code.

The high-level verification methods can be broadly seen as methods for verifying properties of high-level designs and methods for verifying that the translation from high-level design to low-level RTL preserves semantics. We classified the high-level verification area into three main parts, namely high-level property checking, translation validation, and synthesis tool verification. In this book we have explored techniques in each of the above three areas.

9.1 High-Level Property Checking

For high-level property checking, we discussed model checking techniques to verify that a design satisfies a given property such as absence of deadlocks or assertion violations. In particular, we explored two techniques one on execution-based model checking and the other on symbolic model checking.

We described the implementation of Satya, an execution-based model checking tool that combines static and dynamic POR techniques along with high-level semantics of SystemC to intelligently explore all possible behaviors of a SystemC design. This approach reduce the runtime overhead by conservatively computing the dependency information statically and using it during runtime, without significant loss of precision. During experiments Satya was able to automatically find an assertion violation in the FIFO benchmark (distributed as a part of the OSCI repository), which may not have been found by simulation (Sect. 5.10).

We also compared various recent state-of-the-art BMC techniques for concurrent programs. We discussed the operational and non-operational concurrency

S. Kundu et al., *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, DOI 10.1007/978-1-4419-9359-5_9,

semantics. Based on such semantics, we discussed two types of BMC modeling, namely synchronous and asynchronous. For synchronous modeling, we described a representative POR based BMC [205] approach. For asynchronous modeling, we describe two representative approaches, one based on CSSA [204] and another based on token-passing [64, 67]. We provide a comparison for their relative formula size and encoding style.

9.2 Translation Validation

To verify the translation from the high-level design to low-level RTL is correct, we described a translation validation tool called Surya. This algorithm uses a *bisimulation relation* approach to automatically prove the equivalence between two concurrent systems represented as transition diagrams. Surya is used to validate the synthesis process of Spark, a parallelizing HLS framework. Surya validates all the phases (except for parsing, binding and code generation) of Spark against the initial behavioral description. The experiments showed that with only a fraction of the development cost of Spark, Surya can validate the translations performed by Spark, and it even uncovered two previously unknown bugs that eluded long-term use.

9.3 Synthesis Tool Verification

For synthesis tool verification, we described a technique that proves the correctness of optimizations using *Parametrized Equivalence Checking* (PEC). This approach proves the correctness of the optimizations *once and for all*, before it is ever run. The PEC technique is a generalization of translation validation that proves the equivalence of *parameterized programs*. To highlight the power of PEC, a language is designed for implementing complex optimizations using many-to-many rewrite rules. This language is then used to implement a variety of optimizations including software pipelining, loop unrolling, and loop unswitching. The PEC implementation (described in this book) was able to automatically verify that all the optimizations we implemented using this language preserve program behavior.

9.4 Future Work

In this book we focused on exploring techniques in the area of high-level verification. Recent advances in formal methods and HLS have invigorated interest in high-level verification both in industry and academics. Various verification tools and techniques focused toward high-level design are starting to emerge. However, their adoption is in the early stages and the tools are often limited in the quality

of the results and the kinds of correctness guarantees that are provided. Naturally there are many things to be done in this area. In this section we discuss promising future research areas in verification of high-level designs and the tools associated with them.

Hardware–Software Modeling: High-level hardware languages support many features that are useful for both software and hardware designs. For example, SystemC allows both asynchronous and synchronous semantics of concurrency, and also both software and hardware data types. However, existing symbolic analysis tools including the ones discussed here often either target software or hardware. For example, most software model checkers only support software data types and asynchronous semantics of concurrency, and most hardware model checker only support hardware data types and synchronous semantics of concurrency. As a result, researchers often use abstraction or complicated techniques while modeling the non-supported features of a given model checker. This gap points to a possible research direction that would unify techniques for hardware models and techniques for software models into combined methodology for reasoning about hardware–software models.

Compositional Techniques: Although many techniques presented in this book use compositional methods to make the verification problem tractable, these techniques are still limited in their application. Even after decomposition using the current techniques the problem is still quite large and complex. Advanced and more efficient methods are needed for decomposing a computationally demanding global property into local properties whose verification is simpler.

Modular Framework: One observation of this book is that there are variety of overlap between various verification techniques. However, each verification tool is highly tuned toward the particular problem it is solving with its own input language or API, and as such it is hard to modify or extend. Hence, every time a new methodology is proposed a new tool has to be written, often from scratch. Unfortunately, the tools discussed in this book are not designed from a software engineering perspective. Thus, there is a need for a modular and reusable framework, which can be quickly used to prototype new ideas and test them.

Debugging: The tools discussed in this book provide only limited feedback to the user. When a bug is found, these tools cannot typically pin-point the error in the code. All the methods are able to output an error trace, but figuring out the cause of the error from it, is not straightforward and requires expertise in formal methods. Although not directly related to high-level verification, there has been work in this area [10], however adapting such techniques to this domain is still a challenge. Another limitation of our methods is that they often stop searching when a bug is found, rather than providing a list of all bugs. More broadly, the goal should be to fit formal verification into the regular develop–edit–debug flow, which would require the development of verification tools for speed and ease of use.

Synthesis-For-Verification: HLS process focuses mainly on three design constraints: area, timing and power. These methodologies tend to ignore verification,

which takes about 70% of the design cycle, as a constraint. Recently, Ganai et al. [68] proposed a new paradigm 'Synthesis-For-Verification' which involves synthesizing "verification-aware" designs that are more suitable for functional verification. Therefore, another research direction may be to use existing infrastructure of HLS to generate "verification friendly" models that are relatively easier to verify using state-of-the-art techniques.

Compiler Techniques: Many techniques used in HLS are similar to those used for compilers. As a result, advances in fields like compiler correctness can provide inspiration for developing techniques for high-level verification. For example, the translation validation and synthesis tool verification work presented here are inspired from the work done in the area of compiler correctness such as Necula's translation validation technique [160], Zuck et al. [73] work on proving various non-structure preserving transformation, and Lerner et al. [131,133] approach to automatically prove the correctness of compiler optimizations once and for all. There are many other techniques that have been successfully applied to the compiler domain, and can provide new directions for verification of the HLS process.

References

1. EP2: Electronic Payment 2. www.eftpos2000.ch
2. Adve, S.V., Hill, M.D., Miller, B.P., Netzer, R.H.B.: Detecting data races on weak memory systems. SIGARCH Computer Architecture News **19**(3), 234–243 (1991). DOI http://doi.acm.org/10.1145/115953.115976
3. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state-space exploration. Formal Methods System Design **18**(2), 97–116 (2001). DOI http://dx.doi.org/10.1023/A:1008767206905
4. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: Exploiting program structure for model checking concurrent software. In: CONCUR 04: 15th International Conference on Concurrency Theory, *LNCS*, vol. 3170, pp. 1–15. Springer Verlag (2004)
5. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using smt solvers instead of sat solvers. Int. J. Softw. Tools Technol. Transf. **11**(1), 69–83 (2009). DOI http://dx.doi.org/10.1007/s10009-008-0091-0
6. Ashar, P., Bhattacharya, S., Raghunathan, A., Mukaiyama, A.: Verification of RTL generated from scheduled behavior in a high-level synthesis flow. In: ICCAD '98: Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design, pp. 517–524 (1998). URL citeseer.ist.psu.edu/ashar98verification.html
7. Ashar, P., Raghunathan, A., Gupta, A., Bhattacharya, S.: Verification of scheduling in the presence of loops using uninterpreted symbolic simulation. In: ICCD '99: Proceedings of the 1999 IEEE International Conference on Computer Design, pp. 458–466. IEEE Computer Society, Washington, DC, USA (1999)
8. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, pp. 73–85. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1217935.1217943
9. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic predicate abstraction of C programs. In: PLDI '01: Proceedings of the 2001 ACM SIGPLAN conference on Programming Language Design and Implementation (2001). URL citeseer.ist.psu.edu/ball01automatic.html
10. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: Proceedings of the 30th ACM Symposium on Principles of Programming Languages, pp. 97–105 (2003)
11. Barnett, M., yuh Evan Chang, B., Deline, R., Jacobs, B., Leino, K.R.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science, pp. 364–387. Springer (2006)
12. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proceedings of the 31st ACM Symposium on Principles of Programming Languages (2004)

S. Kundu et al., *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, DOI 10.1007/978-1-4419-9359-5,

13. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Springer-Verlag (2004)
14. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast: Applications to software engineering. International Journal on Software Tools for Technology Transfer **9**(5), 505–525 (2007). DOI http://dx.doi.org/10.1007/s10009-007-0044-z
15. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using sat procedures instead of bdds. In: DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation, pp. 317–320. ACM, New York, NY, USA (1999). DOI http://doi.acm.org/10.1145/309847.309942
16. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, pp. 193–207. Springer-Verlag, London, UK (1999)
17. Blank, C.: Formal verification of register binding. In: WAVE '00: Proceedings of the Workshop on Advances in Verification (2000)
18. Borrione, D., Dushina, J., Pierre, L.: A compositional model for the functional verification of high-level synthesis results. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **8**(5), 526–530 (2000). DOI http://dx.doi.org/10.1109/92.894157
19. Boyer, R., Moore, J.: A Computational Logic. Academic Press (1979)
20. Boyer, R., Moore, J.: A Computational Logic, Second Edition. Academic Press (1998)
21. Bozzano, M., Bruttomesso, R., Cimatti, R., Junttila, T., Rossum, P.V., Schulz, S., Sebastiani, R.: The mathsat 3 system. In: Automated Deduction: Proceedings of the 20th International Conference, volume 3632 of Lecture Notes in Computer Science, pp. 315–321. Springer (2005)
22. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3), 293–318 (1992)
23. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic Model Checking: 10^{20} States and Beyond. In: Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 1–33. IEEE Computer Society Press, Washington, D.C. (1990). URL citeseer.ist.psu.edu/burch90symbolic.html
24. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 12–21. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1273442.1250737
25. Bustan, D., Grumberg, O.: Simulation based minimization. In: D.A. McAllester (ed.) Proceedings of the International Conference on Automated Deduction, *LNCS*, vol. 1831, pp. 255–270. Springer Verlag (2000)
26. Cachera, D., Jensen, T., Pichardie, D., Rusu, V.: Extracting a data flow analyser in constructive logic. In: Proceedings of the 13th European Symposium on Programming (ESOP 2004), *Lecture Notes in Computer Science*, vol. 2986. Springer-Verlag (2004)
27. Cai, L., Gajski, D.: Transaction Level Modeling: an overview. In: Proceedings of International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS) (2003)
28. Chaki, S., Clarke, E., Groce, A.: Modular verification of software components in C. In: IEEE Transactions on Software Engineering, pp. 385–395 (2003)
29. Chaki, S., Ouaknine, J., Yorav, K., Clarke, E.: Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In: Proceedings of the Workshop on Software Model Checking (SoftMC), *ENTCS*, vol. 89 (2003). URL citeseer.ist.psu.edu/chaki03automated.html
30. Chandy, K.M.: Parallel program design: a foundation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1988)
31. Clarke, E., Kroening, D., Yorav, K.: Behavioral consistency of c and verilog programs using bounded model checking. In: DAC '03: Proceedings of the 40th Conference on Design automation, pp. 368–371. ACM, New York, NY, USA (2003). DOI http://doi.acm.org/10.1145/775832.775928

32. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs, Workshop, pp. 52–71. Springer-Verlag, London, UK (1982)
33. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems **8**(2), 244–263 (1986)
34. Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. Tech. Rep. CMS-CS-89-145, School of Computer Science, Canergie Mellon University (1989)
35. C.N. Ip, D.L. Dill: Better verification through symmetry. In: D. Agnew, L. Claesen, R. Camposano (eds.) Computer Hardware Description Languages and their Applications, pp. 87–100. Elsevier Science Publishers B.V., Amsterdam, Netherland, Ottawa, Canada (1993). URL citeseer.ist.psu.edu/ip96better.html
36. Constable, R., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T., Smith, S.F.: Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall, NJ (1986)
37. Cook, B., Kroening, D., Sharygina, N.: Symbolic model checking for asynchronous boolean programs. In: SPIN '05: Proceedings of the 12th international workshop on Model Checking Software, pp. 75–90 (2005)
38. Corporation, I.: Rational Statemate. www.telelogic.com/products/statemate
39. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM Symposium on Principles of Programming Languages, pp. 238–252. Los Angeles CA (1977)
40. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM Symposium on Principles of Programming Languages, pp. 269–282. San Antonio, Texas (1979)
41. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Portland OR (2002)
42. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 84–97. ACM Press, New York, NY, Tucson, Arizona (1978)
43. Das, S., Dill, D.L., Park, S.: Experience with predicate abstraction. In: CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification, pp. 160–171. Springer-Verlag, London, UK (1999)
44. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the Association for Computing Machinery **52**(3), 365–473 (2005)
45. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975). DOI http://doi.acm.org/10.1145/360933.360975
46. Dill, D.L.: The murphi verification system. In: CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification, pp. 390–393. Springer-Verlag, London, UK (1996)
47. Dushina, J., Borrione, D., Jerraya, A.A.: Formal verification of the allocation step in high level synthesis. In: Forum on Design Languages (FDL'98). Lausanne, Switzerland (1998). URL citeseer.ist.psu.edu/57380.html
48. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Proceedings of the 18th Computer-Aided Verification conference, *LNCS*, vol. 4144, pp. 81–94. Springer-Verlag (2006)
49. Dutertre, B., Schneider, S.: Using a PVS embedding of CSP to verify authentication protocols. In: TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics, Lecture Notes in Artificial Intelligence. Springer-Verlag (1997)
50. (EDG), E.D.G.: C/C++ Front End (1992). www.edg.com
51. Eisenbiegler, D., Blumenröhr, C., Kumar, R.: Implementation issues about the embedding of existing high level synthesis algorithms in hol. In: TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics, pp. 157–172. Springer-Verlag, London, UK (1996)

52. Emerson, E.A., Clarke, E.M.: Characterizing correctness properties of parallel programs using fixpoints. In: Proceedings of the 7th Colloquium on Automata, Languages and Programming, pp. 169–181. Springer-Verlag, London, UK (1980)
53. Emerson, F.A., Sistla, A.P.: Symmetry and model checking. Formal Methods in System Design: An International Journal **9**(1/2), 105–131 (1996). URL citeseer.ist.psu.edu/emerson94symmetry.html
54. Eveking, H., Hinrichsen, H., Ritter, G.: Automatic verification of scheduling results in high-level synthesis. In: DATE '99: Proceedings of the conference on Design, automation and test in Europe, p. 12. ACM Press, New York, NY, USA (1999). DOI http://doi.acm.org/10.1145/307418.307449
55. Fisler, K., Vardi, M.Y.: Bisimulation and Model Checking. In: Proceedings of the 10th Conference on Correct Hardware Design and Verification Methods. Bad Herrenalb Germany CA (1999)
56. Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. Sci. Comput. Program. **71**(2), 89–109 (2008). DOI http://dx.doi.org/10.1016/j.scico.2007.12.001
57. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (2005)
58. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI '02: Proceedings of the 2002 ACM SIGPLAN conference on Programming Language Design and Implementation (2002)
59. Flanagan, C., Qadeer, S.: Transactions for software model checking. Electronic Notes in Theoretical Computer Science **89** (2003). URL citeseer.ist.psu.edu/flanagan03transactions.html
60. Floyd, R.W.: Assigning meanings to programs. In: J.T. Schwartz (ed.) Mathematical Aspects of Computer Science, *Proceedings of Symposia in Applied Mathematics*, vol. 19, pp. 19–32. American Mathematical Society, Providence, Rhode Island (1967)
61. Gajski, D.D., Dutt, N.D., Wu, A.C.H., Lin, S.Y.L.: High-Level Synthesis: Introduction to Chip and System Design. Kluwer Academic (1992)
62. Gajski, D.D., Ramachandran, L.: Introduction to high-level synthesis. IEEE Design and Test of Computers **11**(4), 44–54 (1994). DOI http://dx.doi.org/10.1109/54.329454
63. Ganai, M., Gupta, A.: SAT-Based Scalable Formal Verification Solutions (Series on Integrated Circuits and Systems). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
64. Ganai, M., Kundu, S.: Reduction of Verification Conditions for Concurrent System using Mutually Atomic Transactions. In: SPIN '09: Proceedings of the 16th International SPIN Workshop on Model Checking of Software (2009)
65. Ganai, M., Wang, C.: Interval Analysis for Concurrent Trace Programs using Transaction Sequence Graphs. In: Proceedings of Runtime Verification (2010)
66. Ganai, M.K., Gupta, A.: Accelerating high-level bounded model checking. In: ICCAD '06: Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design, pp. 794–801. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1233501.1233664
67. Ganai, M.K., Gupta, A.: Efficient modeling of concurrent systems in bmc. In: SPIN '08: Proceedings of the 15th international workshop on Model Checking Software, pp. 114–133. Springer-Verlag, Berlin, Heidelberg (2008). DOI http://dx.doi.org/10.1007/978-3-540-85114-1_10
68. Ganai, M.K., Mukaiyama, A., Gupta, A., Wakabayshi, K.: Synthesizing "verification aware" models: Why and how? VLSI Design '07: 20th International Conference on VLSI Design **0**, 50–56 (2007). DOI http://doi.ieeecomputersociety.org/10.1109/VLSID.2007.151
69. Girkar, M., Polychronopoulos, C.D.: Automatic extraction of functional parallelism from ordinary programs. IEEE Transaction on Parallel Distributed Systems (1992)
70. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Ph.D. thesis, Univerite De Liege (1995). URL citeseer.ist.psu.edu/godefroid95partialorder.html
71. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages (1997)

72. Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods (extended abstract). In: CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification, pp. 438–449. Springer-Verlag, London, UK (1993)
73. Goldberg, B., Zuck, L., Barrett, C.: Into the loops: Practical issues in translation validation for optimizing compilers. Electronic Notes in Theoretical Computer Science **132**(1), 53–71 (2005)
74. Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat-solver. Discrete Applied Mathematics **155**(12), 1549–1561 (2007). DOI http://dx.doi.org/10.1016/j.dam.2006.10.007
75. Gordon, M.: HOL: A proof generating system for higher-order logic. In: G. Birtwistle, P. Subrahmanyam (eds.) VLSI Specification Verification and Synthesis, pp. 73–128. Kluwer Academic Publishers (1988)
76. Graf, S., Saidi, H.: Construction of abstract state graphs of infinite systems with PVS. In: CAV '97: Proceedings of the international conference on Computer Aided Verification (1997)
77. Grobe, D., Ebendt, R., Drechsler, R.: Improvements for constraint solving in the SystemC verification library. In: GLSVLSI '07: Proceedings of the 17th ACM Great Lakes symposium on VLSI, pp. 493–496. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1228784.1228901
78. Grötker, T., Liao, S., Martin, G., Swan, S.: System Design with SystemC. Kluwer Academic Publishers (2002)
79. Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 122–131. ACM, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1040305.1040316
80. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: SPIN '07: Proceedings of the 14th International SPIN Workshop on Model Checking of Software, pp. 95–112 (2007)
81. Gupta, A.: Formal hardware verification methods: a survey. Formal Methods in System Design **1**(2-3), 151–238 (1992). DOI http://dx.doi.org/10.1007/BF00121125
82. Gupta, R., Brewer, F.: High-Level Synthesis: A Retrospective. In: High-Level Synthesis from Algorithm to Digital Circuit. Springer (2008)
83. Gupta, R.K., Liao, S.Y.: Using a programming language for digital system design. IEEE Design and Test **14**(2), 72–80 (1997). DOI http://dx.doi.org/10.1109/54.587745
84. Gupta, S., Dutt, N., Gupta, R., Nicolau, A.: Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In: International. Conference on VLSI Design (2003). URL citeseer.ist.psu.edu/gupta03spark.html
85. Guttman, J., Ramsdell, J., Wand, M.: VLISP: A verified implementation of Scheme. Lisp and Symbolic Computation **8**(1-2), 33–110 (1995)
86. Habibi, A., Tahar, S.: Design for verification of SystemC Transaction Level Models. In: DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, pp. 560–565. IEEE Computer Society, Washington, DC, USA (2005). DOI http://dx.doi.org/10.1109/DATE.2005.112
87. Hanna, F.K., Daeche, N., Longley, M.: Formal Synthesis of Digital Systems. In: L. Claesen (ed.) Proc IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, pp. 532–548. Elsevier (1989). URL http://www.cs.kent.ac.uk/pubs/1989/422. Leuven, Belgium
88. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. Higher Order Symbolic Computation **13**(4), 315–353 (2000). DOI http://dx.doi.org/10.1023/A:1026599015809
89. Helmstetter, C., Maraninchi, F., Maillet-Contoz, L., Moy, M.: Automatic generation of schedulings for improving the test coverage of Systems-on-a-Chip. In: FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design, pp. 171–178. IEEE Computer Society, Washington, DC, USA (2006). DOI http://dx.doi.org/10.1109/FMCAD.2006.10
90. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th ACM Symposium on Principles of Programming Languages (2002)

91. Hinrichsen, H.: Language of labelled segments documentation. Tech. rep., Darmstadt University of Technology (1998). URL http://www.rs.e-technik.tudarmstadt.de/~hinni/document/index.html
92. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). DOI http://doi.acm.org/10.1145/363235.363259
93. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International (1985)
94. Holzmann, G.J.: The model checker SPIN. Software Engineering **23**(5), 279–295 (1997). URL citeseer.ist.psu.edu/holzmann97model.html
95. Initiative, O.S.: IEEE Standard 1666 SystemC Language Reference Manual (2005). www.systemc.org
96. Isobe, Y., Roggenbach, M.: A generic theorem prover of CSP refinement. In: TACAS '05: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science (LNCS)*, vol. 1503, pp. 103–123. Springer-Verlag (2005)
97. Ivanicic, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient sat-based bounded model checking for software verification. Theoretical Computer Science **404**(3), 256–274 (2008). DOI http://dx.doi.org/10.1016/j.tcs.2008.03.013
98. Johnson, S., Bose, B.: DDD — a system for mechanized digital design derivation. In: ACM/SIGDA Workshop on Formal Methods in VLSI Design. Miami, Florida (1991). URL citeseer.ist.psu.edu/johnson97ddd.html
99. Josephs, M.B.: A state-based approach to communicating processes. Distributed Computing **3**(1), 9–18 (1988)
100. Joshi, R., Nelson, G., Randall, K.: Denali: a goal-directed superoptimizer. In: Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation. Berlin, Germany (2002)
101. Jr., E.M.C., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
102. Jussila, T., Niemel, I.: Parallel program verification using bmc. In: In: ECAI 2002 Workshop on Model Checking and Artificial Intelligence, pp. 59–66 (2002)
103. Kahlon, V., Gupta, A., Sinha, N.: Symbolic Model Checking of Concurrent Programs Using Partial Orders and On-the-Fly Transactions. In: CAV '06: Proceedings of the 18th international conference on Computer Aided Verification, pp. 286–299 (2006)
104. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In: CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification, pp. 398–413. Springer-Verlag, Berlin, Heidelberg (2009). DOI http://dx.doi.org/10.1007/978-3-642-02658-4_31
105. Kahng, A.B.: Design technology productivity in the DSM era (invited talk). In: ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation, pp. 443–448. ACM, New York, NY, USA (2001). DOI http://doi.acm.org/10.1145/370155.370510
106. Karfa, C., Mandal, C., Sarkar, D., Pentakota, S.R., Reade, C.: A formal verification method of scheduling in high-level synthesis. IEEE International Symposium on Quality Electronic Design **0**, 71–78 (2006). DOI http://doi.ieeecomputersociety.org/10.1109/ISQED.2006.10
107. Katz, S., Peled, D.: Defining conditional independence using collapses. Theor. Comput. Sci. **101**(2), 337–359 (1992). DOI http://dx.doi.org/10.1016/0304-3975(92)90054-J
108. Kauffmann, M., Boyer, R.: The Boyer-Moore theorem prover and its interactive enhancement. Computers and Mathematics with Applications **29**(2), 27–62 (1995)
109. Kelly, W., Pugh, W.: Finding legal reordering transformations using mappings. In: Proceedings of Languages and Compilers for Parallel Computing (1994)
110. Kern, C., Greenstreet, M.R.: Formal verification in hardware design: a survey. ACM Transactions on Design Automation of Electronic Systems (TODAES) **4**(2), 123–193 (1999). DOI http://doi.acm.org/10.1145/307988.307989
111. Kim, Y., Kopuri, S., Mansouri, N.: Automated formal verification of scheduling process using finite state machines with datapath (fsmd). In: ISQED '04: Proceedings of the 5th International Symposium on Quality Electronic Design, pp. 110–115. IEEE Computer Society, Washington, DC, USA (2004)

112. Koelbl, A., Burch, J.R., Pixley, C.: Memory modeling in ESL-RTL equivalence checking. In: DAC '07: Proceedings of the 44th annual conference on Design automation, pp. 205–209. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1278480.1278530
113. Kozen, D.: Kleene algebra with tests. ACM Transactions on Programming Langauges and Systems **19**(3), 427–443 (1997)
114. Kroening, D., Sharygina, N.: Formal verification of SystemC by automatic hardware/software partitioning. In: Proceedings of Third ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE) (2005)
115. Kundu, S., Ganai, M., Gupta, R.: Partial Order Reduction for Scalable Testing of SystemC TLM Designs. In: DAC '08: Proceedings of the 45th annual conference on Design Automation, pp. 936–941. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1391469.1391706
116. Kundu, S., Lerner, S., Gupta, R.: Automated Refinement Checking of Concurrent Systems. In: ICCAD '07: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design, pp. 318–325. IEEE Press, Piscataway, NJ, USA (2007)
117. Kundu, S., Lerner, S., Gupta, R.: Validating High-Level Synthesis. In: CAV '08: Proceedings of the 20th international conference on Computer Aided Verification, pp. 459–472. Springer, Princeton, NJ, USA (2008)
118. Kundu, S., Lerner, S., Gupta, R.: High-Level Verification. IPSJ Transactions on System LSI Design Methodology (2009). (Invited Paper)
119. Kundu, S., Lerner, S., Gupta, R.: Translation Validation in High-Level Synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **29**, 566 – 579 (2010)
120. Kundu, S., Tatlock, Z., Lerner, S.: Proving Optimizations Correct using Parameterized Program Equivalence. In: PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (2009)
121. Lacey, D., Jones, N.D., Wyk, E.V., Frederiksen, C.C.: Proving correctness of compiler optimizations by temporal logic. In: Proceedings of the 29th ACM Symposium on Principles of Programming Languages (2002)
122. Lahiri, S.K., Ball, T., Cook, B.: Predicate abstraction via symbolic decision procedures. In: CAV '05: Proceedings of the 17th International Conference on Computer Aided Verification, pp. 24–38 (2005)
123. Lam, M.: Software pipelining: an effective scheduling technique for VLIW machines. In: PLDI '88: Proceedings of the 1988 ACM SIGPLAN conference on Programming Language Design and Implementation (1988)
124. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). DOI http://doi.acm.org/10.1145/359545.359563
125. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess program. IEEE Transactions on Computers **28**(9), 690–691 (1979). DOI http://dx.doi.org/10.1109/TC.1979.1675439
126. Larsson, M.: Improving the result of high-level synthesis using interactive transformational design. In: TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics, pp. 299–314. Springer-Verlag, London, UK (1996)
127. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A framework for comparing models of computation. IEEE Transactions on CAD of Integrated Circuits and Systems **17**(12), 1217–1229 (1998)
128. Lee, J., Padua, D.A., Midkiff, S.P.: Basic compiler algorithms for parallel programs. SIGPLAN Not. **34**(8), 1–12 (1999). DOI http://doi.acm.org/10.1145/329366.301105
129. Lerda, F., Sinha, N., Theobald, M.: Symbolic model checking of software. Electronic Notes in Theoretical Computer Science **89**(3), 480–498 (2003). DOI:10.1016/S1571-0661(05)80008-8. URL http://www.sciencedirect.com/science/article/B75H1-4G6H70X-8/2/789a8a6cd28544600d61a7540d5a51b8. SoftMC 2003, Workshop on Software Model Checking (Satellite Workshop of CAV '03)
130. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. San Diego CA (2003)

131. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: PLDI '03: Proceedings of the 2003 ACM SIGPLAN conference on Programming Language Design and Implementation (2003)
132. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Long Beach CA (2005)
133. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (2005)
134. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (2006)
135. Levin, V., Palmer, R., Qadeer, S., Rajamani, S.K.: Sound transaction-based reduction without cycle detection. In: SPIN '05: Proceedings of the 12th international workshop on Model Checking Software, pp. 106–122 (2005)
136. Lin, Y.L.: Recent developments in high-level synthesis. ACM Transactions on Design Automation of Electronic Systems. **2**(1), 2–21 (1997). URL citeseer.ist.psu.edu/lin97recent.html
137. Ltd., F.S.E.: Failures-divergence refinement: FDR2 user manual. Oxford, England, June 2005.
138. Mansouri, N., Vemuri, R.: Automated correctness condition generation for formal verification of synthesized RTL designs. Formal Methods in System Design: An International Journal **16**(1), 59–91 (2000). URL citeseer.ist.psu.edu/mansouri99automated.html
139. Mazurkiewicz, A.: Trace theory. In: Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency, pp. 279–324. Springer-Verlag New York, Inc., New York, NY, USA (1987)
140. McCune, W.W.: Solutions of the Robbins problem. Journal of Automated Reasoning **19**(3), 262–276 (1997)
141. McFarland, M.C.: Formal analysis of correctness of behavioral transformations. Formal Methods in System Design **2**(3), 231–257 (1993). DOI http://dx.doi.org/10.1007/BF01384133
142. McFarland, M.C.: Formal verification of sequential hardware: A tutorial. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **12**(5), 654–663 (1993)
143. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Norwell, MA, USA (1993)
144. McMillan, K.L.: A methodology for hardware verification using compositional model checking. Science of Computer Programming **37**(1-3), 279–309 (2000)
145. Meftali, S., Vennin, J., Dekeyser, J.L.: A fast SystemC simulation methodology for Multi-Level IP/SoC design. In: IFIP International Workshop on IP Based SoC Design (2003)
146. Mendías, J.M., Hermida, R., Molina, M.C., Penalba, O.: Efficient verification of scheduling, allocation and binding in high-level synthesis. In: DSD '02: Proceedings of the Euromicro Symposium on Digital Systems Design, p. 308. IEEE Computer Society, Washington, DC, USA (2002)
147. Micheli, G.D.: Guest editorial: High-level synthesis of digital circuits. IEEE Transactions on Design Test **7**(5), 6–7 (1990)
148. Micheli, G.D.: Synthesis and Optimization of Digital Circuits. McGraw-Hill (1994)
149. Microelectronics, S.: Transaction Accurate Communication (TAC) platform (2005). www.greensocs.com/TACPackage
150. Moore, J.S.: Symbolic simulation: An acl2 approach. In: Proceedings of Formal Methods in Computer-Aided Design, pp. 334–350 (1998). URL citeseer.ist.psu.edu/moore98symbolic.html
151. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference (DAC'01) (2001). URL citeseer.ist.psu.edu/moskewicz01chaff.html
152. Moura, L.D., Bjrner, N.: Z3: An efficient smt solver. In: TACAS '08: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2008)

153. Moy, M., Maraninchi, Maillet-Contoz: Lussy: A toolbox for the analysis of Systems-on-a-Chip at the Transactional Level. In: Proceedings of International Conference on Application of Concurrency to System Design (ACSD) (2005)
154. Muchnick, S.: Advanced Compiler Design And Implementation. Morgan Kaufmann Publishers (1997)
155. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: CMC: A pragmatic approach to model checking real code. In: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (2002)
156. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 446–455. ACM, New York, NY, USA (2007)
157. Narasimhan, N.: Theorem proving guided development of formal assertions and their embedding in a high-level vlsi synthesis system. Ph.D. thesis, University of Cincinnati (1998)
158. Narasimhan, N., Teica, E., Radhakrishnan, R., Govindarajan, S., Vemuri, R.: Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis. Formal Methods in System Design **19**(3), 237–273 (2001). DOI http://dx.doi.org/10.1023/A:1011250531814
159. Narasimhan, N., Vemuri, R.: On the effectiveness of theorem proving guided discovery of formal assertions for a register allocator in a high-level synthesis system. In: J. Grundy, M. Newey (eds.) Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98, pp. 367–386. Springer-Verlag, Canberra, Australia (1998). URL citeseer.ist.psu.edu/341177.html
160. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI '00: Proceedings of the 2000 ACM SIGPLAN conference on Programming Language Design and Implementation (2000)
161. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems **1**(2), 245–257 (1979)
162. Nieuwenhuis, R., Oliveras, A.: Dpll(t) with exhaustive theory propagation and its application to difference logic. In: In CAV05 LNCS 3576, pp. 321–334. Springer (2005)
163. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: PVS: Combining specification, proof checking, and model checking. In: Proceedings of Computer-Aided Verification, CAV '96, *Lecture Notes in Computer Science*, vol. 1102, pp. 411–414. Springer-Verlag, New Brunswick, NJ (1996)
164. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In: D. Kapur (ed.) Proceedings of 11th International Conference on Automated Deduction (CADE), *Lecture Notes in Artificial Intelligence*, vol. 607, pp. 748–752. Springer-Verlag, Saratoga, NY (1992). DOI http://www.csl.sri.com/papers/cade92-pvs/
165. Paulson, L.C.: Isabelle: A generic theorem prover, *Lecure Notes in Computer Science*, vol. 828. Springer Verlag (1994)
166. Peled, D.: All from one, one for all: on model checking using representatives. In: CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification, pp. 409–423. Springer-Verlag, London, UK (1993)
167. Peled, D.: Ten years of partial order reduction. In: CAV '98: Proceedings of the international conference on Computer Aided Verification (1998)
168. Pfenning, F., Schurmann, C.: Sytsem description: Twelf – A meta-logical framework for deductive systems. In: H. Ganzinger (ed.) Proceedings of the 16th International Conference on Automated Deduction (CADE-16), *Lecture Notes in Artificial Intelligence*, vol. 1632, pp. 202–206. Springer-Verlag (1999)
169. Pnueli, A.: The temporal logic of programs. In: SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE Computer Society, Washington, DC, USA (1977). DOI http://dx.doi.org/10.1109/SFCS.1977.32
170. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science*, vol. 1384, pp. 151–166 (1998)

171. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM **8**, 4–13 (1992)
172. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS'05: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 93–107. Springer (2005)
173. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: Proceedings of the 5th Colloquium on International Symposium on Programming, pp. 337–351. Springer-Verlag, London, UK (1982)
174. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: CAV '05: Proceedings of the 17th international conference on Computer Aided Verification, pp. 82–97 (2005)
175. Radhakrishnan, R., Teica, E., Vermuri, R.: An approach to high-level synthesis system validation using formally verified transformations. In: HLDVT '00: Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00), p. 80. IEEE Computer Society, Washington, DC, USA (2000)
176. Rajan, S.P.: Correctness of transformations in high level synthesis. In: S.D. Johnson (ed.) CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications, pp. 597–603. Chiba, Japan (1995). URL citeseer.ist.psu.edu/rajan95correctness.html
177. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Transactions on Programing Language Systems **22**(2), 416–430 (2000). DOI http://doi.acm.org/10.1145/349214.349241
178. Rinard, M., Marinov, D.: Credible compilation. In: Proceedings of the FLoC Workshop Run-Time Result Verification (1999)
179. Rinard, M.C., Diniz, P.C.: Commutativity analysis: a new analysis framework for parallelizing compilers. In: PLDI '96: Proceedings of the 1996 ACM SIGPLAN conference on Programming Language Design and Implementation (1996)
180. Robinson, A., Voronkov, A. (eds.): Handbook of Automated Reasoning. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands (2001)
181. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M.H., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical compression for model-checking csp or how to check 1020 dining philosophers for deadlock. In: TACAS '95: Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems, pp. 133–152. Springer-Verlag, London, UK (1995)
182. Sadowski, C., Freund, S.N., Flanagan, C.: Singletrack: A dynamic determinism checker for multithreaded programs. In: ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems, pp. 394–409. Springer-Verlag, Berlin, Heidelberg (2009). DOI http://dx.doi.org/10.1007/978-3-642-00590-9_28
183. Sander, I., Jantsch, A.: System modeling and transformational design refinement in forsyde [formal system design]. IEEE Transactions on CAD of Integrated Circuits and Systems **23**(1), 17–32 (2004)
184. Sen, A., Garg, V.K.: Formal verification of simulation traces using computation slicing. IEEE Transactions on Computers (2007)
185. Sharp, R., Rasmussen, O.: The T-Ruby design system. Formal Methods in System Design: An International Journal **11**(3), 239–264 (1997). URL citeseer.ist.psu.edu/article/sharp95truby.html
186. Shyamasundar, R., Doucet, F., Gupta, R., Kruger, I.: Compositional reactive semantics of SystemC and verification with RuleBase. In: Proceedings of the GM R&D Workshop (2007)
187. Sittampalam, G., de Moor, O., Larsen, K.F.: Incremental execution of transformation specifications. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Venice Italy (2004)
188. Slaney, J.K., Fujita, M., Stickel, M.E.: Automated reasoning and exhaustive search: Quasigroup existence problems. Compuers and Mathematics with Applications **29**(2), 115–132 (1995)

189. Steffen, B.: Data flow analysis as model checking. In: T. Ito, A. Meyer (eds.) Theoretical Aspects of Computer Science (TACS), Sendai (Japan), *Lecture Notes in Computer Science (LNCS)*, vol. 526, pp. 346–364. Springer-Verlag (1991)
190. Stoller, S.D.: Model-checking multi-threaded distributed java programs. In: International Journal on Software Tools for Technology Transfer, pp. 224–244. Springer (2000)
191. Stoller, S.D., Cohen, E.: Optimistic synchronization-based state-space reduction. Formal Methods in System Design **28**(3), 263–289 (2006). DOI http://dx.doi.org/10.1007/s10703-006-0003-4
192. Swan, S.: Systemc transaction level models and rtl verification. In: DAC '06: Proceedings of the 43rd annual conference on Design automation, pp. 90–92. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1146909.1146937
193. Systems, C.D.: Slec system. www.calypto.com/slecsystem.php
194. Technologies, E.: Scade design verifier. www.esterel-technologies.com/
195. Tej, H., Wolff, B.: A corrected failure divergence model for csp in isabelle/hol. In: FME '97: Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods, pp. 318–337. Springer-Verlag, London, UK (1997)
196. The VIS Group, R.K.B., Sangiovanni-Vincentelli, A.L.: Vis: A system for verification and synthesis. Tech. Rep. UCB/ERL M95/104, EECS Department, University of California, Berkeley (1995). URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1995/2923.html
197. Tjiang, S.W.K., Hennessy, J.L.: Sharlit – A tool for building optimizers. In: PLDI '92: Proceedings of the 1992 ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 82–93 (1992)
198. Tristan, J.B., Leroy, X.: Verified validation of lazy code motion. In: Proceedings of the 35th ACM Symposium on Principles of Programming Languages (2008)
199. Tristan, J.B., Leroy, X.: Formal verification of translation validators: a case study on instruction scheduling optimizations. In: PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (2009)
200. Valmari, A.: Stubborn sets for reduced state space generation. In: Proceedings of the 10th International Conference on Applications and Theory of Petri Nets, pp. 491–515. Springer-Verlag, London, UK (1991)
201. Vardi, M.Y.: Formal techniques for SystemC verification. In: DAC '07: Proceedings of the 44th annual conference on Design Automation (2007)
202. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: Automated Software Engineering Journal, pp. 3–12. Press (2000)
203. Walker, R., Camposano, R.: A Survey of High-Level Synthesis Systems. Kluwer Academic Publishers, Boston, MA, USA (1991)
204. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: FM '09: Proceedings of the 2nd World Congress on Formal Methods, pp. 256–272. Springer-Verlag, Berlin, Heidelberg (2009). DOI http://dx.doi.org/10.1007/978-3-642-05089-3_17
205. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: TACAS '08: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 382–396 (2008)
206. Wang, L., Stoller, S.D.: Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 137–146. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1122971.1122993
207. Whitfield, D.L., Soffa, M.L.: An approach for exploring code improving transformations. ACM Transactions on Programming Languages and Systems **19**(6), 1053–1084 (1997)
208. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: Memory-model-sensitive data race analysis. In: ICFEM '04: Proceedings of 6th International Conference on Formal Engineering Methods, pp. 30–45 (2004)

209. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: A framework for axiomatic and executable specifications of memory consistency models. In: International Parallel and Distributed Processing Symposium (IPDPS) (2003)
210. Yi, K., Harrison III, W.L.: Automatic generation and management of interprocedural program analyses. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 246–259 (1993)
211. Young, W.D.: A mechanically verified code generator. Journal of Automated Reasoning **5**(4), 493–518 (1989)
212. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B.: VOC: A methodology for the translation validation of optimizing compilers. Journal of Universal Computer Science **9**(3), 223–247 (2003)
213. Zuck, L., Pnueli, A., Goldberg, B., Barrett, C., Fang, Y., Hu, Y.: Translation and run-time validation of loop transformations. Formal Methods in System Design **27**(3), 335–360 (2005). DOI http://dx.doi.org/10.1007/s10703-005-3402-z

Index