Combining Diverse Translation and Verification Tools to Detect Faults in SCR Specifications

David Owen, Bojan Cukic, *Member, IEEE,* and Tim Menzies, *Member, IEEE*

Abstract

It can be difficult to determine which verification strategy is best for a particular software system. Researchers have observed complementary relationships between verification tools and argued that there is no single best verification tool: as users' needs change, the choice of tool should change as well. We provide further evidence of complementary relationships between verification strategies, specifically considering tools for automatically translating from the Software Cost Reduction (SCR) modeling language to several different verification and debugging tools for formal models. We show how verification strategies—each with its own formal modeling languages, automatic translator and verification tool—may be considered complementary in terms of both accuracy and scalability. Rather than providing guidance for users deciding between strategies, we argue that a verification strategy combining results from multiple tools will yield the most accurate results, i.e., the results worthy of the greatest trust.

Index Terms

Software verification, model checking, mutation of specifications, automatic translation, fault detection, scalability of verification and debugging tools.

I. Introduction

There are a variety of ways to improve the scalability of automated verification tools. Some verification strategies limit the scope of verification to improve scalability. For example, the scope may be limited by restricting the types of input models or properties that can be verified. Or the results of verification—properties proved or errors detected—may have to be validated manually or with another tool.

Diverse verification strategies, each attempting to improve scalability in different ways, tend to be complementary. In the experiments below, for example, verification of a given input model may require require much more or much less time and memory, depending on the model translation and verification tools used. In addition, because different translation tools implelement different scalability-improving assumptions, property violations present in a model may be missed by one verification strategy but caught by another. And even when two different strategies catch the same property violation, one strategy may be much faster, require much less memory, or produce simpler and more useful information for correcting the input model.

Cobleigh et.al. recognize complementary relationships between modeling languages, translation and verification tools and recommend the use of a verification framework in which a variety of strategies are available, so that, as software models and users' needs change, the right strategy is available [1]. In this article we consider the basic verification challenge, to determine whether a software model is consistent with a formal specification of correctness properties, and argue for the use of several verification strategies—each with its own automatic translation and verification tools—together.

Using multiple complementary translation and verification tools together on the same input model yields two types of advantages. First, hidden assumptions and idiosyncrasies of the tools are brought to light, so that individual tools may be used more effectively and the user has reason for increased confidence in the results. This is especially important when tools are used in a verification framework that includes elements not developed by the user, e.g., automatic translators or modeling tools, as in the experiments we present below. If automated verification tools are to be practical and cost-effective, expert knowledge of the inner workings of a tool must not be a prerequisite for use. If multiple tools, representing diverse modeling languages, translation assumptions, and verification algorithms, can be run on the same input to produce consistent results, the user can be much more confident in the results' correctness.

A second advantage of multiple-tool verification strategies is improved scalability. Tools may be cascaded in such a way that input models difficult for one tool are passed on to another. If tools' performance is sufficiently complementary, most imput models will be easy for at least one tool even if they are difficult for one or more other tools. Cascading multiple tools may result in an overall verification strategy much less sensitive, in terms of time and memory requirements, to minor changes in the input model [2].

In previous work we experimented with a simple random search to detect errors in software models [3]. Our strategy was as follows: start at the initial state of the model; choose the next state at random from those possible; quit when no next state is possible or a user-specified depth limit is reached. Search results followed a pattern: at first, many unique states were explored, but soon the number of unique states explored reached a peak and stopped increasing; from then on the same states were explored over and over. When allowed to run to this *saturation* point, at which the proportion of unique states drops off, the random search produces surprisingly consistent results. We concluded that random search, in spite of its general (worst-case) unreliability, was sufficiently consistent to pursue as an efficient strategy for detecting errors in software models [4]–[6].

Further work on random search led to the development of Lurch, a tool for detecting violations of generic and user-defined logical properties in finite-state machine models [7]. We compared Lurch's performance to the model checkers SPIN [8] and Cadence SMV [9] and found that with random search it is often possible to detect property violations far more quickly and with orders of magnitude less memory. We also showed how Lurch could be used together with a conventional verification tool to greatly improve average verification performance: use Lurch first for a relatively short time and only run the verification tool on input models for which Lurch finds no property violations [10].

In addition, random search can be used as a *sanity check* on verification results produced by a model checker: to improve performance in real-world applications of model checking, the input model may be simplified, based on assumptions about its structure. If the model checker detects no faults in this simplified model, random search can be run on an un-simplified version of the model. Assuming the random search also detects no faults, we can be more confident in the simplifying assumptions used to make the input model small enough for verification with the model checker [11].

In the experiments below we consider complementary relationships between diverse verification strategies including random search, different types of model checking, and a specialized tool for proving invariant properties. We found complementary relationships between these different strategies, especially between the translation tools used to generate the different input models needed for each strategy. Our primary contribution is to show how complementary strategies like those we considered can be used together in a single robust verification strategy. Complementary relationships considered include both complementary scope—tools performing different types of analysis—and complementary performance—execution time and memory consumption.

In this article, we propose and evaluate a multiple-tool verification strategy for software system and property specifications written in the Software Cost Reduction (SCR) modeling language [12]. In addition, we suggest the following more general claim: diverse verification strategies, each attempting to improve scalability in different ways, may be integrated to produce a single strategy that is both more scalable and more reliable. Improved reliability is the result of insight into the use of each tool gained by comparing results from different tools.

II. Related Work

This section provides a brief overview of the research behind the tools used in the experiments described later: symbolic and explicit-state model checking tools, increasingly powerful testing tools influenced by ideas from model checking research, and random search as a way to efficiently detect faults in formal software models.

A. Model Checking

Model checking is probably the most widely used automated verification technique. Model checking tools carry out an exhaustive exploration of the behavior represented by an abstract program model to check for consistency with a specification of desired properties [13]. Model checking has been effective in many domains including computer hardware design, networking, security and telecommunications protocols, and automated control systems [14]–[16]. Model checking has been used in safety critical NASA projects [17]–[19]. Microsoft research has developed a proprietary model checking framework for use on critical components of Windows [20].

The input to a model checking tool is the specification of a finite-state concurrent system. In order to verify that the properties hold, the model checker must construct a single composite finite-state machine to represent all possible behaviors of the individual concurrent machines in the model as they interact with each other. In practice this composite finite-state machine may be very large. This is the *state-space explosion* referred to in the literature: if there are many concurrent machines in the input model, making many transitions in parallel, the number of global states in the composite machine may grow exponentially, compared to the number of concurrent machines in the original model [15].

Model checking techniques originated in the 1980's. In the early 1990's Clarke and colleagues began using binary-decision diagrams, or BDDs, to succinctly represent the global system [15]. This new *symbolic model checking* technique, implemented in a tool called SMV (the Symbolic Model Verifier), made it possible to verify much larger input models. In the experiments described below we use two popular versions of SMV: Cadence SMV [9] and NuSMV [16].

Symbolic model checking works well with models representing synchronous systems, including integrated circuit designs, which tend to have many small, symmetrical components. But software systems are often asynchronous. In an asynchronous system several things may be going on in parallel with no synchronization point and different interleavings are possible. If all possible interleavings must be checked, the state space required tends to grow very large.

Unlike SMV, the SPIN model checker is designed specifically for asynchronous software models [14]. To handle models with many possible interleavings of parallel behaviors, SPIN uses *partial order reduction*: only interleavings relevant to the property specification are checked. That is, if the specified properties are unaffected by the order of some set of events, only one possible ordering of those events will be checked [14], [15]. SPIN has been used to verify a wide range of algorithms, protocols, and system implementations [8], [14], [21].

Partial order reduction decreases the number of states that must be explored to verify the input model. To decrease the amount of memory required for each state, SPIN offers options for lossless or lossy compression. The lossless methods save memory, but tend to require a lot of time [8]. For example, in the experiments presented in Section III-B, without compression SPIN would have required 6.8 Gb of memory for the verification run (this statistic is provided by SPIN when used with the compression option). With compression, the run required only about 270 Mb, but took 30 minutes on a computer with a 2.5GHz processor.

SPIN's lossy compression options run quickly and scale to large systems, but sacrifice completeness; that is, there is the possibility of missing property violations present in the system. In the current version of SPIN these are the hash compaction and bitstate hashing options [8]. In the past, "scatter search," an incomplete random search technique that limited which states were explored rather than limiting the amount of information stored with each state, was added to SPIN [22]. Although this idea was eventually given up, results from that work suggest that if there is a fault in the model, it is likely to affect a large portion of the state space. This idea is an important assumption in our work on random search as a scalable alternative to model checking [3].

SMV also implements incomplete but scalable search options. Success in the development of algorithms for solving satisfiability (SAT) queries has enabled the development of a symbolic search technique known as bounded model checking [23]. To perform bounded model checking, a SAT query is used to represent the state space up to a user-specified depth, and a SAT solving algorithm determines whether the query is satisfiable, which corresponds to determining whether the input model is consistent with the property specification. Bounded model checkers are very effective; nevertheless, bounded model checking is incomplete and can miss faults because of the search depth restriction. If no property violation is found, we only know that there is no violation within the user-specified search depth.

The use of incomplete techniques is controversial, since the primary goal of model checking is verification, not debugging. But in practice, even when complete verification is not possible model checkers have proven useful. For example, they can automatically detect complex errors in systems too large for complete verification [1], [24], find counter examples to aid in fixing known errors [25], and be used to automatically generate test cases [26], [27].

Incomplete strategies that nevertheless provide some of the benefits of model checking can be thought of as part of a growing set of testing tools with capability inspired by model checking. Much work is being done on testing strategies that are increasingly automated and capable of detecting more complex types of faults. For example, tools running directly on source code and large production models can detect classes of faults previously in the realm of formal verification [24], [28]–[30].

It is difficult to compare techniques' scalability and scope, because different approaches work well with different types of models, and some researchers advocate a framework in which several complementary approaches are available [1], [24], [31]. For example, an explicit-state model checker (SPIN) might quickly find many long counter examples, while a symbolic model checker (SMV) might require more time and memory but find much shorter counter examples [1], [32]. So it may depend on whether a verification practitioner requires, e.g., fault detection alone versus short counter examples to facilitate debugging. Related to this is the sensitivity of a testing strategy to minor changes in a model. For a given testing strategy, a small change in the input can make a significant difference in the time and memory required to detect a fault.

B. Random Search Applied to Formal Models

Almost twenty years ago West explored the idea of using a simple incomplete technique, random search, to detect faults in finite-state models of software systems [33]. Random search, although incomplete, was shown to be surprisingly quick and effective. West's explanation of the success of random search is helpful in understanding the success of various heuristics and incomplete verification strategies available today. He noted that faults detected in concurrent systems are often much less complex than the overall system [33]. That is, a fault involving a small subset of processes is present in many global system states—processes not relevant to the fault may be in any local state as long as the relevant processes are in the local states that together constitute the error.

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

III. MOTIVATING EXAMPLES

This section presents three examples in which different verification strategies produced inconsistent results when run on the same SCR input model. In each case the inconsistency was eventually resolved, and we gained a better understanding of the translation and verification tools in the process. Also, in each case it would have been possible to use a single verification strategy to get an invalid result, with no indication that translation and verification tools had been used incorrectly. The first two examples illustrate a key limitation in the use of model checking tools: although a fault detected by the tool can be manually confirmed or disconfirmed by inspecting the counter example trace provided by the model checker, if the model checker reports that the model is correct no proof is generated to certify this result. (Others have developed ways to determine whether a the "all clear" result from a single model checker is valid [34]; in this work we show how multiple tools can be used to validate each other without detailed knowledge of any particular tool.) In the third example, the inconsistency brought to light is less critical: the use of multiple strategies, rather than preventing a violation from being missed, has the practical benefit of showing that a violation detected by one strategy is not actually present in the original input model.

```
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
```

Fig. 1. Inconsistent outputs from Cadence SMV (top) and NuSMV (bottom) running on the same fault-seeded specification.

A. Inconsistent Results from Two Symbolic Model Checkers

Figure 1 shows the outputs from two versions of the SMV symbolic model checker, Cadence SMV [9] and NuSMV [16], running on the same input model.[1] The model was generated automatically from the specification of a security system, written in the SCR modeling language and described in more detail in section IV-E. As shown in Figure 1 Cadence SMV and NuSMV disagree about whether one of the assertions included in the input model is true or false—the assertion `(cGuardAlarm = On) <=> (cUserDisplay = SeeOfficer)`.

The input model in this example was generated from a fault-seeded version of the specification known to be correct in the original version. The fault-seeded version 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 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 with just the second mutation, both Cadence SMV and NuSMV reported that the assertion `(cGuardAlarm = On) <=> (cUserDisplay = SeeOfficer)` was false. This suggests that the assertion violation reported by Cadence SMV is present in the input model, but somehow masked by the first mutation for NuSMV.

[1]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
```

Fig. 2. Output from SPIN running on a model generated from the same fault-seeded specification used to generate the models for which Cadence SMV and NuSMV outputs are shown in figure 1.

To confirm the Cadence SMV result, we ran SPIN on an input model generated from the same fault-seeded specification. Figure 2 shows the result from SPIN, consistent with the result from Cadence SMV. Based on this, we contacted the devlopers of NuSMV and via several emails determined that the SCR-to-SMV translator we were using produced a syntatically correct but outdated input model. Specifically, the keyword `SPEC` used to mark assertions was not being interpreted the same way by NuSMV as by the older Cadence SMV. As a result, NuSMV was checking assertions in the input model only for a limited set of possible execution paths. By replacing `SPEC` with `INVARSPEC` in the input model before running NuSMV, we were able to get the desired behavior. After making this change the output from NuSMV was consistent with Cadence SMV, reporting a violation of the assertion.

The inconsistency between NuSMV and Cadence SMV in this example was not due to a bug in either tool, but to an outdated translator. A more experienced user of NuSMV may have seen right away that the input model produced by the translator wasn't right. For us, however, it is only because the output produced by NuSMV was compared to that of other verification tools that we discovered and corrected the translation problem.

B. Inconsistent Results from Model Checking and Random Search

Figure 3 shows outputs from the explicit-state model checker SPIN [8] and our tool Lurch running on input models generated from a second fault-seeded version of the security system specification mentioned above.[2] SPIN reports that the input model is correct but Lurch reports

[2]For a more detailed explanation of the example described here see [11].

```
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
```

Fig. 3. Inconsistent outputs from SPIN (top) and Lurch (bottom) running on input models generated from the same fault-seeded specification.

an assertion violation. Because Lurch is an incomplete 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 not expect the result shown here: Lurch, an incomplete tool, reports a violation, while SPIN reports no violation.

Unlike the example in the previous section, in which NuSMV and Cadence SMV were run on the same input model, in this case SPIN and Lurch ran on two different input models, generated from the fault-seeded specification using two different translation tools. We initially assumed that the inconsistent outputs shown in Figure 3 were due to an error in the translator to generate the Lurch input model, since it was newly developed as part of the research presented here. So, to determine whether the property violation detected by Lurch was present in the original fault-seeded specification and not due to an error in the translator, we used an SCR simulation tool to step through the fault-seeded specification according to the execution trace output by Lurch.

Figure 4 shows part of the log produced by stepping through the fault-seeded specification. The log indicates that one of the functions in the specification is not *disjoint*; that is, the function is nondeterministic as a result of overlap between two conditions that should be mutually exclusive. This general disjointness error does not necessarily mean that a specific assertion in the specification will be violated. We observed, however, that the translation tool used to generate the input model for SPIN uses a feature of the SPIN input language in a way that would not be compatible with the nondeterminism indicated by the disjointness error shown in

```
--- Initial State --------------------------------------
mDigit4 = Blank                         tNumCReads = 0...

--- State 36 -------------------------------------------
DISJOINTNESS ERROR: Function cGuardDisplay can be assigned both Blank and
SeeOfficer.  The first comes from the discriminant at row 1 column 1.  The
second comes from the discriminant at row 1 column 2.  Using first assign.
```

Fig. 4. Simulator log produced by stepping through execution trace output by Lurch.

Figure 4.

The SPIN input language allows blocks to be marked as deterministic steps with the key word `d_step`. SPIN assumes such blocks are deterministic and therefore checks only one path through the block. For blocks that are not deterministic, this results in some of the behavior of the input model being ignored. For the fault-seeded specification in this example, that ignored behavior happened to include a violation of one of the assertions, the violation detected by Lurch. After removing the relevant `d_step` marker from the input model and running SPIN again, it quickly detected the assertion violation previously detected only by Lurch.

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

C. Inconsistent Results from an Invariant Checker and a Model Checker

Figure 5 shows inconsistent results from the invariant checker Salsa [35] and the model checker SPIN running on input models generated from a third fault-seeded version of the security system specification used in the previous two examples. Salsa, a specialized tool implementing ideas from model checking and theorem proving to prove assertions in SCR models, is described in more detail in section IV. Salsa reports that the property `PINEntry` is true (top of Figure 5) but SPIN reports a violation of the assertion corresponding to the property. As discussed in section

```
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
```

Fig. 5. Inconsistent outputs from Salsa (top) and SPIN (bottom) running on input models generated from the same fault-seeded specification.

IV, Salsa is capable of proving properties true; however, if a property cannot be proved 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 proved true by Salsa.

Eventually, we determined the reason for this inconsistency: again, it was due to the translation tool, which ignores a feature of the SCR modeling language when translating to SPIN. SCR allows the use of `NATURE` constraints to limit the behavior of variables representing inputs from the environment. Using `NATURE` constraints, environment variables and variable change events may be directly linked in ways that would be very difficult to represent in Promela, SPIN's input language. And ignoring `NATURE` constraints does not cause SPIN to miss property violations, since they can only be used to limit modeled behavior. So it makes sense that the translator would ignore them.

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, found a violation of the property. This explanation of the discrepancy between Salsa and SPIN was confirmed by removing the constraint from the Salsa input model. Rerunning Salsa on an input model without the constraint, we found that Salsa could no longer prove the property true.

This inconsistency is less critical than the two in the previous examples, because there was no possibility of missing a genuine property violation. But it does show a practical benefit of combining complementary strategies. If only SPIN (and hence only the `NATURE`-ignoring translator) were 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 unnecessary to track down the cause of the violation found by SPIN. In addition, this example underscores the need to validate faults detected by SPIN or any model checker. The fault may be related to a mistake in the portion of the input model representing the environment rather than the critical system to be verified.

IV. Modeling and Verification Framework

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

A. 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 [12]. 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:

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

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.

During the last 15 years automated tools have been developed to enable more effective and less costly analysis of SCR specifications. The current version of 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 the 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. (Disjointness is explained in the discussion of the second motivating example above; coverage violations occur when, from a certain state of the system, for a given input, no next state is specified.)
5) Model Checker(s): Automatic translation from SCR to the SMV and SPIN model checkers.
6) Theorem Prover: Automatic translation to TAME [36], a simplified theorem proving tool.
7) Invariant Checker: Automatic translation from SCR to the Salsa invariant checker.
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 our Lurch random search tool. Through these scripts and the tools listed above, it is thus possible to automatically translate from an SCR specification into the input languages of all five of the testing and verification tools described below.

B. The Cadence SMV and NuSMV Symbolic Model Checkers

The Cadence SMV [9] and NuSMV [37] symbolic model checkers are two freely available versions of SMV, the "symbolic model verifier". The input languages for Cadence SMV and NuSMV are basically the same. As described earlier, however, there are slight differences. Further, the SCR Toolset's translator to SMV simplifies the input model (and improves scalability as a result) by restricting the type of assertions allowed to only those involving the current state of the system. For example, any assertion using the SCR *Next* (′) operator is removed from the model before translating to SMV.

C. *The SPIN Explicit-State Model Checker*

The SPIN explicit-state model checker is a widely used and freely available automated verification tool [8]. 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 such 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. The different approach taken by the SCR-to-SPIN translator (compared to the SCR-to-SMV translator) makes SPIN's verification result more comprehensive, since the input model is closer to the original SCR model. But this also makes the verification run require much more time and memory, compared to verification with Cadence SMV or NuSMV. [3]

D. *The Salsa Invariant Checker*

The Salsa invariant checker uses a combination of ideas from theorem proving and symbolic model checking to prove disjointness and coverage properties, as well as user-specified assertions, for input models written in a modified form of the SCR specification language [30], [35]. 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

[3]Some performance differences 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.

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 the two-state counterexamples produced by Salsa are valid; that is, whether the first state in the counter example is reachable from the system's initial state.

E. 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 [7]. Lurch runs until it reaches a property violation, the end of a path, or a user-specified depth limit. This is repeated until a user-specified number of paths has 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 early [4].

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 our scripts to 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 [11]. 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 explorations through the model. For a more detailed description of Lurch, the additional features and the process of translating from SCR to Promela to Lurch, see [2].

V. CASE STUDY

Section IV-E outlines the experimental procedure and summarizes general results from the main case study example presented in this article. We describe the input model used, an SCR specification for a personnel access control system (PACS) written as an example to show how to develop a high quality software requirements specification. We then describe our process for automatically generating a set of fault-seeded versions of the original SCR specification. Section V-C first describes the experimental process carried out for each fault-seeded specification—the order in which verification tools were run, the settings and precise way in which each tool was used, and the data collected. Finally, we summarize experimental results: fault-seeded specifications are divided into subsets based on which tools were able to detect faults in each specification, and average time and memory requirements reported for each tool running on each of these subsets.

A. PACS SCR Specification

In the three motivating examples and in the experiments below, we used an SCR specification for a Personnel Access Control System (PACS) described in a prose requirements document from the National Security Agency [38]. These requirements have been used by others to compare the effectiveness of process-based and formal-methods-based strategies for developing reliable software [39]. The SCR specification was derived from that document as an example to demonstrate how to write a high quality formal requirements specification and to evaluate compositional verification methods [40].

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

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

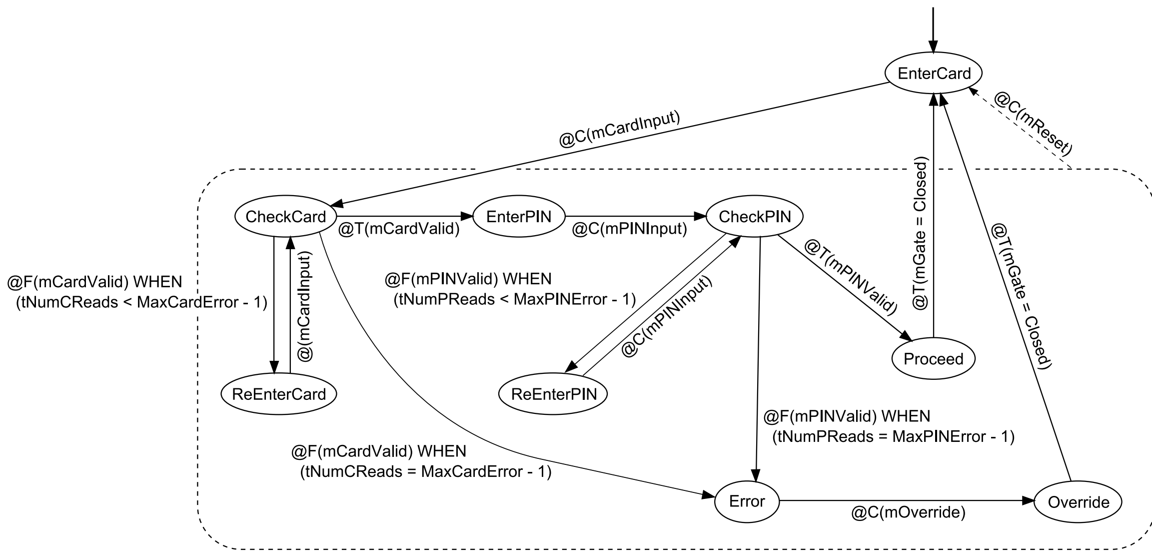

Fig. 6. PACS mode finite-state machine.

PIN, the PACS unlocks the gate and instructs the user to *Please Proceed*. After 10 seconds, the system automatically closes the gate and resets itself for the next user.

Figure 6 shows a finite-state machine representing the core mode logic of the SCR model of the PACS requirements. Initially, the mode is *EnterCard*; when a card is entered the mode changes to *CheckCard*. If the card is not valid, a limited number of retries are allowed, during which time the mode alternates between *CheckCard* and *ReEnterCard*. If the card is valid, the mode changes to *EnterPIN*; when a PIN is entered the mode changes to *CheckPIN*. Similar to *CheckCard*, from *CheckPIN* the user has a limited number of retries if an invalid PIN is entered, during which time the mode alternates between *CheckPIN* and *ReEnterPIN*. If a valid PIN is entered the mode changes to *Proceed*. In mode *Proceed*, the user is able to enter through the gate. Once the gate closes the system is reset to *EnterCard*.

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

The SCR specification of the PACS is approximately 200 lines and includes 15 assertions. To

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

Fig. 7. Mutation operators used to generate fault-seeded versions of the PACS SCR specification.

give an idea of the type of properties checked in the experiments below, one of the assertions is: the user display shows ”Retry” if and only if the ID card has been read at least once but fewer than three times. For the full SCR model, see [2].

B. Generating Fault-Seeded Versions of the PACS SCR Specification

Figure 7 shows mutation operators used to generate fault-seeded versions of the PACS SCR specification. This set of operators is adapted from a set of five determined to be sufficient by Offutt et.al. for Fortran programs [41]. Operators AOR, LCR and ROR are taken directly from that set. UOD is similar to the unary operator insertion (UOI) mutation operator included in the set from Offutt et.al. but is easier to implement without a full parse of the input specification.

The set of five sufficient mutation operators from Offutt et.al. also includes an absolute value insertion (ABS) operator, which replaces an entire arithmetic expression with zero, a positive value, or a negative value. To avoid fully parsing the input specification, and because SCR does not assign a logical value to arithmetic expressions (i.e., SCR does not define 1 to be true and 0 to be false), ABS was not used. Instead we used CRP (from [41] but not one of the five selected) and EVR, which replace individual 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. We developed a mutation tool which automatically generated 323 mutant SCR specification models. Due to space limitations we omit details of the mutant generation algorithm, which are available in [2].

In Andrews et.al. [42] a similar set of mutation operators, adapted from Offutt et.al. to use with C programs, is used to accurately evaluate test suites; that is, these mutation operators produce fault-seeded programs realistic enough that a given set of tests will achieve approximately the same level of program coverage, in terms of widely used coverage criteria, that the given set of tests would achieve for programs with real faults. Also, automatic fault seeding with these mutation operators is compared to manual fault seeding and found to yield results that are actually *more* realistic.

C. Experiments

For each fault-seeded specification, the SCR Toolset was used to run basic generic checks and generate Salsa, SMV and SPIN versions of the specification. We then ran Salsa, SMV and SPIN on the appropriate generated input models; Lurch was run on an input model generated from the SPIN version of the specification.

In the SCR Toolset, we used the command-line program *testtool* via scripts to automatically run consistency checks on the specification to check for syntax and type errors, duplicate names, unspecified or unused variables, missing or inconsistent initial values, circular definitions, and violations of disjointness or coverage properties. For our verification experiments we used only fault-seeded specifications for which no errors were detected by *testtool*. Any specification that failed any of the checks was removed from the set to be used in the experiments. Our focus is on back-end verification tools, so there is no reason to use models that fail these basic checks.[4]

To create NuSMV versions of the SMV models offered by the SCR Toolset's translator, we developed a script to substitute `INVARSPEC` for `SPEC` and delete `AG` from the portion of the SMV model representing assertions in the original SCR specification. This was to remove the possibility of inconsistent results between NuSMV and Cadence SMV, discussed in section III. To create Lurch versions of the SPIN models output by the SCR Toolset's translator we created the script mentioned earlier.

[4]In addition to a *pass* or *fail* result, it is possible to get a *warning* from the SCR Toolset consistency checker. These *warning* specifications were not removed from the experiments since in practice additional back-end verification tools would be used to determine whether the *warning* result corresponded to a real error.

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

1) Those for which Salsa was able to prove *fewer* of the assertions than could be proved for the original SCR specification (94 specifications).
2) Those for which Salsa could prove *additional* assertions (16).
3) Those for which Salsa's results for the assertions matched results on the original specification but Salsa proved *fewer* generic properties (36).
4) Those for which results for the assertions matched the original but Salsa proved *more* generic properties (7).
5) Those for which Salsa's results, for assertions and generic properties, were identical to results on the Salsa version of the original SCR specification (170).

Cadence SMV and NuSMV were next run on the SMV version of each fault-seeded specification, with NuSMV running on a modified version with the changes described in section III. With these minor changes for compatibility with NuSMV, Cadence SMV and NuSMV results were always consistent. Property violations were detected in 141 specifications; no violations were detected in 182 specifications. As mentioned above, the SCR Toolset's translator to SMV restricts the type of assertions to those involving only the current state of the system. For our experiments this meant that *only 9 of the 15 assertions* in the original SCR specification were checked by Cadence SMV and NuSMV. This limitation in the effectiveness of SMV (both Cadence SMV and NuSMV) running on models generated from SCR specifications is also beneficial, because it simplifies the input models and is one possible explanation for the very low time and memory requirements of Cadence SMV and NuSMV, compared to SPIN (described below).

Because some assertions possible in an SCR specification are not included in the SCR Toolset's SMV version of the specification, Cadence SMV or NuSMV can be used as a preprocessor in cases where no faults are detected. Properties proved true by SMV can be removed from the specification so that later verification tools can be run on a simpler input model.

Time and memory requirements were recorded for each run of Cadence SMV and NuSMV. The average time required for Cadence SMV was 0.107 seconds, and for NuSMV 1.21 seconds; the average memory required for Cadence SMV was 3.48 megabytes, for NuSMV 13.2 megabytes,

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)		

Fig. 8. Lurch results on fault-seeded PACS specifications: number of times violations detected vs. search runs, number of specifications in parentheses. For example, “10 / 10 — (175)” means that, for 175 of the fault-seeded specifications, Lurch found violations in 10 of 10 runs on that specification.

all with minimal variance.

Next, we ran Lurch on versions of the fault-seeded specifications generated from SPIN versions of the specifications produced by the SCR Toolset. Because Lurch's random search does not necessarily return consistent results, Lurch was run between 10 and 50 times on each input model. Only in cases where Lurch detected a property violation at least 10 times was Lurch counted as having detected the violation. If Lurch found a violation ten times before 50 runs, we stopped running Lurch on that input model. As shown in Figure 8, 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) in which a violation was detected did Lurch find the violation in less than 75% of runs.

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

Because input models used with Lurch were based on SPIN versions of the specifications produced by the SCR tools, all assertions in the original SCR specification (including assertions not included in SMV versions of the specifications) were checked by Lurch. This is why Lurch detected a larger number of property violations than SMV. In addition, because Lurch input models were derived from the SCR Toolset's SPIN version of the fault-seeded specifications, `NATURE` constraints are ignored by Lurch as well (see Section III).

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

1) First, run SPIN with default settings (default depth limit 10,000).
2) If no violation found, run with settings necessary to get complete verification runs even on input models for which no violations were detected (compile with minimized automaton memory compression, run with depth limit 2,000,000).
3) If (still) no violation found, run on input model with final `d_step` marker removed, as described in section III-B, and with settings necessary for complete verification runs on models for which no violations were detected (compile with minimized automaton memory compression, run with depth limit 3,200,000).

With default settings (option 1), SPIN detected property violations in 205 of 323 input models, averaging 3.41 seconds and 45.9 megabytes per verification run. With settings adjusted to insure a complete verification run, SPIN was able to detect violations in 26 additional models. For SPIN run in this way, the average time required was 561 seconds, and the average memory 488 megabytes. Section III-B showed that in order to get a fully reliable verification result running SPIN on an input model translated from an SCR specification by the SCR tools, it is necessary to remove the final `d_step` marker from the model. Running SPIN on input models with this change, with settings adjusted to enable a complete verification run, SPIN was able to detect property violations in two more of the models. The average time required by SPIN run in this way was 1,340 seconds, and the average memory 475 megabytes. As stated above, SPIN's minimized automaton compression option was used for these runs, which is why they require less memory but far more time than the second set of SPIN runs.

SPIN requires much more time and memory, in most cases, then the other tools described above. But in our experimental framework (primarily because of the translation tools available)

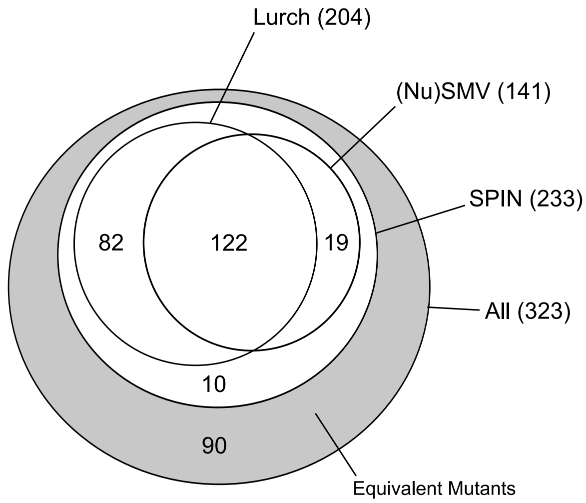

Fig. 9. Summary of verification results for all tools except Salsa—sets of fault-seeded specifications for which each tool detected property violations.

only SPIN can be used to fully verify all 15 of the assertions in the original SCR specification. Based on results from SPIN, we determined that 90 of the fault-seeded specifications were equivalent mutants; that is, they specify behavior identical to the original, as far as the assertions are concerned. Also, as stated earlier, SPIN in the context of the SCR Toolset ignores SCR `NATURE` constraints, so property violations reported by SPIN must be validated using Salsa or one of the SMV model checkers, or manually using the SCR Simulator.

Figure 9 summarizes experimental results for all tools except Salsa, showing sets of specifications in which property violations were detected by all (122 of 233 non-equivalent mutants); Lurch and SPIN only (82); Cadence SMV, NuSMV and SPIN only (19); SPIN only (10); and equivalent mutants (90). Results for Cadence SMV and NuSMV are shown together, denoted (Nu)SMV, since these tools detected property violations in exactly the same set of specifications. Results for Salsa are not shown, because, as explained above, when Salsa fails to prove a property it does not necessarily mean that the property is violated. Thus there is no straightforward way to include the results from Salsa in this kind of diagram. In section V-C we consider how Salsa results might be integrated with other verification tools in a useful way.

For the 122 specifications in which all four tools detected a property violation, using Cadence SMV (the fastest) vs. SPIN (the only tool capable of fully verifying all specifications) saves 345

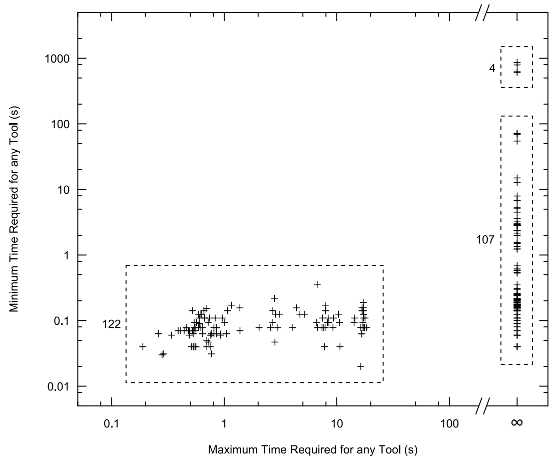

Fig. 10. Specifications plotted to show maximum and minimum time requirements for any tool.

seconds, or about 6 minutes. For the set of 82 specifications in which property violations were detected only by Lurch and SPIN, running Lurch vs. SPIN saves 2,463 seconds, or about 41 minutes. The greatest time benefit is for the set of 19 specifications in which Cadence SMV, NuSMV and SPIN, but not Lurch, detected property violations. For this set running Cadence SMV vs. SPIN alone saves 12,043 seconds, or about 3.5 hours. So even though this is a small number of specifications, they are among the easiest for Cadence SMV and the most difficult for SPIN (assuming a framework using the SCR-to-SMV and SCR-to-SPIN translation tools available to us). Similarly, running Cadence SMV on these specifications requires far less memory than SPIN. Section V-C considers these results in more detail, from various perspectives.

VI. Discussion

A. Performance Variations Between Verification Strategies

Figure 10 shows each fault-seeded specification, excluding equivalent mutants, plotted as a point whose x-coordinate is the maximum time required for any tool to detect a property violation and y-coordinate is the minimum time required for any tool to detect a property violation. For example, the point in the lower right corner of the largest dotted box represents a specification for which the fastest tool to detect a property violation required about 0.020 seconds, while the slowest tool to detect a property violation required about 20 seconds. Points plotted with an x-coordinate of infinity represent specifications for which one or more tools were never able to

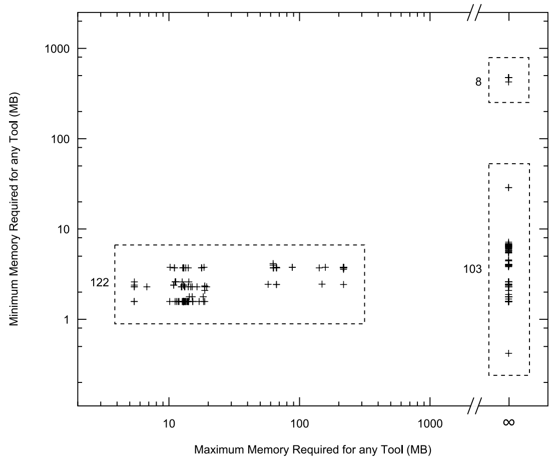

Fig. 11. Specifications plotted to show maximum and minimum memory requirements for any tool.

detect a property violation, regardless of time allotted. Nearly half of fault-seeded specifications for which a fault was detected are in this category. Two reasons why there were such a large number of specifications for which one or more tools could not detect a violation are 1) Cadence SMV and NuSMV could not check 6 of the 15 properties, since two-state assertions are not compatible with the SCR Toolset's SMV translator; and 2) although Lurch was able to check for violations of all 15 properties, it is incomplete.

Figure 11 is similar to Figure 10, except that points represent the maximum and minimum *memory* required by the tools. Again, points plotted with an x-coordinate of infinity represent specifications for which one or more tools were unable to detect any property violation.

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

B. Tool-Specific Lessons Learned

For each verification tool used in our experiments—Salsa, SMV, NuSMV, Lurch and SPIN—we list here characteristics of the tool brought out by comparison of that tool's results with results from other tools. Note that comparisons below are based on input models using SCR-specific translation tools available to us. We do not intend to suggest general conclusions about the relative performance of the verification tools.

Salsa is generally slower than Cadence SMV but faster than NuSMV, Lurch and SPIN, and requires very little memory. Assertions and generic properties can be proved automatically using Salsa, but any assertions that Salsa fails to prove must be checked manually or with some other tool.

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

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

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

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

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

The input model generated by the SCR tools for SPIN encloses code representing transition tables in a `d_step` block, which saves time and memory but is valid only if all tables are disjoint. In general, the final `d_step` marker must be removed to fully verify the specification. In practice, it may also be necessary to use memory compression options and increase the depth limit for the verification run to terminate, as we have in the PACS specification experiments. We found that only SPIN's *minimized automaton* compression option, the slowest but most memory-efficient lossless compression available in SPIN, was sufficient to enable full verification of the PACS specification. We also had to increase the depth limit from the default value of 10,000 to 3.2 million.

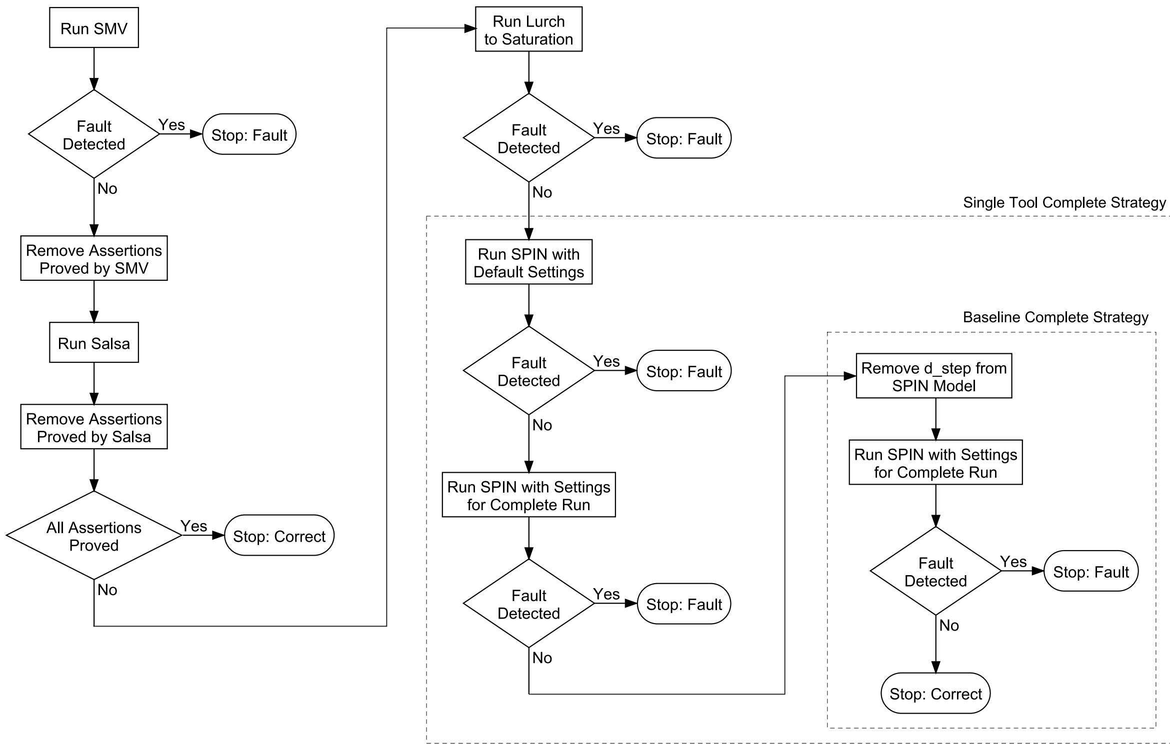

Fig. 12. Combined strategy exploiting complementary variations in performance and accuracy.

C. A General Multiple-Tool Verification Strategy

Figure 12 shows a flowchart representing a multiple-tool verification strategy inferred from both performance variations and characteristics of tools, in the context of the SCR Toolset, relevant to the accuracy of verification results. First, we run Cadence SMV to either detect a fault or prove all single-state assertions. If no fault is detected, single-state assertions are removed from the model and we run Salsa, to attempt to prove some or all of the two-state assertions. Any two-state assertions proved by Salsa are then removed from the model. If all assertions have been proved at this point, the model is correct.

If there are still assertions to be checked in the model, Lurch is run next to detect violations of these remaining two-state assertions. Rather than an arbitrary time cutoff, Lurch is run to 25% saturation. If Lurch detects a fault, we stop. It is possible at this point that the fault detected by Lurch is not actually present in the model, due to Lurch's ignoring `NATURE` constraints, so it needs to be validated. (In our experiments just 2 of 323 mutant specifications contained spurious property violations when `NATURE` constraints were ignored.) Although we mentioned above that assertion violations detected by Lurch or SPIN can sometimes be confirmed or disconfirmed automatically using Salsa or Cadence SMV, if a violation is detected at this point in the flowchart by Lurch it must be validated manually using the SCR Simulator, because Salsa and Cadence SMV have already been run, and any violation that would have been disconfirmed by Salsa or Cadence SMV has already been removed from the model.

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

The dotted rectangles in figure 12 show two alternative complete verification strategies using only SPIN. The outer rectangle shows how SPIN might be used interactively, modifying settings as needed to minimize resource requirements on models for which violations can be detected quickly, but to enable full verification eventually. The inner rectangle shows how SPIN would be used if it were to be run once on each input model with options preset to enable full verification.

VII. Conclusion

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

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

In this article we considered a specific modeling and verification framework, the SCR Toolset, including the consistency checker and its command-line version, *testtool*, and several integrated back-end verification tools. For the case study experiments, we used the Salsa invariant checker for SCR specifications, the Cadence SMV and NuSMV symbolic model checkers, the SPIN explicit-state model checker, and our Lurch random search tool for debugging formal models.

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

Likewise with verification strategies, the goal should not be to simply make sure they all

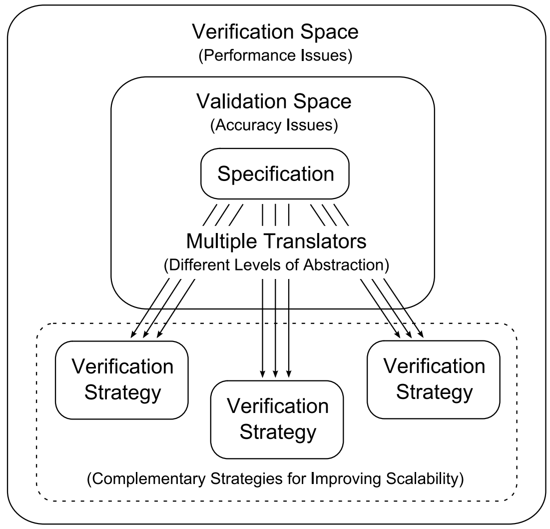

Fig. 13. A conceptual model of the challenges involved in using automatic verification tools.

produce equivalent results and then pick the one with the best performance. Instead, it is desirable that they be complementary, in terms of what kinds of defects they can detect and the kinds of properties they can prove. Further, it is likely that their performance will be complementary too. Rather than a single best strategy for efficient, scalable verification, we confirm the reports that different strategies have different strengths and weaknesses. Not only will a particular strategy be preferable for certain classes of input models, but, for a single input model, changes to the model that seem insignificant can make a large difference in the effectiveness of a particular verification strategy.

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

Figure 13 is meant to illustrate some of the challenges described previously in this section and, along with that, how these challenges may be addressed by a combination of diverse modeling and verification strategies. At the center of Figure 13 is the *specification*, or more generally, the software artifact, which may be anything from prose requirements to source code, along with properties to be verified in whatever form available. The specification and properties must be translated into a formal description suitable for verification and then verified. Thus the information from the specification moves through a *validation space*, in which the goal is to generate accurate models capturing necessary and sufficient information, to a *verification space*, in which the goal is to efficiently determine whether the part of the specification representing behavior is consistent with the part of the specification representing desired properties.

We offer two general contributions. First, we proposed that complementary translation (and modeling) strategies should be combined to address accuracy issues in the validation space. Second, we demonstrated that complementary verification strategies can be combined to address performance issues in the verification space. Through the experiments presented above, we have attempted to show how multiple translation and verification techniques, available within the framework of the SCR Toolset and integrated back-end tools, can be combined to achieve higher confidence and decreased user effort and computational cost.

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

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

What would the ideal set of tools for verification look like? We suggest that it should look like the one in Figure 13. Multiple strategies for improving the scalability of automatic verification would be integrated, through multiple tools, or possibly through multiple scalability strategies available in the same tool. These strategies would be complementary, some emphasizing quick proof of a subset of the properties (as Salsa and SMV were used in our experiments) and some emphasizing quick detection of errors (as SMV, Lurch, and SPIN in the first two modes, were used in our experiments). For each strategy, translation methods would be available at different levels of abstraction, some emphasizing similarity to the behavior of the source model and property specification (to address validation challenges) and some emphasizing structural simplicity (to address verification challenges). If these kinds of translation and verification tools are available, combination strategies like the one we proposed for SCR will provide better performance and higher confidence in the verification result.

References

[1] 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.

[2] D. Owen, "Combining Complementary Formal Verification Strategies to Improve Performance and Accuracy," Ph.D. dissertation, West Virginia University, 2007.

[3] ——, "Random Search of AND-OR Graphs Representing Finite–State Models," Master's thesis, West Virginia University, 2002.

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

[5] D. Owen, B. Cukic, and T. Menzies, "An Alternative to Model Checking: Verification by Random Search of AND-OR Graphs Representing Finite-State Models," in *Proc. International Symposium on High-Assurance Systems Engineering*, 2002.

[6] D. Owen, T. Menzies, and B. Cukic, "What Makes Finite-State Models More (or less) Testable?" in *Proc. International Conference on Automated Software Engineering*, 2002.

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

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

[9] K. McMillan, "The SMV System," 2000, available at www.kenmcmil.com/tutorial.ps.

[10] 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. IEEE / NASA Software Engineering Workshop*, 2003.

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

[12] 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*, vol. 20, no. 1, 2005.

[13] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2000.

[14] G. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, 1997.

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

[16] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A New Symbolic Model Checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, 2000.

[17] K. Havelund, M. Lowry, J. Penix, W. Visser, and J. White, "Formal Analysis of the Remote Agent Before and After Flight," in *Proc. NASA Langley Formal Methods Workshop*, 2000.

[18] P. Glück and G. Holzmann, "Using SPIN Model Checking for Flight Software Verification," in *Proc. IEEE Aerospace Conference*, 2002.

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

[20] T. Ball and S. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," *Lecture Notes in Computer Science*, vol. 2057, 2001.

[21] G. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[22] ——, "Automated Protocal Validation in Argos: Assertion Proving and Scatter Searching," *IEEE Transactions on Software Engineering*, vol. 13, no. 6, 1987.

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

[24] A. Groce and W. Visser, "Heuristic Model Checking for Java Programs," in *Proc. International SPIN Workshop on Model Checking of Software*, 2002.

[25] Y. Dong, X. Du, G. Holzmann, and S. Smolka, "Fighting Livelock in the GNU i-Protocol: a Case Study in Explicit-State Model Checking," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 4, 2003.

[26] A. Gargantini and C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," in *Proc. Joint European Software Engineering Conference and ACM Sigsoft International Symposium on Foundations of Software Engineering*, 1999.

[27] M. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao, "Auto-Generating Test Sequences Using Model Checkers: A Case Study," in *Proc. International Workshop on Formal Approaches to Testing of Software*, 2003.

[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Transactions on Computer Systems*, vol. 15, no. 4, 1997.

[29] P. Godefroid, "Model Checking for Programming Languages Using Verisoft," in *Proc. Symposium on Principles of Programming Languages*, 1997.

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

[31] J. Corbett, “Evaluating Deadlock Detection Methods for Concurrent Software,” *IEEE Transactions on Software Engineering*, vol. 22, no. 3, 1996.

[32] R. Bharadwaj and C. Heitmeyer, “Model Checking Complete Requirements Specifications Using Abstraction,” *Automated Software Engineering*, vol. 6, no. 1, 1999.

[33] C. West, “Protocol Validation in Complex Systems,” *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4, 1989.

[34] H. Chockler, O. Kupferman, and M. Vardi, “Coverage Metrics for Temporal Logic Model Checking,” *Formal Methods in System Design*, vol. 28, 2006.

[35] 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.

[36] M. Archer, C. Heitmeyer, and E. Riccobene, “Proving Invariants of I/O Automata with TAME,” *Automated Software Engineering*, vol. 9, no. 3, 2002.

[37] 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.

[38] “Requirements Specification for Personnel Access Control System. National Security Agency,” 2003.

[39] J. Widmaier, C. Smidts, and X. Huang, “Producing More Reliable Software: Mature Software Engineering Process vs. State-of-the-Art Technology?” in *Proc. International Conference on Software Engineering*, 2000.

[40] D. Desovski, “A Component-Based Approach to Verification and Validation of Formal Software Models,” Ph.D. dissertation, West Virginia University, 2006.

[41] A. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, “An Experimental Determination of Sufficient Mutant Operators,” *ACM Transactions of Software Engineering Methodology*, vol. 5, no. 2, 1996.

[42] J. Andrews, L. Briand, Y. Labiche, and A. Namin, “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria,” *IEEE Transactions on Software Engineering*, vol. 32, no. 8, 2006.