Combining complementary formal verification strategies to improve performance and accuracy

David R. Owen
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation
Owen, David R., "Combining complementary formal verification strategies to improve performance and accuracy" (2007). Graduate Theses, Dissertations, and Problem Reports. 2583.
https://researchrepository.wvu.edu/etd/2583

This Dissertation is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Dissertation in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself. This Dissertation has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.
Combining Complementary Formal Verification Strategies
to Improve Performance and Accuracy

David R. Owen

Dissertation submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer and Information Science

Bojan Cukic, Ph.D., Co-Chair
Tim Menzies, Ph.D., Co-Chair
Edgar Fuller, Ph.D.
Katerina Goseva-Popstojanova, Ph.D.
Arun Ross, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2007

Keywords: D.2 Software Engineering, D.2.1 Requirements / Specifications,
D.2.4 Software / Program Verification, D.2.5 Testing and Debugging,
D.4.5 Reliability, F.3.1 Specifying and Verifying and Reasoning about Programs

Copyright 2007 David R. Owen
Abstract

Combining Complementary Formal Verification Strategies to Improve Performance and Accuracy

David R. Owen

Software is increasingly complex and is used in increasingly critical applications. Sophisticated techniques are available for verifying that software systems work correctly, but these techniques can be very difficult and expensive to use. Researchers have developed tools to automatically verify software models, but using these tools can still be very costly, in terms of manual effort and expertise required to build accurate models and to formally specify required properties, and also in terms of the time and memory required to run these tools. Much work has been done to simplify the process of building software models and to improve the performance of verification tools, resulting in a variety of different modeling languages, each with features designed to reduce effort or improve performance for certain types of input models, and a range of verification tools, each with a different set of strategies available for reducing time and memory requirements.

It can be difficult to determine which verification strategy is best for a particular software system. Others have observed complementary relationships between tools and have argued that there is no single best tool—that as users’ needs change the choice of tool should change as well. This dissertation provides further evidence for complementary relationships between verification tools, specifically considering tools available for specifications of synchronous software systems written in the Software Cost Reduction (SCR) modeling language. We show how verification tools and their associated modeling languages may be complementary in terms of both accuracy and performance. Rather than providing guidance for users deciding between tools, we argue that a verification strategy combining results from multiple tools will yield the most accurate results, i.e., the results worthy of the greatest confidence, and will in most cases perform better, requiring less time and memory, than a strategy based on a single tool.

This dissertation presents several studies in which the use of multiple verification tools resulted in improved accuracy and performance. In some cases the use of a single tool would have produced incorrect results, giving no indication to the user that the tool had been used incorrectly. The use of a second tool, producing results inconsistent with the first, led to a better understanding of both tools and greater confidence in the overall verification result. Further studies show how an efficient debugging tool, based on random search, and a verification tool can be used together so that average time and memory requirements are greatly reduced, and so that performance is much less sensitive to minor changes in the input model. We then discuss in detail a larger case study that produced experimental results consistent with these smaller studies, showing how four verification tools can be used together effectively to verify software specifications written in the SCR modeling language.
Contents

1 Introduction ............................................................... 1
   1.1 Background ......................................................... 1
   1.2 Combining Complementary Formal Verification Strategies ....... 2
   1.3 Overview of Previous Work ....................................... 3
   1.4 Contributions ...................................................... 6
   1.5 Organization ...................................................... 7

2 Related Work ............................................................ 8
   2.1 Software Testability ............................................... 9
      2.1.1 General Definitions of Testability ....................... 9
      2.1.2 Testability as Reachability ............................... 10
   2.2 Verification ....................................................... 11
      2.2.1 Formal Methods ............................................ 11
      2.2.2 Model Checking ............................................ 13
      2.2.3 Complete Strategies for Scalable Model Checking ........ 14
      2.2.4 Incomplete Strategies for Scalable Testing of Formal Models ... 16
      2.2.5 Model Checking and Testing ............................... 17
      2.2.6 Random Search Applied to Formal Models ................ 18
   2.3 Random Search .................................................... 19
      2.3.1 Benefits and Disadvantages of Randomized Algorithms ... 19
      2.3.2 Problem Structures Favorable to Random Search .......... 20
   2.4 Summary ........................................................... 22

3 Motivation ............................................................... 24
   3.1 Using Multiple Verification Tools to Improve Accuracy ........ 24
      3.1.1 Inconsistent Results from Two Symbolic Model Checkers ... 25
      3.1.2 Inconsistent Results from Model Checking and Random Search ... 27
      3.1.3 Inconsistent Results from an Invariant Checker and a Model Checker ... 29
   3.2 Using Multiple Verification Tools to Improve Performance ...... 30
      3.2.1 Combination of Explicit-State Model Checking and Random Search ... 31
      3.2.2 Combination of Symbolic Model Checking and Random Search .... 32
      3.2.3 Combining Tools to Improve Robustness .................. 34
   3.3 Summary ........................................................... 39
List of Figures

1.1  A series of random search runs plotted to show saturation—many unique states explored initially but later only repeat states explored. .................................................. 4

1.2  Time and memory required for Lurch and SPIN (5 modes) running on fault-seeded protocol model with increasing number of processes. ................................. 5

2.1  Hard problems exhibit a phase transition (left); this can be exploited by a simple strategy (right). .......................................................... 21

3.1  Inconsistent outputs from Cadence SMV (top) and NuSMV (bottom) running on the same input model. .......................................................... 25

3.2  Output from SPIN running on a model generated from the same fault-seeded specification used to generate the models for which Cadence SMV and NuSMV outputs are shown in figure 3.1. .......................................................... 26

3.3  Inconsistent outputs from SPIN (top) and Lurch (bottom) running on the same input model. .......................................................... 27

3.4  Simulator log produced by stepping through execution trace output by Lurch. .......................................................... 28

3.5  Inconsistent outputs from Salsa (top) and SPIN (bottom) running on the same input model. .......................................................... 29

3.6  Dining philosopher processes: with loop (top), without loop (bottom). .......................................................... 29

3.7  Time (s) and memory (MB) required for Lurch and SPIN running on fault-seeded version of the leader election protocol input model. (Combined strategy complete but otherwise identical to Lurch.) .................. 38

4.1  Cruise control system finite-state machine. .......................................................... 43

4.2  SCR specification for cruise control system. .......................................................... 44

4.3  Salsa version of cruise control specification. .......................................................... 46

4.4  Salsa output for correct (top) and fault-seeded (bottom) versions of the cruise control system model. .......................................................... 47

4.5  SMV version of cruise control specification (top) and changes to assertion definitions needed for NuSMV (bottom). .......................................................... 49

4.6  SMV output for correct (top) and fault-seeded (bottom) versions of the cruise control system model. .......................................................... 50

4.7  NuSMV output for correct (top) and fault-seeded (bottom) versions of the cruise control system model. .......................................................... 50

4.8  SPIN version of the cruise control specification. .......................................................... 51
4.9 SPIN output for correct (top) and fault-seeded (bottom) versions of the cruise control system model.

4.10 Lurch Input Model representing Dekker’s solution to the two-process mutual exclusion problem (translated from a model written for SPIN in [41]).

4.11 Lurch’s basic random search procedure.

4.12 step function modified for synchronous execution of finite-state machines.

4.13 main function modified for hierarchical execution of finite-state machines.

4.14 Lurch version of the cruise control specification.

4.15 Lurch output for correct (top) and fault-seeded (bottom) versions of the cruise control system model.

5.1 PACS mode finite-state machine.

5.2 Mutation operator(s) and the number of specifications generated for each (pair).

5.3 Summary of verification results for all tools except Salsa—sets of fault-seeded specifications for which each tool detected property violations.

6.1 Specifications plotted to show maximum and minimum time requirements for any tool.

6.2 Specifications plotted to show maximum and minimum memory requirements for any tool.

6.3 Combined strategy exploiting complementary variations in performance and accuracy. (Baseline complete and single-tool complete strategies enclosed in dotted boxes.)

6.4 Comparison of results for baseline complete strategy, single tool complete strategy, and combined strategy. (Medians are marked by large shapes.)

7.1 A conceptual model of the challenges involved in using automatic verification tools.
List of Tables

3.1 Time (s) required for Lurch, SPIN and a simple combined strategy, running on fault-seeded versions of the security system models. 31
3.2 Memory (MB) required for Lurch, SPIN and a simple combined strategy, running on fault-seeded versions of the security system model. 31
3.3 Time (s) required for Lurch, NuSMV, and a simple combined strategy, running on fault-seeded versions of the flight guidance system model. 34
3.4 Time (s) required for SPIN, NuSMV, and combined strategy, running until deadlock detected, on two versions of the dining philosophers problem. 35
3.5 Memory (MB) required for SPIN, NuSMV, and combined strategy, running until deadlock detected, on two versions of the dining philosophers problem. 36
3.6 Time (s) required for Lurch, SPIN, and combined strategy, running on fault-seeded versions of the leader election protocol model. 36
3.7 Memory (MB) required for Lurch, SPIN, and combined strategy, running on fault-seeded versions of the leader election protocol model. 37

5.1 Mutation operators used to generated fault-seeded versions of the PACS SCR specification. 68
5.2 Lurch results on fault-seeded PACS specifications: number of times violations detected vs. search runs, number of specifications in parentheses. 73
5.3 Average time and memory, and standard deviation values for averages, required by SPIN for fault-seeded specifications, with settings adjusted to run in three ways. 75
5.4 Summary of verification results for non-equivalent mutants (average time in seconds and standard deviation for time values). 77
5.5 Summary of verification results for non-equivalent mutants (average memory in megabytes and standard deviation for memory values). 77

6.1 Average time (s) required by combinations of tools. 84
6.2 Average memory (MB) required by combinations of tools. 84
6.3 Average time (s) required by individual tools for sets of specifications distinguished by Salsa results (only statistically significant results shown). 85
6.4 Average memory (MB) required by individual tools for sets of specifications distinguished by Salsa results (only statistically significant results shown). 86
6.5 Average time (s) required by individual tools for sets of specifications distinguished by mutation operator(s) (only statistically significant results shown). 88
6.6 Average memory (MB) required by individual tools for sets of specifications distinguished by mutation operator(s) (only statistically significant results shown). 89
6.7 Average time (s) required by individual tools for sets of specifications distinguished by the number of mutations (only statistically significant results shown). .......................... 91
6.8 Average memory (MB) required by individual tools for sets of specifications distinguished by the number of mutations (only statistically significant results shown). .......................... 91
6.9 Number of specifications in which property violations were detected by tools at different stages of the flowchart shown in figure 6.3. ................................................................. 98
6.10 Average time required by baseline, single tool and combined verification strategies running on all, equivalent and nonequivalent mutant specifications. ................................................. 99
6.11 Average memory required by baseline, single tool and combined verification strategies running on all, equivalent and nonequivalent mutant specifications. ................................................. 99
Acknowledgments

I am very grateful to Tim Menzies and Bojan Cukic, for encouragement, ideas, and support throughout my time as a graduate student at West Virginia University. Tim suggested many of the ideas behind the work presented in this dissertation, including the use of random search to explore software models and even the name “Lurch.” Bojan offered much advice on how to organize and present these ideas to the software engineering research community.

I am thankful to my fellow graduate students in computer science at WVU, especially Dejan Desovski and Jon Crowell. Dejan provided the PACS specification, the SCR Toolset and Salsa software, through his contacts at the Naval Research Laboratory, Constance Heitmeyer, Ramesh Bharadwaj, and others. Jon encouraged me to come to West Virginia in the first place and provided lots of healthy competition as we worked on our Master’s degrees together.

I am grateful also for my professors at WVU, including Mark Shereshevsky, for his course in the theory of computation; K. Subramani and Elaine Eschen, for algorithms courses I survived and learned a lot (including LaTeX) from; Katerina Goseva-Popstojanova for her course in distributed systems; and to Arun Ross and Eddie Fuller for serving on my Ph.D. committee.

Many others contributed to the research presented in this dissertation: Mats Heimdahl and Jimin Gao, at the University of Minnesota; John Powell, Martin Feather and Allen Nikora, at the NASA Jet Propulsion Laboratory; Willem Visser, Charles Pecheur and John Penix, at the NASA Ames Research Center; Ken McGill, Wes Deadrick, Marcus Fisher and Lisa Montgomery, at the WVU/NASA IV&V Facility. Funding was provided by the NASA Office of Safety and Mission Assurance and IV&V Facility, through four research initiatives: A Spectrum of IV&V Modeling Techniques, Integrating Model Checking and Procedural Languages, A Compositional Approach to Validation of Formal Models, and Formal Methods Analysis Framework.

In addition, I am thankful to Frank Gmeindl, Jitesh Gandhi, Nick Hein, Rodney Queen, Mary
Daugherty, Rob Best and Dan McCaugherty, for friendship, understanding, flexibility and support during the time I was both working for ProLogic and working on my Ph.D. For help with vim, bash, sed, awk, and all things Linux, I thank Aron Griffis and everyone else on #noise.

Finally, I am very grateful to my wife Gretta, for patience, flexibility, encouragement and her sense of humor, and to our Lord Jesus Christ, “in whom are hidden all the treasures of wisdom and knowledge.”
Chapter 1

Introduction

Chapter 1 introduces the topic of software verification and summarizes our goal of addressing verification challenges by combining complementary strategies. Section 1.1 summarizes the background for our work, describing trade-offs involved in strategies for improving the scalability of automatic verification methods. Section 1.2 briefly explains the thesis of this dissertation, that complementary formal verification strategies can be combined so that the accuracy and performance of the overall verification strategy is improved, compared to any single strategy used alone. Section 1.3 summarizes our previous research work on random search as an efficient way of detecting errors in software models. In section 1.4 we state this dissertation’s intended contributions to related research. Section 1.5 describes the structure of the rest of the dissertation.

1.1 Background

Software is increasingly complex and is used in increasingly critical applications. Verifying that software systems work correctly thus grows both more difficult and more necessary. Researchers have developed sophisticated mathematical methods for proving the correctness of software, but these techniques are costly. Because of the expertise and manual effort required, and because of the time and computing resources involved, most software developers do not consider formal verification a practical option. Software quality is assessed primarily through testing, relatively late in the development process. As a result fewer errors are detected, and errors that are detected are very expensive to fix.

Many researchers have worked to make formal verification methods more practical, developing automated tools capable of determining whether a simplified model of a software system conforms
to a set of user-defined or generic correctness properties. These model checking tools are gaining popularity, but can require large amounts of time and memory to verify even modest-sized input models. Researchers therefore continue to look for ways to improve the scalability of automated tools for software verification.

There are a variety of strategies, implemented in various tools, to improve the scalability of model checking. Some strategies retain the ability to fully verify—to mathematically prove—the correctness of the input model, while other strategies limit the scope of verification to improve scalability. Scope may be limited in various ways. For example, the types of input models or properties that can be verified may be restricted; properties proved or errors detected may have to be validated manually or with another tool. These various strategies for improving scalability, implemented aggressively in different model checking tools, so that time and memory use is decreased as much as possible, tend to make the tools complementary to each other. That is, for a given input model and property specification, one verification strategy may be much more effective than another, but for a different input model and property specification the first strategy may be much less effective than the second. Also, different strategies emphasize different goals. One may emphasize speed but require much more memory, while another requires less memory but is slower; one strategy may provide simpler and more useful information for correcting detected errors but may require much more time and memory to produce such information.

1.2 Combining Complementary Formal Verification Strategies

Others have recognized complementary relationships between modeling languages and verification tools and recommended the use of a verification framework in which a variety of strategies are available, so that, as software models and users’ needs change, the right strategy is available at the right time [14]. In this dissertation, we consider the basic verification challenge, to determine whether a software model is fully consistent with a formal specification of correctness properties, and argue that the best verification strategy is, in a sense, to use all of the strategies, all of the time.

Using multiple complementary verification tools on the same input model yields two types of advantages. First, hidden assumptions and idiosyncrasies of the tools are brought to light, so that individual tools may be used more effectively and the user has reason for increased confidence in
the results. This is especially important when tools are used in a framework including elements not developed by the user, e.g., when automatic translators or modeling tools are used, as in the experiments presented later in this dissertation. If automated verification tools are to be practical and cost-effective for software developers in industry, expert knowledge of the inner workings of a tool must not be a prerequisite for use. If multiple tools, implementing somewhat different verification strategies, are used on the same input model and yield consistent results, the user can be much more confident in the results, without knowing a great deal about the different verification strategies used by the tools.

A second advantage of multiple-tool verification strategies is improved performance. Tools may be cascaded in such a way that input models difficult for one tool are passed on to another. If tools’ performance is sufficiently complementary, most input models will be easy for at least one tool even if they are difficult for one or more others. In addition, cascading multiple tools may result in an overall verification strategy much less sensitive, in terms of time and memory requirements, to minor changes in the input model.

1.3 Overview of Previous Work

The research work presented in this dissertation began with work on a random search strategy for efficiently detecting errors in software models [58]. This strategy was as follows: start at the initial state of the model; choose the next state at random from those possible; quit when no next state is possible or a user-specified depth limit is reached. Results from this simple random strategy followed a pattern: at first, many unique states were explored, but soon the number of unique states explored reached a peak and stopped increasing; from then on the same states were explored over and over.

When allowed to run to this saturation point, at which the proportion of unique states drops off, the random search produces surprisingly consistent results. We concluded that random search, in spite of its general (worst-case) unreliability, was sufficiently consistent to pursue as an efficient strategy for detecting errors in software models [54, 59, 62]. Figure 1.1 shows results from a large number of random search runs on a simplified TCP protocol model from [69], comparing the number of unique states explored to the total number of states explored. All search runs reach
saturation relatively quickly, after exploring 28 unique states.

Further work on random search led to the development of Lurch, a tool for detecting violations of generic and user-defined logical properties in finite-state machine models like those checked by other automated verification tools [61]. We compared Lurch’s performance to the widely used verification tools SPIN [41] and Cadence SMV [49] and found that with random search it is possible in many cases to detect property violations far more quickly and with orders of magnitude less memory required. For example, figure 1.2 shows time and memory required for Lurch and SPIN running on a scalable protocol model.

For the experiment represented by figure 1.2, SPIN was run in 5 different modes, each using different options for reducing memory requirements. Data points are shown for verification runs in which SPIN required less than 256 megabytes of memory. With default options, SPIN can detect a fault in models with up to 10 processes. Using much more time-consuming memory compression options, SPIN’s range can be extended to 14 processes. But Lurch is able to detect a fault very quickly in models with up to 26 processes. Beyond 26, Lurch’s results were inconsistent, but models of that size are far too large for verification using SPIN. The results plotted here are for a fault-seeded version of the multi-process leader election protocol model from [40]. A version of
Figure 1.2: Time and memory required for Lurch and SPIN (5 modes) running on fault-seeded protocol model with increasing number of processes.
this model with different seeded faults was used in the experiments described in section 3.2.3.

We also showed how Lurch could be used together with a conventional verification tool to greatly improve average verification performance: use Lurch first for a relatively short time and only run the verification tool on input models for which Lurch finds no property violations [63]. In addition, random search can be used to confirm the results of model checking. This might not be expected, since a model checker carries out a complete exploration of all the behavior represented by the input model—how could a random sample include any additional behavior? However, in real-world applications of model checking, assumptions are often made about the structure of the input model in order to improve performance. Random search of an input model with fewer or different assumptions can thus be used as a sanity check on the verification results produced by a model checker [60].

The work presented in this dissertation places the complementary relationships between random search and model checking in a broader context, considering complementary relationships between a variety of verification strategies including random search, different types of model checking, and a specialized tool for proving invariant properties. Just as the combination of random search and model checking improves accuracy and performance, further improvements are possible by combining additional strategies.

1.4 Contributions

The primary contribution of this dissertation is to show how a variety of verification tools may be used together in a single, generally applicable verification strategy based on complementary relationships between tools. Complementary relationships considered include both complementary scope—different tools are capable of different things—and complementary performance—different tools do the same thing, but in some cases one performs much better than another. Our proposed multiple-tool verification strategy can be applied directly to verification of specifications of synchronous software systems written in the Software Cost Reduction (SCR) modeling language. In addition, we argue that, in general, diverse strategies for improving the scalability of model checking may be integrated to produce a single strategy that is both more scalable and more reliable. Improved reliability is the result of insight into the use of each tool gained by comparing
the results from different tools.

Additional unique contributions made by this dissertation and related research work include:

1. Lurch, a random search tool for efficiently detecting errors in software models.
3. Guidance in correct use of the model checkers SPIN and NuSMV to verify software specifications written in the SCR modeling language.

1.5 Organization

Chapter 2 summarizes related work in the area of software testability, verification, and strategies for improving the scalability of automated verification tools. Random search, one focus of our work, is presented as a scalable strategy for detecting errors in formal software models. Next, chapter 3 gives six specific examples to illustrate how using multiple tools can result in more accurate and more efficient verification. Chapter 4 describes in detail all of the tools used in our experiments, paying special attention to our random search tool, Lurch. Chapter 5 explains how we carried out our experiments—how fault-seeded input models were generated and how each tool was used to check these models. In chapter 6, we consider our experimental results from different perspectives to better understand complementary relationships between tools; we then propose a generally applicable multiple-tool verification strategy based on those complementary relationships and compare results from the multiple-tool strategy to results produced by a single tool. Finally, chapter 7 summarizes our experimental results and conclusions and gives suggestions for future work.
Chapter 2

Related Work

Chapter 2 summarizes others’ work related to the research presented in this dissertation. Section 2.1 describes software testability, the idea that one important attribute of software is whether defects tend to be hidden or exposed to testing methods. Here testability is viewed as relative to the testing method: a single software artifact may tend to hide defects from one testing method but tend to expose defects to another. A particular software artifact will thus be most testable by a testing strategy combining complementary methods.

Section 2.2 summarizes current research in the area of software verification, focusing on automated techniques implementing formal verification methods, i.e., formal methods. Model checking, the most widely used automatic formal verification method, uses a finite-state model to represent the software artifact to be verified. Unfortunately the number of states (and therefore the amount of memory) required to represent even modest-sized software models tends to be very large. Because of this state-space explosion, much of the research in model checking centers on scalability. The discussion of scalability below is divided into two sections: first, complete techniques, which maintain verification accuracy, because a complete exploration of the state space is carried out; and second, incomplete techniques, which explore only a sample of the state space.

A major part of the research presented in this dissertation is concerned with one incomplete technique for debugging formal models, the random search procedure implemented in the Lurch tool, described in detail in chapter 4. Related to this, section 2.3 below summarizes others’ work to explain the success of random search strategies in many different domains, presenting ideas about the structure of complex problems’ search spaces that make them amenable to a randomized solution strategy.
2.1 Software Testability

Section 2.1 gives definitions of software testability from several different sources. The focus of this dissertation, that complementary methods of detecting software faults can be used together to create a more robust and effective overall strategy, is consistent with the view of testing as a search problem and with the corresponding view of testability as reachability. Faults within a software artifact will be detected most easily by a combination of complementary detection methods.

2.1.1 General Definitions of Testability

In any engineering discipline, accurate and meaningful measurement is the key to controlling projects, choosing among alternative strategies, and improving quality over time [15]. This is obviously true in engineering disciplines involving the physical sciences, but has been controversial in software engineering. Some researchers believe that “important software attributes like dependability, quality, usability and maintainability are simply not quantifiable” [20]. On the other hand, there are those who believe that, if the most important attributes are unmeasurable, we should be unsatisfied with the state of the art. We ought to use whatever imperfect measurement techniques are available to advance our understanding of software attributes and to improve measurement techniques in the process [15, 20].

Throughout the software life-cycle the concern of developers generally shifts from achieving quality to assessing quality. Surprisingly, it often costs even more to assess quality than to achieve it in the first place [22]. Because of this, the testability (a measure of how difficult it is to assess the quality) of software is extremely important. According to the IEEE Glossary of Software Engineering Terminology [24], testability is more formally defined as “the degree to which a system of components facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.” Voas and Miller [71], and later Bertolino and Stringini [5] argue for a similar but perhaps more practical definition, that testability is “the probability that the program will fail under test if it contains at least one fault.”

Note that in [20] the same point is being made that we make here; we have simply re-worded the beginning of the sentence.
2.1.2 Testability as Reachability

It is reasonable to assume that, the greater the proportion of a program’s execution space is exercised by testing, the more likely a fault will be found and cause the test to fail (if some fault exists). That is, a highly testable program is one for which a relatively small number of tests tell us a relatively large amount. This has been called the reachability view of software testing and testability [53].

Consider a graph representing all possible executions of a program, with a fault as some location in the graph, and a test execution as some pathway through the graph. Testability can be thought of as the probability that the pathway reaches the fault. That is, for a highly testable program the probability that a given pathway reaches a given fault will be high. In previous work we presented a technique for measuring how much of a program graph can be reached by a series of test runs and how quickly that happens (the sum of the length of the runs) [58]. In that study, we proposed two values as indicators of testability: first, the amount of the program reached by our tests, and second, the number of tests required to reach it. We showed how the structure of programs may in some cases be manipulated to increase these values and thus make a program more testable [58].

One important point about this definition of testability, relevant to the work presented in this dissertation, is that it is a relative definition. That is, the degree to which a particular piece of software is testable depends on the test strategy chosen. Taking the reachability view defined above, a program is more testable, by a particular testing strategy, if that strategy is able to reach a larger portion of the program behavior or requires a smaller number of tests. Using this definition, we expect that programs will be more testable by combinations of complementary testing strategies than by one strategy alone.

The method of measuring testability (relative to a particular testing strategy) used in our previous work was based on the idea of saturation [54, 58]. As mentioned in the previous chapter and illustrated in figure 1.1, we tracked the progress of a random search procedure used to explore software models and found an interesting pattern. Initially a great deal of new behavior was explored, but soon the same redundant behavior was explored over and over, so that if we plotted new information found by the search versus time, the plot would rise quickly and then saturate,
tapering off to a nearly level plateau. As stated in the previous chapter, this behavior is key to the (perhaps surprisingly) consistent results produced by random search. In addition, it provides an informal measure of relative testability: given a particular testing strategy, a more testable program yields a faster rise to a higher plateau; or, for a particular program, that program is more testable by a testing strategy that results in a faster rise to a higher plateau.

2.2 Verification

Section 2.2 provides an overview of related work in the area of software verification, describing advantages and disadvantages of powerful but expensive formal methods for proving the correctness of abstract models of software. Next, model checking is described briefly; this is a more specific area within formal methods research in which software models are represented by finite-state machines and automated tools are available to verify the correctness of these models.

Section 2.2 continues with short explanations of several strategies used to improve the scalability of model checking, so that larger and more realistic software models can be verified; or, if verification is not possible, so that complex faults can be detected that would be extremely difficult to find manually. Finally, random search is presented as a technique to improve the scalability of model checking, looking ahead to section 2.3 and later chapters in which the Lurch random search tool is used in conjunction with other verification tools on various software models.

2.2.1 Formal Methods

One important area of software testing and verification encompasses a set of techniques known collectively as formal methods. The goal of formal methods in software engineering is to mathematically prove that a program is correct, with respect to a specification of exactly what it should do. To do this manually, even for very simple programs, is very difficult (and error prone). Furthermore, in general it is not possible to automatically verify arbitrary programs (this is shown, for example, by Turing’s famous halting problem [68]).

Still, there are a variety of techniques and tools available to aid manual verification or to automatically verify abstract models restricted to a subset of the kinds of behaviors found in arbitrary programs. In fact, Wolper [75] argues that the key to making any verification approach truly for-
One is the availability of practical tools to automate the process. Without such tools the amount of error-prone manual effort required seriously undermines any claim of formal correctness.

Where formal methods have been used in software development, there is little doubt they have improved software safety and reliability, e.g., [40]. Unfortunately, it can be very difficult, time-consuming (and therefore expensive) to employ formal methods. In previous work we have categorized the cost of software development processes that include formal methods as follows [54]:

1. The cost of writing the formal model and property specification: often, a combination of Ph.D.-level mathematical expertise and detailed knowledge of the system and application domain is required to write an accurate formal model of the system, necessary properties, and relevant environmental factors.

2. The cost of running automated tools: for realistic models of large systems, the time and memory required by formal methods tools can be huge. In addition, different tools have their own input languages and their own unique features carefully chosen to support algorithms implemented in the tool. Because of this a detailed knowledge of the tool and its configuration options may be required.

3. The cost of rewriting the model: for realistic systems, often several rounds of abstraction are needed to produce a model suitable for the tool being used. To do these abstractions requires great care, so that relevant behavior is not hidden.

Many researchers have worked to reduce these costs using a variety of methods including the design of special restricted modeling languages [37,73], tools to aid in developing models directly from program source code [26, 27, 43], and optimizations that exploit models’ symmetry [13, 17]. Much progress has been made to reduce the cost of writing (and rewriting) formal models and property specifications, mostly by improving tools and their user interfaces. In some cases special features and constructs have been added to support particular domains, such as a variable type representing a communication channel in a distributed system [41].

In practice however, the cost of writing, running, and rewriting formal models still severely limits the use of formal methods. While the writing cost can, at least in principle, be solved using simplified modeling languages and tools for straightforward specification of properties, the high
running cost remains. And the rewriting cost also remains, because it is a consequence of the high running cost; that is, if the resource requirements could be made manageable, it would not be necessary to continually rewrite models to make them small enough to be verified with available tools.

Related to the high running cost of formal methods, in some cases the range of properties that can be verified is limited in order to improve the scalability and simplicity of a tool. For example, many techniques scale to large systems but cannot detect liveness property violations, which requires cycle detection [26, 27, 66]. And even complete techniques may use optimizations limiting the scope of properties that can be verified [41]; these optimizations must be turned off, sacrificing scalability, in order to verify the full range of properties.

2.2.2 Model Checking

Within the area of formal methods, model checking tools are probably the most widely used automated tools. Model checking tools carry out an exhaustive exploration of the behavior represented by an abstract program model to check for consistency with a specification of desired properties [44]. The model is written as a finite-state concurrent system, or set of communicating finite-state machines (two names for the same thing, not two possible ways to write the model), and the specification is written in a form called temporal logic [12, 44].

Automatic verification by model checking has been shown by researchers to be effective in many domains including computer hardware design, networking, security and telecommunications protocols, automated control systems and others [11, 12, 40]. More recently, model checking has been used in several safety critical NASA projects [25, 34, 42]. In addition, Microsoft researchers have developed a proprietary model checking framework for use on critical components of Windows [4].

The finite-state concurrent system formalism adds no computational power to the familiar finite-state machine model, but it is a convenient way of describing software systems with concurrency. The system model is made up of a set of finite-state transducers, which communicate with each other by means of (a finite amount of) shared memory.

Temporal logic, in which properties to be verified by a model checker are written, is an extension of Boolean logic, adding time-related operators like always or eventually. A temporal logic
truth claim can be converted into a Büchi automaton, which is similar to a finite-state transducer except that acceptance is not defined in terms of a set of accept states, but a set of acceptance cycles. A Büchi automaton accepts an input when that input causes it to remain in an acceptance cycle indefinitely. Büchi automata are needed to represent temporal logic claims because they can involve infinite input strings. For more on finite-state machines, transducers, concurrent systems, and Büchi automata see, e.g., [39, 68].

As a model checker explores the behavior of an input model, if a state or cycle is detected representing a violation of a user-specified temporal logic property (or certain generic properties defined by the model checker), a counter example trace is output by the model checker. This is a state-by-state path through the behavior of the model showing exactly how the state or cycle representing the property violation was reached. The ability to produce a counter example trace makes model checkers very useful, not only for verifying correctness but for guiding users in correcting errors.

Model checking has been very successful in detecting errors and providing high assurance for complex critical software. Still, just as with other formal methods technologies, model checking can require a great deal of time and memory when applied to even moderately large systems. Thus much of the research in this area is concerned with the issue of scalability.

### 2.2.3 Complete Strategies for Scalable Model Checking

As stated above, the input to a model checking tool is a finite-state concurrent system. In order to verify that properties hold for the system, the model checker must construct a single composite finite-state machine to represent all possible behavior of the individual concurrent machines in the model as they interact with each other. In practice this composite finite-state machine may be very large. This is the state-space explosion referred to in the literature: if there are many concurrent machines in the input model, making many transitions in parallel, the number of global states in the composite machine may grow exponentially, compared to the number of concurrent machines in the original model [12].

The general model checking technique originated in the 1980’s. At that time the maximum number of states in tractable models was between $10^4$ and $10^5$. In the early 1990’s, however, researchers at Carnegie Mellon University began using binary-decision diagrams, or BDDs, to
succinctly represent the global system [12]. This new *symbolic model checking* technique made it possible to check a model without ever explicitly constructing its composite state machine, and input models with up to $10^{20}$ states became tractable (i.e., models which would have required $10^{20}$ states to be represented explicitly, rather than symbolically with BDDs). Continuing work on BDDs has pushed the limit to $10^{120}$ [12]. The most well known tool implementing such a method is the Symbolic Model Verifier (SMV), available in several versions including Cadence SMV [49] and NuSMV [11], which are used in the experiments described in later chapters.

Symbolic model checking has worked well on models representing synchronous systems, including integrated circuit designs, which tend to have many small, symmetrical components, but has not always worked as well for software, which is often asynchronous. In an asynchronous system several things may be going on in parallel with no synchronizing clock, so that many different interleavings are possible. If all possible interleavings must be checked, the state space required tends to grow very large.

Unlike SMV, the SPIN (not an acronym) model checker is designed especially for asynchronous software models [40]. To handle models with many possible interleavings of parallel behaviors, SPIN uses an optimization strategy called *partial order reduction*, in which only interleavings relevant to the property specification are checked; that is, if the specified properties are unaffected by the order of some set of events, only one possible ordering of those events will be checked [12, 40]. SPIN has been used to verify a range of algorithms, protocols, and system implementations [39–41].

The strategies offered by SPIN for improving scalability fall into two general categories: ways to decrease the number of states in the global system (partial order reduction is an example) and ways to decrease the amount of memory required for each global system state [41]. Note that the use of BDDs in symbolic model checking does not really fall into either of these categories; it is a technique for compact representation of the entire state space, not individual states. Besides partial order reduction, SPIN uses a statement merging strategy to decrease the number of states required in the system, combining two or more transitions into one atomic unit where possible.

To decrease the amount of memory required for each state, SPIN offers a variety of options for lossless or lossy compression. The lossless methods save memory, but tend to require a lot of time [41]. For example, in the experiments presented in section 3.1.2, without compression SPIN
would have required 6.8 gigabytes of memory for the verification run (this statistic is output by SPIN when you use the compression option). With compression, the run required only about 270 Mb, but took 30 minutes on a computer with a 2.5 gigahertz processor.

2.2.4 Incomplete Strategies for Scalable Testing of Formal Models

SPIN’s lossless compression options, mentioned above, provide greater scalability but can require much more time for the verification run. SPIN’s lossy compression options run quickly and scale to large systems, but sacrifice completeness; that is, there is the possibility of missing property violations present in the system. In the current version of SPIN these are the hash compaction and bitstate hashing options [41]. These compression techniques decrease the amount of information recorded for each global system state, rather than decreasing the number of states searched. In the past, “scatter search,” an incomplete random search technique that limited which states were explored rather than limiting the amount of information stored with each state, was added to SPIN [38]. Although this idea was eventually given up, results from that work suggested that if there was a fault in the model, it was likely to affect a large portion of the state space. This idea is one important assumption in our work on Lurch, described in chapter 4, in which random search is used as a scalable alternative to model checking [58].

The SMV model checker also implements incomplete but scalable search options. Success in the development of algorithms for solving satisfiability (SAT) queries\(^2\) has enabled the development of a symbolic search technique known as bounded model checking [8]. To do bounded model checking, a SAT query is used to represent the state space up to a user-specified depth, and a SAT solving algorithm is used to determine whether the query is satisfiable, which corresponds to determining whether the input model is consistent with the property specification. Bounded model checkers have been very effective in practice. Nevertheless, bounded model checking is incomplete and can miss faults because of the search depth restriction. If no property violation is found, we only know that there is no violation within the user-specified search depth.

Although SPIN and SMV include the incomplete search options mentioned above, completeness is usually thought of as the key distinction between full-fledged model checking tools and debugging tools capable of finding errors but not proving their absence. In this sense a technique is

\(^2\) For a discussion of the satisfiability problem, a canonical example in computational complexity, see e.g., [68].
either complete or not complete; it doesn’t make sense to say that one approach is *more complete* than another. On the other hand, formal techniques cannot truly prove the correctness of a system; what they can do is verify that a model of the system is consistent with a set of user-specified properties [26, 27, 33]. And even where complete verification is not possible, model checkers may be very useful: they can automatically detect complex errors in systems too large for complete verification [14, 27], find counter examples to aid in fixing known errors [19], and generate test cases [23, 36].

### 2.2.5 Model Checking and Testing

As described above, there are incomplete strategies capable of efficiently providing some of the functionality of complete model checking tools. These can also be thought of as part of a developing set of testing methods with capability inspired by model checking. Much work is being done on testing strategies that are increasingly automated and capable of detecting more complex kinds of errors (i.e., more like model checkers). For example, tools running directly on source code and large production models can detect classes of errors previously beyond the scope of anything but formal verification [26, 27, 66, 67]. The less complete tools tend to work on much larger and more complex input systems, while the more complete tools tend to work on smaller, more abstract models.

Related to scalability, an important criterion for comparing these various kinds of testing and verification strategies is *scope*: the types of errors that can be detected or, for complete verification techniques, the types of properties that can be verified, is often limited in order to improve the scalability and simplicity of a tool. For example, many incomplete techniques scale to large systems but cannot detect liveness property violations, which requires cycle detection [26, 27, 66]. Even complete techniques may use optimizations limiting the scope of properties that can be verified (e.g., SPIN); these optimizations must be turned off, sacrificing scalability and simplicity (i.e., shorter counter examples), in order to verify the full range of temporal logic properties.

It is difficult to compare techniques’ scalability and scope, because different approaches work

---

3 To use a model checker to generate test cases, goals for the test are described by a negated temporal logic property, and the model checker is used to find a path to a violation of the property, i.e., a test case that goes to the goal.
well with different kinds of models, and some researchers advocate a framework in which several complementary approaches are available [14, 16, 27]. As pointed out by Cobleigh et.al. [14], the expectations of users vary through different phases of software development, while the scalability of a particular tool may depend on what it is being used for. For example, an explicit-state model checker (SPIN) might quickly find many long counter examples, while a symbolic model checker (SMV) might require more time and memory but find much shorter counter examples [6, 14]. So it may depend on whether a user requires, e.g., error detection alone versus short counter examples to facilitate fixing the errors, which tool scales better to a particular model. Related to this is the sensitivity of a testing strategy to minor changes in a model. As mentioned above, an apparently small change in the input model can make a large difference in the time and memory required by a model checker to detect errors.

2.2.6 Random Search Applied to Formal Models

Almost twenty years ago West explored the idea of using a simple incomplete technique, random search, to detect errors in finite-state models of software systems [72]. In that work random search, although incomplete, was surprisingly quick and effective at detecting errors. West’s explanation of the success of random search is helpful in understanding the success of various heuristics and incomplete verification strategies described above. He noted that faults detected in concurrent systems are often much less complex than the overall system [72]. That is, a fault involving a small subset of processes is present in many global system states—processes not relevant to the fault may be in any local state as long as the relevant processes are in the local states that together constitute the fault.

Faults may also be less complex than the overall system in another way: even if the fault is present in a very small number of global system states, there may be many paths that lead to those states. This kind of structure is exploited by the partial order reduction strategy mentioned above in reference to SPIN, which avoids exploring interleavings of behaviors irrelevant to the properties being verified. In models for which partial order reduction is effective we expect random search or other incomplete techniques to perform well also: where any one of a large number of interleaved paths is sufficient to represent all relevant behavior, the search only needs to explore one, so an
incomplete search may be sufficient.

Section 2.3 below gives a brief overview of the many random search algorithms and problem solving strategies used in software applications. Chapter 4 includes a more detailed description of the development and current version of Lurch, through which we apply random search, as West did, to fault detection in finite-state software models.

2.3 Random Search

Section 2.3 begins with a brief overview of the benefits of randomized algorithms, listing several well-known examples, and then discusses some of the trade-offs associated with a randomized approach in software testing. This is followed by a summary of two theories attempting to explain, in terms of the structure of search spaces representing complex problems, why apparently simple random search strategies may be surprisingly effective at exploiting such structure to find solutions.

2.3.1 Benefits and Disadvantages of Randomized Algorithms

Randomized algorithms work well on many kinds of problems. Sometimes a randomized algorithm is faster or more efficient (or both) than any known deterministic alternative. Also, randomized algorithms may be much simpler to implement and their performance may be less susceptible to minor changes in the input [56]. For example:

1. A randomized min-cut algorithm outperforms and is much simpler to implement than the best known optimization algorithms for network flow [56].

2. A randomized version of quicksort can be expected to perform just as well regardless of the order of the input [56].

3. Simulated annealing algorithms can be used to efficiently approximate optimal solutions to complex search problems [45].

4. Probabilistic skip lists are much simpler to implement than AVL trees [64].

5. Model-based diagnosis tools nondeterministically explore mutually exclusive but equally plausible solutions [31].
6. Genetic algorithms can be used to automate the process of designing circuits [46].

7. Machine learning can be used to predict software development effort estimates more reliably than alternative manual or algorithmic methods [9].

On the other hand, any nondeterministic strategy brings with it fundamental challenges: if the technique is not repeatable—given the same inputs it is not expected to produce the same output—how do we know whether to trust the output? Or, since many randomized algorithms do not perform an exhaustive exploration of possible solutions, how can we avoid missing the best (or any) solution? Finally, some randomized strategies have no inherent mechanism guaranteeing termination—without heuristic stopping rules, they are not algorithms at all. Leveson goes as far as saying, “nondeterminacy is the enemy of reliability” [47]. Despite these issues, however, we believe that nondeterministic strategies have much to offer even in the area of testing and verification of potentially critical software.

Several studies have compared partition testing of software, in which inputs are chosen to represent meaningful partitions of the input space, to testing randomly selected inputs, e.g., [21,28,30]. Generally the authors’ concern has been whether partition testing is worth the extra effort required, compared to random testing, which is usually much easier to implement. Two perhaps non-controversial conclusions from these studies are that random testing can be surprisingly effective and that it can work well together with deterministic alternatives. Given its simplicity, it makes sense as a first pass approach for detecting faults. And random testing is in some sense unbiased, i.e., more likely to find faults that strategies guided by domain knowledge or heuristic assumptions might miss [29].

2.3.2 Problem Structures Favorable to Random Search

Section 2.3.2 summarizes two theories that attempt to explain why simple randomized problem solving strategies are often surprisingly effective at solving complex problems. First, the phase transition theory states that for many kinds of complex problems, although the problem in general is difficult to solve, most specific problem instances are actually either very easy to solve or very easy to show unsolvable. Only a small set of problem instances, in the phase transition between these two groups, are hard to solve or hard to show unsolvable. The second theory discussed in
this section states that in complex problems involving many variables a small set of key variables largely determine the structure of the entire problem space. From the point of view of an algorithm searching for a solution, these funnel variables tend to guide the (randomized) algorithm towards a solution, or at least towards the same area of the search space that more complex alternative algorithms would be likely to reach.

Figure 2.1 illustrates the phase transition observed in, e.g., [10, 35, 52] in constraint satisfaction problems. As the number of constraints increases, there is a relatively abrupt transition from under-constrained, and therefore easy to solve, problem instances to over-constrained, and therefore easy to show unsolvable, problem instances. Problem instances in the phase transition region, where a great deal of effort is required to solve them or show that they are unsolvable, are rare. This phase transition region, in the words of Cheeseman et.al., is “where the really hard problems are” [10].

If this kind of phase transition is present in other types of search problems, it might explain why apparently simple solution strategies can be surprisingly effective. The right side of figure 2.1 shows how a simple solution strategy might be used to exploit easy problems but avoid wasting effort on problems that are very hard or unsolvable [61]. A relatively small amount of effort is put into solving the problem with a simple strategy (effort could be time, memory, or some other limited resource). If the problem is easy, it will be solved easily. If the problem cannot be solved, we conclude that it is either difficult or impossible. There is nothing revolutionary about this approach. The key point is that the phase transition region is narrow. A very simple strategy is therefore capable of solving nearly everything that could be solved by much more sophisticated strategies, but with much less effort. In practice a simple strategy may actually be more effective
because its efficiency allows it to scale to much larger problem instances.

Menzies' funnel theory [55] summarizes and attempts to explain in a different way why simple strategies have been surprisingly effective in solving apparently difficult computational problems. It could be because these systems contain funnels: small sets of key variables that determine the structure of the search space representing the problem. The key variables form a virtual funnel in the search space, so that a large proportion of the possible search paths are forced to go through the small part of the search space represented by the funnel. A simple search strategy quickly finds the funnel, because so many possible search paths lead to it. A more systematic search strategy yields little (if any) new information because all of the paths it systematically checks lead to the same funnel.

This is similar to the notion of saturation described above: individual tests at first yield a great deal of information, but as testing progresses its efficiency quickly tapers off, so that more and more tests yield less and less new information. These kinds of test results make sense if software systems tend to have funnels in their structure. Early tests quickly find the funnels, and therefore also find the bulk of system behavior likely to be exercised by testing. Later tests continue to find the same funnels, which lead to the same behavior already explored.

2.4 Summary

It can be very challenging to assess the quality of software. Because of this one focus of software engineering research is the testability of software artifacts. If software testing is viewed as a search through the space of a program’s possible behaviors, testability can be viewed as reachability. A more testable program is one for which a larger portion of the space of behavior can be reached using a smaller amount of testing effort. Thus testability is relative to the testing method chosen, and a particular software artifact will be most testable by a set of complementary methods.

In addition to testing, formal verification methods are used to assess the quality of software, especially relatively simple software to be used in critical applications. Automated tools have made formal methods much easier to use and much more effective, but the high costs, in terms of technical expertise, manual effort, and computing resources, have severely limited use in industry. Where formal methods are being used, however, the most popular tools are model checkers. Model
checkers are used to verify that an abstract finite-state machine model of a system is consistent with a property specification written in temporal logic. The biggest challenge in model checking is scalability, and various strategies have been developed to make model checking more scalable. Complete strategies retain the ability to prove correctness; incomplete strategies lack the ability to prove correctness but may be capable of very efficiently detecting faults. An incomplete strategy that has been a focus of the research presented in this dissertation is random search.

Randomized algorithms have various benefits, including speed and simplicity, but are generally not relied upon in critical applications because of the uncertainty of results. We note, however, that random search is surprisingly effective in problem solving applications, and discuss several possible explanations above. Also, as with other incomplete strategies for improving the scalability of model checking, random search can be used effectively in combination with complete strategies.
Chapter 3

Motivation

Chapter 3 provides several motivating examples in which complementary verification methods are combined to create an overall strategy more effective than any individual method. Section 3.1 describes three examples in which the use of multiple complementary tools improved the accuracy of verification results. The complete verification tools used in these examples are designed to detect any fault present in the input model. If no fault is detected the user should have reason for confidence that no fault is present. But it is possible to use a verification tool incorrectly and miss faults present in the model or, as shown in the third example, to detect spurious property violations actually not present. In these examples inconsistent results from multiple verification tools running on the same input model led to a better understanding of the individual tools and a more accurate overall result.

Section 3.2 describes several examples in which multiple complementary verification tools can be combined so that the overall performance is significantly improved. Two kinds of performance improvement are described. First, combining complementary tools may significantly decrease time and memory required for verification. Second, combining tools may result in a far more robust verification strategy, i.e., a strategy much less sensitive, in terms of time and memory requirements, to minor changes in the input model.

3.1 Using Multiple Verification Tools to Improve Accuracy

Section 3.1 presents three examples in which multiple verification tools were run on the same input model and produced inconsistent results. In each case the inconsistency was eventually resolved, and we gained a better understanding of the tools in the process. Also, in each case it would have
Model checking results
======================
(AG ((¬(cGuardAlarm=On))|(cUserDisplay=SeeOfficer))&{(¬(cUserDisplay=.....false

*** This is NuSMV 2.4.1 zchaff (compiled on Tue Jan 30 19:33:47 UTC 2007)...  
-- specification AG {{!(cGuardAlarm = On) | cUserDisplay = SeeOfficer)
& {{!(cUserDisplay = SeeOfficer) | cGuardAlarm = On}) is true

Figure 3.1: Inconsistent outputs from Cadence SMV (top) and NuSMV (bottom) running on the same input model.

been possible to use a single tool to get an invalid verification result, with no indication that the tool had been used incorrectly. These first two examples illustrate a key limitation in the use of model checking tools: although a fault detected by the tool can be manually confirmed or disconfirmed by inspecting the counter example trace output by the model checker, if the model checker reports that the model is correct no proof is generated to certify this result. Still, it can be confirmed or disconfirmed by the use of a second model checking tool. In the third example, the inconsistency brought to light is not as critical as the inconsistencies found in the first two examples. In the third example the use of multiple tools, rather than preventing a violation from being missed, has the practical benefit of showing that a violation detected by one tool is actually not present in the original input model.

3.1.1 Inconsistent Results from Two Symbolic Model Checkers

Figure 3.1 shows the outputs from two versions of the SMV symbolic model checker, Cadence SMV [49] and NuSMV [11], running on the same input model.¹ The model was generated automatically from a fault-seeded software requirements specification for a security system, written in the SCR modeling language mentioned above and described in more detail in chapter 4. As shown in figure 3.1 Cadence SMV and NuSMV disagree about whether one of the assertions included in the input model is true or false—the assertion (cGuardAlarm = On) <=> (cUserDisplay = SeeOfficer).

The input model used in this example was generated from a fault-seeded version of a specifi-

¹ For clarity many lines of output have been deleted in this figure and similar figures below.
Figure 3.2: Output from SPIN running on a model generated from the same fault-seeded specification used to generate the models for which Cadence SMV and NuSMV outputs are shown in figure 3.1.

ication known to be correct in the original version. The fault-seeded version contained two mutations, or minor changes, so our first step in attempting to resolve the inconsistency between Cadence SMV and NuSMV was to look at the results from running these tools on input models generated from specifications that each had just one of the mutations. Results on these single-mutation versions were consistent: for an input model generated from the specification with just the first mutation, both Cadence SMV and NuSMV reported that all assertions included in the input model were true; for an input model generated from the specification with just the second mutation, both Cadence SMV and NuSMV reported that the assertion \((c_{GuardAlarm} = \text{On}) \iff (c_{UserDisplay} = \text{SeeOfficer})\) was false. This suggests, although not conclusively, that the assertion violation reported by Cadence SMV is present in the input model, but somehow masked by the first mutation for NuSMV.

To confirm the Cadence SMV result, the model checker SPIN was run on an input model generated from the fault-seeded specification used in this example. Figure 3.2 shows the result from SPIN, consistent with the result from Cadence SMV. Based on this, we next contacted the developers of NuSMV and via several emails determined the reason for NuSMV’s incorrect verification result for the example input model. Although the input model was syntactically valid for NuSMV, the keyword \texttt{SPEC} used to mark assertions is not interpreted the same way by NuSMV as by Cadence SMV. As a result, NuSMV was checking assertions in the input model only for a limited set of possible execution paths. To force NuSMV to check assertions for all valid execution paths, it was necessary to replace \texttt{SPEC} with a NuSMV-specific key word, \texttt{INVARSPEC}. After doing this and running NuSMV on the modified input model, the output was consistent with Cadence SMV, reporting a violation of the assertion \((c_{GuardAlarm} = \text{On}) \iff (c_{UserDisplay} = \text{...}
Figure 3.3: Inconsistent outputs from SPIN (top) and Lurch (bottom) running on the same input model.

This example might be dismissed as not relevant to the question of whether a particular verification tool is more accurate than another. The cause of the inconsistency between NuSMV and Cadence SMV was not a bug in the verification tool, but an incompatibility between the automatic translation tool used to generate the input model (which used SPEC rather than INVARSPEC to mark assertions) and the NuSMV input language. But from the point of view of the user, the cause of the inconsistency is not relevant. The issue is that a verification tool, promising 100% confidence, reported that an incorrect input model was correct. It is only because this output was compared to that of other verification tools that the model was shown to be incorrect and eventually corrected.

### 3.1.2 Inconsistent Results from Model Checking and Random Search

Figure 3.3 shows outputs from the model checker SPIN and the random search tool Lurch [61] running on input models generated from another fault-seeded version of the software specification mentioned in the previous section and described in detail in chapter 5.\(^2\) SPIN reports that the input model is correct but Lurch reports an assertion violation. Because Lurch is an *incomplete* random search tool, which can detect property violations but not verify correctness, we would expect to sometimes see violations missed by Lurch but detected by SPIN. But SPIN is a *complete* verification tool, so we would never expect the result shown in figure 3.3—Lurch, an incomplete tool, reports a violation, while SPIN runs to completion but reports no violation.

Unlike the example in the previous section, in which NuSMV and Cadence SMV were run on

\(^2\) For a more detailed explanation of the examples in sections 3.1.2 and 3.2.1 see [60].
Figure 3.4: Simulator log produced by stepping through execution trace output by Lurch.

the same input model, in this case SPIN and Lurch ran on two different input models generated from the fault-seeded specification using two different translation tools. We initially assumed that the inconsistent outputs shown in figure 3.3 were due to an error in the translator used to generate the Lurch input model, since it had been newly developed as part of the research presented in this dissertation (see chapter 4 for a description of the translator). So, to determine whether the property violation detected by Lurch was present in the original fault-seeded specification and not due to an error in the translator, we used an SCR simulation tool to step through the fault-seeded specification according to the execution trace output by Lurch.

Figure 3.4 shows part of the log produced by stepping through the fault-seeded specification according to the execution trace output by Lurch. The log indicates that one of the functions in the specification is not disjoint; that is, the function is nondeterministic as a result of overlap between two conditions that should be mutually exclusive. This general disjointness error does not necessarily mean that a specific assertion in the specification will be violated; however, we observed that the translation tool used to generate the input model for SPIN exploits a feature of the SPIN input language not compatible with the nondeterminism indicated by the disjointness error shown in figure 3.4.

The SPIN input language allows blocks to be marked as deterministic steps with the key word d_step. SPIN assumes such blocks are deterministic, i.e., there is only one possible execution path through the block, and therefore SPIN checks only one path through the block. For blocks that are not deterministic, this results in some of the behavior of the input model being ignored. For the fault-seeded specification in this example, behavior ignored by SPIN included a violation of one of the assertions, the violation detected by Lurch. After removing the relevant d_step marker from the input model and running SPIN again, it quickly detected the assertion violation previously detected only by Lurch.
Checking disjointness of all modules...
Checking coverage of all modules...
Checking guarantees in all modules...
Checking PINEntry ... {1,0,1}:0 - {1,1,0}:0 pass...

Depth= 499462 States= 1e+06 Transitions= 1.02634e+06 Nodes= 17543 Memory= 60.608
panic: assertion violated ((mPINInput_OLD==mPINInput_NEW)||((mcStatus_OLD==10)||(mcStatus_OLD==5)))
(at depth 833676)...

(Spin Version 4.2.4 -- 14 February 2005)...
State-vector 32 byte, depth reached 833689, errors: 1

Figure 3.5: Inconsistent outputs from Salsa (top) and SPIN (bottom) running on the same input model.

In this experiment, if only the SPIN had been used, there would have been no way of knowing that this particular specification had a disjointness error and a related assertion violation. And this is not because of any bug in SPIN, but because of an assumption made in the translation—an assumption which makes sense most of the time and greatly improves SPIN’s performance on automatically translated models, but an assumption which was not valid in this case. Using Lurch as well, we were able to uncover the assumption and better understand how to use SPIN to get accurate verification results.

3.1.3 Inconsistent Results from an Invariant Checker and a Model Checker

Figure 3.5 shows inconsistent results from the invariant checker Salsa [7] and the model checker SPIN running on another fault-seeded version of the input model used in the previous two sections. Salsa, a specialized tool implementing ideas from model checking and theorem proving to prove assertions in SCR models, is described in more detail in chapter 4. Salsa reports that the property PINEntry is true (top of figure 3.5) but SPIN reports a violation of the assertion corresponding to the property. Salsa, described in more detail in chapter 4, is an invariant checker capable of proving properties true; however, if a property cannot be proved true by Salsa it is not necessarily false. In this way Salsa is different from a model checker like SPIN, which is designed to detect only genuine property violations. Strangely, in this example SPIN reports a violation of a property proved true by Salsa.
Eventually we determined the reason for the inconsistency: one feature of the SCR modeling language is ignored by the translation tool used to generate the SPIN version of the input model. SCR allows the use of \textsc{nature} constraints to limit the behavior of variables representing inputs from the environment. In this case one of the \textsc{nature} constraints is necessary in the model for the property \textsc{pinentry} to be true. Thus Salsa, running on an input model including the relevant \textsc{nature} constraint, found that the property \textsc{pinentry} was true. But SPIN, running on a model without the constraint, found a violation of the property. This explanation of the discrepancy between Salsa and SPIN was confirmed by removing the constraint from the Salsa input model. Rerunning Salsa on an input model without the constraint, we found that Salsa could no longer prove the property true.

This inconsistency is perhaps less critical than the two in the previous examples, because there was no possibility of missing a genuine fault. But it does show a practical benefit of combining complementary tools. If only SPIN were used, much manual effort might be expended attempting to find and correct the input model so that the property \textsc{pinentry} would not be violated. Using Salsa makes it unnecessary to track down the cause of the violation found by SPIN. In addition, this example underscores the need to validate faults detected by SPIN or any model checker. The fault may be related to a mistake in the portion of the input model representing the environment rather than the critical system to be verified.

\section{Using Multiple Verification Tools to Improve Performance}

Section 3.2 briefly summarizes the results of several experiments in which combining complementary tools greatly improves performance without sacrificing completeness for the set of input models used in the experiments. First, overall results from the experiment described in section 3.1.2 are given, showing that the SPIN model checker can be used together with the Lurch random search tool, not just to improve accuracy, as stated above, but to save time and decrease memory requirements. Next, results are given from a past experiment in which the symbolic model checker NuSMV and the Lurch random search tool, used together, retain the completeness of NuSMV but greatly reduce the average time required for an input model. Finally, two experiments on scalable research problems are used to show how verification tools’ performance can be very sensitive
Tables 3.1 and 3.2 summarize the results for the full experiment from which the example in section 3.1.2 was taken. Out of 50 fault-seeded input models, 38 contained property violations and 12 were equivalent mutants. Of the 38 with property violations, Lurch detected property violations in 34. Average time and memory values are listed in the tables for Lurch, SPIN, and a simple combination strategy, in which Lurch was run for 7 seconds, and if no property violation was detected by Lurch SPIN was run. The cutoff value of 7 seconds was the optimal (integer) choice for this set of experiments, but any value between 5 and 10 seconds produced approximately the same overall result.

In tables 3.1 and 3.2, Lurch values are only shown for the 34 specifications in which Lurch found property violations. This is because Lurch’s resource requirements in cases where Lurch detected no property violation are somewhat arbitrary—time and memory required would be equal
to whatever their values are at the point when the user decides to give up running Lurch. SPIN values shown represent a process of running SPIN first in default mode; then, if no violation was detected, with slower but more memory efficient settings required to complete the verification run; and then, if still no violation was detected, with the d_step marker removed, which further slowed the verification but was necessary to catch the property violation, as described in section 3.1.2.

Values shown in tables 3.1 and 3.2 for the combination of Lurch and SPIN are averages of values for all specifications, including some values from Lurch and some from SPIN. This is why these values are not simply the minimum of the Lurch and SPIN values. For example, in the second row of table 3.1, the average time required for the combination of Lurch and SPIN is 2.31 seconds, which is less than the average time for Lurch (3.74 seconds) and less than the average time for SPIN (43.3 seconds). This is because there are some cases where Lurch required more than 7 seconds to detect a property violation, but SPIN detected a violation very quickly. In these cases the time required for the combination of Lurch and SPIN is less than the time required for Lurch alone. There are enough of these cases to make the average time required by the combination of Lurch and SPIN less than the average time required by Lurch alone.

The results in tables 3.1 and 3.2 show that, in addition to the benefit of improved accuracy described in section 3.1.2, using Lurch and SPIN together can significantly improve the overall performance for a set of verification runs. Using random search, as implemented in Lurch, as a first step before running the complete verification tool SPIN, saves on average over 30 seconds per verification run. Because time values varied greatly between runs, standard deviation values are also given for each tool’s results running on each set of input models. As will be discussed further in section 3.2.3, combining multiple tools sometimes greatly decreases the variability of resource requirements for verification runs on similar input models.

3.2.2 Combination of Symbolic Model Checking and Random Search

This section describes a past experiment in which we ran Lurch and NuSMV on fault-seeded versions of a model of the mode logic for a commercial flight guidance system developed in collaboration between Rockwell Collins, Inc. and the University of Minnesota [63]. The model was written in RSML [48, 73], a formal specification language based on Statecharts [32], which is similar to the SCR language, in which the original software specification for the last three exam-
ples was written. The flight guidance system model was translated automatically to NuSMV and Lurch through Nimbus [70], the development environment for RSML$^{-e}$. In preparation for this experiment a set of 100 fault-seeded versions of the original model were created. Mutation operators used to generate fault-seeded versions of the model were based on developers’ revision history and include:

1. Variable Replacement: a variable reference was replaced with a reference to another variable of the same type.$^3$
2. Condition Insertion: a condition originally marked *don’t care* was changed to *true*.
3. Condition Removal: a condition originally marked *true* was changed to *don’t care*.
4. Condition Negation: a condition originally marked *true* was changed to *false* (or vice versa).

Using NuSMV, we then attempted to verify 300 properties for the 100 fault-seeded versions of the flight guidance system model. 45 of the 100 fault-seeded models violated at least one of the original properties. The actual properties violated made up 60 of the original 300. For our experiment we used these 45 fault-seeded models and these 60 properties. The specific properties used were considered proprietary information by Rockwell Collins, Inc., but the following informal examples should be sufficient to illustrate the general type of properties used in the experiment:

1. If the flight director cues are off, the flight director cues shall not be turned on when the Transfer Switch is pressed, provided that no lateral or vertical mode is selected and... (some additional conditions).

2. If mode annunciations are off, auto pilot engagement shall cause ROLL mode to be selected, provided... (some additional conditions).

Table 3.3 shows average times required for Lurch, NuSMV, and a simple combined strategy—run Lurch for two minutes and then run NuSMV if no property violations are detected by Lurch—running on fault-seeded versions of the (very large) flight guidance system model. On the whole, Lurch ran much faster than NuSMV and was almost as effective at finding property violations.

$^3$ These mutation operators correspond to the VRP and SOR operators described in chapter 5.
Table 3.3: Time (s) required for Lurch, NuSMV, and a simple combined strategy, running on fault-seeded versions of the flight guidance system model.

Running Lurch for two minutes on each input model would have saved on average about two hours per input model, compared to running NuSMV alone. In this experiment a cutoff value of 2 minutes was nearly optimal, but even if the 7 second cutoff value from the previous section were used, the combined strategy would still save on average about 40 minutes per input model.

Much of the time difference between Lurch and NuSMV might be explained by the fact that Lurch is an explicit-state strategy, exploring the state space one path at a time and reporting violations as soon as they are detected, while NuSMV is a symbolic model checker and builds a representation of the entire state space before exploring it. Others have shown that explicit state strategies can be much faster at detecting faults, as opposed to proving correctness or finding short paths to faults [6, 14, 16]. As the other experiments in this section show, however, although Lurch is based on an explicit-state strategy its performance is significantly different from both explicit-state (e.g., SPIN) and symbolic model checkers. So the key difference between Lurch and NuSMV is probably Lurch’s random search strategy, rather than the fact that Lurch uses an explicit-state representation of the state space.

### 3.2.3 Combining Tools to Improve Robustness

In the previous section, standard deviation values shown in tables 3.1, 3.2 and 3.3 show that combining verification strategies may result in a more robust overall strategy; that is, the variability of time and memory requirements from one input model to another (apparently very similar) input model may be greatly reduced. In this section, results from experiments on two scalable research problems further illustrate this benefit of a combined verification strategy. First, two very similar versions of the dining philosophers problem are given—one that is difficult for SPIN but easy for
Figure 3.6: Dining philosopher processes: with loop (top), without loop (bottom).

<table>
<thead>
<tr>
<th>Sets of Input Models</th>
<th>SPIN</th>
<th>NuSMV</th>
<th>Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>All (16)</td>
<td>12.2</td>
<td>11.2</td>
<td>3.84</td>
</tr>
<tr>
<td></td>
<td>42.0</td>
<td>10.2</td>
<td>10.3</td>
</tr>
<tr>
<td>With Loop (8)</td>
<td>0.0538</td>
<td>14.6</td>
<td>0.0538</td>
</tr>
<tr>
<td></td>
<td>0.0130</td>
<td>4.89</td>
<td>0.0130</td>
</tr>
<tr>
<td>Without Loop (8)</td>
<td>24.4</td>
<td>7.80</td>
<td>7.62</td>
</tr>
<tr>
<td></td>
<td>58.6</td>
<td>13.2</td>
<td>13.9</td>
</tr>
</tbody>
</table>

Table 3.4: Time (s) required for SPIN, NuSMV, and combined strategy, running until deadlock detected, on two versions of the dining philosophers problem.

NuSMV, and another that is easy for SPIN but difficult for NuSMV. Both versions of the problem are easy for a strategy combining SPIN and NuSMV. Second, results are given for a scalable protocol model; strangely, versions of the model with an even number of processes were much easier for SPIN than versions with an odd number of processes. Both versions were easy for Lurch. Although using Lurch alone for verification would not guarantee completeness, SPIN and Lurch can be used together on both even and odd versions of the protocol model in a way that is complete but not sensitive, in terms of time and memory requirements, to whether the input model has an even or odd number of processes.

Tables 3.4 and 3.5 show average time and memory requirements for SPIN and NuSMV running on two versions of the well-known dining philosophers problem [18], which represents multiple processes sharing multiple resources. $N$ philosophers are seated around a table with $N$ forks placed one between each pair of philosophers. Each philosopher process has 3 states: satisfied, waiting and eating; each process transitions through the states as shown at the top of figure 3.6: picking up the left and right forks, eating, and dropping the forks. This is the first version (with loop in tables
Sets of Input Models | SPIN | NuSMV | Combination
---|---|---|---
All (16) | memory | 10.7 | 29.7 | 5.25
| stdv | 25.1 | 66.1 | 5.31
With Loop (8) | memory | 2.82 | 47.4 | 2.82
| stdv | 0.254 | 92.9 | 0.0254
Without Loop (8) | memory | 18.6 | 12.0 | 7.67
| stdv | 34.8 | 3.60 | 6.86

Table 3.5: Memory (MB) required for SPIN, NuSMV, and combined strategy, running until deadlock detected, on two versions of the dining philosophers problem.

Sets of Input Models | Lurch | SPIN | Combination
---|---|---|---
All (8) | time | 0.0950 | 23.1 | 0.0950
| stdv | 0.0302 | 61.6 | 0.0302
Even Number of Processes (4) | time | 0.0850 | 0.0425 | 0.0850
| stdv | 0.0129 | 0.00500 | 0.0129
Odd Number of Processes (4) | time | 0.105 | 46.2 | 0.105
| stdv | 0.0412 | 86.2 | 0.0412

Table 3.6: Time (s) required for Lurch, SPIN, and combined strategy, running on fault-seeded versions of the leader election protocol model.

3.4 and 3.5). In the second version, each philosopher process reaches the eating state at most once, after which the left and right forks are dropped and the process moves to the finished state.

Interestingly, for SPIN the first version of the problem is easier than the second, while for NuSMV the second version is easier. A simple combined strategy, in which SPIN is run for one second and then, if SPIN fails to find the deadlock, NuSMV is run to completion, preserves the best performance of both tools, decreasing not only the average time and memory for all 16 input models, but also significantly decreasing the standard deviation values associated with time and memory.

Tables 3.6 and 3.7 show average time and memory requirements for SPIN and Lurch running on a fault-seeded version of a leader election protocol model originally published as an example for which SPIN's partial order reduction optimization, described in section 2.2.3, significantly improves performance [40]. This is a scalable model for which the number of (identical) processes

---

4 In communication via email with the developer of SPIN, we learned that one reason SPIN would not work as well on the second (without loop) version of the dining philosophers problem has to do with SPIN’s dynamic process deletion and creation. Results shown in tables 3.4 and 3.5 are for models rewritten, according to Holzmann’s instructions, so that philosopher processes are not deleted and recreated. Even with the suggested changes made, the second version of the problem is much harder for SPIN.
can be modified. For one particular fault-seeded version, in which two message passing statements were changed so that the wrong message types were used, scaled versions of the model with an even number of processes were much easier for SPIN than versions with an odd number of processes. Conceptually, it seems that for versions with an even number of processes a property violation associated with the seeded faults was located in a part of the state space explored early by SPIN; but for versions with an odd number SPIN had to explore much more of the state space before finding a property violation. Lurch, on the other hand, searches randomly through the state space, so it makes sense that performance is approximately the same for even or odd versions of the protocol model.

In this example, Lurch was able to detect property violations for all fault-seeded input models. But in general, because Lurch is not complete, to fully verify the model requires a combined strategy in which SPIN would be used if Lurch failed to detect a property violation after a specified time cutoff. For this experiment, since Lurch detected property violations in under one second for all versions of the input model, the only difference between the combined strategy and Lurch is that the combined strategy is (technically) complete, while Lurch is not. Figure 3.7 shows graphically the clear performance difference for SPIN between even and odd versions of the protocol model. SPIN’s complete verification algorithm is extremely sensitive to what would seem to be a very minor difference between two input models—whether there is an even or odd number of (identical) processes. But Lurch’s random search, and therefore the combined strategy as well, is not very sensitive to the difference.

Table 3.7: Memory (MB) required for Lurch, SPIN, and combined strategy, running on fault-seeded versions of the leader election protocol model.
Figure 3.7: Time (s) and memory (MB) required for Lurch and SPIN running on fault-seeded version of the leader election protocol input model. (Combined strategy complete but otherwise identical to Lurch.)
3.3 Summary

Multiple verification tools should be used together for two reasons: improved accuracy and improved performance. Using the symbolic model checkers Cadence SMV and NuSMV, we found that a modification to the input model for NuSMV resulted in more accurate results. Without the modification a fault was missed; with the modification it was detected. Using the explicit-state model checker SPIN and the random search tool Lurch we found that a feature used in the input model for SPIN was not valid in certain cases. Modifying the model resulted in SPIN’s detection of a fault otherwise detected only by Lurch. Finally, the use of SPIN together with the invariant checker Salsa brought out the fact that certain kinds of constraints in the input model were ignored in the translator to SPIN, so that SPIN sometimes reported faults not actually present in the original model. We found that Salsa could also be used to validate faults reported by Lurch.

Using SPIN together with Lurch on a set of fault-seeded input models, we found that a simple cascaded strategy, running Lurch first and then SPIN if no fault was detected by Lurch, resulted in significant performance improvements. We found similar results in a different experiment using NuSMV and Lurch on fault-seeded versions of a very large input model. Verification tools may also be used together to improve robustness, i.e., to decrease the sensitivity of the overall verification strategy to minor changes in the input model. An individual tool’s performance on one input model may be very good; a minor change in the model may result in much worse performance. But a combination of tools will likely perform more consistently from one input model to the next. We found this to be true for SPIN and NuSMV, running on two versions of the dining philosophers problem, and for Lurch and SPIN running on fault-seeded versions of a protocol model.
Chapter 4

Experimental Framework

Chapter 4 describes the modeling and verification tools that together make up the framework for experiments presented in chapter 5. Continuing in the vein of the previous chapter, section 4.1 describes other researchers’ related work—work on which the case study experiments presented in later chapters depend directly. Specifically, section 4.1 covers verification tools developed by other researchers and used in our experiments, including the SCR Toolset, for modeling, simulation and testing of software specifications written in the SCR language; the Salsa invariant checker, for SCR specifications; the Cadence SMV and NuSMV symbolic model checking tools, and the SPIN explicit-state model checking tool. The input language and basic output of each tool is illustrated using an example SCR specification for a cruise control system.

Section 4.2 describes Lurch, the random search debugging tool for formal models developed as a major part of the research work done in preparation for this dissertation. In addition to the verification tools listed above, developed by other researchers, we use Lurch in our experiments in later chapters as an alternative tool for efficiently debugging formal models. The original version of the random search technique now implemented in Lurch used an intermediate AND-OR graph representation based on Menzies’ work in hypothesis testing applications. The current version is somewhat simpler in its basic search procedure, which has made it possible to add several features, including support for synchronous models (e.g., models based on SCR specifications). Chapter 4 concludes by describing in section 4.2 an automatic method of translating SCR specifications to Lurch models. This is illustrated by a Lurch version of the cruise control specification example from section 4.1.
4.1 Existing Tools Used in Case Study

This section briefly describes the software modeling and verification tools developed by others that are used in the experiments below: the SCR Toolset, the Salsa invariant checker, the Cadence SMV and NuSMV symbolic model checkers, and the SPIN explicit-state model checker. The SCR Toolset, in addition to features for working directly on SCR specifications, includes the ability to automatically translate SCR specifications into the input languages for Salsa, SPIN, and SMV. (Translation to NuSMV involves minor modifications, described in section 3.1.1, to the SMV version of the specification.)

4.1.1 SCR Modeling, Simulation and Testing Tools

The SCR requirements specification language, a tabular notation for concise, unambiguous description of functional requirements, has been developed by Heitmeyer and others over the last twenty years and has been used in a variety of research and industrial applications [37]. An SCR specification includes two types of variables: monitored variables represent environmental quantities monitored by the system; controlled variables represent quantities controlled by the system. Monitored variables may change nondeterministically, but behavior within the system, causing changes to controlled variables, must be deterministic. In general, changes in controlled variables are triggered by conditioned events of the form:

\[ \mathit{@T(c)} \quad \text{WHEN} \quad d \quad \overset{\text{def}}{=} \quad \neg c \land c' \land d \]

This event could be read: “c changes from false to true when d is true.” The \( \mathit{@T(c)} \) portion of the event is a two-state predicate and is true if the condition \( c \) is false in the current state but true in the next state. For the entire event to be true (including WHEN \( d \)) the condition \( d \) must be true in the current state.

Early on, analysis of SCR specifications was done manually, but during the last 15 years automated tools have been developed to enable more effective and less costly analysis. The current version of the SCR Toolset includes:

1. Specification Editor: Enables user-friendly viewing, editing, and search of specifications; also provides access to the other tools through a single interface.
2. Simulator: Allows user to observe and control execution of the specification, to follow a path
to an error discovered by one of the model checking tools, for example.

3. Dependency Graph Browser: Constructs and displays a graph showing relationships between
variables in the specification.

4. Consistency Checker: Detects various kinds of errors including syntax errors, invalid values,
circular definitions, and violations of disjointness or coverage properties.

5. Model Checker(s): Automatic translation from SCR to the SMV [49] and SPIN [41] model
checkers.


7. Property Checker: Automatic translation from SCR to Salsa [7], a more powerful tool (than
the consistency checker) for proving disjointness or coverage properties, or user-specified
assertions.

8. Invariant Generator: Automatically generates state invariants for the specification.

Figure 4.1 shows a finite-state machine representing a simplified version of the cruise control
system specification in [3]. The cruise control mode is initially Off. It changes to Inactive once the
key is turned in the ignition. From Inactive, the mode changes to Cruise when the cruise control
lever is moved to Activate, if the engine is running and the brake is not pressed. From Cruise, the
mode may change to Override if the brake is pressed or the lever is moved to Deactivate, assuming
the key is still in the ignition and the engine is still running.

From Override the mode changes back to Cruise if the lever is moved to Resume or Activate,
as long as the key is still in the ignition, the engine is still running, and the brake is not pressed.
From Cruise or Override, if the engine stops running but the key is still in the ignition, the mode
changes to Inactive. From Inactive, Cruise, or Override, if the key is removed from the ignition
the mode changes to Off.

Figure 4.2 shows an SCR specification representing the cruise control system in figure 4.1. The
specification begins with definitions of an enumerated type to represent the positions of the
cruise control lever (lines 5–6) and an SCR modeclass to represent the possible modes of the cruise
Figure 4.1: Cruise control system finite-state machine.
// This file contains an SCRTool system specification...

SPECIFICATION; VERSION "1.7";

5 TYPE "CruiseLever"; BASETYPE "Enumerated"; UNITS "";
VALUES "Activate, Deactivate, Resume";

MODECLASS "Mode"; MODES "Off, Inactive, Cruise, Override";
INITMODE "Off";

10 MON "Brake"; TYPE "Boolean"; INITVAL "FALSE";
MON "EngRun"; TYPE "Boolean"; INITVAL "FALSE";
MON "Ignited"; TYPE "Boolean"; INITVAL "FALSE";
MON "Lever"; TYPE "CruiseLever"; INITVAL "Deactivate";

15 ASSERTION "S1"; EXPR "(Mode = Off) => (NOT Ignited)";
ASSERTION "S2"; EXPR "(Mode = Inactive) => Ignited";
ASSERTION "S3"; EXPR "(Mode = Cruise) => (Ignited AND EngRun AND (NOT Brake) AND
(NOT (Lever = Deactivate)))";

20 ASSERTION "S4"; EXPR "(Mode = Override) => (Ignited AND EngRun)";

MODETRANS "Mode";
FROM "Off" EVENT "@T(Ignited)" TO "Inactive";
FROM "Inactive" EVENT "@F(Ignited)" TO "Off";
FROM "Inactive" EVENT "@T(Lever = Activate) WHEN (EngRun AND (NOT Brake))" TO "Cruise";
FROM "Cruise" EVENT "@F(Ignited)" TO "Off";
FROM "Cruise" EVENT "@F(EngRun) WHEN (Ignited)" TO "Inactive";
FROM "Cruise" EVENT "@T(Brake OR (Lever = Deactivate)) WHEN (Ignited AND EngRun)" TO "Override";
FROM "Override" EVENT "@F(Ignited)" TO "Off";
FROM "Override" EVENT "@F(EngRun) WHEN (Ignited)" TO "Inactive";
FROM "Override" EVENT "@T((Lever = Activate) OR (Lever = Resume)) WHEN (Ignited AND EngRun AND (NOT Brake))" TO "Cruise";

Figure 4.2: SCR specification for cruise control system.
control system (lines 8–9). Next, four *monitored* variables are declared to represent inputs to the system from the environment—from the brake, engine, ignition and cruise control lever (lines 11–14). Variable declarations are followed by four assertions (lines 16-20), stating:

1. If the cruise control mode is *Off*, the key is not in the ignition (line 16).

2. If the mode is *Inactive*, the key *is* in the ignition (line 17).

3. If the mode is *Cruise*, the key is in the ignition, the engine is running, the brake is not pressed and the lever is not set to *Deactivate* (lines 18–19).

4. If the mode is *Override*, the key is in the ignition and the engine is running (line 20).

Finally, the specification lists mode transitions and the events that trigger them (lines 22–34). These lines correspond to the finite-state machine in figure 4.1.

### 4.1.2 Salsa Invariant Checker

The Salsa invariant checker uses a combination of ideas from theorem proving and symbolic model checking to prove generic disjointness and coverage properties, as well as user-specified assertions, for input models written in a modified version of the SCR specification language [7, 67]. Like an automated theorem proving tool, Salsa attempts to carry out an inductive proof using decision procedures. Like a symbolic model checking tool, Salsa uses binary decision diagrams (BDDs) to represent the state space of the input model in a very compact way.

Salsa either determines that a property is true or outputs a counter example. In some cases Salsa is unable to prove properties that are actually true, so the user must determine whether counter examples produced by Salsa are valid; that is, whether the first state in the counter example is reachable from the system’s initial state. In our experiments Salsa has been more effective than the SCR Toolset’s consistency checker but less helpful than model checking tools for detecting and confirming property violations. In chapter 6 we consider whether Salsa might be used as a preprocessing step to improve the performance of model checking tools used later.

Figure 4.3 shows the cruise control system as a Salsa input model equivalent to the the SCR model in figure 4.2, although the code has been edited to save space. (See appendix A for the full version.) The Salsa model begins with type definitions and declarations of *monitored* and
// This file contains the SAL version of an SCR specification...

module cc.ssl

5 type definitions
  CruiseLever : { Activate, Deactivate, Resume };
  prefix0_Mode_modes : { Off, Inactive, Cruise, Override };

monitored variables
10   EngRun : bool;
    Brake : bool...

internal variables
    Mode : prefix0_Mode_modes;

15 assumptions
    init = initially EngRun = FALSE and Brake = FALSE and...

guarantees
20   S4 = (Mode = Override) => (Ignited and EngRun);
    S3 = (Mode = Cruise)...

definitions

25 /*---- Begin mode transition table: Mode -----------------------------*/
var Mode initially Off :=
  case Mode
    [] Cruise ->
      ev
        [] @F(Ignited) -> Off
        [] @F(EngRun) when Ignited -> Inactive...
      ve
    [] Inactive ->
      ev
        [] @F(Ignited) -> Off...
      ve
    otherwise -> Mode
  esac

end module

Figure 4.3: Salsa version of cruise control specification.
Analyzing SAL specification in file: cc.ssl.sal.
Checking disjointness of all modules...

Checking coverage of all modules...

Checking guarantees in all modules...

Checking S4 ... (1,0,1):0 - (1,1,0):0 pass.
Checking S3 ... (1,0,1):0 - (1,0,1):0 fail.
Checking S2 ... (1,0,1):0 - (1,1,0):0 pass.
Checking S1 ... (1,0,1):0 - (1,0,1):0 fail.

Checks failed for: S1, S3
Number of failed/passed verification conditions: 2/2

Analyzing SAL specification in file: cc.ssl.sal.
Checking disjointness of all modules...

Checking coverage of all modules...

Checking guarantees in all modules...

Checking S4 ... (1,0,1):0 - (1,1,0):0 pass.
Checking S3 ... (1,0,1):0 - (1,0,1):0 fail.
Checking S2 ... (1,0,1):0 - (1,1,0):0 fail.
Checking S1 ... (1,0,1):0 - (1,0,1):0 fail.

Checks failed for: S1, S2, S3
Number of failed/passed verification conditions: 3/1

Figure 4.4: Salsa output for correct (top) and fault-seeded (bottom) versions of the cruise control system model.

internal variables (lines 5–14); internal variables include SCR controlled variables and modeclass variables. Next, initial values are listed as assumptions (lines 16–17), and assertions are listed as guarantees (lines 16–21). Finally, the transitions for the system mode are given in an indented block structure (lines 25–40).

Figure 4.4 shows the output from Salsa for two versions of the cruise control system model, the correct version, based on the SCR specification in figure 4.2, and a fault-seeded version, in which the condition WHEN EngRun has been added to the event in line 24 of figure 4.2. For the correct version Salsa is able to prove assertions S2 and S4, while for the fault-seeded version Salsa is only able to prove S4. This suggests that S2 is violated in the fault-seeded version, but the result is not conclusive. (Even in the correct version of the model Salsa fails to prove S1 and S3.)
4.1.3 SMV and NuSMV Symbolic Model Checking Tools

Figure 4.5 shows the SMV version of the cruise control specification from figure 4.2 (see appendix A for the full version). The SMV model begins with variable declarations (lines 5–8). As with other model checking tools, SMV variables must be discrete and if possible limited to a small number of values. The ASSIGN section is used to specify the initial and next values for each variable (lines 15–28). For example, the initial value of EngRun is zero (line 11), and each time the executing system steps forward the next value of EngRun is chosen randomly from the set \{0, 1\} (line 15). The choice of the next value for Mode is more complicated, since it is based on the current value of Mode and changes in other variables (i.e., events). For example, line 23 states that, if the value of Ignited changes from 0 to 1 (false to true) and Mode is Off, the next value of Mode is Inactive.

The TRANS section adds an additional constraint to the system, that only one of the monitored variables may change each time the executing system steps forward (lines 30-37). Next, assertions are listed in terms of their corresponding temporal logic operator, \(\text{AG}\), or always, globally, which means that the assertion must hold in every global state (lines 40–43). As stated in section 3.1.1, assertions must be stated differently to be compatible with the NuSMV model checker. The bottom of figure 4.5 shows how the final section of the model should be changed to work with NuSMV.

Figure 4.6 shows the output from Cadence SMV for the correct version of the cruise control model, as given in figure 4.5 (top) and the fault-seeded version described in section 4.1.2 (bottom). Figure 4.7 shows output from NuSMV for these same two versions of the cruise control model. Note that for Cadence SMV the default behavior is to state the first property found to be false, whether or not there are others, while for NuSMV, by default all properties are listed as either true or false. In the experiments below, this may be the reason Cadence SMV is consistently faster and requires less memory than NuSMV, because NuSMV’s default settings force it to work harder and output a more comprehensive verification result.

4.1.4 SPIN Explicit-State Model Checker

Figure 4.8 shows the cruise control specification from figure 4.2 written in Promela, the input language for SPIN (see appendix A for the full version). Compared to the input languages for Salsa and SMV, Promela is much more like a typical structured programming language (e.g., C). The model begins with \#define macros and global variable declarations (lines 8–11). Promela
-- This file contains the SMV version of an SCR specification...

MODULE main

5 VAR
    Mode : {Off, Inactive, Cruise, Override};
    EngRun : boolean;
    Brake...

10 ASSIGN
    init(EngRun) := 0;
    init(Brake) := 0;
    init(Lever) := Deactivate...

15 next(EngRun) := {0,1};
    next(Brake) := {0,1};
    next(Lever) := (Activate, Deactivate, Resume);
    next(Ignited)...

20 ---- Begin mode transition table: Mode -----------------------------
next(Mode) :=
    case
        (next(Ignited) & ! Ignited) & (Mode = Off) : Inactive;
        (Ignited & ! next(Ignited)) & (Mode = Inactive) : Off;
    esac;

25 1 : Mode;

30 TRANS
    -- One Input Assumption
    !(next(EngRun) = EngRun) & (next(Brake) = Brake) & (next(Lever) = Lever) &
    (next(Ignited) = Ignited) |
    (next(EngRun) = EngRun) & !(next(Brake) = Brake) & (next(Lever) = Lever) &
    (next(Ignited) = Ignited) |
    (next(EngRun) = EngRun) & (next(Brake) = Brake) & !(next(Lever) = Lever) &
    (next(Ignited) = Ignited)...

    -- Assertions
    SPEC
    AG(!(Mode = Override) | (Ignited & EngRun))
    SPEC
    AG(!(Mode = Cruise)...
Figure 4.6: SMV output for correct (top) and fault-seeded (bottom) versions of the cruise control system model.

Figure 4.7: NuSMV output for correct (top) and fault-seeded (bottom) versions of the cruise control system model.
/* This file contains the PROMELA/spin version of an SCRTool specification...

#define TRUE 1
#define FALSE 0
#define Activate 0
#define Deactivate...

bool Brake_OLD = FALSE;
bool Brake_NEW = FALSE;
bool EngRun_OLD = FALSE;
bool EngRun_NEW...

init {
    /* main processing loop */
    do 
        /* "any state" specification asserts */
        assert(!(Mode_NEW == Off)) || (!Ignited_NEW));
        assert(!(Mode_NEW == Inactive))...

    /* update each variable and mode class for this state change */
    d_step {
        Brake_OLD = Brake_NEW;
        EngRun_OLD...
    }

    /* simulate monitored variable changes */
    if 
        :: (Brake_OLD) -> Brake_NEW = FALSE
        :: (!Brake_OLD) -> Brake_NEW = TRUE
    fi;
    ::if
        :: (EngRun_OLD) -> EngRun_NEW...
    fi;
    fi;

    /* executions of the functions in dependency order */
    d_step {
        if 
            :: (Ignited_NEW && (!Ignited_OLD)) && (Mode_OLD == Off)
                -> Mode_NEW = Inactive;
            :: (Ignited_OLD && (!Ignited_NEW)) && (Mode_OLD == Inactive)
                -> Mode_NEW = Off;
            :: (((Lever_NEW == Activate)...
                fi;
        }
    od /* end of main processing loop */
}

Figure 4.8: SPIN version of the cruise control specification.
supports small, memory-saving data types including bool and byte, for 2 and 8-bit variables. Note that two copies are declared for each variable from the original SCR specification, var_OLD and var_NEW, where var is the original variable name. This is necessary because SCR allows model behavior to be based not just on the current value of a variable but on events defined to occur when the value of the variable changes. In Promela, the only way to know the previous value of a variable is to save it in another variable. So in the Promela version of the specification, the SCR event @T(var) would be represented: (var_NEW && !var_OLD).

A typical Promela model includes the process called init, which like main in a C program is called automatically when the model is executed (lines 13–51). In this model, the entire init process is comprised of a loop. Each iteration through the loop represents one step forward in the execution of the original SCR specification. Inside the loop, the process begins with single-state assertions (lines 17–19). (Two-state assertions, described later in comparing SMV and SPIN models produced by the SCR Toolset’s translators, involve both the _OLD and _NEW copies of each variable and would be listed at the end of the loop.) Next, each variable’s _OLD copy is updated by setting it equal to the _NEW copy of that variable (lines 21–25); then one monitored variable is chosen and its _NEW copy’s value is chosen at random from the set of possible values, excluding the value of its _OLD copy. (This is only done for one of the monitored variables: SPIN’s nondeterministic if structure executes just one :: branch.)

The model continues with the cruise control Mode transition function (lines 39–48). This is enclosed in a d_step block, which means it is treated by SPIN as a deterministic step, i.e., SPIN treats this block as if only one path through it is possible. If more than one path is possible, behavior on other paths besides the first will not be checked by SPIN. Although d_step is used earlier in the model (line 22) it has no effect there since no branching control structures are used. This use of d_step (line 40), however, referred to in section 3.1.2, can potentially hide some of the behavior of the original SCR specification from SPIN, if any of the transition functions inside this d_step block are nondeterministic.

Figure 4.9 shows the output from SPIN for the correct (top) and fault-seeded (bottom) versions of the cruise control system model used in previous examples above. Note that the depth reached in the fault-seeded version (bottom) is significantly less than in the correct version (top). This is because SPIN stops as soon as a property violation is detected. In general, far more time and
memory is required to verify a correct model than to detect a property violation in an incorrect model, because SPIN may only have to explore (and store) a small fraction of the global state space before an existing property violation is detected, but the entire global state space must be explored to verify that no property violations are possible. In this way SPIN, an on-the-fly model checker, works differently from the symbolic model checkers SMV and NuSMV, which build the entire global state space first (in a compact symbolic representation based on BDDs) and then explore it. Time and memory requirements for SMV and NuSMV thus do not tend to vary as much, compared to SPIN’s time and memory requirements, between different versions of the same input model.

### 4.2 Lurch Random Search Tool

Section 4.2 begins with background on the random search procedure used in our random search tool Lurch, which was originally based on techniques for assessing AND-OR graphs. Next, Lurch’s input language is illustrated with a simple example, and then the random search procedure used in the current version (no longer using AND-OR graphs) is presented, along with explanations of additional features relevant to synchronous, hierarchical finite-state machine models like those used in SCR specifications. The section concludes with a description of the process of translating from SCR to Lurch, done by way of the SCR Toolset’s translator to Promela for SPIN. The translation is illustrated by using the cruise control example from the previous section.
4.2.1 Random Search of AND-OR Graphs

The random search implemented in Lurch, which will be described in detail below, was originally based on a technique called abduction. Abduction runs on an AND-OR graph representing what is usually a vague domain—the model contains contradictions and gaps that reflect incomplete knowledge about the domain [50]. Abduction is a way of extracting from these models competing hypotheses that can be ranked according to their predictive power.

Menzies et.al. describes a generalized abductive hypothesis testing procedure called HT4, which performs an exhaustive search (requiring exponential time) for internally consistent worlds implicit in an AND-OR graph [51]. In order to find the best world (the world whose hypotheses have the most predictive power), HT4 finds every possible world in the AND-OR graph. To evaluate HT4, HT4-dumb was created; it simply chooses a single world (instead of looking for the best one). Surprisingly, HT4-dumb performed nearly as well as HT4. This prompted the development of a randomized hypothesis testing procedure, HT0, which randomly samples the space of possible hypotheses. HT0 is able to do in approximately $O(n^2)$ time what takes HT4 exponential time; in addition, HT0 works on models much too large for HT4 [51].

In previous work we modified HT0’s randomized hypothesis testing technique to apply to models of processes changing through time [58, 61]. Rather than searching an AND-OR graph representing a knowledge base for one internally consistent world, the new procedure searches a similar AND-OR graph representing a model of a process for a series of worlds. This series of worlds represents a valid execution path in the process model. For more details on the construction of AND-OR graphs representing finite-state models, as well as our random search procedure to run on those models, see [58].

4.2.2 Lurch Input Language

Figure 4.10 shows a simple finite-state model written for Lurch, the current implementation of the random search procedure for finite-state models mentioned in the previous section. The model begins with C code (lines 1–12), which may be referred to in finite-state machines defined later. (The special function before() is called by Lurch internally to set C variables to their initial values.) After the %% symbol (line 13), finite-state machines are listed. A blank line indicates the beginning of a new machine, and the first state listed in each machine is defined to be the initial
#define FALSE 0
#define TRUE 1

denum { A, B } turn = A;

int a = TRUE, b = TRUE;

void _before()
{
    turn = A;
    a = b = TRUE;
}

%%

a1; -; (a = TRUE;); a2;
a2; -; (turn = B;); a3;
a3; (!b); -; acs;
a3; (turn == A); -; acs;
acs; -; (a = FALSE;); a5;

b1; -; (b = TRUE;); b2;
b2; -; (turn = A;); b3;
b3; (!a); -; bcs;
b3; (turn == B); -; bcs;

bcs; -; (b = FALSE;); b5;

ok; acs,bcs; -; _fault;

Figure 4.10: Lurch Input Model representing Dekker’s solution to the two-process mutual exclusion problem (translated from a model written for SPIN in [41]).
state.

Each line in a finite-state machine description represents a transition and has four fields: the current state, input conditions, output or side affects of the transition, and the next state. Fields two and three may refer to the C code section of the model, either as input conditions, in field two, or as side affects in field three, to be executed with the transition. The final line is a two-state machine representing a safety property: the transition from ok to _fault is enabled if the first two machines are both in their critical sections (acs and bcs) simultaneously. State names that begin with an underscore are recognized by Lurch to represent safety property violations.
4.2.3 Basic Search Procedure

Figure 4.11 shows the core random search procedure used by Lurch. Lurch uses a Monte Carlo, not Las Vegas, random search algorithm [56]; that is, there is no guarantee that every state will be explored, but as the search runs the probability of exploring every state increases. This is in contrast to, for example, a Las Vegas random depth-first search, in which all nodes are explored once but the order in which child nodes are explored is random. Where random search has been used in model checking by others, e.g., [27], it has been primarily the Las Vegas approach; this is fundamentally different from our approach since it requires the record keeping of conventional deterministic model checking (the search must keep track of states previously visited) and therefore cannot scale to models as large.

The main search procedure is shown in lines 18–31. User-defined parameters max_paths and max_depth (line 18) determine how long the search will run. max_paths is the number of iterations, each of which generates a path from the initial state through the state space. Path length is limited by max_depth, although shorter paths may be generated if a state is reached from which no more transitions are possible. The state vector (line 21) includes an entry for each machine in the system. The value of a particular entry represents a local state, i.e., a state in a machine actually enumerated in the input model; a global state is a tuple with a local-state value for each machine. Lines 20–21 set all machines to their initial local state. Lines 23–31 generate a path of global states, checking each to see if it represents a fault.

The step function (lines 1–8) first inputs a queue of unblocked transitions and the state vector, then removes a random transition from the queue, and then modifies the state vector according to the effect of the transition. The queue is emptied in lines 7–8.

The check function (lines 10–15) checks the current state to see if it represents a fault. The state is checked for:

1. Local state faults, e.g., assertion violations.
2. Deadlocks—if any local state is not a legal end state, the global state represents a deadlock.
3. Cycle-based faults, including no-progress cycles and violations of user-specified temporal logic properties, which are represented by acceptance cycles.
To do cycle detection Lurch stores a hash value for each unique global state in the current path. This requires some additional time and memory and can be turned off if the user is not interested in looking for cycle-based faults. Using the hash value storage needed for cycle detection, and based on the saturation idea described above, an early-stopping mechanism is implemented in Lurch that works in the following way. For each path generated, hash values are saved for all unique global states visited. (This is done already for cycle detection.) The number of new values (values which are not hash collisions) is compared to the total number of values, including hash collisions. When the percentage of new values drops below a user-defined threshold (default is 0.01%), the search is terminated. In our experiments, saturation is usually achieved quickly; also, when Lurch is allowed to run to saturation it nearly always produces consistent results.

### 4.2.4 Additional Features

Lurch’s basic search procedure, shown in figure 4.11, simulates asynchronous execution of the finite-state machines in the input model. Each time step only one machine (selected at random) is given a chance to transition forward. In this way it is possible for Lurch to explore any of the machines’ possible interleavings. Figure 4.12 shows how the `step` function can be modified to simulate synchronous execution of machines, as is needed for, e.g., input models representing SCR specifications. The first transition chosen is executed (line 6), and then any other transitions from its machine are disqualified for the current step (lines 8–9). Then, instead of clearing the queue, all transitions in the queue but not yet disqualified are processed. In effect, all (not just one) of the machines are given a chance to transition forward in each step. But interleavings of the individual machines within a step are ignored.

If the code in figure 4.13 is substituted for lines 18–31 in the original search procedure (fig-
main(max_paths, max_depth)
  for (i in 1...max_paths)
    for (m in machines) /* set all machines to initial state */
      state[m] = 0
  for (d in 1...max_depth) /* generate a global state path */
    for (tr in transitions)
      for (g in machine_groups)
        if (check_inputs(tr) /* see if transition is blocked or comes
           AND check_group(g, tr)) from a machine in the wrong group */
          push(Q, tr) /* if not, put it in the queue */
        step(Q, state) /* get next global state */
        check(state) /* see if next state represents a fault */

Figure 4.13: main function modified for hierarchical execution of finite-state machines.

Lurch also uses this scheme to separate Büchi automata, used to represent temporal logic property violations, from the rest of the machines in the model. In this way the Büchi automaton is executed as a sort of observer of the rest of the system: at the end of each step in a global state path, when all other machines have finished, the Büchi automaton is given a chance to react to what has happened.

4.2.5 Translating from SCR to Lurch

Although SCR specifications are somewhat similar to Lurch input models, there are two significant challenges in translating from SCR to Lurch. First, individual finite-state machines in an SCR specification are executed in order according to variable dependencies. Lurch’s support for hier-
archival finite-state machine models, described in the previous section, makes it possible to force individual machines to be executed in a specified order, but the order must be made explicit in the Lurch model. (The order is not explicit in the SCR specification.)

The second significant difference between SCR and Lurch has to do with what information is included in the global state. SCR defines the current global state to include both the current state and the previous state of each local machine. This makes it possible to detect events, i.e., it is possible to inspect the global state and see that, for a particular machine, its current value is not equal to its previous value. Lurch, on the other hand, includes only the current value of each machine in the global state; with Lurch there is no straightforward way to detect events involving local state changes.

Fortunately for this research, Lurch’s input language and execution model are based on SPIN, and the existing translator from SCR to SPIN included in the SCR Toolset already resolves both the dependency order and event detection issues described in the previous two paragraphs. Instead of translating directly from SCR to Lurch we therefore wrote a simple translator from the automatically generated SPIN models produced by the SCR Toolset to Lurch. (This is not a full translator from Promela to Lurch, but takes advantage of the very predictable structure of models automatically generated by the SCR Toolset.) The code of the translator, written in AWK, is included in appendix C.

Figure 4.14 shows the Lurch version of the cruise control example model used in section 4.1. The model begins with #define macro definitions and global variable declarations, much like the Promela version above, but this time in standard C (lines 3–9). Next, the update_variables function sets the _OLD copy of each monitored variable equal to the _NEW copy (lines 11–28), and then chooses a random monitored variable and sets its _NEW copy to a value chosen randomly from all possible values for that variable, excluding the value of its _OLD copy. This function will be called once at the beginning of each step through the execution of the model.

The special Lurch functions _before and _hash (lines 30–40) are called by Lurch at particular times during its random search: _before is called at the beginning of each global state path and is used here to reset variables to their initial values; _hash is called at each step along a global state path to include values of variables in the C-code section of the Lurch model in the hash value calculated internally by Lurch for each global state. The %% symbol (line 42) marks the end of the
/* This file contains the Lurch version of an SCR Specification... */

#define TRUE 1
#define FALSE 0
#define Activate 0
#define Deactivate...

char Brake_OLD = FALSE;
char Brake_NEW...

void update_variables(void)
{
    int _p;
    Lever_OLD = Lever_NEW;
    Mode_OLD...

    switch (_next_int(3))
    {
    case 0: _p = 0;
        if ((Brake_OLD) && !_next_int(_p--)) { Brake_NEW=FALSE; break; }
        else if (!Brake_OLD && !_next_int(_p--)) { Brake_NEW=TRUE; break; }
    case 1: _p = 0;
        if (((EngRun_OLD) && !_next_int(_p--))...}
    default: break;
    }
}

void _before(void)
{
    Brake_OLD = FALSE;
    Brake_NEW...
}

void _hash(unsigned int *h)
{
    _hash_int(Brake_OLD, h);
    _hash_int(Brake_NEW, h)...
}

1: assert0; (!((!(Mode_NEW == Cruise)) || (((Ignited_NEW && EngRun_NEW) &&
45  ![Brake_NEW]) && ![Lever_NEW == Deactivate])))); -; _assert0_violated;
2: assert1; (((!(Mode_NEW == Override)) || (Ignited_NEW && EngRun_NEW))); -;
5: update_variables; -; [update_variables()]; update_variables;
6: Mode; [[[Levold==Activate]| ![Levold==Activate]]&&(EngRun_OLD&&
6: Mode; [[Ignited_OLD&&[Ignited_OLD])&&(Mode_OLD==Cruise)]; {Mode_OLD==Off};
55 Mode...

Figure 4.14: Lurch version of the cruise control specification.
C-code section of the model and the beginning of finite-state machine definitions.

Each assertion from the original SCR specification is represented by a finite-state machine with just one transition (lines 44–48). These are executed first in the dependency order (indicated by, e.g., 1:, 2:, etc.). Next, a single-transition finite-state machine is used to call the `update_variables` function (line 50). Since this transition has no input condition specified, it will be called every time Lurch reaches this point in the execution of the model, i.e., once corresponding to each step forward in the execution of the original SCR specification. The model concludes with a finite-state machine representing the cruise control mode transition function (lines 52–55). Note that from Lurch’s point of view the `Mode` finite-state machine has a single state, `Mode`. Each time Lurch attempts to execute a transition from this machine, C variables representing the cruise control mode and monitored variables are checked, and based on their values the C variable `Mode_NEW` is updated (e.g., line 53).

The Lurch model of the cruise control system is structured in a somewhat strange way. The finite-state machine portion of the model, which would normally include the main logic and behavior of the system, is instead a sort of dummy system acting as an interface between Lurch’s random search procedure and the C-code portion of the model, which in this case includes the real logic and behavior of the system. In spite of this strange use of Lurch, very different from what was originally intended, Lurch works well, compared to the other verification tools described above, in the experiments in chapter 5.

Figure 4.15 shows Lurch’s output running on the correct (top) and fault-seeded (bottom) versions of the cruise control specification used as an example in section 4.1. The `% new` and `col` statistics report information related to the hash values calculated by Lurch for each global state. The `% new` value of 97.1 indicates that at the point when the assertion violation was detected, of all
global state hash values calculated by Lurch, 97.1% were unique. The col value of 0 means that the current global state path, from initial conditions to the state in which the assertion violation was detected, is made up entirely of unique global states. If the global state path contained one or more hash collisions, this would indicate that the path (very likely) contained a cycle.

4.3 Summary

The SCR Toolset (and integrated back-end verification tools) provide a convenient framework for multiple-tool verification experiments. The SCR Toolset includes a specification editor for developing executable models written in the SCR tabular notation, a simulator for interactively executing SCR models, a dependency graph browser for viewing relationships between variables, and a consistency checker for detecting syntax errors and checking certain kinds of generic properties. In addition, automatic translators are provided for the symbolic and explicit-state model checking tools SMV and SPIN, the simplified theorem proving tool TAME, and the invariant checker Salsa. Also, we have written scripts to 1) make minor modifications to the SCR tools’ SMV version of a specification to support NuSMV, and 2) to translate from the SCR tools’ SPIN version of a specification to our random search tool, Lurch.

Lurch was originally based on a technique for searching AND-OR graphs representing finite-state machine models. Surprisingly, a quick random search of such AND-OR graphs can detect errors almost as effectively as a much more resource-intensive systematic search. Lurch allows a combination of C-code and a simple language for representing finite-state machine transition functions. Lurch’s basic search algorithm executes the input model asynchronously, allowing for all possible interleavings of concurrent machines’ behavior. Lurch checks for single-state faults, deadlocks, and cycle-based faults—no progress cycles and cycles representing violations of temporal logic properties. Two features added to Lurch’s basic algorithm, relevant to the experiments presented in this dissertation, are 1) the ability to execute machines synchronously, ignoring interleavings, and 2) the ability to force execution of machines to follow a specified order. Finally, to produce a Lurch version of an SCR specification, we translate from the SCR Toolset’s SPIN version of a specification to Lurch, rather than directly from SCR to Lurch, because the major challenges in translation to Lurch are the same as for SPIN and already dealt with in the SCR
Toolset’s SPIN translator.

The next chapter describes experiments using the SCR Toolset and back-end verification tools Salsa, Cadence SMV, NuSMV and SPIN, developed by other researchers and integrated together into a framework for modeling and verification of SCR specifications. For the experiments below we also use Lurch, integrated into the SCR Toolset as part of our research work done in preparation for this dissertation. We consider strengths and weaknesses of these tools when applied to SCR specifications, using the automatic translators provided in the SCR Toolset and the translator we developed for Lurch, and propose a general verification strategy combining the tools in a way that exploits complementary relationships between them.
Chapter 5

Case Study

Chapter 5 outlines the experimental procedure and summarizes general results from the main case study example presented in this dissertation. Section 5.1 describes the input model used, an SCR specification for a personnel access control system (PACS) previously published as an example to show how to develop a high quality software requirements specification. Section 5.2 delineates our process for automatically generating a set of fault-seeded versions of the original SCR specification. We describe in detail mutation operators used to produce seeded faults and then give summary information about the set of fault-seeded specifications used in the experiments.

Section 5.3 first describes the experimental process carried out for each fault-seeded specification: the order in which verification tools were run, the settings and precise way in which each tool was used, and the data collected. Next, we summarize experimental results: fault-seeded specifications are divided into subsets based on which tools were able to detect faults in each specification, and average time and memory requirements are reported for each tool running on each of these subsets. Section 5.3 concludes by comparing experimental results for fault-seeded specifications generated automatically, according to the procedure described in section 5.2, versus fault-seeded specifications generated manually for earlier the experiments described in chapter 3.

5.1 PACS SCR Specification

In the experiments presented below and in sections 3.1.2 and 3.2.1, we used an SCR specification for a personnel access control system (PACS) described in a prose requirements document from the National Security Agency [65]. These requirements have been used in others’ work comparing the effectiveness of processed-based and formal-methods-based strategies for developing reliable
software [74]. The SCR specification was derived from that document as an example to demonstrate how to write a high quality formal requirements specification and has been used by others to evaluate compositional verification methods [17].

The PACS checks information on magnetic cards and PIN numbers to limit physical access to a restricted area to authorized users. To gain access, the user swipes an ID card containing the user’s name and social security number (SSN) through a card reader. After using its database of names and SSNs to make sure that the user has the required access privileges, the system instructs the user to enter a four-digit personal identification number (PIN). If the entered PIN matches a stored PIN in the system database, the PACS allows the user to enter the restricted area through a gate. To guide the user through this process, the PACS includes a single-line display screen. A security officer monitors and controls the PACS using a console with a second single-line display screen, an alarm, a reset button, and a gate override button.

To initiate the validation process, the PACS displays the message Insert Card on the user display and, upon detecting a card swipe, validates the user name and SSN. If the card is valid, the PACS displays Enter PIN. If the card is unreadable or the information on the card fails to match information in the database, the PACS displays Retry for a maximum of three tries. If after three tries the user’s card is still invalid or there is no match, the system displays See Officer on both the user display and the officer display and turns on an alarm on the officer’s console. Before system operation can resume, the officer must push the reset button. The user, who has three tries to enter a PIN, has a maximum of five seconds to enter each of the four digits before the PACS displays the Invalid PIN message. If three times either an invalid PIN is entered or the time limit is exceeded, the system displays See Officer on both the user and the officer display. After receiving a valid PIN, the PACS unlocks the gate and instructs the user to Please Proceed. After 10 seconds, the system automatically closes the gate and resets itself for the next user.

Figure 5.1 shows a finite-state machine representing the core mode logic of the SCR model of the PACS requirements. (The full SCR model is included in appendix B.) Initially, the mode is EnterCard; when a card is entered the mode changes to CheckCard. If the card is not valid, a limited number of retries are allowed, during which time the mode alternates between CheckCard and ReEnterCard. If the card is valid, the mode changes to EnterPIN; when a PIN is entered the mode changes to CheckPIN. Similar to CheckCard, from CheckPIN the user has a limited number
of retries if an invalid PIN is entered, during which time the mode alternates between CheckPIN and ReEnterPIN. If a valid PIN is entered the mode changes to Proceed. In mode Proceed, the user is able to enter through the gate. Once the gate closes, the system is reset to EnterCard.

In modes CheckCard and CheckPIN, if the maximum number of retries is reached after repeated invalid card or PIN entries, the mode changes to Error. From Error the officer may override the PACS, the mode of which then changes to Override. The user may then enter through the gate. When the gate is closed, the mode changes to EnterCard. Also, if the system is reset by the officer in any mode (except EnterCard), the mode is reset to EnterCard (note the dotted line transition in the upper right part of figure 5.1).

5.2 Generating Fault-Seeded Versions of the PACS SCR Specification

Section 5.2 describes the process we used to create a set of fault-seeded versions of the PACS SCR specification. Fault-seeded versions were created by randomly applying mutation operators based on those given in [57]. After a description of the mutation operators, the section concludes with a summary of the fault-seeded specifications used in the experiments below.
<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOR</td>
<td>Arithmetic Operator Replacement</td>
<td>+ → −</td>
</tr>
<tr>
<td>CRP</td>
<td>Constant Replacement</td>
<td>1 → 2</td>
</tr>
<tr>
<td>EVR</td>
<td>Enumerated Type Value Replacement</td>
<td>a → b (where a and b are possible values for the enumerated type)</td>
</tr>
<tr>
<td>IOR</td>
<td>Implication Operator Replacement</td>
<td>=&gt; → &lt;=&gt;</td>
</tr>
<tr>
<td>LCR</td>
<td>Logical Connector Replacement</td>
<td>AND → OR</td>
</tr>
<tr>
<td>ROR</td>
<td>Relational Operator Replacement</td>
<td>&lt; → &lt;=</td>
</tr>
<tr>
<td>SND</td>
<td>SCR Next Operator Deletion</td>
<td>′ →</td>
</tr>
<tr>
<td>SOR</td>
<td>SCR Event Operator Replacement</td>
<td>@T → @F</td>
</tr>
<tr>
<td>UOD</td>
<td>Unary Operator Deletion</td>
<td>NOT →</td>
</tr>
<tr>
<td>VRP</td>
<td>Variable (same type) Replacement</td>
<td>x → y (where x and y are variables of the same type)</td>
</tr>
</tbody>
</table>

Table 5.1: Mutation operators used to generate fault-seeded versions of the PACS SCR specification.

### 5.2.1 Mutation Operators

Table 5.1 shows mutation operators used to generate fault-seeded versions of the PACS SCR specification. This set of operators is adapted from a set of five determined to be sufficient by Offutt et.al. for Fortran programs [57]. Operators AOR, LCR and ROR are taken directly from that set. UOD is similar to the unary operator insertion (UOI) mutation operator included in the set from Offutt et.al. but is easier to implement without a full parse of the input specification.

The set of five sufficient mutation operators from Offutt et.al. also includes an absolute value insertion (ABS) operator, which replaces an entire arithmetic expression with zero, a positive value, or a negative value. To avoid fully parsing the input specification, and because SCR does not assign a logical value to arithmetic expressions (i.e., SCR does not define 1 to be true and 0 to be false), ABS was not used. Instead we used CRP, from Offutt et.al. but not one of the five selected, and EVR, which replace individual integers or, for variables of enumerated type, other legal values. IOR, SOR and SND are similar to LCR and UOD, but deal with SCR-specific features. VRP, like CRP comes from Offutt et.al. but was not one of the five sufficient operators selected. We used it because it (arguably) makes more sense in an SCR-specific context, where typically a small number of variables are declared for each of several user-defined enumerated types.

In Andrews et.al. [1] a similar set of mutation operators, adapted from Offutt et.al. to use with C
programs, is used to accurately evaluate test suites; that is, these mutation operators produce fault-seeded programs realistic enough that a given set of tests will achieve approximately the same level of program coverage, in terms of widely used coverage criteria, that the given set of tests would achieve for programs with real faults. Also, automatic fault seeding with these mutation operators is compared to manual fault seeding and found to yield more realistic results.

### 5.2.2 Summary of Fault-Seeded Specifications Used in Case Study

Figure 5.2 shows how many fault-seeded specifications were generated for each mutation operator, or for specifications generated using two mutation operators, for each pair of operators. Different numbers of specifications were created for different operators because, for each operator, there are a different number of possible places to apply it in the original specification. For example, there are many more places in the original specification where the EVR (enumerated type value replacement) operator may be applied than there are where the SND (SCR Next operator deletion) may be applied.

The program we used to (randomly) generate fault-seeded specifications therefore generated many more unique specifications with an EVR mutation than with an SND mutation. In fact, there were so few possible places in the original specification where the UOD operator could be applied that no fault-seeded specifications were produced with that operator. Also, the EVR-s operator, a variation on EVR in which enumerated type values in a list are swapped instead of randomly
replacing a value with some other possible value, was used in five of the specifications produced.

5.3 Case Study Experiments

Section 5.3 lists the tools used in our experiments, describing how each was used and summarizing accuracy and performance results. This is followed by a general summary of results considering all tools and all fault-seeded specifications.

5.3.1 Experimental Procedure

For each fault-seeded specification, the SCR Toolset was used to run basic generic checks and generate Salsa, SMV and SPIN versions of the specification. Salsa was run on the SCR Toolset’s Salsa version of the specification. SMV was run on the SCR Toolset’s SMV version, and NuSMV was run on version of the SMV model with the minor modifications described in section 3.1.1. Lurch was run on a model generated from the SPIN version of the specification by the script in appendix C, and then SPIN was run three times, each configured differently, on the SPIN version of the specification.

SCR tools

From the SCR Toolset, the command-line program testtool was used via scripts to automatically run generic checks on the specification—checks that would be run by the consistency checker if run from the SCR Toolset GUI—to check for syntax and type errors, duplicate names, unspecified or unused variables, missing or inconsistent initial values, circular definitions, and violations of disjointness or coverage properties [37]. As explained in section 3.1.2, disjointness violations occur when, from a certain state of the system, for a given input, the specification allows more than one possible next state. This nondeterminism is considered an error in SCR. Coverage violations occur when, from a certain state of the system, for a given input, no next state is specified.

The sets of fault-seeded specifications listed in figure 5.2 include only specifications for which no errors were detected by testtool. Any specification that failed any of the checks run by testtool was removed from the set to be used in the experiments, since our focus is on back-end verification tools, which there would be no reason to use if the basic SCR tools were able to detect errors in
the specification.

**Translation**

Each fault-seeded specification that passed all of the checks done by testtool was then opened in the SCR Toolset GUI and double-checked manually using the checks provided in the GUI. If the specification passed these checks, the SCR Toolset’s translators were used to automatically generate Salsa, SMV and SPIN versions of the specification. In addition to a pass or fail result, it is possible to get a warning from the SCR Toolset consistency checker. These warning specifications were not removed from the set to be used in the experiments, since in practice additional back-end verification tools would be used to determine whether the warning result corresponded to a real error.

To create NuSMV versions of the SMV models output by the SCR tools’ translator, a simple script was used to substitute INVARSPEC for SPEC and delete AG from the portion of the SMV model representing assertions in the original SCR specification. (This was to remove the possibility of inconsistent results between NuSMV and Cadence SMV, as discussed in section 3.1.1.) To create Lurch versions of the SPIN models output by the SCR Toolset’s translator, we used the script included in appendix C, written in the AWK interpreted language.

**Salsa**

Next, we ran Salsa on the Salsa version of the fault-seeded specification and compared the results to those produced by running Salsa on the Salsa version of the original correct specification. Specifications were divided into five categories based on the results produced by Salsa:

1. Those for which Salsa was able to prove fewer of the assertions than could be proved for the original SCR specification (94 specifications).
2. Those for which Salsa could prove more of the assertions (16).
3. Those for which Salsa’s results for the assertions matched results on the original specification but Salsa proved fewer generic properties (36).
4. Those for which results for the assertions matched the original but Salsa proved *more* generic properties (7).

5. Those for which Salsa’s results, for assertions and generic properties, were identical to results on the Salsa version of the original SCR specification (170).

Section 6.2.1 provides more details about specifications from these sets, including relationships between Salsa results and the results of other verification tools. Time and memory requirements were recorded for each Salsa run on each specification, although there was little variation from one run to another. Average time required was 1.20 seconds; memory required for every run was the same: 1.59 megabytes. Section 6.3 discusses how a multiple-tool verification strategy in which specifications are modified based on Salsa results so that when other tools are run properties already proved true by Salsa are ignored. Section 3.1.3 describes how Salsa can be used as an efficient method for validating property violations detected by SPIN, which may be spurious due to the fact that \textit{NATURE} constraints are ignored by the SCR Toolset’s SPIN translator.

**Cadence SMV and NuSMV**

Cadence SMV and NuSMV were next run on the SMV version of each fault-seeded specification, with NuSMV running on a modified version with the changes described in section 3.1.1. With the minor changes for compatibility with NuSMV, Cadence SMV and NuSMV results were always consistent. Property violations were detected in 141 specifications; no violations were detected in 182 specifications. The SCR Toolset’s translator to SMV restricts the type of assertions used to those involving only the current state of the system. That is, any assertion using the SCR \textit{Next} operator (‘) or any event operator (@T, @F, @C) is removed from the SMV version of the specification. For our experiments this meant that only 9 of the 16 assertions in the original SCR specification were checked by Cadence SMV and NuSMV. This limitation in the effectiveness of SMV (Cadence SMV or NuSMV) running on models generated from SCR specifications is also beneficial, because it simplifies the input models and is one possible explanation for the very low time and memory requirements of Cadence SMV and NuSMV, compared to SPIN (described below).

Because some kinds of assertions possible in an SCR specification are not included in the
SCR Toolset’s SMV version of the specification, Cadence SMV or NuSMV can be used as a preprocessor in cases where no faults are detected. Properties proved true by SMV can be removed from the specification so that later verification tools can be run on a simpler input model. This is described in section 6.3, along with similar use of Salsa.

Time and memory requirements were recorded for each run of Cadence SMV and NuSMV. There was more variation than with Salsa, especially in the amount of memory required by NuSMV, but still the amount of variation was very small compared to SPIN (described below). The average time required for Cadence SMV was 0.107 seconds, for NuSMV 1.21 seconds; the average memory required for Cadence SMV was 3.48 megabytes, for NuSMV 13.2 megabytes.

### Lurch

Next, we ran Lurch on versions of the fault-seeded specifications generated by the script in appendix C from SPIN versions of the specifications produced by the SCR Toolset. Because Lurch’s random search does not necessarily return consistent results, Lurch was run between 10 and 50 times on each input model. Only in cases where Lurch detected a property violation at least 10 times was Lurch counted as having detected the violation. If Lurch found a violation ten times before 50 runs, we stopped running Lurch on that input model. As shown in table 5.2, for most

<table>
<thead>
<tr>
<th>10 violations in under 50 runs</th>
<th>Less than 10 violations in 50 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 / 10 (175)</td>
<td>0 / 50 (117)</td>
</tr>
<tr>
<td>10 / 11 (16)</td>
<td>1 / 50 (2)</td>
</tr>
<tr>
<td>10 / 12 (5)</td>
<td></td>
</tr>
<tr>
<td>10 / 13 (4)</td>
<td></td>
</tr>
<tr>
<td>10 / 17 (1)</td>
<td></td>
</tr>
<tr>
<td>10 / 27 (1)</td>
<td></td>
</tr>
<tr>
<td>10 / 38 (1)</td>
<td></td>
</tr>
<tr>
<td>10 / 39 (1)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Lurch results on fault-seeded PACS specifications: number of times violations detected vs. search runs, number of specifications in parentheses.

---

1 For all tools memory requirements were determined by using the /proc/id/stat pseudo-file, where id is the ID number of a running process, available in Cygwin, UNIX emulation software running on Windows XP. For all tools except Cadence SMV, memory requirements shown in /proc/id/stat were consistent with memory requirements shown in the Windows Task Manager program. For Cadence SMV, memory requirements shown in /proc/id/stat were about 1/3 what was shown in Windows Task Manager. For our experimental results we used the lower values.
input models (292 of 323) Lurch either detected a property violation ten times in the first ten runs or detected no violation in 50 runs. In only a few cases (6 of 206) in which a violation was detected did Lurch find the violation in less than 75% of runs.

For input models in which Lurch detected a property violation at least 10 times, average time and memory requirements for all runs, including those in which no property violations were detected, were recorded for comparison with the other tools. So, for example, if Lurch had to run 20 times to detect violations in 10 of those runs, all twenty runs were included in average time and memory values. Average time required by Lurch for a single input model (in which violations were detected) ranged from 0.144 seconds to 62.7 seconds; overall average time was 2.72 seconds. Memory requirements showed little variation from one model to another, with an overall average of 5.68 megabytes.

Because input models used with Lurch were based on SPIN versions of the specifications produced by the SCR tools, all assertions in the original SCR specification (including assertions not included in SMV versions of the specifications) were checked by Lurch. This is why Lurch detected a larger number of property violations than Cadence SMV or NuSMV. Otherwise, an incomplete tool like Lurch would never detect more violations than a complete tool like Cadence SMV or NuSMV. In addition, because Lurch input models were derived from the SCR Toolset’s SPIN version of the fault-seeded specifications, NATURE constraints are ignored by Lurch as well (see section 3.1.3).

**SPIN**

Finally, SPIN was run on versions of the fault-seeded specifications produced by the SCR Toolset’s Promela translator, in the following three ways:

1. First, run SPIN with default settings (default depth limit 10,000).

2. If no violation found, run with settings necessary to get complete verification runs even on input models for which no violations were detected (compile with minimized automaton memory compression, run with depth limit 2,000,000).

3. If (still) no violation found, run on input model with final d step marker removed, as described in section 3.1.2, and with settings necessary for complete verifications runs on mod-
Table 5.3: Average time and memory, and standard deviation values for averages, required by SPIN for fault-seeded specifications, with settings adjusted to run in three ways.

<table>
<thead>
<tr>
<th>Settings</th>
<th>Time (s)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>average</td>
</tr>
<tr>
<td>Default (205)</td>
<td>0.0300</td>
<td>3.41</td>
</tr>
<tr>
<td>Complete (26)</td>
<td>68.9</td>
<td>561</td>
</tr>
<tr>
<td>Complete, no d_step (2)</td>
<td>1330</td>
<td>1340</td>
</tr>
</tbody>
</table>

Table 5.3 shows time and memory requirements for SPIN running on the fault-seeded specifications, with settings adjusted in three ways. With default settings, SPIN detected property violations in 205 of 323 input models. With settings adjusted to insure a complete verification run, SPIN was able to detect violations in 26 more of the models. Time values given for SPIN with these settings are based on the sum of the time required to run SPIN with default settings and the time required to run SPIN with adjusted settings. Memory values are maximum values for default or adjusted settings.

Section 3.1.2 showed that in order to get a fully reliable verification result running SPIN on an input model translated from an SCR specification by the SCR tools, it is necessary to remove the final \texttt{d\_step} marker from the model. Running SPIN on input models with this change, with settings adjusted to enable a complete verification run, SPIN was able to detect property violations in two more of the models. Time values given for SPIN run in this way are sums of times for running SPIN in all three configurations, since the only way to know the third configuration is necessary is to first run the other two. Memory values are maximum values for all SPIN configurations run on a given specification.

SPIN requires much more time and memory, in most cases, than the other tools described above. But in our experimental framework only SPIN can be used to fully verify all 16 of the assertions in the original SCR specification. Based on results from SPIN, we determined that 90 of the fault-seeded specifications were equivalent mutants; that is, they specify behavior identical to the original, as far as the assertions are concerned. Also, as stated in the section above describing Lurch, SPIN in the context of the SCR Toolset ignores SCR \texttt{NATURE} constraints, so property
violations reported by SPIN must be validated using Salsa, one of the SMV model checkers, or manually using the SCR Simulator.

5.3.2 Overview of Experimental Results

Figure 5.3 summarizes experimental results for all tools except Salsa, showing sets of specifications in which property violations were detected by all (122 of 233 non-equivalent mutants); Lurch and SPIN only (82); Cadence SMV, NuSMV and SPIN only (19); SPIN only (10); and equivalent mutants (90). Results for Cadence SMV and NuSMV are shown together, as (Nu)SMV, since these tools detected property violations in exactly the same set of specifications. Results for Salsa are not shown, because, as explained above, when Salsa fails to prove a property it does not necessarily mean that the property is violated. Thus there is no straightforward way to include the results from Salsa in this kind of diagram. Sections 6.2.1 and 6.3 discuss how Salsa results might be integrated with the other verification tools in a useful way.

Tables 5.4 and 5.5 show average time and memory requirements for Cadence SMV, NuSMV, Lurch and SPIN running on the sets of fault-seeded specifications shown in figure 5.3. Since in some cases values vary a great deal from one specification to another, standard deviation values are
<table>
<thead>
<tr>
<th>Sets of Specifications</th>
<th>Cd. SMV</th>
<th>NuSMV</th>
<th>Lurch</th>
<th>SPIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>All detected a fault (122)</td>
<td>0.980</td>
<td>1.28</td>
<td>1.36</td>
<td>3.81</td>
</tr>
<tr>
<td></td>
<td>0.0420</td>
<td>2.30</td>
<td>2.38</td>
<td>6.33</td>
</tr>
<tr>
<td>Lurch but not (Nu)SMV detected a fault (82)</td>
<td>4.76</td>
<td></td>
<td></td>
<td>34.8</td>
</tr>
<tr>
<td></td>
<td>10.4</td>
<td></td>
<td></td>
<td>208</td>
</tr>
<tr>
<td>(Nu)SMV but not Lurch detected a fault (19)</td>
<td>0.167</td>
<td>0.733</td>
<td></td>
<td>634</td>
</tr>
<tr>
<td></td>
<td>0.0440</td>
<td>0.226</td>
<td></td>
<td>358</td>
</tr>
<tr>
<td>Only SPIN detected a fault (10)</td>
<td>316</td>
<td></td>
<td></td>
<td>358</td>
</tr>
</tbody>
</table>

Table 5.4: Summary of verification results for non-equivalent mutants (average time in seconds and standard deviation for time values).

<table>
<thead>
<tr>
<th>Sets of Specifications</th>
<th>Cd. SMV</th>
<th>NuSMV</th>
<th>Lurch</th>
<th>SPIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>All detected a fault (122)</td>
<td>3.45</td>
<td>13.1</td>
<td>5.62</td>
<td>51.6</td>
</tr>
<tr>
<td></td>
<td>0.534</td>
<td>3.07</td>
<td>0.315</td>
<td>81.6</td>
</tr>
<tr>
<td>Lurch but not (Nu)SMV detected a fault (82)</td>
<td>5.78</td>
<td></td>
<td></td>
<td>43.7</td>
</tr>
<tr>
<td></td>
<td>0.792</td>
<td></td>
<td></td>
<td>90.5</td>
</tr>
<tr>
<td>(Nu)SMV but not Lurch detected a fault (19)</td>
<td>3.64</td>
<td>13.4</td>
<td></td>
<td>493</td>
</tr>
<tr>
<td></td>
<td>0.638</td>
<td>1.14</td>
<td></td>
<td>66.3</td>
</tr>
<tr>
<td>Only SPIN detected a fault (10)</td>
<td></td>
<td></td>
<td></td>
<td>381</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>186</td>
</tr>
</tbody>
</table>

Table 5.5: Summary of verification results for non-equivalent mutants (average memory in megabytes and standard deviation for memory values).
also shown for each average value in the tables. For the 122 specifications in which all four tools
detected a property violation, using Cadence SMV (the fastest) vs. SPIN (the only tool capable
of fully verifying all specifications) saves 345 seconds, or about 6 minutes. For the set of 82
specifications in which property violations were detected only by Lurch and SPIN, running Lurch
vs. SPIN saves 2,463 seconds, or about 41 minutes. The greatest time benefit is for the set of
19 specifications in which Cadence SMV, NuSMV and SPIN, but not Lurch, detected property
violations. For this set running Cadence SMV vs. SPIN alone saves 12,043 seconds, or about
3.5 hours. So even though this is a small number of specifications, they are among the easiest
for Cadence SMV and the most difficult for SPIN. Similarly, running Cadence SMV on these
specifications requires far less memory than SPIN. Chapter 6 considers these results in more detail,
from various perspectives.

Results shown in figure 5.3 and tables 5.4 and 5.5, as well as experimental results discussed
in chapter 6, are based on 323 fault-seeded specifications: 278 generated automatically accord-
ing to the process described in section 5.2 and 45 generated manually for the earlier experiments
described in sections 3.1.2 and 3.2.1. We decided to include the manually generated specifica-
tions, first, because the same mutation operators were used to generate them, and fault-seeding
was not based on any expert knowledge of the system; second, because performance results for
different verification tools was very similar for the manually seeded specifications, compared to
the automatically generated fault-seeded specifications. The only statistically significant difference
in performance ($p = 95\%$) was in memory requirements for NuSMV, the average for which just
changed from 11.9 to 13.5 megabytes, a very minor difference compared to the overall variation in
memory requirements, considering all tools.

5.4 Summary

For the main case study experiments presented in this dissertation, we used fault-seeded versions
of the PACS SCR specification. The PACS model represents a security system limiting access to
a restricted area, so that only users with a valid card and PIN number may enter. Fault-seeded
versions of the PACS specification were generated by applying a set of mutation operators, based
on a minimal set of sufficient operators developed by others for Fortran (and later C) programs, to
evaluate testing methods.

Fault-seeded specifications were checked first using the SCR Toolset. We excluded from our experiments any specifications for which the basic SCR checks failed, since our focus is on verification tools, which would not make sense to use on specifications already known to contain syntax errors or other generic kinds of faults. We next used the SCR Toolset to generate, for each specification, a Salsa version, an SMV version, and a SPIN version. The SMV version was modified to create a NuSMV version, and the SPIN version was translated into a Lurch version. We then ran Salsa, Cadence SMV, NuSMV, Lurch and SPIN on each specification. SPIN was run in three different modes, each with different configuration settings, to eventually produce a fully formal verification result. (Other tools contribute to the result, but within our experimental framework are not capable of fully verifying specifications alone.) A comparison of results from the different verification tools showed clear evidence for complementary relationships between the tools. For example, 19 fault-seeded specifications were found to be among the easiest for Cadence SMV and NuSMV and among the most difficult for SPIN.
Chapter 6

Discussion

In chapter 6 we look more closely at the experimental results summarized in the previous chapter, breaking the data into different subsets and considering implications brought out from these different perspectives. In section 6.1 each fault-seeded specification is reduced to four data values: the maximum and minimum time required for any tool running on that specification, and the maximum and minimum memory required for any tool running on that specification. These values are plotted to illustrate a key idea motivating a combination of complementary verification strategies, that nearly all specifications are easy for at least one tool, even if they are hard for others. Based on that idea, section 6.1 continues by comparing experimental results for two simple performance-based combination strategies.

Section 6.2 examines the effect of mutation operators on the performance of individual tools, compared to the combination strategies described in section 6.1. First, fault-seeded specifications are divided into subsets based on which mutation operator was used to generate the specification, and average results for each tool and combination strategy are reported for each of these subsets. In some cases, looking at particular subsets makes complementary relationships between tools clear. Next, specifications are divided into subsets based on whether one or two mutations were used to generate the specification, and results for each tool and combination strategy are reported for each of these subsets. Dividing specifications in this way brings out a complementary relationship between symbolic and explicit-state model checking techniques.

Finally, section 6.3 proposes a combination strategy designed to exploit relationships between tools complementary in terms of both accuracy and performance. Cadence SMV and Salsa are used first to filter out properties easily proved true using these tools. Next, if no property violation is
detected by Cadence SMV, Lurch is used to catch whatever faults can be detected quickly. Finally, SPIN is used fully verify any input models with no faults detected by Cadence SMV or Lurch. This combination strategy is then compared to two complete strategies that use only SPIN.

### 6.1 Performance-Based Combined Strategy

In this section, we consider evidence for complementary relationships between tools, that in most cases a particular specification will be easy, in terms of time and memory requirements, for one or more tools, even if it is difficult or impossible for another tool. Based on this, we propose two simple performance-based multiple tool verification strategies. Average resource requirements for these combined strategies are then compared to resource requirements for individual tools.
6.1.1 Performance Variations Between Tools

Figure 6.1 shows each fault-seeded specification, excluding equivalent mutants, plotted as a point whose x-coordinate is the maximum time required for any tool to detect a property violation and whose y-coordinate is the minimum time required for any tool to detect a property violation. For example, the point in the lower right corner of the largest dotted box represents a specification for which the fastest tool to detect a property violation required about 0.02 seconds, while the slowest tool to detect a property violation required about 20 seconds. (Note that a logarithmic scale is used for both axes.) Points plotted with an x-coordinate of infinity represent specifications for which one or more tools were never able to detect a property violation, regardless of time allotted. Nearly half of fault-seeded specifications for which a fault was detected are in this category (equivalent mutants are not shown in figure 6.1). Two reasons why there were such a large number of specifications for which one or more tools could not detect a violation are 1) Cadence SMV and NuSMV could not check 6 of the 15 properties, since two-state assertions are not compatible with the SCR Toolset’s SMV translator; and 2) although Lurch was able to check for violations of all 15 properties, it is incomplete.

Figure 6.2 is similar to figure 6.1, except that points represent the maximum and minimum memory required by the tools. Again, points plotted with an x-coordinate of infinity represent specifications for which one or more tools were unable to detect any property violation.

These figures are meant to illustrate the complementary relationships between tools used in our experiments. If tools were not complementary, we would expect to see points plotted along a 45-degree line from the origin to the upper right corner of the graph, indicating that specifications easy for a given tool are easy for all tools and that specifications difficult for one tool are difficult for all tools. We do see a large set of specifications (122) easy for all tools. (For these specifications, no tool requires much more than 10 seconds or 100 megabytes.) But nearly all specifications that represent significant challenges for some tools require less than 100 seconds (107 specifications) or less than 50 megabytes (103 specifications) for at least one other tool. That is, the tools are complementary: nearly all specifications are relatively easy to check for at least one of the verification tools, including specifications difficult or impossible for one or more other tools.
Figure 6.2: Specifications plotted to show maximum and minimum memory requirements for any tool.
### 6.1.2 Combined Strategy Based on Performance Variations

The *Best* and *Worst* columns in tables 6.1 and 6.2 show average time and memory requirements for the sets of specifications marked in figures 6.1 and 6.2. The *Worst* column is left blank for specifications impossible for one or more tools to check. The columns labeled *SLS* and *LSS* show average time and memory required for two simple multiple-tool verification strategies based on the performance variations described in the previous section. For *SLS*, Cadence SMV was run first; then if no fault was detected by SMV, Lurch was run for seven seconds; then if no fault was detected by Lurch, SPIN was run in one, two, or three configurations as described in section 4.1. For *LSS*, Lurch was run first, then SMV, then SPIN. Seven seconds was the optimal whole-integer time cutoff for Lurch, but we found similar results with cutoff values ranging from 1 to 30 seconds.

Both combination strategies achieve close to the *Best* time and memory performance, although *SLS* performance is a little bit better than *LSS*. These results only consider performance variations in detection of faults. In section 6.3 below, we show how, by taking into account both accuracy...
Table 6.3: Average time (s) required by individual tools for sets of specifications distinguished by Salsa results (only statistically significant results shown).

and performance, it is possible to build a more sophisticated combined strategy, superior to any individual tool in both fault detection and verification of correct input models.

6.2 Performance of Individual Tools on Subsets of Specifications

In this section, we compare average resource requirements of individual tools, as well as the multiple-tool verification strategies proposed in the previous section, running on sets of specifications distinguished by different criteria. First, sets are distinguished by results from the Salsa invariant checker. Next, sets are distinguished by the mutation operator(s) used to generate specifications in each set. Finally, sets are distinguished by whether one or two mutations are present in the specification.

6.2.1 Specifications Categorized by Salsa Results

In section 5.3.1 five possible categories of Salsa results are listed. Tables 6.3 and 6.4 show sets of fault-seeded specifications divided based on those five categories: specifications for which Salsa proved fewer assertions, more assertions, fewer generic properties, more generic properties, and specifications for which Salsa results matched results on the original correct specification. Only
Table 6.4: Average memory (MB) required by individual tools for sets of specifications distinguished by Salsa results (only statistically significant results shown).

<table>
<thead>
<tr>
<th>Sets of Specifications</th>
<th>SMV (Incomplete)</th>
<th>NuSMV</th>
<th>Lurch</th>
<th>SPIN (Complete)</th>
<th>Best</th>
<th>SLS</th>
<th>LSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>All (233)</td>
<td>3.48</td>
<td>13.2</td>
<td>5.68</td>
<td>99.0</td>
<td>19.3</td>
<td>25.6</td>
<td>26.7</td>
</tr>
<tr>
<td></td>
<td>± 0.551</td>
<td>2.88</td>
<td>± 0.562</td>
<td>163</td>
<td>85.1</td>
<td>89.1</td>
<td>88.8</td>
</tr>
<tr>
<td>Salsa Proved Fewer assertions (94)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>36.1</td>
<td>2.84</td>
<td>7.12</td>
<td>8.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>± 78.5</td>
<td>1.53</td>
<td>12.2</td>
<td>11.9</td>
</tr>
<tr>
<td>Salsa Proved More assertions (16)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6.29</td>
<td>57.1</td>
<td>98.7</td>
<td>98.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>± 0.573</td>
<td>157</td>
<td>172</td>
<td>172</td>
</tr>
<tr>
<td>Salsa Proved Fewer generic properties (36)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12.8</td>
<td>5.56</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>± 2.67</td>
<td>195</td>
<td>113</td>
<td>113</td>
</tr>
</tbody>
</table>

specifications in which property violations were detected by one or more tools are shown; equivalent mutants have been left out. Also, the set of all fault-seeded specifications for which a property was detected is included for comparison.

For each set of fault-seeded specifications, average time and memory requirements are shown for verification tools Cadence SMV, NuSMV, Lurch and SPIN, and then for the Best tool and the two performance-based combined strategies, as described in the previous section. Values are not shown if the difference between a tool’s resource requirements for a particular set and for the rest of the specifications was not statistically significant ($p = 90\%$). Also, incomplete strategies (Cadence SMV, NuSMV and Lurch) are distinguished from complete strategies (SPIN, Best, SLS and LSS). Recall that, because of characteristics of the SCR to SMV translation and of the Lurch random search tool, these tools can not fully verify all of the assertions in the original SCR specification. Results for these tools, shown in tables 6.3 and 6.4, thus show average time and memory requirements only for the specifications in which the tool could detect a property violation. This is why, for example, table 6.3 shows that, for specifications for which Salsa proved fewer assertions, SMV required on average 0.986 seconds, while the Best tool required on average 4.70 seconds. The time shown for SMV reflects only those specifications for which SMV detected a fault, while the time shown for Best is an average value for all specifications in the set for which Salsa proved fewer assertions. So, for incomplete strategies, it makes sense to compare results within columns,
but from one column to the next results may reflect a different set of specifications. For complete strategies, all results in a row are for the same set of specifications.

As can be seen in the tables, the sets of specifications for which Salsa performed better than on the original, whether by proving more assertions or generic properties, are quite small—only 23 specifications total. Thus the lack of evidence for a statistically significant difference between results on these sets and the rest of the specifications is not surprising. Considering the other sets of specifications, however, there are some interesting things to point out. First, the set of specifications for which Salsa was able to prove fewer assertions than for the original (94 specifications) is somewhat easier for Lurch and much easier for SPIN (and, as a result, the other complete strategies). This means that Salsa might be used to predict the performance of complete verification strategies involving Lurch or SPIN.

On the other hand, the set of specifications for which Salsa proved fewer generic properties is much more difficult for Lurch (and therefore also the three combination strategies). Perhaps specifications with structure such that it is more difficult to prove desirable generic properties tend not to have the kind of structure most easily exploited by random search; that is, structure in which many paths lead to a relatively small number of key states, as described in section 2.2.6.

It is also interesting to compare tools’ performance on set of specifications for which Salsa’s results matched results on the original (170 specifications) to overall average time and memory requirements. Interestingly, Lurch detected faults in these specifications about twice as fast as the overall average time for Lurch, but SPIN was about twice as slow. If the four other sets are combined, to form the set of specifications for which Salsa’s results differed from the original (not shown in the tables—a total of 153 specifications), Lurch’s average time is 3.62 seconds and SPIN’s is 35.9 seconds. It would still make sense to run Lurch first on these specifications, since SPIN is likely to require more time to detect a violation. But perhaps Lurch ought to be run longer on these, since the expected time for Lurch to detect a violation is greater than for specifications for which Salsa’s results matched results on the original.

### 6.2.2 Specifications Categorized by Mutation Operator

Tables 6.5 and 6.6 show average time and memory requirements for tools and multiple-tool strategies running on sets of fault-seeded specifications distinguished by mutation operator. The acronyms
<table>
<thead>
<tr>
<th>Sets of Specifications</th>
<th>SMV</th>
<th>NuSMV</th>
<th>Lurch</th>
<th>SPIN</th>
<th>Best</th>
<th>SLS</th>
<th>LSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>All (157)</td>
<td>0.108</td>
<td>1.10</td>
<td>3.62</td>
<td>94.8</td>
<td>15.3</td>
<td>18.2</td>
<td>19.5</td>
</tr>
<tr>
<td></td>
<td>0.100</td>
<td>3.53</td>
<td>6.85</td>
<td>236</td>
<td>88.6</td>
<td>96.3</td>
<td>96.0</td>
</tr>
<tr>
<td>CRP (8)</td>
<td>time</td>
<td>stdv</td>
<td>99.4</td>
<td>101</td>
<td>101</td>
<td>280</td>
<td>282</td>
</tr>
<tr>
<td>EVR (43)</td>
<td>0.724</td>
<td>0.947</td>
<td>49.9</td>
<td>178</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVR, EVR (7)</td>
<td>1.85</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVR, ROR (11)</td>
<td>3.55</td>
<td>0.641</td>
<td>63.3</td>
<td>90.2</td>
<td>90.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVR, VRP (11)</td>
<td>5.44</td>
<td>1.39</td>
<td>187</td>
<td>238</td>
<td>238</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IOR (9)</td>
<td>time</td>
<td>stdv</td>
<td>198</td>
<td>377</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCR (14)</td>
<td>0.842</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR (23)</td>
<td>0.554</td>
<td>1.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOR (18)</td>
<td>5.99</td>
<td>4.01</td>
<td>308</td>
<td>516</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRP (13)</td>
<td>1.84</td>
<td>14.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.67</td>
<td>19.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: Average time (s) required by individual tools for sets of specifications distinguished by mutation operator(s) (only statistically significant results shown).
<table>
<thead>
<tr>
<th>Sets of Specifications</th>
<th>SMV</th>
<th>NuSMV</th>
<th>Lurch (Incomplete)</th>
<th>SPIN</th>
<th>Best</th>
<th>SLS</th>
<th>LSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>All (157)</td>
<td>3.54</td>
<td>13.2</td>
<td>5.71</td>
<td>117</td>
<td>24.4</td>
<td>29.0</td>
<td>30.1</td>
</tr>
<tr>
<td></td>
<td>3.53</td>
<td>13.1</td>
<td>5.71</td>
<td>110</td>
<td>90.6</td>
<td>90.8</td>
<td>90.2</td>
</tr>
<tr>
<td>CRP (8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVR (43)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVR, EVR (7)</td>
<td></td>
<td></td>
<td></td>
<td>30.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>58.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVR, ROR (11)</td>
<td></td>
<td></td>
<td></td>
<td>88.1</td>
<td>71.4</td>
<td>72.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>191</td>
<td>178</td>
<td>178</td>
<td></td>
</tr>
<tr>
<td>EVR, VRP (11)</td>
<td></td>
<td></td>
<td></td>
<td>179</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>195</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IOR (9)</td>
<td></td>
<td></td>
<td></td>
<td>66.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>141</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCR (14)</td>
<td></td>
<td></td>
<td></td>
<td>81.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>163</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR (23)</td>
<td></td>
<td></td>
<td></td>
<td>189</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>192</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOR (18)</td>
<td></td>
<td></td>
<td></td>
<td>167</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VRP (13)</td>
<td></td>
<td></td>
<td></td>
<td>210</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6: Average memory (MB) required by individual tools for sets of specifications distinguished by mutation operator(s) (only statistically significant results shown).
on the left match the mutation operators described in section 5.2.1. Different quantities of fault-seeded specifications were generated for each operator because the number of possible places in the original PACS SCR specification where a particular operator could be applied varied greatly from one operator to another. Only operators (or pairs of operators) for which at least ten fault-seeded specifications were generated are shown. The reason why, for example, only 8 CRP specifications are represented in the tables, is that only non-equivalent mutants are represented by results in the table; the 8 considered in the table, plus equivalent mutants with the CRP mutation, would together make up at least ten. Also, as with the tables in the previous section, only statistically significant results are shown ($p = 90\%$).

When categorized by mutation operator, the sets of specifications are relatively small, but there are still a few clear performance variations. The CRP (constant replacement) operator seems to force combined strategies to rely on SPIN; hence time requirements for $Best$, $SLS$ and $LSS$ are very close to what would be required by SPIN alone. (The performance for SPIN was not significantly different for CRP specifications, so we can assume it would be fairly close to the 94.8 seconds overall average for SPIN.) The combination of EVR (enumerated type value replacement) and ROR (relational operator replacement) has a similar effect on results for the combined strategies. The EVR operator was much easier for SPIN but the IOR (implication operator replacement) mutation operator much more difficult. The EVR, ROR combination was most difficult for NuSMV, SOR (SCR Event Operator Replacement) was most difficult for SPIN, and VRP was most difficult for Lurch.

Although it would be difficult to say why a particular operator would be difficult for one tool but easy for another, here again we see, as in section 6.1.1, that automated verification tools tend to be complementary. Rather than some specifications being just plain hard, regardless of tool choice, and others being easy, we see that specifications difficult for one tool will often be easy for another.

### 6.2.3 Specifications Categorized by Number of Mutations

As mentioned in the previous section, some fault-seeded specifications used in our experiments were seeded with a single mutation, while others had two mutations. Tables 6.7 and 6.8 compare performance for the various tools and strategies running on sets of specifications distinguished by
<table>
<thead>
<tr>
<th>Sets of Specifications</th>
<th>SMV (Incomplete)</th>
<th>NuSMV</th>
<th>Lurch</th>
<th>SPIN (Complete)</th>
<th>Best</th>
<th>SLS</th>
<th>LSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>All (233)</td>
<td>0.107</td>
<td>1.21</td>
<td>2.72</td>
<td>79.5</td>
<td>14.1</td>
<td>15.2</td>
<td>16.4</td>
</tr>
<tr>
<td></td>
<td>0.0481</td>
<td>2.15</td>
<td>6.99</td>
<td>248</td>
<td>95.2</td>
<td>96.2</td>
<td>96.0</td>
</tr>
<tr>
<td>One Mutation (142)</td>
<td>0.818</td>
<td>3.49</td>
<td></td>
<td>112.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.23</td>
<td>8.73</td>
<td></td>
<td>296</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two Mutations (91)</td>
<td>1.72</td>
<td>1.62</td>
<td></td>
<td>27.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.88</td>
<td>2.89</td>
<td></td>
<td>128</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.7: Average time (s) required by individual tools for sets of specifications distinguished by the number of mutations (only statistically significant results shown).

<table>
<thead>
<tr>
<th>Sets of Specifications</th>
<th>SMV (Incomplete)</th>
<th>NuSMV</th>
<th>Lurch</th>
<th>SPIN (Complete)</th>
<th>Best</th>
<th>SLS</th>
<th>LSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>All (233)</td>
<td>3.48</td>
<td>13.2</td>
<td>5.68</td>
<td>99.0</td>
<td>19.3</td>
<td>25.6</td>
<td>26.7</td>
</tr>
<tr>
<td></td>
<td>0.551</td>
<td>2.88</td>
<td>0.562</td>
<td>163</td>
<td>85.1</td>
<td>89.1</td>
<td>88.8</td>
</tr>
<tr>
<td>One Mutation (142)</td>
<td>3.55</td>
<td>12.8</td>
<td></td>
<td>119</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.490</td>
<td>2.28</td>
<td></td>
<td>172</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two Mutations (91)</td>
<td>3.38</td>
<td>13.6</td>
<td></td>
<td>67.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.612</td>
<td>3.48</td>
<td></td>
<td>142</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.8: Average memory (MB) required by individual tools for sets of specifications distinguished by the number of mutations (only statistically significant results shown).
the number of mutation operators used to generate the fault-seeded specification. As with tables in
the previous two sections, only statistically significant values are shown, and equivalent mutants
are excluded.

One might expect that it would be easier for all tools to detect property violations in fault-
seeded specifications with two mutations, compared to those with only one. But NuSMV is actu-
ally significantly faster (and requires less memory) to detect property violations in single-mutation
specifications than in specifications with two mutations. On the other hand, Lurch and SPIN are
faster on two-mutation specifications. A possible explanation for the difference is that NuSMV, a
symbolic model checker, builds the entire state space and then runs a breadth-first search; Lurch
and SPIN, on the other hand, are based on depth-first-search and tend to either detect faults early,
long before the entire state space would have been explored, or not at all. That is, NuSMV will
tend to do a similar amount of computational effort regardless of the number of faults present in
the system, while Lurch and SPIN will tend to do much less effort if many faults are present. That
being said, why does Cadence SMV not show the same pattern as NuSMV? Probably because
NuSMV’s default settings are such that all properties are checked, while Cadence SMV by default
stops at the first violation detected. But in this case performance variations for Cadence SMV were
quite small, so further experiments would be needed to explore the possibility that Cadence SMV
and NuSMV show complementary behavior based in some way on the number of faults present in
the input model.

6.3 Combined Strategy Based on Performance and Accuracy

Chapter 3 documents our process of discovering and resolving several apparent inconsistencies in
results between verification tools and describes variations in performance between tools from one
input model to another. Chapter 4 describes basic differences, including strengths and weaknesses,
of each tool used in our experiments. Previous sections of this chapter describe performance
variations between tools observed when input models are broken into subsets based on different
criteria. Also, in both chapter 3 and this chapter, we compare the results of simple multiple-tool
verification strategies based on performance (i.e., time cutoffs) to the results of individual tools.
In this section, we summarize lessons learned from all of our experiments and propose a more
generally applicable verification strategy combining tools based on both performance variations and characteristics of tools, within the framework of SCR and automatic translation provided by the SCR tools, relevant to the accuracy of verification results.

6.3.1 Lessons Learned

For each of the verification tools used in our experiments—Salsa, SMV, NuSMV, Lurch and SPIN—we list here characteristics of the tool brought out by comparison of that tool’s results with results from other tools. The SCR Toolset is not included, since our focus here is on developing a verification strategy effective on models that have already passed all of the checks implemented in the SCR Toolset GUI. We assume that back-end verification tools would only be used after a specification has already passed the SCR Toolset’s generic checks.

Salsa

Salsa is generally slower than Cadence SMV but faster than NuSMV, Lurch and SPIN, and requires very little memory. Assertions and generic properties can be proved automatically using Salsa, but any assertions that Salsa fails to prove must be checked manually or with some other tool. Salsa’s ability to prove a particular assertion is, at least in our experiments, independent of the presence or absence of other assertions in the specification. For example, if a copy of the specification is made for each assertion, so that only one assertion is present in each copy, and Salsa is run on each copy, we do not find that Salsa is able to prove any more assertions than if it were run on a single copy that includes all of the assertions.

Although we observed some interesting variations in the performance of other tools running on sets of specifications categorized according to classes of Salsa results, none of these variations can be readily exploited in a multiple-tool verification strategy. However, as shown in the next section, if assertions proved by Salsa are removed from a specification it may greatly improve SPIN’s performance. Also, NATURE constraints in an SCR specification are compatible with Salsa but not with SPIN or Lurch (assuming models are generated from the automatic SPIN translator included with the SCR tools). So Salsa can be used to automatically validate assertion violations reported by SPIN or Lurch, which may be the result of ignoring NATURE constraints. Since Salsa is not complete, however, some violations may still need to be validated manually using the SCR
Simulator.\(^1\)

**SMV and NuSMV**

As stated in chapter 3, the SMV version of a specification generated by the SCR Toolset requires minor modifications to be compatible with NuSMV. With these changes, results from Cadence SMV and NuSMV were consistent, in terms of accuracy, in all our experiments. We found also that Cadence SMV was consistently faster and required less memory, although NuSMV resource requirements were still small compared to SPIN. Since Cadence SMV was faster we use it in the combined strategy described in the next section. Depending on the application it may be preferable to use NuSMV, however, because it is an open-source tool with a less restrictive license.

As far as the differences between Cadence SMV and Salsa, Lurch or SPIN, Cadence SMV is 1) very fast, and requires very little memory, 2) respects \textit{NATURE} constraints (like Salsa), and 3) can only verify single-state assertions. Thus it makes sense to run Cadence SMV first. If an assertion violation is detected, it should not be necessary to validate it using another tool, since \textit{NATURE} constraints are respected (of course it can be validated using another tool if desired). If no error is detected, all single-state assertions can be removed from the model before running other tools to check two-state assertions (i.e., assertions containing the SCR \textit{Next} operator or event operators).

**Lurch**

Lurch is usually slower than Salsa, Cadence SMV and NuSMV, but often faster than SPIN, at detecting assertion violations. Lurch is able to check both single state and two-state assertions, but assertion violations reported by Lurch must be validated, because the input model for Lurch is generated from the SCR Toolset’s SPIN version of the specification, which ignores \textit{NATURE} constraints. Lurch is incomplete, so if no assertion violations are detected by Lurch a complete tool (i.e., SPIN) must be used to fully verify the specification.

In the simple performance-based multiple tool verification strategies described in chapter 3 and previous sections of this chapter, Lurch was run until an arbitrary time cutoff. For smaller models

\(^1\) One additional minor point about the use of Salsa within the framework of the SCR tools: the filename of the SCR specification must not include a hyphen (\(-\)), or else the automatically translated Salsa version of the specification will not work with Salsa.
a cutoff of 1 second was used; for the PACS SCR specification a cutoff of 7 seconds was used; for
the flight guidance system experiments in section 3.2.2 a cutoff of 2 minutes was used. In each
case these were nearly optimal, but a wide range of cutoffs gave similar results. For example, in
the PACS SCR experiments 7 seconds was used, but a cutoff value anywhere between 1 and 30
seconds produced similar results. Still, it would be preferable to have a generally applicable cutoff
value not related to running time. For this reason we propose below to use Lurch’s saturation
statistic, a measure of the ratio of new states to repeat states explored by the search. A saturation-
based cutoff value of 25%, which roughly corresponds to the 7 second time cutoff used in the
PACS experiments, is used in the final set of experimental results presented below.

**SPIN**

Within our experimental framework, comprised of the SCR Toolset, our script for generating Lurch
input models, and minor modifications necessary for NuSMV and SPIN described in chapter 3, the
only tool capable of fully verifying all types of assertions present in an SCR specification is SPIN.
So, although SPIN in some cases requires far more time and memory than other tools, it is a
necessary component of any complete verification strategy. We should point out also that SPIN’s
completeness is related to its resource requirements. If, for example, a translator was written to
produce SMV input models in a less abstract way, so that two-state assertions could be checked by
SMV, this would likely increase Cadence SMV’s and NuSMV’s time and memory requirements
significantly.

The input model generated by the SCR Toolset for SPIN encloses code representing transition
tables in a \texttt{d_step} block, which saves time and memory but is valid only if all tables are disjoint.
In general, the final \texttt{d_step} marker must be removed to fully verify the specification. In practice,
it may also be necessary to use memory compression options and increase the depth limit for the
verification run to terminate, as we have done for the PACS specification experiments. We found
that only SPIN’s \textit{minimized automaton} compression option, the slowest but most memory-efficient
lossless compression option available in SPIN, was sufficient to enable full verification of the
PACS specification. We also had to increase the depth limit from the default value of 10,000 to 3.2
million.
Figure 6.3: Combined strategy exploiting complementary variations in performance and accuracy. (Baseline complete and single-tool complete strategies enclosed in dotted boxes.)

### 6.3.2 Generally Applicable Multiple-Tool Verification Strategy

Figure 6.3 shows a flowchart representing a multiple-tool verification strategy based on both performance variations and characteristics of tools, in the context of the SCR tools framework, relevant to the accuracy of verification results. First, we run Cadence SMV to either detect a fault or prove all single-state assertions. If no fault is detected, single-state assertions are removed from the model and we run Salsa, to attempt to prove some or all of the two-state assertions. Any two-state assertions proved by Salsa are then removed from the model. If all assertions have been proved at this point, the model is correct.

If there are still assertions to be checked in the model, Lurch is run next to detect violations of these remaining two-state assertions. Rather than an arbitrary time cutoff, Lurch is run to 25% saturation. If Lurch detects a fault, we stop. It is possible at this point that the fault detected by Lurch is not actually present in the model, due to Lurch’s ignoring NATURE constraints, so it
needs to be validated. Although we mentioned above that assertion violations detected by Lurch or SPIN can sometimes be confirmed or disconfirmed automatically using Salsa or Cadence SMV, if a violation is detected at this point in the flowchart by Lurch it must be validated manually using the SCR Simulator, because Salsa and Cadence SMV have already been run, and any violation that would have been disconfirmed by Salsa or Cadence SMV has already been removed from the model.

If Lurch does not detect a fault, SPIN is run next with default options. This is the fastest and most memory-expensive mode in which to run SPIN. It is incomplete, because of the depth limit of 10,000 states, but often detects assertion violations very quickly. If SPIN with default options detects no assertion violation, SPIN is next run with options set to allow the run to terminate normally. In our experiments this required using the minimized automaton compression option described in the previous section (set to 28) and increasing the depth limit to 2 million. Finally, if no violation is detected with these options SPIN is run again, with options set to allow a full verification run, on a modified version of the input model with the final d_step marker removed. In our experiments, in order to get a full verification run on input models modified in this way we again used the minimized automaton compression and increased the depth limit to 3.2 million.

The dotted rectangles in figure 6.3 show two alternative complete verification strategies using only SPIN. The outer rectangle shows how SPIN might be used interactively, modifying settings as needed to minimize resource requirements on models for which violations can be detected quickly, but to enable full verification eventually. The inner rectangle shows how SPIN would be used if it were to be run once on each input model with options preset to enable full verification. Figure 6.4 shows a comparison of results for these two ways of using SPIN and for the complete strategy, using all the tools, represented by the entire flowchart in figure 6.3. Results for each strategy are plotted with a point for each specification (a square, circle or triangle) whose x-coordinate is the amount of time required by that strategy to verify or detect a property violation in that specification and whose y-coordinate is the amount of memory required by that strategy.

Table 6.9 shows the number of specifications in which property violations were detected by different tools in the order the tools were used, according to the flowchart shown in figure 6.3. The column labeled Combined shows values for tools used in the combined strategy represented by the entire flowchart; the column labeled Single Tool shows values for SPIN run in three different
Figure 6.4: Comparison of results for baseline complete strategy, single tool complete strategy, and combined strategy. (Medians are marked by large shapes.)

<table>
<thead>
<tr>
<th></th>
<th>Combined</th>
<th>Single Tool</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cadence SMV</td>
<td>141</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lurch</td>
<td>72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPIN (default)</td>
<td>3</td>
<td>197</td>
<td></td>
</tr>
<tr>
<td>SPIN (complete)</td>
<td>7</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>SPIN (no d_step)</td>
<td>0</td>
<td>1</td>
<td>223</td>
</tr>
</tbody>
</table>

Table 6.9: Number of specifications in which property violations were detected by tools at different stages of the flowchart shown in figure 6.3.
ways, as represented by the part of the flowchart enclosed in the larger dotted rectangle; and the final column shows values for SPIN run in the single mode represented by the part of the flowchart enclosed in the smaller dotted rectangle.

Unlike the results shown in table 6.9 and results given in previous sections, figure 6.3 includes results for both non-equivalent mutants, in which violations were detected, and equivalent mutants, which were fully verified. Figure 6.4 and tables 6.10 and 6.11 include data for both equivalent and non-equivalent mutants, but do not include data from specifications for which NATURE constraints were relevant to the truth of assertions. In such cases resource data doesn’t always make sense. For example, the two strategies using SPIN alone may require very little time and memory to detect a property violation ruled out by NATURE constraints, while the combined strategy may take much more time and memory to reach the (true) conclusion that the specification is correct. Table 6.9 shows that the number of non-equivalent mutant specifications is 223, not 232 as stated above; and tables 6.10 and 6.11 show that the total number of specifications is 312, rather than the 323 specifications considered in experiments described above, because specifications for which NATURE constraints played a role are not included here.

### Table 6.10: Average time required by baseline, single tool and combined verification strategies running on all, equivalent and non-equivalent mutant specifications.

<table>
<thead>
<tr>
<th>Sets of Specifications</th>
<th>Baseline (time)</th>
<th>Single Tool (time)</th>
<th>Combined (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>All (312)</td>
<td>404</td>
<td>611</td>
<td>343</td>
</tr>
<tr>
<td></td>
<td>592</td>
<td>934</td>
<td>548</td>
</tr>
<tr>
<td>Equivalent Mutants (89)</td>
<td>1300</td>
<td>2040</td>
<td>1180</td>
</tr>
<tr>
<td></td>
<td>258</td>
<td>405</td>
<td>236</td>
</tr>
<tr>
<td>Non-Equivalent Mutants (223)</td>
<td>45.2</td>
<td>42.3</td>
<td>6.56</td>
</tr>
<tr>
<td></td>
<td>107</td>
<td>128</td>
<td>41</td>
</tr>
</tbody>
</table>

### Table 6.11: Average memory required by baseline, single tool and combined verification strategies running on all, equivalent and non-equivalent mutant specifications.

<table>
<thead>
<tr>
<th>Sets of Specifications</th>
<th>Baseline (memory)</th>
<th>Single Tool (memory)</th>
<th>Combined (memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>All (312)</td>
<td>104</td>
<td>207</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>16.3</td>
<td>230</td>
<td>90.1</td>
</tr>
<tr>
<td>Equivalent Mutants (89)</td>
<td>125</td>
<td>484</td>
<td>195</td>
</tr>
<tr>
<td></td>
<td>4.00</td>
<td>106</td>
<td>44.1</td>
</tr>
<tr>
<td>Non-Equivalent Mutants (223)</td>
<td>95.2</td>
<td>96.8</td>
<td>14.5</td>
</tr>
<tr>
<td></td>
<td>10.4</td>
<td>161</td>
<td>36.2</td>
</tr>
</tbody>
</table>
The large square, circle and triangle in figure 6.4 represent the median time and memory requirements for each strategy: for the baseline complete strategy, running SPIN once with minimized automaton compression, median resource requirements were 29.4 seconds and 95.3 megabytes; for the single tool complete strategy, running SPIN in three successively more time-consuming configurations, median requirements were 3.18 seconds and 75.0 megabytes; for the strategy combining all tools shown in figure 6.3, median requirements were 1.75 seconds and 5.45 megabytes. Tables 6.10 and 6.11 show average time and memory requirements, as well as standard deviation values, for the three strategies running on 1) all specifications, 2) equivalent mutants, and 3) non-equivalent mutants.

Consistent with experimental results above, tables 6.10 and 6.11 show that a strategy combining multiple tools is much faster and requires significantly less memory to detect property violations in non-equivalent mutant specifications; that is, the combined strategy is very effective at detecting faults. What these tables show, that is not shown in experimental results reported above, is that a combination strategy exploiting tool variations in both performance and accuracy is also effective for fully verifying correct specifications. On equivalent mutants the combined strategy is fastest (on average) and has memory requirements comparable to the baseline strategy and much lower than the single tool complete strategy using SPIN in 3 configurations. All in all, the combined strategy saves about 1 minute per specification, compared to the baseline single SPIN run, for a total savings of about 5.5 hours; compared to the 3-mode SPIN results, the combined strategy saves almost 5 minutes per specification, for a total savings of over 26 hours on all 312 specifications considered in figure 6.4 and tables 6.10 and 6.11.

6.4 Summary

Verification tools available in the framework of the SCR Toolset exhibit complementary performance, in terms of time and memory required to detect assertion violations. For nearly all fault-seeded specifications used in our experiments, excluding equivalent mutants, at least one verification tool was able to detect a violation quickly and without using a large amount of memory, even for specifications much more difficult or impossible for one or more other verification tools. Based on tools’ complementary performance, we proposed simple multiple-tool combinations:
run faster but less thorough tools first; run slower but more thorough tools later. Two combination strategies were proposed, one beginning with Cadence SMV and one with Lurch. Both performed nearly as well as a hypothetical Best strategy, in which the single best tool for each specification is presciently chosen and used.

Next we considered the performance of individual tools and performance-based combined strategies running on sets of specifications distinguished by 1) Salsa results, 2) mutation operator(s), and 3) the number of mutations per specification. Even for the relatively small sets of specifications in some of these categories, comparing performance of the different tools and strategies provides additional evidence for complementary relationships. In several cases a set of specifications more difficult than average for one tool was found to be less difficult than average for another tool.

Finally, we proposed a more generally applicable combined verification strategy based on both performance variations between tools and the different capabilities of tools within the framework of the SCR Toolset. Salsa can prove assertions true but not false and is fast and efficient, so it makes sense to use it to filter out whatever assertions it can prove before moving to more resource-hungry tools. Cadence SMV and NuSMV produce identical results, so we use Cadence SMV here because of better performance. (NuSMV could be used if a more thoroughly documented, open-source tool was desired.) Cadence SMV can be used to quickly check single-state assertions, but cannot check two-state assertions, so it makes sense to use it early. If no violations are detected by Cadence SMV, it can be used like Salsa to filter out whatever assertions it proves before moving on to other tools.

Lurch can sometimes quickly detect violations but cannot prove properties true. Also, Lurch requires a cutoff, i.e., a point at which to give up if no violation is detected. In previous combination strategies time-based cutoffs were used. Here we use a more generally applicable cutoff based on saturation, the previously described pattern in our random search results, that the proportion of unique states explored tends to drop off after a relatively short time. SPIN can be time-consuming and may require a great deal of memory, but it is the only tool capable of fully verifying specifications in our experimental framework. So we propose it be used last, if no previous tool has detected a property violation.

Results from this final combination strategy, compared to SPIN used in two different ways,
show that complementary relationships between tools can be exploited to produce a multiple-tool verification strategy that is in many cases significantly faster and more memory efficient than alternative strategies using a single tool. In addition, the use of multiple tools will tend to expose hidden assumptions or invalid use of individual tools, so that the combination strategy is not only more efficient, but more reliable as well, compared to the use of a single tool.
Chapter 7

Conclusion

Chapter 7 begins in section 7.1 with a broad overview of the content from previous chapters: automatic verification tools offer great benefit but at significant cost to users. Different tools implement different strategies for reducing these costs. We argue here that costs are reduced (and benefits increased) most effectively by a strategy combining multiple tools. Section 7.2 presents a general model of software verification challenges based on our experiments within the framework of the SCR Toolset. The model illustrates costs described in the previous section and also shows a generic framework for automated verification. Beginning with an initial specification, multiple complementary verification strategies should each be integrated via translation at multiple levels of abstraction. Finally, section 7.3 proposes open research questions motivated by our work and experimental results, and section 7.4 summarizes this chapter.

7.1 Overview

Researchers have shown that automatic verification tools offer great benefits to developers of complex and critical software systems. These tools can be used to detect subtle, non-repeatable errors that would be extremely difficult to find through conventional testing or manual inspection of source code. Still, developers remain skeptical because of the costs, in user effort and expertise, and in computing resources, of using these tools. These costs may be divided into two general categories, along the lines of the traditional distinction between validation and verification in software quality assurance. There is the cost of building an abstract model and property specification that together accurately represent the essential behavior of the system to be verified (validation), and there is the cost, in computational resources and user expertise in the chosen verification method,
of verifying that the model and properties are consistent with each other.

These two categories of costs, validation cost and verification cost, are not at all independent from each other: action to decrease one may increase the other in unexpected ways. For example, validation cost is decreased if it is possible to automatically translate the software model into the language required by a verification tool. But automatically generated models tend to be much less efficient, compared to carefully handwritten models, and require much more time and memory for verification. On the other hand, verification cost may be greatly decreased by restricting the input language of the verification tool, but this makes it much more difficult to create accurate input models, because the models likely represent systems that could be much more elegantly expressed in a less restrictive language.

In this dissertation we have considered a specific modeling and verification framework, the SCR Toolset, including the consistency checker and its command-line version, testtool, and several integrated back-end verification tools. For the main case study experiments, we used the Salsa invariant checker for SCR specifications, the Cadence SMV and NuSMV symbolic model checkers, the SPIN explicit-state model checker, and our Lurch random search tool for debugging formal models. Lurch works in some ways like an explicit-state model checker, but instead of carrying out a systematic, complete exploration of the behavior represented in the model, it explores a series of random paths through the model.

In attempting to use this wide range of tools, we initially expected the primary validation challenge would be to make sure that automatic translation, from the original SCR specification to input models for Salsa, Cadence SMV, NuSMV, SPIN and Lurch, was done correctly. Over time, however, with more experience using the automatic translators and verification tools, our view of the validation challenge shifted: the challenge is not to make sure that all translators produce correct output, where correct is understood to mean perfectly equivalent models for each verification tool. Instead, the primary validation challenge is to clearly understand the differences between the models produced by each translator. It is actually beneficial to have different, non-equivalent versions of the model, at different levels of abstraction and with different features present. Verification results are validated when results from different verification tools, running on different (i.e., non-equivalent in behavior) models of the system, can be synthesized into a coherent whole.

Likewise with verification strategies, the goal should not be simply to make sure they all pro-
duce equivalent results and then pick the one with the best performance. Instead, it is desirable that they be complementary, in terms of what kinds of defects they can detect and what kinds of properties they can prove. And it is very likely that their performance will be complementary, so that it is not possible to choose the one with the best performance. Rather than a single best strategy for efficient, scalable verification, we observe (and others report in the literature) that different strategies have different strengths and weaknesses. Not only will a particular strategy be preferable for certain classes of input models, but, for a single input model, changes to the model that seem insignificant can make a large difference in the effectiveness of a particular verification strategy.

By combining diverse strategies for verification, we can increase the scope of the overall strategy, so that a wider range of properties can be checked, and we can increase confidence in the validity of the results of the verification, as different tools confirm or disconfirm each other’s results. In addition, combining strategies with complementary performance makes it possible to integrate incomplete but efficient strategies, such as random search, without sacrificing the completeness of the overall strategy, so that much more time (or memory) consuming methods are used only when absolutely necessary.

### 7.2 A Conceptual Model of Software Verification Challenges

Figure 7.1 is meant to illustrate some of the challenges described in the previous section and, along with that, how these challenges may be addressed by a combination of diverse modeling and verification strategies. At the center of figure 7.1 is the specification, or more generally, the software artifact, which may be anything from prose requirements to source code, along with properties to be verified in whatever source form is available. The specification and properties must be translated into a formal description suitable for verification and then verified. Thus the information from the specification moves through a validation space, in which the goal is to generate accurate models capturing necessary and sufficient information, to a verification space, in which the goal is to efficiently determine whether the part of the specification representing behavior is consistent with the part of the specification representing desired properties.

There are two general contributions of this dissertation: first, to propose that complementary translation (and modeling) strategies should be combined to address accuracy issues in the valida-
tion space, and second, to propose that complementary verification strategies should be combined to address performance issues in the verification space. Through the experiments presented above, we have attempted to show how multiple translation and verification techniques, available within the framework of the SCR Toolset and integrated back-end tools, can be combined to achieve higher confidence and decreased user effort and computational cost for software specifications written in SCR.

The SCR Toolset’s translators used in our experiments are complementary, for example, in the sense that the output for SMV is a smaller model than the output for SPIN, and so can be verified more efficiently; yet the output for SPIN is a more complete representation of the specification, including both single-state and two-state assertions, so verification using SPIN is more comprehensive. Also, the Salsa and SMV models generated by the SCR tools respect \textit{nature} constraints, which is relatively easy to do in the input languages for these tools. But the output for SPIN does not—to do so would require much additional complexity in the portion of the model representing the environment and \textit{monitored} variables.

Verification tools used in our experiments were complementary as well. SPIN was slowest,
and required the most memory, but is the only tool used in our experiments capable of fully verify-
ing SCR specifications. Salsa, Cadence SMV and NuSMV sometimes proved particular properties
much more quickly than SPIN, and running SPIN on specifications with these already proven prop-
erties removed was much less time consuming than running SPIN with these properties still present
in the model. Cadence SMV, NuSMV and Lurch detected property violations in certain specifi-
cations more quickly than SPIN, and for these specifications a more time-consuming SPIN run was
not necessary. In addition, SPIN showed huge performance variations from one fault-seeded spec-
fication to another. Some specifications contained property violations almost as difficult for SPIN
to detect as it was for SPIN to verify the correct model. But for many of these same specifications,
property violations could be detected very quickly using Cadence SMV, NuSMV or Lurch.

What would the ideal set of tools for verification look like? We propose that it would like
figure 7.1. Multiple strategies for improving the scalability of automatic verification would be in-
tegrated, through multiple tools, or possibly through multiple scalability strategies available in the
same tool. These strategies would be complementary, some emphasizing quick proof of a subset
of the properties (as Salsa and SMV were used in our experiments) and some emphasizing quick
detection of errors (as SMV, Lurch, and SPIN in the first two modes, were used in our experi-
ments). For each strategy, translation methods would be available at different levels of abstraction,
some emphasizing similarity to the behavior of the source model and property specification (to
address validation challenges) and some emphasizing structural simplicity (to address verification
challenges). If these kinds of translation and verification tools are available, combination strategies
like the one we proposed for SCR will provide better performance and higher confidence in the
verification result.

7.3 Open Research Questions

How confident can we be in the results of any verification method? Is there a way to quantify
confidence and make comparisons between methods? Researchers in model checking, and formal
methods generally, have tried to address these questions by providing verification methods that
promise 100% confidence. As shown above, however, caveats and assumptions made in practice
tend to undermine 100% confidence in any particular verification method. If no method provides
100% confidence, a combination of complementary tools may provide more confidence than individual tools used alone. But how much confidence does the combination of tools provide? Is there a practical way to measure the level of confidence provided by any incomplete verification strategy? If we don’t know how large the space of behaviors represented by a software model really is, how can we know how much of it has been explored? Or, if we can completely explore the space of behaviors represented by a software model, how can we know for sure that assumptions and abstractions implicit in the model and verification strategy are valid? Our work shows that increased confidence can be obtained through combining complementary strategies, compared to a single strategy used alone. But it also suggests a very difficult practical question: if we can’t achieve 100% confidence in the correctness of a software model, how do we know when a sufficient level of confidence has been achieved?

The experiments presented in this dissertation were done on a software specification, with relatively simple structure, carefully written by experts in the modeling language and related tools, as an example of how to write a clean, high quality software specification. After a significant amount of effort we developed a strategy combining several tools in a way that allowed us to more effectively verify fault-seeded versions of the specification. Though the effort was significant, in our case there was less work to do, because we started with a software artifact very carefully written in a formal modeling language, and much of the work was already done—other than the translator from the SCR Toolset’s SPIN version of the model to Lurch, which we developed, all verification tools and automatic translators were already integrated within the SCR Toolset. Readers might reasonably wonder whether our results, however interesting, could be applied in a situation where the model, properties, and integrated verification framework are not already available. Our results are probably more applicable to researchers working on modeling and verification tools than to software developers, unless they are working within the framework of the SCR Toolset. We suggest further work on integrated frameworks, along the lines of figure 7.1.

Our results indicate complementary relationships between verification strategies, in terms of time and memory required to detect property violations in fault-seeded software models. Based on these results, we advocate using a combination of tools. But we have not attempted to understand why, for each of these strategies, some input models are much easier to check than others. Is there something measurable about the structure of a formal model that could be used to predict the
performance of a particular verification strategy? For example, for deterministic, complete search methods, we have offered above the idea that a minor change in the input model might place a property violation in a different location in the space of behaviors represented by the model, thereby greatly affecting the performance of the verification tool. Would it be useful to offer a reverse search option in a model checker, so that property violations on the opposite side (conceptually) of the state space would be found quickly? That is, just as apparently minor changes in the input model may greatly affect the performance of a particular verification strategy, might apparently minor changes in the search algorithm have just as great an affect on performance?

Our work on random search as an efficient method of detecting errors in formal models, implemented in Lurch, raises its own version of some of these general questions about verification strategies. For Lurch runs in which no property violation is detected, is there any way to measure coverage, to estimate the size of the space of behaviors left unexplored, or to characterize the structure of the state space? Monitoring a random search run for saturation is helpful in knowing when to stop trying to use random search to find new behavior in the model, but we have not done experiments to try to use saturation-related metrics to estimate the size of the remaining unexplored behavior or to predict the performance of other tools. Because of the unbiased nature of a random search, compared to a systematic, deterministic search through a space of behaviors, it may be possible to use it to quickly gain approximate information about the structure or size of the state space. With a better understanding of the relationship between model structure and alternative verification strategies, information from random search might be helpful in guiding the use of other tools.

7.4 Summary

Automatic verification tools offer significant benefits for software assurance, but developers remain skeptical because of the high costs of using researchers’ methods and tools. There are validation costs, to make sure the software model and properties accurately represent the essential behavior of the critical system to be verified, and there are verification costs—user expertise in the verification method and system to be verified, and time and memory requirements to run the verification. These two categories of cost are not independent. Strategies to decrease one may increase the other.
in unexpected ways. We suggest that tools for modeling and verification should be integrated into frameworks providing multiple complementary strategies for minimizing validation cost (modeling and translation tools) and multiple complementary strategies for minimizing verification costs (incomplete and complete strategies emphasizing a range of verification tasks).

The experimental results presented in this dissertation raise several questions for future research work. First, how confident can we be in the results of any verification method? How can we quantify and measure the appropriate level of confidence for individual methods, and how can these values be combined to produce a confidence measure for a combination of methods? Second, to what degree are our results, which are based on a very clean, well-written and relatively simple software specification, analyzed in a tightly integrated framework—to what degree are these results generalizable to others’ work on much less ideal software artifacts, in much less integrated tool frameworks? Third, what are the underlying reasons for complementary relationships between verification strategies? Can our observations be used to motivate new strategies, rather than just the combination of existing strategies? Finally, how can random search be used to better understand the structure of formal models and to predict the results of other more sophisticated verification methods?
Appendix A

Cruise Control Specification Models

SCR

(The SCR version of the specification is included in figure 4.2.)

Salsa

// This file contains the SAL version of an SCR specification

module cc.ssl

type definitions
  CruiseLever : { Activate, Deactivate, Resume };
  prefix0_Mode_modes : { Off, Inactive, Cruise, Override };

monitored variables
  EngRun : bool;
  Brake : bool;
  Lever : CruiseLever;
  Ignited : bool;

internal variables
  Mode : prefix0_Mode_modes;

assumptions
  init = initially EngRun = FALSE and Brake = FALSE and Lever = Deactivate and Ignited = FALSE;

guarantees
  S4 = (Mode = Override) => (Ignited and EngRun);
  S3 = (Mode = Cruise) => (((Ignited and EngRun) and not Brake) and not (Lever = Deactivate));
  S2 = (Mode = Inactive) => Ignited;
  S1 = (Mode = Off) => not Ignited;

definitions

  /*---- Begin mode transition table: Mode -----------------------------------*/
  var Mode initially Off :=
case Mode
  [] Cruise ->
    ev
      [] @F(Ignited) -> Off
      [] @F(EngRun) when Ignited -> Inactive
      [] @T(Brake or (Lever = Deactivate)) when (Ignited and EngRun) -> Override
  ev
  [] Inactive ->
    ev
      [] @F(Ignited) -> Off
      [] @T(Lever = Activate) when (EngRun and not Brake) -> Cruise
  ev
  [] Off ->
    ev
      [] @T(Ignited) -> Inactive
  ev
  [] Override ->
    ev
      [] @F(Ignited) -> Off
      [] @F(EngRun) when Ignited -> Inactive
      [] @T((Lever = Activate) or (Lever = Resume)) when ((Ignited and EngRun) and not Brake) -> Cruise
  ev
  otherwise -> Mode
esac
end module

SMV

-- This file contains the SMV version of an SCR specification

MODULE main

VAR
  Mode : {Off, Inactive, Cruise, Override};
  EngRun : boolean;
  Brake : boolean;
  Lever : {Activate, Deactivate, Resume};
  Ignited : boolean;

ASSIGN
  init(EngRun) := 0;
  init(Brake) := 0;
  init(Lever) := Deactivate;
  init(Ignited) := 0;
  init(Mode) := Off;

  next(EngRun) := {0,1};
  next(Brake) := {0,1};
  next(Lever) := {Activate, Deactivate, Resume};
  next(Ignited) := {0,1};
--- Begin mode transition table: Mode -----------------------------
next(Mode) :=
case
  (next(Ignited) & ! Ignited) & (Mode = Off) : Inactive;
  (Ignited & ! next(Ignited)) & (Mode = Inactive) : Off;
  ((next(Lever) = Activate) & !(Lever = Activate)) & (EngRun & ! Brake)) & (Mode = Inactive) : Cruise;
  (Ignited & ! next(Ignited)) & (Mode = Cruise) : Off;
  ((EngRun & ! next(EngRun)) & Ignited) & (Mode = Cruise) : Inactive;
  (((next(Brake) | (next(Lever) = Deactivate)) & !(Brake | (Lever = Deactivate)) &
      (Ignited & EngRun)) & (Mode = Cruise) : Override;
  (Ignited & ! next(Ignited)) & (Mode = Override) : Off;
  ((EngRun & ! next(EngRun)) & Ignited) & (Mode = Override) : Inactive;
  (((next(Lever) = Activate) | (next(Lever) = Resume)) & !(Lever = Deactivate)) & (Ignited & EngRun) & ! Brake)) & (Mode = Override) : Cruise;
1 : Mode;
esac;

TRANS
-- One Input Assumption
(!((next(EngRun) = EngRun) & (next(Brake) = Brake) & (next(Lever) = Lever) & (next(Ignited) = Ignited) &
  (next(Ignited) = Ignited)) |
  (next(EngRun) = EngRun) & ! (next(Brake) = Brake) & (next(Lever) = Lever) & (next(Ignited) = Ignited) &
  (next(Ignited) = Ignited)) |
  (next(EngRun) = EngRun) & (next(Brake) = Brake) & ! (next(Lever) = Lever) & (next(Ignited) = Ignited) &
  (next(Ignited) = Ignited)) |
  (next(EngRun) = EngRun) & (next(Brake) = Brake) & (next(Lever) = Lever) & !(next(Ignited) = Ignited)

-- Assertions
SPEC
AG(! (Mode = Override) | (Ignited & EngRun))
SPEC
AG(! (Mode = Cruise) | (((Ignited & EngRun) & ! Brake) & !(Lever = Deactivate)))
SPEC
AG(! (Mode = Inactive) | Ignited)
SPEC
AG(! (Mode = Off) | ! Ignited)

SPIN

/* This file contains the PROMELA/spin version of an SCRTool specification */

#define TRUE 1
#define FALSE 0
#define Activate 0
#define Deactivate 1
#define Resume 2
#define Off 0
#define Inactive 1
#define Cruise 2
#define Override 3
bool Brake_OLD = FALSE;
bool Brake_NEW = FALSE;
bool EngRun_OLD = FALSE;
bool EngRun_NEW = FALSE;
bool Ignited_OLD = FALSE;
bool Ignited_NEW = FALSE;
byte Lever_OLD = Deactivate;
byte Lever_NEW = Deactivate;
byte Mode_OLD = Off;
byte Mode_NEW = Off;

init {
    /* main processing loop */
    do ::
        /* "any state" specification asserts */
        assert(!(Mode_NEW == Off)) || !Ignited_NEW);
        assert(!(Mode_NEW == Inactive)) || Ignited_NEW);
        assert(!(Mode_NEW == Cruise)) || ((Ignited_NEW && EngRun_NEW) && (!Brake_NEW)) && 
                (!Lever_NEW == Deactivate)));
        assert(!(Mode_NEW == Override)) || (Ignited_NEW && EngRun_NEW));

        /* update each variable and mode class for this state change */
        d_step {
            Brake_OLD = Brake_NEW;
            EngRun_OLD = EngRun_NEW;
            Ignited_OLD = Ignited_NEW;
            Lever_OLD = Lever_NEW;
            Mode_OLD = Mode_NEW;
        }

        /* simulate monitored variable changes */
        if ::
            :: (Brake_OLD) -> Brake_NEW = FALSE
            :: (!Brake_OLD) -> Brake_NEW = TRUE
            fi;
        :: if
            :: (EngRun_OLD) -> EngRun_NEW = FALSE
            :: (!EngRun_OLD) -> EngRun_NEW = TRUE
            fi;
        :: if
            :: (Ignited_OLD) -> Ignited_NEW = FALSE
            :: (!Ignited_OLD) -> Ignited_NEW = TRUE
            fi;
        :: if
            :: (Lever_OLD != Activate) -> Lever_NEW = Activate
            :: (Lever_OLD != Deactivate) -> Lever_NEW = Deactivate
            :: (Lever_OLD != Resume) -> Lever_NEW = Resume
            fi;
        fi;

        /* executions of the functions in dependency order */
d_step {

    /* the PROMELA version of the Mode function */
    if
    :: (Ignited_NEW && !Ignited_OLD) && (Mode_OLD == Off)
    -> Mode_NEW = Inactive;
    :: (Ignited_OLD && !Ignited_NEW) && (Mode_OLD == Inactive)
    -> Mode_NEW = Off;
    :: ((Lever_NEW == Activate) && (!(Lever_OLD == Activate))) && (EngRun_OLD &&
      (!Brake_OLD)) && (Mode_OLD == Inactive)
    -> Mode_NEW = Cruise;
    :: (Ignited_OLD && !Ignited_NEW) && (Mode_OLD == Cruise)
    -> Mode_NEW = Off;
    :: ((EngRun_OLD && !EngRun_NEW) && Ignited_OLD) && (Mode_OLD == Cruise)
    -> Mode_NEW = Inactive;
    :: (((Brake_NEW || (Lever_NEW == Deactivate)) && (!Brake_OLD)) && (Ignited_OLD && EngRun_OLD)) && (Mode_OLD == Cruise)
    -> Mode_NEW = Override;
    :: (Ignited_OLD && !Ignited_NEW) && (Mode_OLD == Override)
    -> Mode_NEW = Off;
    :: ((EngRun_OLD && !EngRun_NEW) && Ignited_OLD) && (Mode_OLD == Override)
    -> Mode_NEW = Inactive;
    :: (((Lever_NEW == Activate) || (Lever_NEW == Resume)) && (!(Lever_OLD == Activate) || (Lever_OLD == Resume))) && (Ignited_OLD && EngRun_OLD) && (!Brake_OLD)) && (Mode_OLD == Override)
    -> Mode_NEW = Cruise;
    :: else skip;
    fi;
    od /* end of main processing loop */
}

Lurch

/* This file contains the Lurch version of an SCR specification */

#define TRUE 1
#define FALSE 0
#define Activate 0
#define Deactivate 1
#define Resume 2
#define Off 0
#define Inactive 1
#define Cruise 2
#define Override 3

cchar Brake_OLD = FALSE;
cchar Brake_NEW = FALSE;
cchar EngRun_OLD = FALSE;
cchar EngRun_NEW = FALSE;
cchar Ignited_OLD = FALSE;
cchar Ignited_NEW = FALSE;
cchar Lever_OLD = Deactivate;
char Lever_NEW = Deactivate;
char Mode_OLD = Off;
char Mode_NEW = Off;

void update_variables(void)
{
  int _p;

  Lever_OLD = Lever_NEW;
  Mode_OLD = Mode_NEW;
  Brake_OLD = Brake_NEW;
  EngRun_OLD = EngRun_NEW;
  Ignited_OLD = Ignited_NEW;

  switch (_next_int(3))
  {
    case 0: _p = 0;
    if ((Brake_OLD) && !_next_int(_p--)) { Brake_NEW=FALSE; break; }
    else if (!Brake_OLD && !_next_int(_p--)) { Brake_NEW=TRUE; break; }
    case 1: _p = 0;
    if ((EngRun_OLD) && !_next_int(_p--)) { EngRun_NEW=FALSE; break; }
    else if (!EngRun_OLD && !_next_int(_p--)) { EngRun_NEW=TRUE; break; }
    case 2: _p = 0;
    if ((Ignited_OLD) && !_next_int(_p--)) { Ignited_NEW=FALSE; break; }
    else if (!Ignited_OLD && !_next_int(_p--)) { Ignited_NEW=TRUE; break; }
    case 3: _p = 1;
    if ((Lever_OLD!=Resume) && !_next_int(_p--)) { Lever_NEW=Resume; break; }
    else if ((Lever_OLD!=Activate) && !_next_int(_p--)) { Lever_NEW=Activate; break; }
    else if ((Lever_OLD!=Deactivate) && !_next_int(_p--)) { Lever_NEW=Deactivate; break; }
    default: break;
  }
}

void _before(void)
{
  Brake_OLD = FALSE;
  Brake_NEW = FALSE;
  EngRun_OLD = FALSE;
  EngRun_NEW = FALSE;
  Ignited_OLD = FALSE;
  Ignited_NEW = FALSE;
  Lever_OLD = Deactivate;
  Lever_NEW = Deactivate;
  Mode_OLD = Off;
  Mode_NEW = Off;
}

void _hash(unsigned int *h)
{
  _hash_int(Brake_OLD, h);
  _hash_int(Brake_NEW, h);
  _hash_int(EngRun_OLD, h);
  _hash_int(EngRun_NEW, h);
_hash_int(Ignited_OLD, h);
_hash_int(Ignited_NEW, h);
_hash_int(Lever_OLD, h);
_hash_int(Lever_NEW, h);
_hash_int(Mode_OLD, h);
_hash_int(Mode_NEW, h);
}

%%

1: assert0; (!((Mode_NEW == Cruise)) || (((Ignited_NEW && EngRun_NEW) && (Brake_NEW) &&
   (!Lever_NEW == Deactivate))))); -; _assert0_violated;

2: assert1; (!((Mode_NEW == Override)) || (Ignited_NEW && EngRun_NEW))); -;
   _assert1_violated;

3: assert2; (!((Mode_NEW == Off)) || (!Ignited_NEW))); -; _assert2_violated;

4: assert3; (!((Mode_NEW == Inactive)) || Ignited_NEW)); -; _assert3_violated;

5: update_variables; -; {update_variables();}; update_variables;

6: Mode; (((Lever_NEW==Activate)&&(Lever_OLD==Activate)))&&(EngRun_OLD&&(Brake_OLD))));
   (Mode_OLD==Inactive)); {Mode_NEW=Cruise;}; Mode;
6: Mode; ((Ignited_OLD&&!Ignited_NEW))&&(Mode_OLD==Cruise)); {Mode_NEW=Off;}; Mode;
6: Mode; (((EngRun_OLD&&(EngRun_NEW))&&Ignited_OLD)&&(Mode_OLD==Cruise)); {Mode_NEW=
   Inactive;}; Mode;
6: Mode; (((Brake||Lever_NEW==Deactivate))&&(Brake_OLD||Lever_OLD==Deactivate)))&&
   (Ignited_OLD&&EngRun_OLD))&&(Mode_OLD==Cruise)); {Mode_NEW=Override;}; Mode;
6: Mode; ((Ignited_OLD&&!Ignited_NEW))&&(Mode_OLD==Override)); {Mode_NEW=Off;}; Mode;
6: Mode; (((EngRun_OLD&&(EngRun_NEW))&&Ignited_OLD)&&(Mode_OLD==Override)); {Mode_NEW=
   Inactive;}; Mode;
6: Mode; (((Lever_NEW==Activate)&&(Lever_OLD==Resume)))&&((Lever_OLD==Activate))&&
   (Lever_OLD==Resume))));(((Ignited_OLD&&EngRun_OLD)&&(Brake_OLD))&&(Mode_OLD==Override));
   {Mode_NEW=Cruise;}; Mode;
6: Mode; ((Ignited_NEW&&!Ignited_OLD))&&(Mode_OLD==Off)); {Mode_NEW=Inactive;}; Mode;
6: Mode; ((Ignited_OLD&&!Ignited_NEW))&&(Mode_OLD==Inactive)); {Mode_NEW=Off;}; Mode;
Appendix B

PACS SCR Specification

// This file contains an SCRTool system specification...

SPECIFICATION; VERSION "1.7";

TYPE "yAlarm"; BASETYPE "Enumerated"; UNITS ""; VALUES "On, Off";
TYPE "yGate"; BASETYPE "Enumerated"; UNITS ""; VALUES "Open, Closed";
TYPE "yGuardDisplay"; BASETYPE "Enumerated"; UNITS ""; VALUES "Blank, SeeOfficer";
TYPE "yKeys"; BASETYPE "Enumerated"; UNITS ""; VALUES "Blank, Number, Delete";
TYPE "yUserDisplay"; BASETYPE "Enumerated"; UNITS ""; VALUES "InsertCard, Retry, EnterPIN, InvalidPIN, PleaseProceed, SeeOfficer";
TYPE "CReads"; BASETYPE "Integer"; UNITS ""; VALUES "[0, MaxCardError]";
TYPE "TReads"; BASETYPE "Integer"; UNITS ""; VALUES "[0, MaxPINError]"

MODECLASS "mcPIN"
    MODES "Init, GetDigit1, GetDigit2, GetDigit3, GetDigit4"; INITMODE "Init"
MODECLASS "mcStatus"
    MODES "EnterCard, CheckCard, EnterPIN, CheckPIN, Proceed, ReEnterCard, ReEnterPIN, Error, Override"; INITMODE "EnterCard"

CONSTANT "MaxCardError"; TYPE "Integer"; VAL "3"
CONSTANT "MaxPINError"; TYPE "Integer"; VAL "3"

CON "cGate"; TYPE "yGate"; INITVAL "Closed"
CON "cGuardAlarm"; TYPE "yAlarm"; INITVAL "Off"
CON "cGuardDisplay"; TYPE "yGuardDisplay"; INITVAL "Blank"
CON "cUserDisplay"; TYPE "yUserDisplay"; INITVAL "InsertCard"
MON "mCardInput"; TYPE "Boolean"; INITVAL "FALSE"
MON "mCardValid"; TYPE "Boolean"; INITVAL "FALSE"
MON "mDigit1"; TYPE "yKeys"; INITVAL "Blank"
MON "mDigit2"; TYPE "yKeys"; INITVAL "Blank"
MON "mDigit3"; TYPE "yKeys"; INITVAL "Blank"
MON "mDigit4"; TYPE "yKeys"; INITVAL "Blank"
MON "mGate"; TYPE "yGate"; INITVAL "Closed"
MON "mOverride"; TYPE "Boolean"; INITVAL "FALSE"
MON "mPINValid"; TYPE "Boolean"; INITVAL "FALSE"
MON "mReset"; TYPE "Boolean"; INITVAL "FALSE"
TERM "tPINInput"; TYPE "Boolean"; INITVAL "FALSE"
TERM "tNumCReads"; TYPE "CReads"; INITVAL "0"
TERM "tNumPReads"; TYPE "TReads"; INITVAL "0"
ASSERTION "AlarmStatus";
EXPR "((cGuardAlarm = On) <=> (cUserDisplay = SeeOfficer))";

ASSERTION "CardDisplay1";
EXPR "((tNumCReads > 0 AND tNumCReads < MaxCardError) <=> (cUserDisplay = Retry))";

ASSERTION "CardDisplay2";
EXPR "((tNumCReads = MaxCardError) => (cUserDisplay = SeeOfficer))";

ASSERTION "CardErrors";
EXPR "((cUserDisplay = InsertCard OR cUserDisplay = Retry) AND mcStatus = CheckCard AND @F(mCardValid)) => (tNumCReads' = tNumCReads + 1)";

ASSERTION "CardSuccess";
EXPR "((cUserDisplay = InsertCard OR cUserDisplay = Retry) AND mcStatus = CheckCard AND @T(mCardValid)) => (cUserDisplay' = EnterPIN)";

ASSERTION "GateStatus";
EXPR "((cUserDisplay = PleaseProceed) => (cGate = Open))";

ASSERTION "NumCardErrors";
EXPR "((cUserDisplay = InsertCard OR cUserDisplay = Retry) AND mcStatus = CheckCard AND @F(mCardValid)) => (tNumCReads' = tNumCReads + 1)";

ASSERTION "PINDisplay1";
EXPR "((cUserDisplay = InsertCard OR cUserDisplay = Retry) AND mcStatus = CheckCard AND @T(mCardValid)) => (cUserDisplay' = EnterPIN)";

ASSERTION "PINDisplay2";
EXPR "((cUserDisplay = InsertCard OR cUserDisplay = Retry) AND mcStatus = CheckCard AND @T(mCardValid)) => (cUserDisplay' = EnterPIN)";

NATURE "GateOpen";
EXPR "@T(mGate = Open) => (cGate = Open)";

CONDFUNC "cGate";
CONDITIONS "mcStatus = Proceed OR mcStatus = Override", "NOT (mcStatus = Proceed OR mcStatus = Override)"
ASSIGNMENTS "Open", "Closed";

CONDFUNC "cGuardAlarm";
CONDITIONS "mcStatus = Error", "NOT (mcStatus = Error)"
ASSIGNMENTS "On", "Off";

EVENTFUNC "cGuardDisplay";
EVENTS "@T(mcStatus = EnterCard) OR @T(mcStatus = Override)", "@T(mcStatus = Error)"
ASSIGNMENTS "Blank", "SeeOfficer";

EVENTFUNC "cUserDisplay";
EVENTS "@T(mcStatus = EnterCard)", "@T(mcStatus = ReEnterCard)", "@T(mcStatus = EnterPIN)", "@T(mcStatus = ReEnterPIN)", "@T(mcStatus = Proceed) OR @T(mcStatus = Override)", "@T(mcStatus = Error)"
ASSIGNMENTS "InsertCard", "Retry", "EnterPIN", "InvalidPIN", "PleaseProceed";

ASSIGNMENTS "Open", "Closed";

ASSIGNMENTS "On", "Off";

ASSIGNMENTS "Blank", "SeeOfficer";
EVENTFUNC "tPINInput";
EVENTS "@T(mDigit4 = Number) WHEN (mcPIN = GetDigit4)"
ASSIGNMENTS "NOT tPINInput";

MODETRANS "mcPIN";
FROM "Init" EVENT "@T(mCardValid) WHEN (mcStatus = CheckCard) OR @F(mPINValid) WHEN
(mcStatus = CheckPIN AND tNumPReads < MaxPINError - 1)" TO "GetDigit1";
FROM "GetDigit1" EVENT "@T(mDigit1 = Number)" TO "GetDigit2";
FROM "GetDigit2" EVENT "@T(mDigit2 = Number)" TO "GetDigit3";
FROM "GetDigit2" EVENT "@T(mDigit2 = Delete)" TO "GetDigit1";
FROM "GetDigit3" EVENT "@T(mDigit3 = Number)" TO "GetDigit4";
FROM "GetDigit3" EVENT "@T(mDigit3 = Delete)" TO "GetDigit2";
FROM "GetDigit4" EVENT "@T(mDigit4 = Number)" TO "Init";
FROM "GetDigit4" EVENT "@T(mDigit4 = Delete)" TO "GetDigit3";
FROM "GetDigit1, GetDigit2, GetDigit3, GetDigit4" EVENT "@C(mReset)" TO "Init";

MODETRANS "mcStatus";
FROM "EnterCard" EVENT "@C(mCardInput)" TO "CheckCard";
FROM "CheckCard" EVENT "@T(mCardValid)" TO "EnterPIN";
FROM "EnterPIN" EVENT "@C(tPINInput)" TO "CheckPIN";
FROM "CheckPIN" EVENT "@T(mPINValid)" TO "Proceed";
FROM "Proceed" EVENT "@T(mGate = Closed)" TO "EnterCard";
FROM "CheckCard" EVENT "@F(mCardValid) WHEN (tNumCReads < MaxCardError - 1)" TO
"ReEnterCard";
FROM "CheckCard" EVENT "@F(mCardValid) WHEN (tNumCReads = MaxCardError - 1)" TO "Error";
FROM "ReEnterPIN" EVENT "@F(mPINValid) WHEN (tNumPReads < MaxPINError - 1)" TO "ReEnterPIN";
FROM "CheckPIN" EVENT "@F(mPINValid) WHEN (tNumPReads = MaxPINError - 1)" TO "Error";
FROM "Error" EVENT "@C(mOverride)" TO "Override";
FROM "Override" EVENT "@T(mGate = Closed)" TO "EnterCard";
FROM "EnterCard, CheckCard, EnterPIN, CheckPIN, ReEnterCard, ReEnterPIN, Error, Override,
Proceed" EVENT "@C(mReset)" TO "EnterCard";

EVENTFUNC "tNumCReads"; MCLASS "mcStatus";
MODES "CheckCard" EVENTS "@T(mCardValid) OR @C(mReset)", "@F(mCardValid)"
MODES "Error" EVENTS "@C(mOverride) OR @C(mReset)", "NEVER"
MODES "ReEnterCard" EVENTS "@C(mReset)", "NEVER"
ASSIGNMENTS "0", "tNumCReads + 1";

EVENTFUNC "tNumPReads"; MCLASS "mcStatus";
MODES "CheckPIN" EVENTS "@T(mPINValid) OR @C(mReset)", "@F(mPINValid)"
MODES "Error" EVENTS "@C(mOverride) OR @C(mReset)", "NEVER"
MODES "ReEnterPIN" EVENTS "@C(mReset)", "NEVER"
ASSIGNMENTS "0", "tNumPReads + 1";
Appendix C

SCR to Lurch Translator

# AWK translator from SCR Toolset’s automatically generated Promela code to Lurch

BEGIN {

# Instead of breaking input on newlines, break on key text in Promela code
RS = "\n#define |bool |byte |int |init |" \n    " */\* "any state" specification asserts */" \n    " */\* update each variable and mode class */" \n    " } */\* close state update */" \n    " */\* simulate monitored variable changes */" \n    " */\* randomly select any value */" \n    " */\* toggle the current value */" \n    " d_step { */\* print new value, */" \n    " */\* randomly jump to any value */" \n    " */\* executions of the functions */" \n    " */\* the PROMELA version of the */" \n    " } */\* close calculation d_step */" \n    " od */\* end of main processing loop */";

wt = 1;
case_count = 0;
}

# trigger matches based on previous record terminator - this one is a #define
prt~/define/ {  
    printf("%ss %s", prt, $1, $2);
    if (RT!~/define/) printf("\n");
}

# variable declarations
prt~/bool|byte|int/ {  
    printf("%s", prt~/int/ ? prt : "\nchar ");
    printf("%s = %s", $1, $3);
    bf = sprintf("%s %s = %s\n", bf, $1, $3);
    sv = sprintf("%s _hash_int(%s, h);\n", sv, $1);
    if (RT!~/bool|byte|int/) printf("\n");
}

121
# first set of assertions
prt˜/any state/ {
    split($0, tmp, "assert\[\]");
    for (i in tmp)
        if (tmp[i] != "")
            asa[a++] = sprintf("assert%i; (!(%s)); -; _assert%i_violated;\n\n", l, substr(tmp[i], 0, (index(tmp[i], ")") - 1)), l++;
}

# section where variables are updated (var_NEW value copied to var_OLD)
prt˜/update each variable/ {
    split($0, tmp, "d_step.*\*/( |\t)*");
    split(tmp[2], tmp, "; ( |\t)*");
    printf("\nvoid update_variables(void)\n{\nint _p;\n\nfor (i in tmp)
    if (tmp[i] != "")
        {\n            gsub(" ", ",", tmp[i]);
            printf("%s;", tmp[i]);
        }
    printf("\n");\n}

# (random) change of a monitored variable
prt˜/randomly|jump to|toggle/ {
    _p = split($0, tmp, "::") - 3;
    new_case = 1;
    for (i in tmp)
        if (tmp[i] !~\/*//)
            {\n                gsub(/ |fi;\n/", ",", tmp[i]);
                split(tmp[i], tmp2, "->");
                if (new_case)
                    {\n                        new_case = 0;
                        mv = sprintf("%s case %i: _p = %i;\n\n", mv, case_count++, _p);
                        mv = sprintf("%s if (%s && !_next_int(_p--)) { %s; break; }\n\n", mv, tmp2[1], tmp2[2]);
                    }
                else
                    mv = sprintf("%s else if (%s && !_next_int(_p--)) { %s; break; }\n\n", mv, tmp2[1], tmp2[2]);
            }
} 122
# simulator output of new monitored variable value
prt~/print/ {
  split($0, tmp, "::");
  for (i in tmp)
  {
    gsub(/ |[\n]\(\//, "", tmp[i]);
    split(tmp[i], tmp2, ">")
    if (match(tmp2[2], "printf"))
    {
      gsub("fi", ",", tmp2[2]);
      gsub("printf\(\)\", ",", tmp2[2]);
      gsub("\(\)\", ",", tmp2[2]);
      j = index(tmp2[1], ")");
      if (!j) j = index(tmp2[1], ")");
      j++;
      k = index(tmp2[1], ")") - j;
      nm = substr(tmp2[1], j, k);
      split(tmp2[2], tmp3, nm);
      pr = sprintf("%s if %s sprintf(c, "%%s %s %s %s, c);\n", pr, 
                   tmp2[1], tmp3[1], nm, tmp3[2]);
    }
  }
}

# start of finite-state machine definitions
prt~/executions/ {
  printf("\n switch (_next_int(%i))\n", --case_count);
  printf("  \{");
  printf("  default: break;\n");
  printf("  }\n");
  printf("\n");
  printf("void _before(void)\n");
  printf("\n");
  printf("void _print(char *c)\n");
  printf("\n");
  printf("void _hash(unsigned int *h)\n");
  printf("\n");
  printf("for (i in asa)\n  printf("%i: %s", wt++, asa[i]);
\n123
printf("%i: update_variables; -; {update_variables();}; ", wt++);
printf("update_variables\n\n");
}

# start of individual finite-state machine
prt˜/PROMELA/ {

name = $1;
split($0, tmp, "::");

for (i in tmp)
if (tmp[i]!˜/if|else/) {

gsub(/ \n/, ", tmp[i]);
split(tmp[i], tmp2, "->");

printf("%i: %s; (%s); {%s}; %s;
", wt, name, tmp2[1], substr(tmp2[2], 0, index(tmp2[2], ";")), name);
}
printf("\n");
wt++;
}

# end of model (right after second set of assertions)
prt˜/calculation/ {

split($0, tmp, "assert[]");

for (i in tmp)
if (tmp[i]!";")
printf("%i: assert%i; (!(%s)); -; _assert%i_violated;\n\n", wt++, l, substr(tmp[i], 0, index(tmp[i], ";") - 1)), i++;
}

# prt will always be the previous record terminator
{ prt = RT; }
Bibliography


126


