Increasing Confidence in Verification Results by Combining Diverse Tools: an Empirical Study

David Owen
Department of Mathematical Sciences
Messiah College, Grantham, PA
owen@messiah.edu

Bojan Cukic, Tim Menzies
Lane Department of Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV
cukic@csee.wvu.edu, tim@menzies.us

Abstract

Automated verification tools are capable of detecting subtle errors in models of complex software systems. Unfortunately, it can be difficult to use these tools effectively. Input models must correctly represent essential system behavior. Models must also enable efficient verification in terms of their time and memory requirements. Satisfying these two requirements can be challenging even for experts. But, given a correct model, can a user be confident that the verification results are correct?

One way to assess correctness of verification results is to provide translation tools from a modeling language to the input languages of diverse verification tools. Our experiments with automatic translators from the SCR modeling language to the input languages of various verification tools show that verification results produced from automatically translated models are not always consistent. In the tradition of fault identification and fault tolerance provided through diversity, we argue that various verification tools need to be used in concert. Our fault seeding experiments indicate that the results of different model verification tools can be compared, their discrepancies analyzed and modeling faults corrected. Our experiments further imply that "ensemble verification" should be a viable paradigm for the assessment of high assurance systems in the future.

1. Introduction

Increasingly complex software is being used in critical applications. Powerful techniques are available to verify the correctness of complex systems, but using these techniques effectively can be challenging, in terms of human expertise, and costly, in terms of computational resources [7, 12]. Tools have been developed to aid in constructing abstract models of software, but verification of these models continues to require time and memory resources rarely available in desktop computers. On the other hand, different strategies have been developed to decrease time and memory requirements of automated verification (see e.g., [5, 11, 13, 14]). It can be very challenging to develop software models that take full advantage of performance improving strategies made available by a particular tool and satisfy the assumptions needed for the state space size reduction. Previously, we demonstrated that combining two complementary verification strategies can improve verification accuracy and performance [17]. In those experiments we found inconsistent results between two verification tools, and the process of resolving the inconsistency brought to light hidden assumptions that would have undermined verification results. The experiments described in this paper extend that work, comparing results from five verification tools.

In addition to the inconsistent result reported in [17], we found two more cases of inconsistent results between different verification tools. First, we found inconsistent results between the Cadence SMV [14] and NuSMV [6] symbolic model checkers, resolved by modifying automatically generated input models before running NuSMV. Second, we found inconsistent results between the Salsa invariant checker [4] and the SPIN explicit-state model checker [12], resolved by observing that the model automatically generated for SPIN included some behavior not present in the model generated for Salsa. In each case where inconsistent results were found, there would have been no way to identify the fault in the model if that tool were used alone.

The technical contribution of this paper includes a set of mutation operators, based on the set of operators by Offut et.al. [16], adapted for specifications written in the SCR software modeling language [10]. We used this set of operators to create fault-seeded versions of an SCR specification in the experiments described below. Based on these tests, we comment on the relative merits of using one, or more than one, automated verification tools.

More generally, the results of this paper call for a change to generally accepted automated software verification methodologies. Our proposed approach is quite different than standard model-based verification methods. Others have considered complementary relationships between verification tools [8]. Smith [20] discussed the merits of SPIN vs SMV model checkers and commented that SMV's use of Binary Decision Diagrams (BDD) makes it the most suitable for hardware checking (where repeated structures in the design allow for very efficient BDD representations). These papers provide guidance on selecting the right tool for a user's particular need at a particular time in the software development life cycle. But these approaches assume that users can easily migrate between different verification tools and environments, something that has proven unrealistic for a vast majority of software engineering professionals.

This paper differs because we do not debate the value of different verification tools and/or modeling notations. We describe a practical way to make the most of available verification tools: rather than choosing one tool and attempting to use it optimally, choose several complementary tools and use them together in a way that exploits their different strengths. We show that an *ensemble* of verification tools is typically able to identify more modeling faults than each participating tool. Decades of research yielded numerous verification tools. Instead of debating the differences between these tools, *we should routinely use them all*, provided that the adequate model translation support exists.

The rest of the paper is organized as follows. Section 2 provides background information about verification tools used in our study. Section 3 describes the experimental setup. The results of our experiments are presented in Section 4. We summarize our findings in Section 5 and provide conclusions in Section 6.

2. Background

This section describes the modeling and verification tools that together make up the framework for the experiments described in the next section. First, we briefly describe existing tools which we used in our experiments: the SCR Toolset, the Cadence SMV and NuSMV symbolic model checkers, the SPIN explicit-state model checker, and the Salsa invariant checker. We next describe Lurch, our random search debugging tool for formal models.

2.1. The SCR Toolset

The SCR requirements specification language, a tabular notation for concise, unambiguous description of functional requirements, was developed by Heitmeyer and others over the last twenty years and has been used in a variety of research and industrial applications [10]. An SCR specification includes both *monitored* variables, which represent environmental quantities monitored by the system, and *controlled* variables, which 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:

$$@\mathrm{T}(c) \text{ WHEN } d \quad \stackrel{\text{def}}{=} \quad \neg c \wedge c' \wedge d$$

This event could be read: "c changes from false to true when d is true." The @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.

The SCR Toolset includes the following modeling and verification tools:

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 and SPIN model checkers.
6. Verifier: Automatic translation to TAME [3], a simplified theorem proving tool.
7. Property Checker: Automatic translation from SCR to Salsa, 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.

In addition to these tools, we wrote scripts to automatically translate from an SCR specification to the input language for Lurch, our random search debugging tool [18], described later. Through these scripts and the tools listed above, it is thus possible to automatically translate an SCR specification model into the input languages of all five of the testing and verification tools described below.

2.2. The Cadence SMV and NuSMV Symbolic Model Checkers

The Cadence SMV [14] and NuSMV [6] symbolic model checkers are two freely available contemporary versions of SMV, the "symbolic model verifier," a popular verification tool for formal models of synchronous hardware and software systems.

The input to a model checker is a description of a system of interacting finite-state machines and a set of temporal logic properties. The model checker builds and explores a global finite-state machine representing the overall system and verifies that the properties hold true for all possible paths through the system behavior. If a property is violated, the model checker outputs a counterexample: the full path from initial conditions to the property violation.

The fundamental challenge in model checking is to control the size of the global finite-state machine, so that the amount of time and memory required to fully explore it is not prohibitive. SMV uses a compact symbolic representation of the state space, based on binary decision diagrams (BDDs) to control the size of the global finite-state machine. This strategy affects what features and restrictions are included in the input language, what types of properties can be verified, and what kinds of input models the tools works best on. In general, symbolic model checking has worked well on models of synchronous systems, including computer hardware.

The input languages for Cadence SMV and NuSMV and basically the same. (However, as described below, we initially got inconsistent results between these two model checkers and found it was necessary to modify input models slightly before using NuSMV.) Because of differences between the SCR modeling language and the SMV language, the SCR Toolset's translator to SMV restricts the type of assertions to those involving only the current state of the system. For example, any assertion using the SCR *Next* ($'$) operator is removed from the model before translating to SMV.

2.3. The SPIN Explicit-State Model Checker

The SPIN explicit-state model checker is a widely used and freely available automated verification tool specifically designed to work well on models of asynchronous software systems [12]. Unlike Cadence SMV and NuSMV, which represent sets of states symbolically with BDDs, SPIN represents each global state explicitly in the global finite-state machine representing the behavior of the overall system. SPIN uses other techniques to control the size of the global finite-state machine, including partial order reduction, which decreases the number of states by ignoring redundant parallel paths irrelevant to the properties being verified. SPIN also includes a variety of options for decreasing the amount of memory required for each state stored.

Unlike the SCR modeling language, in which state transitions may be triggered by change events based on the current state and previous state of the system, in Promela, the SPIN input language, state transitions are based only on the current state of the system. Rather than removing behavior from the SCR model before translation to Promela (as is done before translation to SMV), the SCR Toolset's Promela translator makes two copies of every variable in the specification, one for the previous state and one for the current state of the system. Change events (and assertions involving both the previous and current state) can thus be included in the Promela version of the specification. This makes SPIN's verification result more comprehensive, since the input model is closer to the original SCR model, but it also makes the verification run require much more time and memory, compared to verification with Cadence SMV or NuSMV. Some of the performance difference may also be due to the fact that SCR is a synchronous modeling language, and SPIN has been designed for asynchronous models, unlike Cadence SMV and NuSMV, which are designed for synchronous models.

2.4. The 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 form of the SCR specification language [4, 19]. Like an automated theorem proving tool, Salsa attempts to carry out an inductive proof using decision procedures. Like a symbolic model checker, Salsa uses BDDs to represent the global system in a very compact way.

Salsa either determines that a property is true or outputs a two-state counterexample. This is different from the counterexample produced by a model checker, which would include all states along a path from initial conditions to the property violation. In some cases Salsa is unable to prove properties that are actually true, so the user must determine whether counterexamples produced by Salsa are valid; that is, whether the first state in the counter example is reachable from the system's initial state.

2.5. The Lurch Random Search Tool

Lurch, our random search tool for detecting property violations in formal models, explores a sample of paths through the global finite-state machine, choosing randomly when more than one branch is possible [18]. A run ends when Lurch reaches a property violation, the end of a path, or a user-specified depth search limit. Runs are repeated until a user-specified number of paths have been explored, or until *saturation* is achieved; that is, if the percentage of unique global states explored, compared to the total number of global states explored, drops below a user-specified threshold, the search is stopped [15].

Lurch was originally designed to run on asynchronous models, in which the individual finite-state machines in the input model all run in parallel. The experiments below use functionality added later to support synchronous, hierarchical finite-state machine models, in which the individual machines may each execute a tansition at each execution step, but within each step machines are ranked in a dependency order. This is necessary to support models translated from SCR, a synchronous, hierarchical modeling language.

Lurch's input language is similar to Promela, the input language for SPIN, allowing state transitions and assertions to be based only on the current state of the system. Because of this, scripts that translate from SCR to Lurch actually translate from the SCR Toolset's Promela model, generated for SPIN, to Lurch, rather than directly from SCR to Lurch [17]. This makes the Lurch version of an SCR specification larger (like the SPIN version), but does not have much impact of Lurch's performance, since Lurch does a limited number of random runs through the model.

3. Experimental Setup

This section begins with a description of the software model used in our experiments, an SCR specification of a security system. Next, we summarize our process for creating fault-seeded versions of the specification, which we used to evaluate the testing and verification tools listed in the previous section. Finally, for each of the tools, we describe how it was used and give a brief summary of results.

3.1. The 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 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 [9]. This specificaction is large enough that versions of it produced by the SCR Toolset's automatic translators require significant time and memory to fully verify. For example, with default options full verification with SPIN would have required 6.8GB and over 24 hours; with various compression options enabled this could be reduced to 512MB and approximately 30 minutes on a 2.5 GHz desktop machine.

Label	Description	Example
AOR	Arithmetic Operator Repl.	`+` → `-`
CRP	Constant Repl.	`1` → `2`
EVR	Enumerated Type Val. Repl.	`a` → `b` (where `a` and `b` are possible values for enumerated type)
IOR	Implication Operator Repl.	`=>` → `<=>`
LCR	Logical Connector Repl.	`AND` → `OR`
ROR	Relational Operator Repl.	`<` → `<=`
SND	SCR *Next* Operator Del.	`'` →
SOR	SCR Event Operator Repl.	`@T` → `@F`
UOD	Unary Operator Del.	`NOT` →
VRP	Variable (same type) Repl.	`x` → `y` (where `x` and `y` are variables of same type)

Figure 2. Mutation operators used to generated fault-seeded versions of the PACS SCR specification.

Figure 1 shows a finite-state machine representing the core mode logic SCR model of the PACS requirements. Initially, the system mode is *EnterCard*; when a card is entered the mode changes to *CheckCard*. If the card is not valid, a limited number of retries is 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 *Proceed* mode, 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* a security 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 security officer in any mode (except *EnterCard*), the mode is reset to *EnterCard* (note the dotted line transition in the upper right part of Figure 1).

3.2. Fault-Seeded Specifications

Figure 2 shows the set of mutation operators used to create fault-seeded versions of the PACS SCR specification for

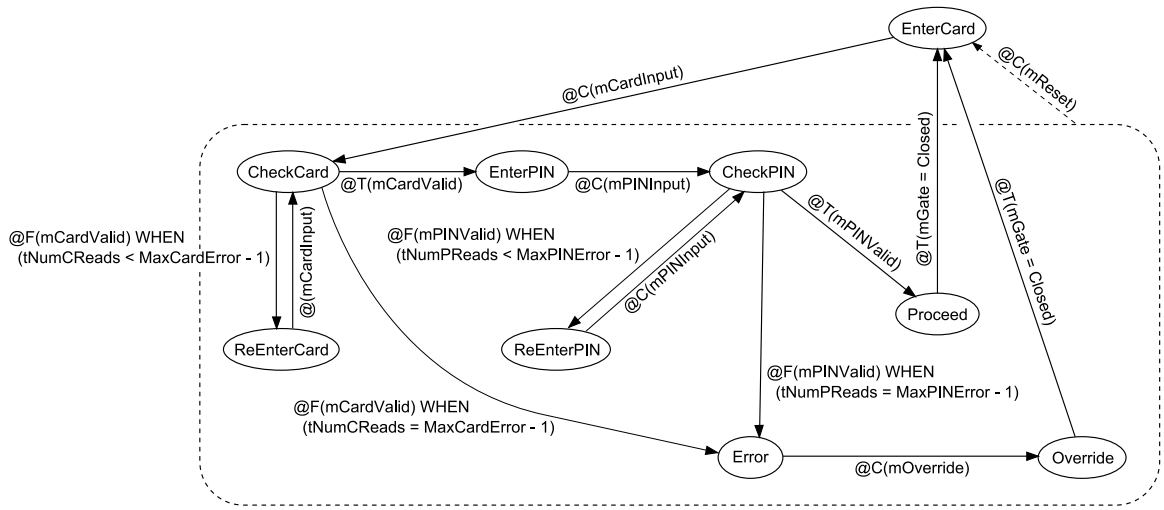

Figure 1. PACS mode finite-state machine.

use in our experiments. A preliminary set of 45 fault-seeded specifications were prepared manually, and later 278 specifications were generated automatically using these operators.[1]. This set of operators is based on the set argued to be sufficient for Fortran programs [16]. Operators AOR, LCR and ROR are taken directly from [16]. UOD is similar to the unary insertion (UOI) operator from [16] but is easier to implement without a complete parse of the input specification.

The set of five sufficient operators in [16] also includes an absolute value insertion (ABS) operator, which replaces an entire arithmetic expression by 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 arithmentic expressions, ABS was not used. Instead we introduced CRP (considered in [16] but not one of the five selected) and EVR, which replace integers or, for variables of enumerated type, other legal values. IOR and SND are similar to LCR and UOD, but deal with SCR-specific features. Andrews et.al. [2] use a similar set of mutation operators with C programs, to accurately evaluate test suites. These mutation operators produce fault-seeded C programs realistic enough that a given set of tests will achieve approximately the same level of program coverage that the given test set achieves for programs with real faults.

Figure 3 shows how many fault-seeded specifications were generated with each mutation operator. For specifications generated using two mutation operators, we indicate both operators. Different number of specifications were cre-

[1]Manually generated specifications were included, since 1) manual fault seeding was not based on any expert knowledge of the PACS, and 2) average performance of verification tools varied little between the sets of manually seeded and automatically seeded specifications.

AOR	(9)	CRP	(9)	EVR	(58)
AOR CRP	(1)	CRP EVR	(1)	EVR-s	(4)
AOR EVR	(6)	CRP LCR	(1)	EVR EVR	(10)
AOR LCR	(1)	CRP ROR	(2)	EVR EVR-s	(1)
AOR SOR	(2)	CRP VRP	(1)	EVR IOR	(3)
				EVR LCR	(1)
				EVR ROR	(11)
				EVR SND	(2)
				EVR SOR	(9)
				EVR VRP	(12)
IOR	(16)	LCR	(20)	ROR	(31)
IOR ROR	(1)	LCR LCR	(3)	ROR ROR	(7)
IOR SOR	(2)	LCR ROR	(5)	ROR SND	(1)
IOR VRP	(2)	LCR SOR	(3)	ROR SOR	(4)
		LCR VRP	(6)	ROR VRP	(7)
SND	(5)	SOR	(30)	VRP	(23)
SND SOR	(1)	SOR SOR	(1)	VRP VRP	(1)
SND VRP	(1)	SOR VRP	(1)		

Figure 3. Mutation operator(s) and the number of specifications generated for each (pair).

ated for different operators because operators apply to different locations 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.

3.3. Use of the SCR Toolset

Each fault-seeded specification was checked first using the command-line program *testtool* and GUI consistency checker provided with the SCR Toolset. These tools check for generic errors including syntax errors, circular definitions, and violations of disjointness or coverage properties. Disjointness violations occur when, from a certain system state, for a given input, more than one transition is possible. Coverage violations occur when, from a certain state of the system, for a given input, no next state is specified.

If, for a particular specification, errors were detected by these basic SCR Toolset checks, we excluded that specification from further experiments. Because our focus is on the accuracy of back-end verification tools, eliminating models where basic SCR Toolset checks detected faults was appropriate (in practice, these models would never be passed to back end verification tools). If the specification contained no faults detected by the SCR Toolset, we created Salsa, SMV and SPIN versions of the specification using the automatic translators included in the SCR Toolset. We developed original scripts to generate a NuSMV version from the SCR Toolset's SMV version (which worked as-is for Cadence SMV but not for NuSMV) and a Lurch version from the SCR Toolset's SPIN version. All together, this process generated 323 fault seeded specification instances.

3.4. Use of The Salsa Invariant Checker

For specifications which passed basic SCR Toolset checks, we the invariant checker on the SCR Toolset's automatically generated Salsa version of the specification. By comparing Salsa's results on each fault-seeded specification to results on the original (correct) specification, fault-seeded specifications were divided into five categories:

1. Salsa proved *fewer* assertions for the fault-seeded specification than for the original (94 fault-seeded specification instances in this category).
2. Salsa proved *more* assertions for the fault-seeded specification than for the original (16).
3. Salsa proved the same number of assertions but *fewer* generic properties than for the original (36).
4. Salsa proved the same number of assertions but *more* generic properties than for the original (7).
5. Salsa results matched results for the original (170).

Salsa alone can not be used to produce complete verification results since assertions not proved true by Salsa may or may not be false. But, if a properties is proved true by Salsa, in principle, it can be removed from the specification before running a model checker. In some cases verifying a lower number of properties can *greatly improve the performance of the model checking tools*. This is a great example of a complementary nature of some verification tools. However, as described later, we witnessed examples where SPIN reported assertion violations in the Promela version of the model which were not present in the original SCR specification. In the spirit of diversity, Salsa might be used in concert with a model checking tool to check for spurious assertion violation reports.

3.5. Use of the Cadence SMV and NuSMV Symbolic Model Checkers

Cadence SMV and NuSMV were next run on the SMV version of each fault-seeded specification. With some minor but important adjustments in the SMV's input model to make it compatible with NuSMV (described later), Cadence SMV and NuSMV results were consistent. Assertion violations were detected in 141 specifications; no violations were detected in 182 specifications. As stated above, the SCR Toolset's translator to SMV restricts the type of assertions it translates to only those that involve the current state of the system. For our experiments this meant that only 9 of the 16 assertions in the original SCR specification could have been checked by Cadence SMV and NuSMV. In cases like this, Cadence SMV or NuSMV can not be used alone to produce complete verification results.

3.6. Use of the Lurch Random Search Tool

Next, we ran Lurch on versions of the fault-seeded specifications generated from Promela versions produced by the SCR Toolset. Lurch's random search does not necessarily alert the user to property violation every time because it randomly searches through the state space model. In some runs it may not "stumble" into a property violation. For this

10 violations in under 50 runs		Less than 10 violations in 50 runs	
10 / 10	(175)	0 / 50	(117)
10 / 11	(16)	1 / 50	(2)
10 / 12	(5)		
10 / 13	(4)		
10 / 17	(1)		
10 / 27	(1)		
10 / 38	(1)		
10 / 39	(1)		

Table 1. Lurch results on fault-seeded PACS specifications: number of times violations detected vs. search runs, number of specifications in parentheses.

reason, 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. Although it is sufficient that Lurch detects property violation only once, we wanted to analyze the consistency of our "random testing approach" to verification of formal models. If Lurch found a violation ten times in less than 50 runs, we stopped running Lurch on that input model. As shown in table 1, for most 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) when a violation was detected did Lurch detest the property violation in less than 75% of runs.

Because input models used with Lurch were based on Promela versions of the specifications produced by the SCR Toolset, 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, an incomplete random search tool, detected a larger number of property violations than the complete verification tools Cadence SMV or NuSMV.

3.7. Use of the SPIN Explicit-State Model Checker

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 where the default search depth limit is set to 10,000.
2. Second, we run SPIN with settings necessary to get complete verification runs. Promela models were compiled with minimized automaton memory compression and run with the depth limit set to 2,000,00.
3. Third, we run the input model with final `d_step` marker removed. This ensures that nondeterministic SCR tables are handled properly, and with settings necessary for complete verifications runs on all models. We compiled models with minimized automaton memory compression and run with depth limit of 3,200,000.

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. As explained in [18] and briefly below, 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 `d_step` marker from the model. Running SPIN on input models with this change, with the depth settings adjusted to enable a complete verification run, SPIN was able to detect property violations in four more models. As described below, two of these were disconfirmed by Salsa. All in all, SPIN detected assertion violations in 233 of 323 input models.

SPIN requires much more time and memory, in most cases, then the other tools described above. But in our experimental framework only SPIN could be used to fully verify all 16 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 the behavior identical to the original, as far as the given assertions are concerned.

4. Results

This section describes three cases of inconsistent results between verification tools and discusses how each was resolved. We then give an overview of experimental results, comparing the accuracy of the results from different verification tools used in our experiments. Note that we are not so much comparing the accuracy of the verification tools in isolation, but the accuracy of the overall system verification results. System verification accuracy depends on factors which include the automatic translation tools from SCR to input models of the verification tools and the configuration settings of the tools chosen to minimize time and memory requirements.

4.1. Inconsistent Results Between Cadence SMV and NuSMV

We mentioned in Section 3.5 that achieving consistent verification results between Cadence SMV and NuSMV model checkers required "minor adjustments". We found that consistent verification between Cadence SMV and NuSMV model checkers required a modification in the SCR Toolset's SMV model translation utility. Prior to

```
Copyright 1996 Cadence Berkeley Labs. Cadence Design Systems...

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 4. Inconsistent results from Cadence SMV (top) and NuSMV (bottom) running on the same input model.

implementing the change, Figure 4 shows the conflicting output produced by the two model checkers for one of the fault-seeded specifications.[2] Cadence SMV and NuSMV disagreed about whether one of the assertions included in the input model is true or false—the assertion `(cGuardAlarm = On) <=> (cUserDisplay = SeeOfficer)`. Needless to say, this result was unexpected and its resolution implied weeks long human analysis effort.

The fault-seeded specification in question contained two *mutations*, 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 `(cGuardAlarm = On) <=> (cUserDisplay = SeeOfficer)` was false. Thus we concluded that the assertion violation reported by Cadence SMV was likely present in the input model, but somehow masked by the first mutation for NuSMV.

To confirm the Cadence SMV result, SPIN was run on an input model generated from the fault-seeded specification for which Cadence SMV and NuSMV produced inconsistent results. Figure 5 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 specific input model. Although the input model was syntactically valid for NuSMV, the keyword `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 `SPEC` with a NuSMV-specific key word, `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 `(cGuardAlarm = On) <=> (cUserDisplay = SeeOfficer)`.

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. Needless to say, it came as a surprise that the two versions of SMV could interpret the same model differently. Further, from the point of view of the user, the cause of the inconsistency is not relevant. The issue is that a verification tool, expected to provide a 100% confidence in the verification result, reported that an incorrect input model was correct. It is only because this output was compared to that of other verification tools that was it shown to be incorrect and eventually corrected.

4.2. Inconsistent Results Between Salsa and SPIN

Figure 6 shows inconsistent results from Salsa and SPIN running on another fault-seeded version of the input model. Salsa reports that the property `PINEntry` is true (top of Figure 6) but SPIN reports a violation of the assertion corresponding to the property. Salsa is capable of proving properties true; however, if a property cannot be proven true by Salsa it is not necessary 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 proven true by Salsa*. Again, the surprising inconsistency implied considerable human analysis.

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

[2] For clarity many lines of output have been deleted in this figure and similar figures below.

```
Depth= 500129 States= 1e+06 Transitions= 1.02631e+06 Memory= 72.780...

pan: assertion violated (( !((cGuardAlarm_NEW==0))||(cUserDisplay_NEW==9))
  &&( !((cUserDisplay_NEW==9))||(cGuardAlarm_NEW==0))) (at depth 859760)...

(Spin Version 4.2.4 -- 14 February 2005)...

State-vector 32 byte, depth reached 859769, errors: 1
```

Figure 5. 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 4.

```
Analyzing SAL specification in file: utpb28.ssl.sal.
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
pan: 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 6. Inconsistent outputs from Salsa (top) and SPIN (bottom) running on the same input model.

version of the specification model. SCR allows the use of NATURE constraints to limit the behavior of variables representing inputs from the environment. In this case one of the NATURE constraints is necessary in the model for the property PINEntry to be true. Thus Salsa, running on an input model including the relevant NATURE constraint, found that the property PINEntry was true. But SPIN, running on a model without the constraint (because the translator ignores it), 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 the input model without the NATURE constraint, we found that Salsa could no longer prove the property true.

This inconsistency is perhaps less critical than the one described in previous section, just because there was no possibility of missing a genuine fault. The analysis of the fault detected by SPIN would indicate that it is a benign one. But this example does show a practical benefit of combining an ensemble of complementary verification tools. If only SPIN was used, much manual effort might be expended attempting to find and correct the input model so that the property PINEntry would not be violated. Using Salsa makes it easier 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 other model checker. The fault may be related to a mistake in the portion of the machine generated input model representing the environment rather than the critical system to be verified.

4.3. Inconsistent Results Between Lurch and SPIN

Figure 7 shows outputs from SPIN and Lurch running on input models generated from a different fault-seeded version of the specification. 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. We would never expect the result shown in Figure 7—*Lurch, an incomplete tool, reports a violation, while SPIN runs to completion but reports no violation*.

As described in [17], we eventually determined that the output from Lurch was correct and that the inconsistency between SPIN and Lurch's results was due to a performance improving feature of SPIN included in the Promela model generated by the SCR Toolset's translator. Promela 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. If a block that is not deterministic is enclosed within d_step, this results in some of the possible interleaving behaviors of the input model being ignored. For the fault-seeded specification in this example, that ignored behavior 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

```
Depth= 500129 States= 1e+06 Transitions= 1.02631e+06 Nodes= 19616 Memory= 144.710...
(Spin Version 4.2.4 -- 14 February 2005)...
State-vector 32 byte, depth reached 1714629, errors: 0
```

```
time  memory   states  sts/sec  % new   col  depth    name...
9.08   7.55   1.2e+05  1.3e+04   49.0     0    155    _assert6_violated
```

Figure 7. Inconsistent outputs from SPIN (top) and Lurch (bottom) running on the same input model.

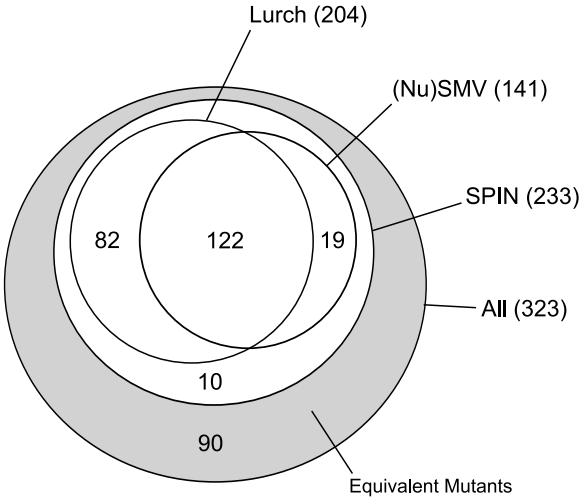

Figure 8. Sets of specifications for which various tools detected assertion violations.

detected only by Lurch.

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. Using Lurch as well, we were able to uncover the assumption and better understand how to use SPIN to get accurate verification results. Note that *the use of* `d_step` *should not be viewed as an error in the translator*. In practice, the SCR Toolset can usually be used to prove disjointness before running SPIN. However, for specifications the SCR Toolset is unable to prove disjoint, *the user is apparently expected to know* that the final `d_step` marker should be removed from the automatically generated Promela version of the specification before running SPIN. How many users are actually aware of this assumption is the question of grave concern.

5. Discussion: Comparing the Results

Figure 8 shows the division of fault-seeded specifications used in our experiments into sets based on which verification tools detected their property violations. Since Salsa was used to prove properties true but not to detect violations, there is no straightforward way to include Salsa results in this diagram. Regardless of the omission of Salsa results, the diagram is instructive. Lurch detected property violations in 204 specifications. After the lesson was learned about the discrepancies between Cadence SMV and NuSMV, they both detected violations in exactly the same set of 141 specifications. SPIN detected violations in 233 specifications. SPIN actually detected violations in 235 specifications, but Salsa proved that 2 of those were false alarms. We note that confirming this result required the domain knowledge or the presence of a domain engineer. In absence of any additional evidence to the contrary, we concluded that the remaining 90 specifications, out of a total of 323, were the mutants logically equivalent to the correct seed specification with respect to the properties checked by the verification tools.

Figure 8 shows that SPIN detected a superset of all property violations detected by any tool. So why not just use SPIN to verify SCR specifications? Firstly, only 205 incorrect models were identified by SPIN when it ran with the default settings, which limit the depth of search to $10,000$ (see Section 3.7). 26 additional models were correctly detected when a complete search was allowed. As expected, the time and memory requirements of complete search can surpass those needed for an optimized search by a factor of a 100 or more. The cost of computational efficiency is limited accuracy. Needless to say, the user does not know when a complete search is needed instead of an optimized one. Users could chose to run complete SPIN searches on all 323 models. With some scripting, this conservative approach would require almost to a year of computing time on the desktop used in this study. A better approach would be to run a complete search only if the optimized one does not return any property violations. Apparently, this approach would save user more than 200 complete search runs of SPIN. Whichever approach a user might chose, two faulty specification models would still remain hidden, as they required human intervention in the automatically generated Promela model (removing the `d_step` marker).

As stated above, learning how to use SPIN effectively was greatly enhanced by our choice to use Lurch, Salsa and SMV. We were able to compare the results from these tools with results from SPIN and eliminate false positive property violations. In addition, our experiments indicated that Ca-

dence SMV and NuSMV required far less time and memory than SPIN running on input models produced from the same fault-seeded specification. Thus it would make sense to use Salsa, Cadence SMV or NuSMV first. If we trusted the results from these tools, we could remove properties proved true by these tools before running SPIN. In our experiments, doing so reduced SPIN's time and memory requirements by approximately half. However, given our findings, unless the system verification engineer is truly an expert with abundance of experience, we would not recommend property elimination as potentially fatal modeling faults may be discovered by an ensemble of verification tools, rather than individual ones.

6. Summary

In this paper, we argue that the interaction between different software and system verification tools can be more informative than the information from each tool, used in isolation. Data and tool diversity have proven to be viable methods for achieving dependability. Our work demonstrates that tool diversity is valuable when it comes to system verification too.

We experimented with an SCR specification model verified by 4 alternative freely available verification tools: Lurch, Salsa, (Nu)SMV, and SPIN. In spite of decades of research, formal system modeling frameworks which allow users to easily utilize these tools are rare. We relied on SCR toolset and its translators from SCR tabular notation to other modeling languages. We applaud its developers for providing automated translators for many verification tools. It would be easy to blame most of the verification result discrepancies on the immaturity of SCR's translation utilities. But our opinion is just the opposite: SCR's open design allows ensemble verification in practice. Therefore, even if its translators are imperfect, the ability to cross compare verification results inspires confidence in the achieved level of system assurance. For us, it was surprisingly simple to study these different verification perspectives. Given the well known difficulties in developing formal models and selecting appropriate state invariants for verification, our results suggest that future formal modeling environments will have to support translation to diverse back end verification tools. While we cannot report any inconsistency related to well known verification tools, we believe that their default parameters which trade off correctness for performance are potentially hazardous. Such assumptions must be disclosed to the users of verification tools more effectively and the default configuration of model checkers should be the complete search.

References

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

[2] J. Andrews, L. Briand, Y. Labiche, and A. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering*, 32(8), 2006.

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

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

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking Without BDDs. *Lecture Notes in Computer Science*, 1579, 1999.

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV2: An Open-Source Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification*, 2004.

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

[8] J. Cobleigh, L. Clarke, and L. Osterweil. The Right Algorithm at the Right Time: Comparing Data Flow Analysis Algorithms for Finite-State Verification. In *Proc. International Conference on Software Engineering*, 2001.

[9] D. Desovski. *A Component-Based Approach to Verification and Validation of Formal Software Models*. PhD thesis, West Virginia University, 2006.

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

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

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

[13] G. Holzmann and R. Joshi. Model-Driven Software Verification. In *Proc. International SPIN Workshop on Model Checking of Software*, 2004.

[14] K. McMillan. The SMV System, 2000. Available at `www.kenmcmil.com/tutorial.ps`.

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

[16] A. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions of Software Engineering Methodology*, 5(2), 1996.

[17] D. Owen, D. Desovski, and B. Cukic. Effectively Combining Software Verification Strategies: Understanding Different Assumptions. In *Proc. International Symposium on Software Reliability Engineering*, 2006.

[18] D. Owen and T. Menzies. Lurch: a Lightweight Alternative to Model Checking. In *Proc. International Conference of Software Engineering and Knowledge Engineering*, 2003.

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

[20] M. Smith. Verifying Autonomous Planning Systems. In *Proc. NASA Software Assurance Symposium*, 2006.