Effectively Combining Software Verification Strategies: Understanding Different Assumptions

David Owen, Dejan Desovski, Bojan Cukic
Lane Department of Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV 26506
{dowen|desovski|cukic}@csee.wvu.edu

Abstract

In this paper we describe an experiment in which inconsistent results between two tools for testing formal models (and a third used to determine which of the two was correct) led us to a more careful look at the way each tool was being used and a clearer understanding of the output of the tools. For the experiment, we created error-seeded versions of an SCR specification representing a real-world personnel access control system. They were checked using the model checker SPIN and Lurch, our random testing tool for finite-state models. In one case a property violation was detected by Lurch, an incomplete tool, but missed by SPIN, a model checking tool designed for complete verification. We used the SCR Toolset and the Salsa invariant checker to determine that the violation detected by Lurch was indeed present in the specification. We then looked more carefully at how we were using SPIN in conjunction with the SCR Toolset and, eventually, made adjustments so that SPIN also detected the property violation initially detected only by Lurch. Once it was clear the tools were being used correctly and would give consistent results, we did an experiment to determine how they could be combined to optimize completeness and efficiency. We found that combining tools made it possible to verify the specifications faster and with much less memory in most cases.

1. Introduction

There are many different tools available for verifying software models. Some have been designed to find relatively simple errors in very complex models or even code, e.g., compilers and integrated development environments that detect common structural errors. Other tools are capable of finding very complex errors in simpler abstract models, e.g., model checking tools for automated verification of temporal logic properties [8,14]. There are also various kinds of hybrid tools between these two extremes, targeting a range of different kinds of errors and models at different levels of complexity [2,10,19]. Different tools restrict in different ways the kinds of errors and models that can be checked. They may also make different assumptions about the models and the kinds of results expected by the user. So it can be difficult to determine in a particular situation which tool should be used. Also, most of these tools have various modes and parameters affecting time and memory requirements, what types of errors can be detected, and what types of input models are acceptable. Even for a single tool, different configurations may reflect very different assumptions.

Some software specification and development frameworks make several complementary verification approaches available [9, 11, 12]. We have been working with one such framework, the SCR (Software Cost Reduction) Toolset [12], which can be used to automatically generate and test models using various tools including Salsa [6], SPIN [14], SMV [7,15] and TAME [4]. Using the SCR Toolset in combination with these tools, it is possible to find many different types of faults or to prove complex correctness properties in specifications written using the tabular SCR notation.

A typical SCR requirements specification consists of two parts: an operational part, which describes how the system operates, and a property-based part, which describes the properties that the system must satisfy. The operational part represents the system as a finite-state machine, in tabular SCR notation, while the property-based part is made up of first-order logic formulas representing state or transition (two-state) invariants which must hold for the system. For an SCR requirements specification to be correct, properties must be stated correctly and there must be no violations of those properties in the operational part

of the specification. The SCR Toolset makes it possible to automatically detect inconsistencies between the property-based and operational parts of the specification.

We developed a translator to produce Lurch (see [17] and the description below) versions of SCR specifications and have been working on combining Lurch with the SPIN model checker as a demonstration of how different tools can be used together. Our goal is to detect a wider range of errors in shorter time with less effort and resource requirements, compared to a single tool used alone. This is an important addition to providing alternative verification tools—to understand how these alternatives can be combined effectively. Eventually, we hope to learn more general guidelines that users could apply throughout the software lifecycle, when selecting and combining debugging and verification tools, but at this point we are focusing on a small set of specific tools that would be applied early in the lifecycle.

As illustrated by the experiments below, our experience has been that combining verification tools, in addition to sometimes improving efficiency, can be very helpful for understanding the tools. We used Lurch, our own simple random testing tool for finite-state models, together with the model checker SPIN, a very powerful verification tool that can be used in many different ways depending on the needs and assumptions of the user. The unexpected result, that Lurch's incomplete random simulation detected a property violation missed by SPIN's exhaustive search, first led to a close look at the way in which we were using Lurch. We then used the SCR Simulator and Salsa tools to validate Lurch's results, finding that the detected property violation indeed existed in the specification.

Through experimentation with SPIN's different compilation and runtime options, and as we looked carefully at the SCR Toolset's Promela translation (Promela is the input language for SPIN), we learned more about SCR, the SCR Toolset, and SPIN. Eventually we were able to modify the SCR-to-Promela translation so that SPIN detected the violation not detected previously. While our experimental results very specifically relate to the types of specification languages and verification tools utilized, we believe there are more general lessons to be learned by practitioners and researchers who combine different model translators and verification tools.

After resolving the inconsistency in results between the tools, we ran an experiment to determine how they could be combined to minimize time and memory requirements while maintaining completeness. Lurch was used first, for a limited amount of time; if no property violations were detected, SPIN was run to fully verify the correctness of the model. We also tried running Lurch, then (if no property violations were detected) SPIN with options set to run a quick but not necessarily complete verification, and then (if still no property violations had been detected) SPIN with options set for complete verification. We found the combination of tools was somewhat faster and required much less memory in many cases, compared to running SPIN alone with options set for complete verification.

This paper is organized as follows. Section 2 describes the Lurch, SPIN and Salsa tools used in our experiments in conjunction with the SCR Toolset. The description of Lurch in section 2.1 is more detailed, while the descriptions of the better-known SPIN and Salsa tools provide references to documentation available elsewhere. Section 2.4 explains how we are using the SCR Toolset's Promela translator to produce Lurch models, and section 2.5 describes the case study and the SCR specification used in our experiments. Section 3 describes the experiments in which we combined Lurch and SPIN to verify models generated from the SCR Toolset. Section 3.1 describes the unexpected result that our random simulation tool Lurch detected a property violation not detected initially by SPIN. Section 3.2 explains how a better understanding of SPIN's utilization within the SCR Toolset helped us understand and resolve the problem. In section 3.3 we present experimental results. Finally, sections 4 and 5 discuss the broader implications of our work, giving general conclusions and suggestions for future work.

2. Background

In the experiment below Three different verification tools were used to check a requirements specification written in SCR: Lurch, our random exploration tool; Salsa, an invariant checking tool; and SPIN, a complete model-checking tool.

Lurch is used for random testing and debugging of finite-state concurrent systems. Its purpose is to quickly identify errors without requiring much memory, even for large systems. Salsa is an invariant checker that uses induction and several decision procedures to prove whether a given property holds for the system. Salsa also can be used to check the completeness and disjointness of SCR specifications. SPIN is a well known model-checking tool that supports different validation and verification techniques (e.g., simulation, exhaustive verification by complete state-based search, approximate verification by state hashing). In this study we were interested in the exhaustive verification provided by SPIN: we used it as a safeguard against the errors missed by the first two tools. Our

```
#define FALSE 0
#define TRUE 1
enum { 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 1. Lurch input model representing Dekker's solution to the two-process mutual exclusion problem (translated from a model written for the model checker SPIN [13]).

intention was to demonstrate that by using incomplete verification tools like Lurch and Salsa we can quickly identify possible errors, reducing the need to perform complete verification and consequently reducing the time and memory requirements for verification of software models.

2.1. Lurch

In this section we describe Lurch [17], our simulation tool for random testing and debugging models of finite-state concurrent systems. Although Lurch's exploration of the state space is not exhaustive, it can be used to detect errors in large systems quickly using much less memory than complete-search alternatives.

Figure 1 shows a simple finite-state model written for Lurch, translated from a model written for the model checker SPIN [13]. The model begins with C code, which may be referred to in the model. The special function *_before()* is called by Lurch internally to set C variables to their initial values. After the *%%* symbol, finite-state machines are listed; each line in a machine description represents a transition and has four fields: the current state, input conditions, output or side effects of the transition, and the next state. In this model the final line represents 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.

Lurch works by simulating the execution of the model, choosing randomly when more than one branch is possible, until it reaches either the end of a path, an error, or a user-specified depth limit. This basic simulation cycle is repeated until a user-specified number of paths have been explored. Lurch checks for deadlocks (the end of a path at which point one or more machines in the model are not in a legal end state), violations of user-specified safety properties, and likely liveness property violations (for example, a path which remains in a no-progress cycle until a user-specified number of cycle repetitions is reached).

We added an early stopping heuristic to Lurch, attempting to take advantage of the "saturation" effect observed in many experiments [16]: random simulation often finds unique information at first but then quickly shifts to finding redundant information. Our saturation-based stopping criterion works in the following way: for each path generated, we save hash values for all unique global states visited and compare the number of collisions with values from previously explored paths to the number of new values on the current path. 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. When Lurch is allowed to run to saturation it almost always produces consistent results.

Lurch was originally designed to run on asynchronous models in which all individual state machines run in parallel. The experiments presented in this paper use functionality added later to support models in which the individual machines "take turns" in a specified order. This is necessary because the models in our case study originated from SCR specifications. These are synchronous models whose components execute in dependency order (no circular dependencies are allowed in SCR). Also, the experiments presented here use a great deal of C code in the first part of the model and in the second part minimal "dummy" state machine descriptions, through which the random search engine accesses the C code. This made it much easier to write an automatic translator for the models. In spite of the fact that we were using Lurch on synchronous models written mostly in C, which is very different from what we had in mind originally for Lurch, we found Lurch to be a helpful addition to SPIN for verifying SCR specifications.

2.2. SPIN

The SPIN model checker [14], developed by Holzmann and others over the last twenty years, is probably

the most popular freely available verification tool for software models. SPIN's input language Promela has a syntax similar to a high-level programming language. Various features available with SPIN make it much less difficult to write input models and specify properties to be verified. SPIN can be run in several modes with many different options, representing a great deal of research on strategies to decrease the time and memory required for verification. Often input models far too large for verification by SPIN with default settings can be verified efficiently in a different mode with the right set of options.

For the experiments presented below, there are a few important points about how we used SPIN. First, the input models were automatically generated from SCR specifications by the SCR Toolset. Verification of automatically generated models tends to require more time and memory than verification of models carefully written by hand, which can be fine-tuned to minimize resource requirements.

Second, we used the "minimized automaton" compression option available with SPIN. When SPIN (with default settings, for example) runs out of memory and quits early, included in the output are instructions for recompiling the verifier with this compression option enabled. Compared to other memory-saving options available with SPIN that preserve completeness, this "minimized automaton" compression option is usually the slowest but uses the least memory. Other compression options that do not preserve completeness, such as bitstate hashing, may require much less memory but only provide approximate results.

Third, in Promela it is possible to mark blocks of the model as deterministic steps. This can greatly improve the efficiency of the verification, but should only be used when the code marked in this way is in fact deterministic. As shown by the experiments presented below, the use of this feature reflects an important assumption that users should understand. When this feature is used in auto-generated models, although it may be necessary for efficiency, there is the risk that some of the behavior in the original specification will be hidden from SPIN, and if the hidden behavior includes a property violation, it will not be detected.

2.3. Salsa

The Salsa tool [6] is an invariant checker. It uses induction to determine if a given state or transition property holds for the SCR specification under analysis. The main idea of the algorithms applied is to prove that a given state property holds for the initial state, and that its conjunction with the transition predicate specifying the evolution of the system would still hold—representing an inductive proof. A similar strategy is used for the transition properties. Salsa employs a combination of decision procedures (e.g., propositional logic, the theory of unordered enumerations, and integer linear arithmetic) based on BDD algorithms together with a linear constraint solver. Although these decision problems are NP-complete, Salsa implements several heuristic optimizations which work very well in practice.

Salsa's invariant checking algorithms are sound, but they are not complete, due to the use of induction. If Salsa reports that the specification passes a check for a particular property, the specification definitely does not violate that property. On the other hand, if Salsa reports that the specification fails a check for a property, it does not necessarily mean that the property is violated. It is interesting to compare Salsa to Lurch as complementary tools: Salsa is designed to prove properties and may not be able to prove them all. On the other hand, Lurch is designed to detect property violations and may not be able to find them all. Therefore Salsa says "pass" or "I don't know," while Lurch says "fail" or "I don't know." Complete tools like SPIN say "pass" or "fail," but may require much more time and memory to reach such a conclusion.

It has been demonstrated that Salsa performs better in disjointness checking than the standard SCR consistency checker included in the SCR Toolset [6]. Our results were consistent with this observation. Salsa was able to quickly detect a disjointness error not detectable with the SCR Toolset's consistency checker. SPIN could detect this error only after changes described below were introduced into the SCR Toolset's automatically generated Promela model.

2.4. SCR to Promela to Lurch Translation

SCR specifications and Lurch models are similar in some ways, but we found that translating from the Promela code automatically generated by the SCR Toolset was much easier than translating directly from SCR. The Promela generator already solves the two main challenges for SCR to Lurch translation:

- The dependency order for execution of individual finite-state machines is not specified directly in the SCR specification, but must be included explicitly in the Promela (or Lurch) model.
- SCR allows transitions to be triggered based on the value of the current and previous state of the system. Promela (and Lurch) transitions are based on the current state only, so a copy of the

previous state must be included as part of the current state.

The SCR Toolset's Promela generator lists machines in dependency order and declares a VAR_NEW and VAR_OLD for each variable in the system. It is also easier to translate from Promela to Lurch because Promela uses the same syntax for macro definitions and boolean expressions (C syntax). Finally, because the Promela model is auto-generated, the Lurch translator does not need to be able to translate arbitrary Promela code (a much bigger challenge). Instead, the Lurch translator can count on certain parts of the Promela model always being written in a specific way, with the same comments marking the sections every time.

The SCR Toolset's Promela translator marks many blocks deterministic, thereby significantly decreasing the size of the state space and greatly improving the performance of SPIN. Since a valid SCR specification must be deterministic, and there are other checks in the SCR Toolset to warn you when a specification is not, it should usually be safe to mark portions of the Promela model as deterministic steps. But it is not always safe to do so, as we found in the experiments below. Our Lurch translator ignores the deterministic step declarations in the Promela model generated by the SCR Toolset (Lurch has no such feature, nor is it present in the original SCR specification). Because of this discrepancy, in one specific case, Lurch was able to detect a property violation initially missed by SPIN.

2.5. PACS SCR Specification

In the experiments presented below, we used an SCR specification for a Personnel Access Control System (PACS) originally described in a prose requirements document from the National Security Agency [1]. The SCR specification had been derived from that document as an example to demonstrate how to write a high quality formal requirements specification.

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 validate 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, 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 systems 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 (either a sound or light) on the officer's console. Before system operation can resume, the officer must reset the PACS by pushing the reset button. The user, who also 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 an invalid PIN is entered three times 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.

To give an idea of the scale of the experiments, the PACS specification was about 150 lines and included 17 variables, 2 mode classes and 46 transitions in the function tables. The Promela model generated for the specification was about 500 lines. For a verification run on the model SPIN explored 180 million states. Without compression verification would have required 6.8 GB, according to statistics output by SPIN. In order to perform a full verification run we used the minimized automaton compression option, set for a state vector of 44 bytes, and had to increase the depth limit for the search to 3.2 million steps (default is 10,000). This verification run took about 30 minutes on a 2.5 GHz desktop machine with 512 MB of memory.

3. PACS Verification Experiment

In order to explore the benefits of combining complementary strategies for verifying formal models, we used the SCR Toolset, SPIN and Lurch, and the PACS SCR specification described above. These tools were chosen because we were already familiar with them and had access to the PACS specification written in SCR. Through the SCR Toolset, we also had access to a simulator, a range of built-in checking features, and translators to several backend verification tools in addition to SPIN.

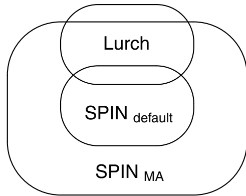

Figure 2. Sets of error-seeded specifications for which property violations were detected by different methods.

3.1. Setup and Unexpected Initial Result

Errors were seeded manually to create 50 modified versions of the PACS requirements specification. These seeded errors were not based on expert knowledge of the system. Instead, we used a simple set of mutations including: switching event operators `@T` (triggered when a value becomes true), `@F` (true when a value becomes false), and `@C` (true when a value changes); we also incremented or decremented numeric values in specifications. We expect our results would be similar to larger scale experiments in which the same kind of mutations would be done automatically. Others have argued that this approach may more accurately represent real errors than seeded errors based on expert knowledge of the correct system [3].

Each error-seeded specification contained one manually seeded error not detectable by automatic checking features of the SCR Tool. For example, since syntax errors or errors involving circular dependencies are detectable by the SCR Tool, these were not used for our seeded errors. We used the translation tools described above to create SPIN and Lurch versions of each error-seeded specification. All error-seeded SCR specifications resulted in syntactically correct SPIN and Lurch input models. We then ran SPIN and Lurch on the 50 error-seeded specifications.

Figure 2 is based on the first set of experimental results.[1] Since Lurch does not perform a complete search of the model, it makes sense that there are property violations detected by SPIN but not by Lurch. SPIN with default settings runs quickly but runs out of memory in many cases before finding a property violation or finishing the verification run, so it also makes sense that there are some property violations detected by Lurch but not by SPIN with default settings.

When SPIN with default options ran out of memory, the output from SPIN indicated that the minimized automaton compression option should be used, with state vector size to 44 bytes. It was also necessary to set the depth limit for SPIN's search of the state space to 1.8 million in order to get a complete verification run. The oval marked "SPIN MA" represents SPIN compiled with these options. The verification run with these options produced a very unexpected result: although SPIN completed the run, there was still one specification in which Lurch had detected a violation but SPIN had not. The property violated in this case was an assertion that the user's display and the officer's display should always match.

After looking carefully at the Lurch model in which the error was detected and not finding any obvious problems, we used the SCR Simulator to check the error trace provided by Lurch. The SCR Simulator allows a user to go step by step through an SCR specification, changing the values of one input variable at a time. As the inputs are changed, the user observes the effects on other variables in the system and can check at each step that none of the specified correctness properties have been violated. Using the simulator we confirmed that the path to the property violation detected by Lurch was indeed present in the original error-seeded SCR specification.

We also used the SCR Toolset to export the specification to SAL format and check it using the Salsa invariant checker described above. Because Salsa sometimes fails to prove properties that are actually true of the input, we ran it on the original correct version of the SCR specification as well and compared that result to the output from the error-seeded specification for which SPIN and Lurch had produced inconsistent results. For one of the tables in the specification, Salsa was able to prove that table disjoint in the original specification but not in the error-seeded version. Looking back at the SCR simulator result, it was clear that this discrepancy corresponded to the property violation reported by Lurch.

3.2. Explanation of Unexpected Result

Considering all these results together and looking carefully at the Promela code generated by the SCR Toolset, we noticed that many blocks of the code were marked as deterministic steps using the "d_step" feature. For a block of Promela representing a disjoint (i.e., deterministic) SCR table, this is valid. But for

[1] This diagram (and the one in figure 3 as well) is only meant to show the different sets of error-seeded specifications for which each verification method detected property violations. It is not meant to show the proportion of specifications in each set. For example, the majority of error-seeded specifications contained property violations detectable by all three verification methods and are represented in figures 2 and 3 by the overlap between all sets.

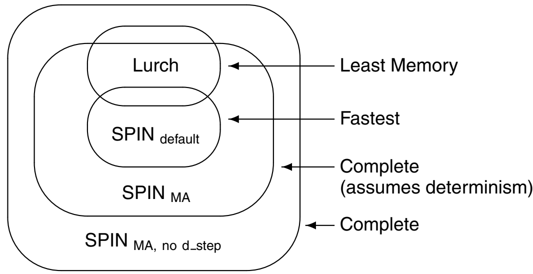

Figure 3. Sets of error-seeded specifications for which property violations were detected by different methods, including SPIN running on Promela models with d_step markers taken out.

any block representing a table that is not disjoint, marking it as a deterministic step has the effect of hiding some of the behavior of the specification from SPIN.[2]

In this particular case one of the tables in the specification was not disjoint, as was shown by running Lurch's trace file through the SCR Simulator. The same observation was suggested by inconclusive results from the SCR Toolset's consistency checker and by Salsa's inability to prove disjointness for the error-seeded specification. Because this table was not disjoint, the use of the d_step in the Promela code generated by the SCR Toolset was not appropriate as it had hidden some of the behaviors of the SCR specification from which it was generated. In this case the hidden behavior included a violation of one of the specified properties—this was the property violation detected by Lurch.

After removing the d_step markers from the Promela, we reran SPIN and it quickly detected the property violation caused by the non-disjoint table in the error-seeded SCR specification. We then repeated the entire experiment with d_step markers removed, to get a baseline for SPIN doing complete verification of the entire behavior of the error-seeded SCR specifications, without assuming disjointness. To do this we had to increase the depth limit to 3.2 million, although we kept the minimized automaton compression option the same. Without the d_step markers SPIN required about twice as much time and about 30% more memory.

[2]Thus is not a bug in the translator, but reflects the assumption that the specification to be translated is disjoint. Users need to understand this assumption in order to fully understand the results of SPIN running on a model produced by the SCR Toolset.

Figure 3 is based on the second round of experimental results, this time including SPIN runs on the error-seeded specifications with the d_step markers taken out. SPIN with default settings runs the fastest but does not catch all property violations. Lurch uses the least memory but does not catch all property violations. SPIN with the minimized automaton compression algorithm (SPIN MA) and depth limit set to 1.8 million detects many more but not all property violations. Finally, SPIN with minimized automaton compression, depth limit set to 3.2 million and the d_step markers taken out detects all property violations.

3.3. Results from Second Round of Experiments

After resolving the issue with the specific error-seeded specification in which Lurch caught a property violation not originally detected by SPIN, we repeated the whole experiment for all 50 error-seeded specifications, producing the results illustrated by the diagram in figure 3. We still wanted to achieve the original goal of this experiment: to learn more about how complementary tools can be combined to more efficiently debug and/or verify software models. To accomplish this, we checked the 50 error-seeded specifications using:

- Lurch, run with default options.
- SPIN, run with default options.
- SPIN, run with the minimized automaton option and depth limit set to 3.2 million, and with d_step markers removed from the input model.

For 35 of the 50 error-seeded specifications, Lurch with default settings detected a property violation. Because of Lurch's incomplete random search, it is possible to obtain different results in different runs. Lurch was therefore run 10 times on each specification, with default options (including the "saturation" stopping criterion described above). The maximum time for any Lurch run was about 15 seconds. For all but two of the specifications, Lurch's results were consistent: violations were found in either all or none of the runs. We only counted Lurch as detecting violations that it found in all ten runs.

For 33 of the 50 specifications, SPIN with default settings was able to detect property violations. As shown in figure 3, these 33 include some violations not detected by Lurch. SPIN with the minimized automaton compression option and increased depth limit, and with d_step markers removed from the input model, detected property violations in 43 of the 50 specifications, including all violations detected by Lurch and

SPIN with default options. We assume that for the remaining 7 specifications the seeded errors did not cause violations of any specified properties. All properties considered in this experiment were simple safety properties, represented as assertions in the SPIN and Lurch models.

Based on the results of running Lurch and SPIN in the two modes on the error-seeded specifications, we considered the following hypothetical combination scenario involving Lurch and SPIN: Run Lurch for n seconds and if no property violations were detected run SPIN with minimized automaton compression, increased depth limit and d_steps removed, so that a complete verification could be done. We created a large spreadsheet with results for the tools, combining them according to $n = 1, 2, 3 ...$ seconds. Balancing time and memory (giving % improvement of time values and % improvement of memory values equal weight), we found the best performance in our computing environment was achieved for n equal to 7 seconds. We also considered a combination of SPIN with these options and SPIN with just the default options, and found the best result running SPIN with default options first for 6 seconds.

Finally, we determined that the best three-way combination was to run Lurch for 3 seconds, then if no property violations were detected to run SPIN with default options for 6 seconds, and then if no property violations were detected at that point to run SPIN with minimized automaton compression, increased depth limit and d_step markers removed, until a complete verification was done.

Figure 4 shows at the top a comparison of the time required for SPIN alone with options set for complete search, vs. SPIN with those options combined with Lurch. The average time for SPIN with options set for complete search (baseline for these experiments) was 76.8 seconds. Combined with Lurch, the average time was reduced to 45.9 seconds. Note that the average is skewed by the very time-consuming cases on the far right of the graph, which is plotted with a logarithmic scale on the y-axis.

The graph in the middle of figure 4 shows a comparison of the time required for SPIN alone with options set for complete search vs. SPIN with those options combined with SPIN running with default options. In this case, the combination decreased the average time required from 76.8 to 51.9 seconds.

The graph at the bottom of figure 4 shows a comparison of the time required for SPIN alone with options set for complete search, vs. Lurch run for 3 seconds, then SPIN with default options run for 6 seconds, and then SPIN with options set to run for a complete verification. Combining all three, the average time decreased from 76.8 to 46.3 seconds.

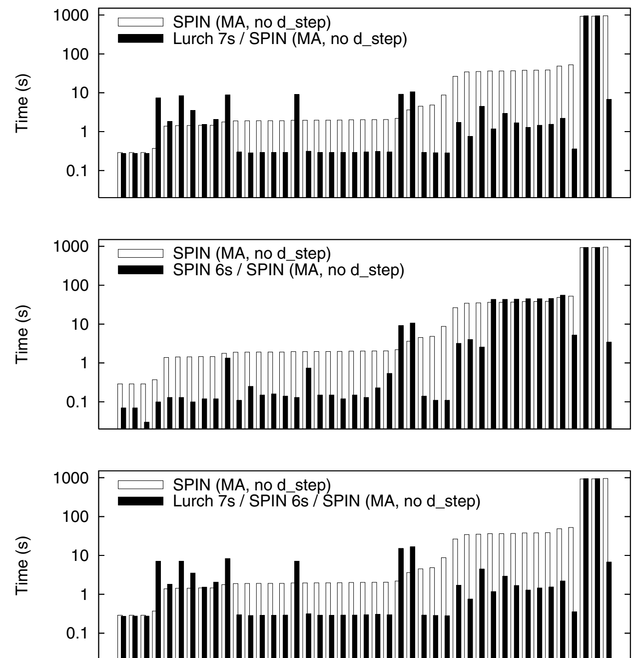

Figure 4. Time (s) used for complete search alone vs. the 3 combined verification strategies.

Figure 5 shows the same three comparisons as figure 4, except this time showing memory requirements. The average memory required by SPIN with options set for complete search was 117 Mb. Adding Lurch reduced the average memory from 117 to 25 Mb. Combining SPIN with default options and SPIN with options set for complete search reduced the average memory required from 117 to 45.7 Mb. And the three-way combination reduced the average memory requirement from 117 Mb to 15.6 Mb.

Having used Salsa to double check the results of Lurch and SPIN, we thought it might be interesting to try a combined verification strategy using all three tools. When Salsa reports that a property is true, it is definitely true, but a negative result from Salsa is inconclusive. So we tried first running error-seeded specifications through Salsa, then deleting from the model properties proven correct by Salsa, and then running Lurch and SPIN to check only the properties Salsa could not prove.

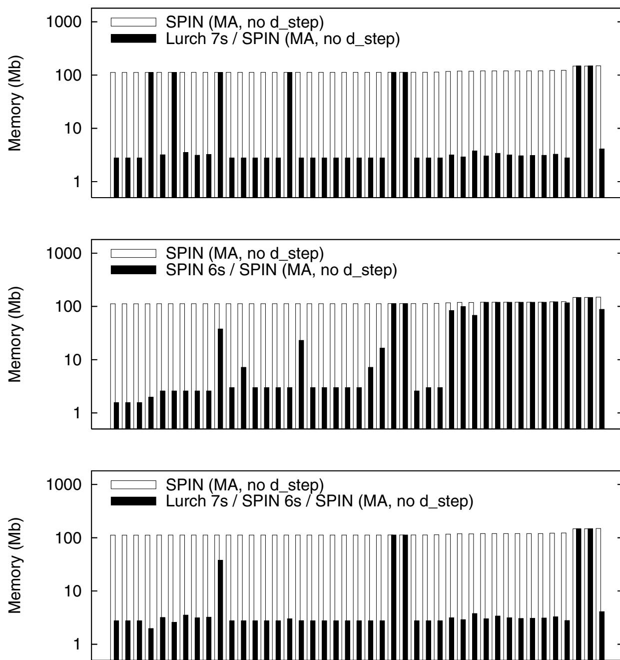

Figure 5. Memory (Mb) used for complete search alone vs. 3 combinations.

We found that adding Salsa in this way as a front end did not significantly change time requirements for Lurch or SPIN, but did reduce memory requirements for SPIN (but not Lurch) by nearly 15% in some cases. On the other hand, if for a particular specification Salsa had been able to prove all the properties true, running Lurch and SPIN would have been unnecessary. Since the SCR Toolset already provides a translator, not much effort is involved in adding Salsa as a front end, assuming that the analyst is familiar with all of these tools. So in general it would make sense to start with Salsa and then proceed with debugging and verification tools like Lurch and SPIN only if Salsa reported that some properties could not be proven.

4. Discussion

The idea of combining complementary strategies to solve problems is not revolutionary, but common sense. It has been suggested by others in the specific context of automated software testing (e.g., [5]). On the other hand, it can be difficult to apply formal verification, even with automated tools. Developers are understandably hesitant to devote the time and resources needed to effectively use a new tool, let alone several complementary tools together. In our experience with attempting to combine verification tools, we encountered several general challenges.

First, it is often difficult to translate a formal model from one tool's input language to that of another. This is not just because of different input syntax, but because the different algorithms used by the tools are amenable to different kinds of fine tuning and modeling tricks. And because of the state space explosion associated with automated verification, tools are designed so that users can "tweak" their models in many ways in order to limit time and memory requirements as much as possible. The use of Lurch in the experiments presented above was possible only because of the similarity between Lurch and SPIN's input languages and the availability of a translator from SCR to Promela. In our previous work we attempted to use Lurch as a front end to the symbolic model checker NuSMV [7] on specifications originally written in RSML [20]. The translation from RSML to Lurch was nontrivial and represented a major challenge [18].

In addition, where researchers have developed user-friendly environments for applying complex verification tools assumptions have been made about which settings best fit the needs of a typical user. Even if these assumptions are documented, a detailed knowledge of the tool and its input language may be necessary to understand their significance. In our case, a careful look at the auto-generated Promela model led to an understanding of one of the assumptions built into the SCR to Promela translator: it was assumed safe to mark portions of the model as deterministic steps, since any non-disjoint tables in the SCR specification would have been caught by the SCR Toolset's consistency checker.

Finally, as far as we know, very limited literature exists that would suggest exactly how complementary tools can be combined most effectively. We attempted to illustrate with one example model (and its error-seeded versions) that a combination between random simulation and model checking can save time and memory. We expect this would be especially true early on in the modeling process, when quick detection of errors (or incorrectly specified properties) is more important than full verification of correctness. Still, to reach a helpful general conclusion, we would need to repeat the experiment with many models in many domains, using a variety of verification tools.

5. Conclusion

It is hard to make a one-size-fits-all tool for software debugging and verification. Different tools make dif-

ferent assumptions and target different types of input models and properties to check for in order to run as efficiently as possible. An attempt to make one strategy robust, by removing all time- and memory-saving assumptions, will likely limit it to only the smallest models. But combining different strategies with complementary goals and assumptions may produce a robust but still efficient overall strategy.

In this paper we demonstrated how the simple random testing tool Lurch and the much more powerful model checker SPIN could be used together to more efficiently verify formal models. We found that a strategy combining the tools can in fact save time and memory. Perhaps more importantly, the process also forced us to look carefully at exactly how the tools were being used within the SCR Toolset and in our own experimental framework. We learned how SPIN could be used to fully verify our error-seeded SCR specifications, but only by using Lurch, the SCR Simulator and Salsa to better understand the specification and the results produced by SPIN.

We suggest conducting more comprehensive studies with a wide range of tools on a wide range of models, considering a wide range of possible error types. The motivation for this work would be, first, to provide guidance for effectively combining complementary strategies; and second, to provide a better understanding of the differences between tools and the assumptions involved when a particular tool is used in combination with others in a general verification framework.

Acknowledgments

We would like to thank Constance Heitmeyer from the Naval Research Laboratory (NRL), Washington DC, for providing invaluable input during the development of the PACS SCR specification and details about the SCR to Promela translation. We also thank Ramesh Bharadwaj from NRL for providing us with the Salsa tool together with details on how to apply it in our case study.

References

[1] Requirements Specification for Personnel Access Control System. National Security Agency, 2003.

[2] P. Anderson, T. Reps, and T. Teitelbaum. Design and Implementation of a Fine-Grained Software Inspection Tool. *IEEE Transactions on Software Engineering*, 29(8), 2003.

[3] J. Andrews, L. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *Proc. International Conference on Software Engineering*, 2005.

[4] M. Archer, C. Heitmeyer, and E. Riccobene. Proving Invariants of I/O Automata with TAME. *Automated Software Engineering*, 9(3), 2002.

[5] S. Berner, R. Weber, and R. Keller. Observations and Lessons Learned from Automated Testing. In *Proc. International Conference on Software Engineering*, 2005.

[6] R. Bharadwaj and S. Sims. Salsa: Combining Constraint Solvers with BDDs for Automatic Invariant Checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, 2000.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification*, 2002.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[9] J. Cobleigh, L. Clarke, and L. Osterweil. FLAVERS: a Finite-State Verification Technique for Software Systems. *IBM Systems Journal*, 41(1), 2002.

[10] P. Godefroid. Software Model Checking: the Verisoft Approach. *Formal Methods in System Design*, 26(2), 2005.

[11] A. Groce and W. Visser. Heuristic Model Checking for Java Programs. In *SPIN Workshop on Model Checking of Software*, 2002.

[12] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for Constructing Requirements Specifications: The SCR Toolset at the Age of Ten. *International Journal of Computer Systems Science and Engineering*, 20(1), 2005.

[13] G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[14] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[15] K. McMillan. The SMV Model Checker. `www-cad.eecs.berkeley.edu/~kenmcmil`.

[16] T. Menzies, D. Owen, and B. Cukic. Saturation Effects in Testing of Formal Models. In *Proc. 13th International Symposium on Software Reliability Engineering*, 2002.

[17] D. Owen and T. Menzies. Lurch: A Lightweight Alternative to Model Checking. In *Proc. 15th International Conference on Software Engineering and Knowledge Engineering*, 2003.

[18] D. Owen, T. Menzies, M. Heimdahl, and J. Gao. On the Advantages of Approximate vs. Complete Verification: Bigger Models, Faster, Less Memory, Usually Accurate. In *Proc. 28th IEEE / NASA Software Engineering Workshop*, 2003.

[19] S. Sims, R. Cleaveland, K. Butts, and S. Ranville. Automated Validation of Software Models. In *Proc. 16th International Conference on Automated Software Engineering*, 2001.

[20] M. Whalen. A Formal Semantics for RSML^{-e}. Master's thesis, Univerisy of Minnesota, 2000.