

Random Testing of Formal Software Models and Induced Coverage

David Owen
Lane Department of CSEE
West Virginia University
Morgantown, WV 26505
downen@csee.wvu.edu

Dejan Desovski
Lane Department of CSEE
West Virginia University
Morgantown, WV 26505
desovski@csee.wvu.edu

Bojan Cukic
Lane Department of CSEE
West Virginia University
Morgantown, WV 26505
cukic@csee.wvu.edu

ABSTRACT

This paper presents a methodology for random testing of software models. Random testing tools can be used very effectively early in the modeling process, e.g., while writing formal requirements specification for a given system. In this phase users cannot know whether a correct operational model is being built or whether the properties that the model must satisfy are correctly identified and stated. So it is very useful to have tools to quickly identify errors in the operational model or in the properties, and make appropriate corrections. Using Lurch, our random testing tool for finite-state models, we evaluated the effectiveness of random model testing by detecting manually seeded errors in an SCR specification of a real-world personnel access control system. Having detected over 80% of seeded errors quickly, our results appear to be very encouraging. We further defined and measured test coverage metrics with the goal of understanding why some of the mutants were not detected. Coverage measures allowed us to understand the pitfalls of random testing of formal models, thus providing opportunities for future improvement.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements / Specifications; D.2.4 [Software Engineering]: Software / Program Verification; D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Experimentation, Verification, Performance

Keywords

random testing, model testing, formal methods

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

RT'06, July 20, 2006, Portland, ME, USA

Copyright 2006 ACM 1-59593-457-X/06/0007 ...\$5.00.

1. INTRODUCTION

Software systems pose significant challenges for their quality assessment. This is mainly caused by the prohibitive complexity of any practical software application. Despite the enthusiasm in the research community about the development of formal methods for software verification and validation, they are rarely used in industry due, in part, to scalability problems. Many software applications are simply too large for most formal verification approaches and tools. Consequently, testing is the dominant methodology currently used in industry to assess the quality of software.

Software testing has a well known set of limitations. It is incomplete, so not finding a fault during the testing phase does not imply that the program is fault-free. If performed manually it leads to increased costs of the software development process since it is labor intensive and depends on the specialized skills of the testers to find bugs in the code. Thus, the tendency is to automate the testing process, either by creating automated test case replay tools, or by randomly generating input sequences and using an oracle to validate the correctness of the output.

In the case of the replay tools, the burden is moved to creation of the test case database and its maintenance as the system evolves. Random Testing [14] on the other hand can generate a large number of input cases automatically. However, the development of the oracle that would detect erroneous results of test executions is difficult for any practical system. If the oracle produced too many false positives needing human review and resolution, it would significantly increase the cost. Any number of false negatives leads to decreased efficiency and undetected faults.

One approach to the oracle problem, especially in the domain of high assurance systems, is to use software requirements specifications written in some formal notation as an oracle [4, 23]. The main assumption in this methodology is that the formal model of the system representing the requirements specification is "correct." Having a correct formal model can be very beneficial in the software development process since it can be used for automated generation of tests based on defined coverage metrics on the model [10], or it can be used for automated generation of source code which might represent an initial implementation of the system [19]. The focus of our research, therefore, are practical methodologies for verification of formal software models.

A specifier, based on the informal specification from the user (or the domain expert), writes the formal system specification. This formal specification is based on the mental model (understanding of the system) that the specifier de-

velops. It should be complete, minimal and consistent with respect to the domain expert’s informal specification. After the mental model has been developed it is mapped into a formal domain, usually written in some form of formal notation or a domain specific language (eg. Software Cost Reduction (SCR), Z, Relational Specifications, etc.)

There are two sources of errors that can arise during the formalization process. The first source is the incomplete understanding of the informal specification (or the domain) resulting in the development of an incorrect system model. Second, the specifier can introduce errors by incorrectly mapping the (possibly correct) mental model into the formal domain. Both of these types of errors can be discovered to some extent by consistency and completeness checks and this procedure can be automated [16]. However, completeness and consistency are necessary but not sufficient conditions in assuring that the developed model is correct.

The formalization process maps informal knowledge to the formal domain and consequently it is impossible to perform verification between the both. In order to perform verification we must define redundant formal properties which the requirements specification must satisfy (e.g. temporal logic formulas used in model checking, or first order logic formulas used in SCR.) The developed system model is then verified against the specified properties in order to demonstrate its correctness.

There are many different tools available for verifying software models. Some have been designed to find relatively simple errors in very complex models or even code, e.g., compilers and integrated development environments that detect common structural errors. Other tools are capable of finding very complex errors in simpler abstract models, e.g., model checking tools for automated verification of temporal logic properties [8,18]. There are also various kinds of hybrid tools between these two extremes, targeting different kinds of errors and models at different levels of complexity [2, 11, 24].

Some software specification and development frameworks make several complementary verification approaches available [9,13,15]. We have been working with one such framework, the SCR (Software Cost Reduction) Toolset [15], which can be used to automatically generate and test models using various tools including Salsa [6], SPIN [18], SMV [7, 20] and TAME [5]. Using the SCR Toolset in combination with these verification tools makes it possible to find many different types of faults or prove complex correctness properties in specifications written using the tabular SCR notation.

A typical SCR requirements specification consists of two parts: a description of the operation of the system and a property-based part describing the properties that the system must satisfy. The operational part represents the system as a finite-state machine through SCR tables. The property-based part is made up of first order logic formulae representing state or transition (two-state) invariants that must hold for the system. Having these two parts makes it possible to verify an SCR requirements specification by detecting possible inconsistencies between the operational model and the properties. These inconsistencies might be caused by errors in the operational model or incorrectly stated properties. Thus, the set of properties basically represent an oracle used for verification of the operational model of the system.

Our motivation is based on observation that random testing tools like Lurch (see [22] and the description below) can

be used most effectively early in the modeling process, e.g., while writing the SCR requirements specification for a given system. In this phase we are still not sure if we have correctly identified or stated the properties that the model must satisfy or built the correct model. So it is useful to have a way to quickly identify errors in the operational model or in the properties themselves, and make appropriate corrections. As we get closer to a final version of the SCR requirements specification, having corrected the errors detected by incomplete tools like Lurch, we can then apply complete verification tools like SPIN, if feasible, to demonstrate the consistency between the operational model and the stated properties.

The paper is organized as follows. Section 2 describes the Lurch tool for random testing of software models. Section 2.4 explains how we are using the SCR Toolset’s Promela translator to produce Lurch models. Section 3 describes the experiments in which we evaluated the effectiveness of our random testing methodology in detecting seeded errors in models generated from the SCR Toolset. The PACS system used for the case study is described in section 3.1. Section 3.2 presents the results obtained from the experiment. In section 3.3 we explore the induced coverage of the performed random tests. Finally, the conclusions and suggestions for future work are given in section 4.

2. RANDOM TESTING OF SOFTWARE MODELS WITH LURCH

In this section we describe Lurch [22], our prototype simulation tool for random testing and debugging of models of finite-state concurrent systems. Although Lurch’s exploration of the state space is not exhaustive, it can be used to detect errors in large systems quickly using much less memory than any complete-search alternatives.

2.1 Example Input Model

Figure 1 shows a simple finite-state model written for Lurch. The model in Figure 1 originates from a Promela (the input language for the model checker SPIN [18]) model in [17]. The model begins with C code, which may be referred to in the model (the special function *_before()* is called by Lurch internally to set C variables to their initial values). After the %% symbol, finite-state machines are listed; a blank line indicates the beginning of a new machine, and the first state listed in each machine is defined to be the initial state. Each line in a machine description represents a transition and has four fields: the current state, input conditions, output or side affects of the transition, and the next state. The final line represents a safety property: the transition from *ok* to *_fault* is enabled if the first two machines are both in their critical sections (*acs* and *bcs*) simultaneously. State names that begin with an underscore are recognized by Lurch to represent faults. The “-” symbol in the input or output fields denotes that the transition has no input condition or produces no outputs.

2.2 Basic Search Procedure

Figure 2 shows the core random search procedure used by Lurch. Lurch uses a Monte Carlo, not Las Vegas, random search algorithm [21] in contrast to, for example, a random depth-first search, in which all nodes are explored but the order is random. Where random search has been used recently in model checking by others, e.g., [13], it has been

```

#define FALSE 0
#define TRUE 1
enum { A, B } turn = A;
int a = TRUE, b = TRUE;
void _before() { turn = A; a = b = TRUE; }

%%

a1; -; {a = TRUE;}; a2;
a2; -; {turn = B;}; a3;
a3; (1b); -; acs;
a3; (turn == A); -; acs;
acs; -; {a = FALSE;}; a5;

b1; -; {b = TRUE;}; b2;
b2; -; {turn = A;}; b3;
b3; (1a); -; bcs;
b3; (turn == B); -; bcs;
bcs; -; {b = FALSE;}; b5;

ok; acs,bcs; -; _fault;

```

Figure 1: Lurch input model representing Dekker’s solution to the two-process mutual exclusion problem (translated from a Promela model written for the model checker SPIN [17]).

primarily the Las Vegas approach; this is fundamentally different from our approach as it requires the record keeping of conventional deterministic model checking (the search must keep track of states previously visited) and therefore can not scale to models as large.

The main search procedure is shown in lines 11–20. User-defined parameters *max_paths* and *max_depth* (line 11) determine how long the search will run. *max_paths* is the number of iterations, each of which generates a path from the initial global state through the global state space. Path length is limited by *max_depth*, although shorter paths may be generated if a global state is reached from which no more transitions are possible. Lines 13–14 set all machines to their initial local state. The state vector (line 14) includes an entry for each machine in the system. The value of a particular entry represents a local state, i.e., a state in a machine actually enumerated in the input model; a global state is a tuple with a local-state value for each machine. Lines 15–20 generate a path of global states, checking each state to see if it represents a fault.

The *step* function (lines 1–6) inputs a queue of unblocked transitions and the state vector, pops the first transition from the queue (when the queue is generated, transitions are pushed in random order), and modifies the state vector according to the effect of the transition. The queue is emptied in lines 5–6.

The *check* function (lines 7–10) checks the current state to see if it represents a fault. The state is checked for:

- Local state faults, e.g., assertion violations;
- Deadlocks—if, at the end of a path, any local state is not a legal end state, the global state represents a deadlock;
- Cycle-based faults, including no-progress cycles and violations of user-specified temporal logic properties, which are represented by acceptance cycles.

To do cycle detection Lurch stores a hash value for each unique global state in the current path. This requires some

```

1 step(Q, state) {
2   if(Q not empty)
3     /* pop the first transition */
4     tr ← pop(Q)
5     /* modify state vector accordingly */
6     execute_outputs(tr, state)
7     /* empty the queue */
8     while(Q not empty)
9       pop(Q) }
10
11 check(state) {
12   local_fault_check(state)
13   deadlock_check(state)
14   /* cycle_check requires storage of hash values */
15   cycle_check(state) }
16
17 main(max_paths, max_depth) {
18   for(i ∈ max_paths)
19     /* set all machines to initial state */
20     for(m ∈ machines)
21       state[m] ← 0
22     /* generate a global state path */
23     for(d ∈ max_depth)
24       for(tr ∈ transitions)
25         /* see if transition is blocked */
26         if(check_inputs(tr))
27           /* if not, put it in the queue at random index */
28           random_push(Q, tr)
29         /* get next global state */
30         step(Q, state)
31         /* see if next state represents a fault */
32         check(state) }

```

Figure 2: Lurch’s random search procedure.

additional time and memory and can be turned off if the user is not interested in looking for cycle-based faults. Using the hash value storage needed for cycle detection, we implemented in Lurch an early stopping mechanism that works in the following way: for each path generated, we save hash values for all unique global states visited (this is done already for cycle detection) and compare the number of collisions to the number of new values. When the percentage of new values drops below a user-defined *saturation* threshold (default is 0.01 %), the search is terminated. In our experiments, saturation is usually achieved quickly; also, when Lurch is allowed to run to saturation it nearly always produces consistent results.

2.3 Ordered Execution of (e.g., hierarchical) Machines

If the search function presented in Figure 3 is used instead of the one in the original search procedure (lines 11–20 in Figure 2), Lurch can simulate the execution of hierarchical finite-state machine models. The code is very similar and involves addition of a **for** loop after line 15 and change to line 17 of the original search function. In this type of models transitions are divided into groups, and groups are ordered according to a hierarchy. The first group in the hierarchy always goes first, the second group always goes second, and so on. Lurch uses this scheme to separate Büchi automata, used to represent temporal logic property violations, from the rest of the machines in the model. In this way the Büchi automaton is executed as a sort of *observer* of the rest of the system: at the end of each time tick, when all other machines have finished doing whatever they are going to do, the Büchi automaton is given a chance to react to what’s happened. In addition, the Lurch models for the experi-

```

1 main(max_paths, max_depth) {
2   for(i ∈ max_paths)
3     /* set all machines to initial state */
4     for(m ∈ machines)
5       state[m] ← 0
6     /* generate a global state path */
7     for(d ∈ max_depth)
8       /* process one transition group at a time */
9       for(g ∈ groups)
10        for(tr ∈ transitions)
11          /* see if transition is from wrong group or blocked */
12          if(check_group(g, tr) ∧ check_inputs(tr))
13            /* if neither, put transition in queue */
14            random_push(Q, tr)
15          /* get next global state */
16          step(Q, state)
17          /* see if next state represents a fault */
18          check(state) }

```

Figure 3: Search function modified for hierarchical machines.

ments described in this paper used hierarchical finite-state machines to represent SCR specifications, in order to enforce the dependency order assumed by SCR.

2.4 SCR to Promela to Lurch Translation

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

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

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

3. EXPERIMENTAL RESULTS

We used two different tools for our case study dealing with the verification of the SCR requirements specification of the Personnel Access Control System (PACS):

- Lurch, our random testing and exploration tool,
- SPIN, a complete model-checking tool.

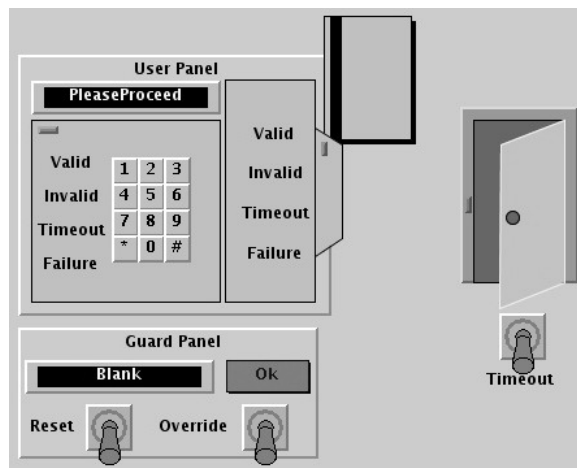


Figure 4: Visual Interface for the PACS specification.

Lurch is used for random testing and debugging of finite-state concurrent systems. Its purpose is to quickly identify errors. It has low memory requirements even for the analysis of large systems. SPIN is a well known model-checking tool that supports different verification techniques (e.g. simulation, exhaustive verification by complete state-based search, approximative verification by state hashing). For the purposes of this study we were interested in the exhaustive verification provided by SPIN: we used it as a safe-guard against the errors which were missed by Lurch. Our intention was to demonstrate that by using incomplete verification tools like Lurch we can quickly identify many errors, limiting the need to repeatedly perform complete verification.

3.1 PACS SCR Specification

The SCR specification of Personnel Access Control System (PACS) originates from a prose requirements document from the National Security Agency [1]. The SCR specification had been derived to demonstrate how to write a high quality formal requirements specification of a secure access system. Figure 4 presents a visual interface used for simulation and testing of the developed PACS specification.

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

To initiate the validation process, the PACS displays the message “Insert Card” on the user display and, upon detecting a card swipe, validates the user name and SSN. If the card is valid, the PACS displays “Enter PIN.” If the

card is unreadable or the information on the card fails to match information in the systems database, the PACS displays “Retry” for a maximum of three tries. If after three tries the user’s card is still invalid or there is no match, the system displays “See Officer” on both the user display and the officer display and turns on an alarm (either a sound or light) on the officer’s console. Before system operation can resume, the officer must reset the PACS by pushing the reset button. The user, who also has three tries to enter a PIN, has a maximum of five seconds to enter each of the four digits before the PACS displays the “Invalid PIN” message. If an invalid PIN is entered three times or the time limit is exceeded, the system displays “See Officer” on both the user and the officer display. After receiving a valid PIN, the PACS unlocks the gate and instructs the user to “Please Proceed.” After 10 seconds, the system automatically closes the gate and resets itself for the next user.

3.2 Random Testing Results

We created 50 error-seeded versions of the specification. Each error-seeded specification contained one error not detectable by automatic checking features of the SCR Tool. For example, since syntax errors or errors involving circular dependencies are detectable by the SCR Tool, we did not use these for seeding. The mutation operations used were:

1. Change of operators;
2. Change of boolean and integer values;
3. Change of variable names.

The seeded errors were not based on expert knowledge of the system. We expect our results would be similar to larger scale experiments in which the same kind of mutations would be done automatically [3].

We used the translation tools described above to create SPIN and Lurch versions of each error-seeded specification. All error-seeded SCR specifications resulted in syntactically correct SPIN and Lurch input models. We then ran SPIN and Lurch on the 50 error-seeded specifications.

To give an idea of the scale of the experiments, for a verification run of the Promela model generated from the original correct SCR specification SPIN explored 180 million states. Without compression verification would have required 6.8 GB of memory. In order to perform a full verification run we used the minimized automaton compression option, set for a state vector of 44 bytes, and had to increase the depth limit for the search to 3.2 million steps (default is 10,000). With these settings SPIN was able to explore the complete state space. The verification run took about 30 minutes on a 2.5 GHz desktop machine with 512 MB of memory.

SPIN with the settings appropriate to perform the complete verification, detected property violations in 43 of the 50 specifications. We assume that for the remaining 7 specifications the seeded errors did not cause violations of any specified properties and consider that they are equivalent to the original correct specification. All defined properties for the PACS specification were simple safety properties, represented as assertions in the SPIN and Lurch models.

For 35 of the 50 error-seeded specifications, Lurch detected a property violation. Because of Lurch’s incomplete random search, it is possible to obtain different results in different runs. So we ran Lurch 10 times on each specification, with default options (including the “saturation” stopping

Table 1: Event table for the $tNumCReads$ variable.

$tNumCReads$ Event Function		
Modes for $mcStatus$	Events	
<i>CheckCard</i>	@T($mCardValid$) OR @C($mReset$)	@F($mCardValid$)
<i>Error</i>	@C($mOverride$) OR @C($mReset$)	NEVER
<i>ReEnterCard</i>	@C($mReset$)	NEVER
$tNumCReads' =$	0	$tNumCReads + 1$

criterion described previously). The maximum time for any Lurch run was about 15 seconds. For all but two of the specifications, Lurch found violations in either all or none of the runs. We counted Lurch as detecting violations only if they were found in all ten runs.

Consequently, the random testing approach in Lurch was successful in identifying 81.39% of the mutants. The average time required to detect a seeded fault by Lurch was 3.7 seconds, much faster than the average time of fault detection by SPIN, which was 76.8 seconds.

3.3 Induced Coverage on SCR models

As a second part of the experiment we wanted to take a closer look at the induced structural coverage of the performed tests on the PACS SCR model. First we should define the structural coverage of interest.

The underlying SCR model represents the environmental quantities as monitored and controlled variables. The environment non-deterministically produces a sequence of input events, i.e., changes in some monitored quantity. The system, represented in the model as a finite state machine, begins execution in some initial state and then responds to each input event by changing its state and, possibly, by producing one or more output events. An output event represents a change in a controlled quantity.

An assumption of the model is that with each state transition exactly one monitored variable changes the value. To concisely capture the system behavior, SCR specifications may include two types of internal auxiliary variables: terms, and mode classes whose values are modes. Mode classes and terms often capture historical information.

To construct the state transition function, SCR uses composition of smaller functions described in a tabular notation, thus improving the readability and understandability of the developed specification. There are three kinds of tables in SCR requirements specifications, event tables, condition tables, and mode tables. These tables describe the values of each dependent variable, that is, each controlled variable, mode class, or term.

Based on the conditions or events that become true, the variables are changed according to their table definitions. Each table cell specifies a guarded condition that must hold true in order for the variable to be assigned some new specified value. Table 1 presents the event table for the $tNumCReads$ term variable. This variable counts the number of unsuccessful card reads for the current user trying to obtain access through the system (see Section 3.1 describing the PACS system). Depending on the current value of the $mcStatus$ variable and the events described in the cells of the table, $tNumCReads$ is either set to zero or increased by one.

Table 2: Cumulative coverage of a single Lurch run on the correct PACS specification.

Cell	1		2		3		4		5		6	
	True	False	True	False	True	False	True	False	True	False	True	False
<i>mcPIN</i>	3762	145220	1078	18885	242	4334	225	4351	60	996	57	999
<i>mPINInput</i>	12	25043										
<i>mcStatus</i>	12685	75881	3758	45614	12	25803	5	35	2	31	3206	46166
<i>cGate</i>	75	175071	175071	75								
<i>cGuardAlarm</i>	63	175083	175083	63								
<i>cGuardDisplay</i>	12443	162703	13	175133								
<i>cUserDisplay</i>	12434	162712	3206	171940	3758	171388	4	175142	14	175132	13	175133
<i>tNumCReads</i>	10921	10323	3219	18025	13	6	0	19	1563	3183	0	4746
<i>tNumPReads</i>	8	11	4	15	13	12	0	25	4	2	0	6

The keyword *NEVER* denotes an event which can never become true. For the details on SCR and its notation we refer the reader to [15].

For the purposes of our study we created tools to measure the branch coverage of each cell in the PACS specification, i.e. we count how many times each event in the cells was evaluated to *TRUE* and to *FALSE*, when the corresponding guard condition was satisfied. For example for the cell in the first row and second column of Table 1 we count how many times the event $@F(mCardValid)$ is evaluated to *TRUE* or to *FALSE*, when the *mcStatus* variable is equal to *CheckCard* during the execution of the test. If there is no guard condition (i.e. the table is not set as being dependent on a specific variable), we consider that the guard is always satisfied.

Instead of traditional coverage measurement which only tracks if a construct is exercised at least once, we wanted to gain more quantitative information about the performed random tests, i.e. tracking the number *TRUE* and *FALSE* evaluations. The intention was to see if there will be noticeable differences in the observed coverage measures between the executions on the correct specification, the detectable mutants, and the mutants which were undetected.

The input sequences generated by the Lurch tool were exercised by instrumented simulators of the SCR models tracking the previously defined coverage measures. Table 2 gives the cumulative coverage numbers for portion of the variable definitions from a single Lurch run on the correct specification. Some of the variables have more than 6 cells and we have truncated the table because of space constraints. The numbers we use when referring to cells of SCR tables are assigned left to right, top to bottom starting from 1. For example: cell 2 of Table 1 contains the event $@F(mCardValid)$, and cell 5 contains the event $@C(mReset)$. In Table 2 we can find the defined branch coverage of all 6 cells of the Table 1 specifying the *tNumCReads* variable. Note that the cells having *NEVER* event (cells 4 and 6) have zeros for their *TRUE* coverage measures, since this event can never become true. We can also see that the event $@F(mCardValid)$ (cell 2 in Table 1) was evaluated to true 3219 times and to false 18025 times when the *mcStatus* variable was equal to *CheckCard*.

From the Table 2 we can conclude that the uniform random testing performed by Lurch does not result in uniformity of the measured coverage across the model. For example, notice the difference in the coverage between the *tNumCReads* variable, and the *tNumPReads* variable. Although these two variables are very similar (one counts the number of incorrect card reads and the other counts the num-

ber of incorrect PIN reads) and they have almost the same specification tables, *tNumCReads* has been more extensively covered than *tNumPReads*. This is caused by the fact that there is a sequence of specific events in the system that needs to be executed for a person to be validated. The probability of the uniform event generation to explore this sequence decreases exponentially with its length.

Thus, having parts of the specification receive low coverage makes it possible for errors not to be detected with the random testing approach. One solution might be to increase the depth of the generated random probes, or to give Lurch more time to exercise the model, or both. We can use the defined coverage measures as a stopping criteria to determine when to stop the random testing.

We compared the induced coverage by the same event sequence generated by Lurch on the correct specification, and the faulty specifications with the idea that deviations might point out the undetected mutant specifications. Our findings were inconclusive - there were mutant specifications which had different degrees of deviation of the coverage from the correct one. We observed faulty specifications with the same coverage measures as the correct one, as well as faulty specifications with large difference in the induced coverage from the correct one. Also, we observed undetected mutants which had complete branch coverage, i.e. each cell of the specification was evaluated to *TRUE* and *FALSE* at least once. This leads to the conclusion that the defined branch coverage should be refined, maybe following the ideas of [25], or propose different coverage measures.

Another approach to even-out the induced coverage by random testing tools like Lurch is to use different probabilities for input event generation instead of generating the input sequences uniformly. The strategy may represent the operational profile of the system under test. These probabilities can be modeled, for example, by using Markov models as described in [26].

In our case study with PACS specification there are two variables, *mReset* and *mOverride*, which have low probability of occurrence in the real system - less than 1%. But, they have a very significant effect on taking the system to the initial state. So, during the course of experiments, change in one of these two variables severely limits the depth of our exploration. As a crude experiment we removed the generation of these events from the Lurch model. This simple change resulted in successful identification of the remaining 8 mutants which were not detected when the two variables were present in the tested specification.

4. CONCLUSION AND FUTURE WORK

We presented the methodology and tools for random testing of formal software models. Although simple, random testing approach is demonstrated to be effective in the early phases of formal modeling by identifying faults and inconsistencies between the operational and the property based description of the system. Because of the inherent incompleteness of the random testing we need to augment our tools by using complete verification tools, like SPIN, in the later phases of the model development. However, we note that this is not always feasible - due to the large state spaces of practical software systems, current complete model checking approaches often hit the state explosion barrier and can not be applied.

For this purpose we plan to investigate further the proposed coverage metrics for SCR formal models. One direction is to use different input generation statistics, that would explore the structure of the model more uniformly. The other goal is to modify our random search methodology to automatically explore uncovered parts of the model leading to higher assurance in the correctness of the model under study. In addition to ideas on how this can be achieved at the source code level [12], we believe that randomized testing strategies for formal models have an important role in the system assurance.

Acknowledgments

We would like to thank Constance Heitmeyer from the Naval Research Laboratory (NRL), Washington DC, for providing invaluable input during the development of the PACS SCR specification and details about the SCR to Promela translation. We also thank the anonymous reviewers for their constructive comments and suggestions on how to improve the paper.

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0093315 and NASA under Grant No. NCC5-685. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

5. REFERENCES

- [1] Requirements Specification for Personnel Access Control System. National Security Agency, 2003.
- [2] P. Anderson, T. Reps, and T. Teitelbaum. Design and Implementation of a Fine-Grained Software Inspection Tool. *IEEE Transactions on Software Engineering*, 29(8), 2003.
- [3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, 2005.
- [4] S. Antoy and R. Hamlet. Automatically Checking an Implementation against Its Formal Specification. *IEEE Transactions on Software Engineering*, 26(1), 2000.
- [5] M. Archer, C. Heitmeyer, and E. Riccobene. Proving Invariants of I/O Automata with TAME. *Automated Software Engineering*, 9(3), 2002.
- [6] R. Bharadwaj and S. Sims. Salsa: Combining Constraint Solvers with BDDs for Automatic Invariant Checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, 2000.
- [7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification*, 2002.
- [8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [9] J. Cobleigh, L. Clarke, and L. Osterweil. FLAVERS: a Finite-State Verification Technique for Software Systems. *IBM Systems Journal*, 41(1), 2002.
- [10] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In *Proc. Joint 7th European Software Engineering Conference and 7th ACM Sigsoft International Symposium on Foundations of Software Engineering*, 1999.
- [11] P. Godefroid. Software Model Checking: the Verisoft Approach. *Formal Methods in System Design*, 26(2), 2005.
- [12] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2005.
- [13] A. Groce and W. Visser. Heuristic Model Checking for Java Programs. In *SPIN Workshop on Model Checking of Software*, 2002.
- [14] R. Hamlet. Random Testing. In J. Maciniak, editor, *Encyclopedia of Software Engineering*. Wiley, 1994.
- [15] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for Constructing Requirements Specifications: The SCR Toolset at the Age of Ten. *International Journal of Computer Systems Science and Engineering*, 20(1), 2005.
- [16] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3), 1996.
- [17] G. Holzmann. On-the-Fly, LTL Model Checking with SPIN. spinroot.com/spin/whatispin.html.
- [18] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.
- [19] E. I. Leonard and C. L. Heitmeyer. Program Synthesis from Formal Requirements Specifications Using APTS. *Higher-Order and Symbolic Computation*, (16), 2003.
- [20] K. McMillan. The SMV Model Checker. www-cad.eecs.berkeley.edu/~kenmcmil.
- [21] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [22] D. Owen and T. Menzies. Lurch: A Lightweight Alternative to Model Checking. In *Proc. 15th International Conference on Software Engineering and Knowledge Engineering*, 2003.
- [23] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *ICSE '92: Proceedings of the 14th International Conference on Software Engineering*, 1992.

- [24] S. Sims, R. Cleaveland, K. Butts, and S. Ranville. Automated Validation of Software Models. In *Proc. 16th International Conference on Automated Software Engineering*, 2001.
- [25] E. Weyuker, T. Goradia, and A. Singh. Automatically Generating Test Data from a Boolean Specification. *IEEE Transactions on Software Engineering*, 20(5), 1994.
- [26] J. A. Whittaker and J. H. Poore. Markov Analysis of Software Specifications. *ACM Transactions on Software Engineering and Methodology*, 2(1), 1993.