SRI International

September 2005

Test Suite Evaluation and Generation

Final Report
Contract NAS1-00079, SRI Project 10905

Submitted by:
John Rushby

Prepared for:
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Abstract

We describe an approach to the synthesis of a benchmark test suite for evaluation of coverage analyzers, and new tools for test generation that have been developed for PVS and SAL.

Contents

Chapter 1

Introduction

This report describes work performed for Task 13, and other work in progress when this project was terminated early. We have unified the presentation around the themes of test suite generation and evaluation, and their use in tool qualification (the topic of Task 13).

Chapter 2

Tool Qualification for Test Coverage Analyzers

Testing plays an important role in the assurance and certification of safety-critical systems. Testing is performed at many levels of the system development hierarchy and those at the higher levels (especially integration tests and hardware in the loop tests) are generally regarded as highly effective, both in revealing errors and providing assurance for their absence. However, it is important to ensure that the basic building blocks of the system are of high quality, and so testing at the unit level is also performed with a high degree of rigor.

The goals of unit testing are to ensure that every requirement is correctly implemented, and that there is nothing present that cannot be traced to a requirement—because this could indicate the possible hazard of unintended function. In the case of software units, the first goal is addressed by constructing tests that systematically examine every requirement, and the second is addressed by measuring the extent to which these requirements-driven tests cover the control structure of the program. There are many measures of structural coverage; for airborne software, DO-178B identifies three measures: statement coverage (SC), decision coverage (DC), and modified condition/decision coverage (MC/DC). Measuring the coverage achieved by a suite of tests requires instrumenting the program (i.e., temporarily modifying either the program source or binary code, or its compilation or execution environments) and this is generally achieved with the aid of a tool: a test coverage analyzer. An earlier report [Cro03] documented testing standards and regulatory guidance for structural coverage in several industries, and surveyed the capabilities of some commercial test coverage analyzers.

A coverage analyzer is a software tool and Section 12.2 of DO-178B states that when its recommended processes "are eliminated, reduced, or automated by the use of a software tool, without its output being verified" then the tool should be "qualified." DO-178B identifies two kinds of tools: software verification tools and software development tools. A test coverage analyzer is a software verification tool, which is one "that cannot introduce errors, but may fail to detect them." Chapter 9 of FAA Order 8110.49 [Fed03] gives guidance on tool qualification. In particular, for a verification tool, Section 9-5 b (2) requires that "the Tool Operational Requirements should be documented and available to the FAA" and 9-5 b (3) requires that "Data showing that all of the requirements in the Tool Operational Requirements have been verified should also be documented and available for FAA review. Sufficient verification data are needed to demonstrate normal operation only and will vary depending on the complexity and purpose of the tool, and how it is used. The applicant may package these verification data in any document they choose." Section 9-6 a specifies that a tool's operational requirements data should provide at least (1) "The tool's functionality in terms of specific requirements verified as part of the tool's qualification tests," (2) "A definition of the tool's operational environment, including operating system and any other considerations. . . ," and (3) "Any other information necessary for the tool's installation or operation (such as User's Manual). . . "

There is room for wide interpretation of what a "tool's functionality in terms of specific requirements" and "data showing that all of the requirements in the Tool Operational Requirements have been verified" might mean for a coverage analyzer. In particular, is there a requirement to demonstrate that a tool that claims to measure MC/DC coverage really does do that correctly, and how is that demonstration accomplished?

We are not privy to the qualification data actually submitted by tool vendors, but their public material makes no mention of how the accuracy of their coverage analysis is verified. There would seem to be value in an open process that could be used for this purpose: both so that customers could gain independent assurance and so that new tool developers could benchmark their product against an accepted measure.

Because commercial vendors of coverage tool do not make the design and source code for their tools available for external scrutiny, the only way to evaluate the accuracy of their coverage analysis seems to be through experiment, and that requires development of a benchmark test suite: that is, a suite of programs each equipped with a set of tests that achieves known coverage. Such a benchmark suite would be consistent with the goals of the Standard Reference Dataset (SRD) project at the National Institute of Standards and Technology, which aims to develop evaluation methods for software tools [Bla05].

The benchmark suite should be constructed against the requirement for accurate coverage estimation by a generic coverage tool, and there are two dimensions to this requirement.

1. Does the tool correctly interpret the coverage measure (e.g., MC/DC) that it is intended to measure?

2. Does the tool correctly interpret the programming language of the software whose coverage it is intended to measure?

These two dimensions are interrelated (e.g., to interpret the DC or MC/DC measures, we have to know what is a decision in the programming language concerned), but it proves helpful to consider them separately.

A benchmark suite for coverage analyzer will comprise a set of programs that include all the control structures of the programming language concerned, and that use them in a way that exercises all of the cases of the coverage measure concerned. The question then arises: how should this benchmark suite be developed to achieve adequate "metacoverage," that is, to provide adequate assurance for the accuracy of the coverage tools that it is applied to?

The term "all" in the previous paragraph might seem to indicate an adequate notion for metacoverage ("all" the control structures, and "all" the cases of the coverage measure), but this is deceptive because we need to consider combinations. If the language provides both `if` statements and `case` statements, is it enough to consider these separately, or do we need to consider situations where one is nested inside the other and, if so, to what depth? And are syntactic considerations sufficient, or do we need to consider semantic ones as well (e.g., aliasing)? There seems no easy resolution to these questions, so our proposal is to adopt a "fault-based" approach: that is, we use historical and other experience to hypothesize elements of the programming language and of the coverage measure that are considered difficult or that admit different interpretations, and then we construct tests that will reveal the difference between analyzers that contain each hypothesized fault and those that do not.

In the next chapter we consider aspects of programming languages that are challenging for coverage analysis and aspects in definitions for coverage that are similarly challenging.

In Chapter 4, we consider how to generate a benchmark suite that will challenge the accuracy of coverage analyzers. There are two steps to this process: the generation of challenging programs, and the generation of test sets for those programs. The first of these can be done by grammar-based testing, which is a form of random test generation; the second requires test generation automated by a model checker.

In Chapters 5 and 6, we provide brief descriptions of the random test generator that we have developed for PVS, and the `SAL-ATG` test generator that we have developed from the model checkers of SAL.

Chapter 7 considers issues in extending automated test generation beyond unit tests into the higher levels of the system development lifecycle.

Chapter 3

Challenges in Coverage Estimation

As mentioned in the introduction, there are two dimensions to accurate coverage analysis or, dually, two sources of difficulty in achieving accurate coverage analysis. These are difficulties in the programming language concerned, and difficulties in the definition of the coverage measure employed. We consider these separately.

3.1 Difficulties Due To The Programming Language And Its Use

Most coverage measures are concerned with elements of the language that affect the flow of control, and in most programming languages these elements are controlled by Boolean expressions. A Boolean expression that affects control flow is called a *decision* and its atomic elements are called *conditions*. Several coverage measures probe for the "meaningful impact" of the conditions within a decision. We can hypothesize that a coverage analyzer might work by finding the constructs in a program that affect its control flow (e.g., `if`, `while`, and `case` or `switch` statements etc.) and then exploring the conditions of the decision within the construct. For example, in the following fragment of a C program

```
    if (k > 0 && k < 13) m = k;
```

the decision and its conditions are easily extracted from the `if` statement. However, this program could have been written as follows

```
    temp = (k > 0 && k < 13);
    ...
    if (temp) m = k;
```

and many lines of code could separate the assignment to the variable `temp` and its use within the `if` statement.

A coverage analyzer could reasonably assume that the only reason for making an assignment to a Boolean variable is that the value will later be used in a control flow decision, and so its notion of condition coverage might be extended from explicit control flow constructs to any assignment to a Boolean variable (a position paper of the Certification Authorities Software Team recommends this treatment [CAS02]). This approach could be straightforward in a strongly typed language such as Ada, but is difficult in C, where there is no explicit Boolean type. An analyzer could look for the appearance of comparison operators (such as `k > 0` in the example above) and Boolean operators (such `as &&` in the example) to infer that an integer is being used as a Boolean, but this is incomplete: any nonzero integer is interpreted as `TRUE` in C, so the effect of comparisons and of Boolean operations can be reproduced by arithmetic computations (in fact, integers are often used to encode decisions for `switch` statements in C). To detect this, a coverage analyzer would need either to perform a much stronger form of type inference, or to trace information flow into the decisions of control constructs.

Of course, our context is safety-critical systems, where we may assume that programmers do not deliberately disguise their intent, and where there usually are strong style guides that strive to ensure safe practices (one example is MISRA-C [Mot04]).

Suppliers of coverage analysis tools should document precisely the subset of the programming language that their tool is capable of analyzing and how compliance with this subset is checked. If the analyzer claims to work with a standardized subset such as MISRA-C, then a program's conformity with the standard can be delegated to a checker for that standard; otherwise the analyzer must have some provision for detecting programs outside the subset that it is able to handle. Notice that subsets such as MISRA-C are intended to avoid usages that are error-prone. These are not necessarily the same as those in which it is difficult to identify the notions of decision and condition, and hence to perform correct coverage analysis; thus, reference to an external standard subset may not absolve a coverage analyzer from dealing with some difficult cases.

Notice, too, that a strict subset may simplify the analyzer's task at the expense of encouraging programming errors. To take an extreme example, an analyzer could disallow Boolean operators, so that the following schematic program

```
if (p && q)
   S1;
else
   S2;
```

would have to be rewritten as follows.

```
if (p)
{
   if (q)
      S1;
   else
      S2;
}
else
   S2;
```

Duplication of the code fragment `S2` required by this transformation is known to be an error-prone process.

We recommend that a benchmark suite should include programs in which the decisions and conditions affecting control flow are nonobvious. Some programming constructions that render decisions and conditions nonobvious are specific to the programming language concerned (e.g., the lack of an explicit Boolean type in C), while others are fairly generic (e.g., aliasing, and exceptions). Identification of constructions that render control flow decisions nonobvious could be a community activity, similar to ones that have identified error-prone constructions and defined safe subsets for safety-critical applications. Most treatments (e.g., [San03]) focus on identifying which Boolean expressions (or those that are implicitly Boolean in the case of C) should be considered as decisions, whereas a promising alternative might be to define decisions as those calculations that are related by information flow to statements in the program that affect control flow.

If the coverage analyzer requires compliance with some external coding standard, then the benchmark suite should first be filtered through the recommended checker for the standard concerned. The programs remaining in the benchmark suite should then either be handled correctly by the coverage analyzer, or be rejected by its own internal subset recognizer.

3.2 Difficulties Due To The Definition Of Coverage

The programming language and the programming style determine the difficulty or otherwise of determining what is a "decision." Given this determination, the task of a coverage analyzer is to check that the control flow choices guarded by each decision are all exercised (for DC) and that the conditions within each decision are individually shown to affect the outcome (for MC/DC). The precise interpretation of "individually shown to affect the outcome" is rather delicate. For example in the C expression `A || (B && C)` there are three conditions (`A`, `B`, and `C`) and one interpretation of MC/DC is that for each of these we need a pair of tests that vary the condition concerned while holding the others constant. This is easily achieved when `A`, `B`, and `C` are unrelated. If, however, `B` were actually `!A`, then this interpretation of the criterion cannot be achieved because we cannot vary the first condition while holding the second constant: these conditions are "coupled."

MC/DC [CM94, HVCR01] is an instance of a general class of coverage criteria that attempt to measure the "meaningful impact" of each condition within a decision [WGS94]. A good survey of these criteria and how they deal with issues such as coupling is [AOH03]. MC/DC itself has numerous interpretations that deal with issues such as coupling; some examples are *unique-cause*, *coupled-cause*, and *masking* MC/DC. A position paper of the Certification Authorities Software Team recommends the masking interpretation [CAS01].

It is not the task of a benchmark suite to impose any particular interpretation of MC/DC; rather its task is to verify that an MC/DC coverage tool accurately measures the interpretation that it claims to. In the next chapter we consider how to generate test suites that comply with any specified meaningful impact strategy.

Chapter 4

Generating Benchmark Suites for Coverage Analyzers

Benchmark suites for evaluating coverage analyzers should comprise both “real” examples culled from industrial and academic projects, and “artificial” examples designed explicitly to probe the potential difficulties in coverage analysis identified in the previous chapter. We propose to develop the artificial examples by semi-automatic means. First, we will generate programs that systematically enumerate the constructs of the programming language concerned, and that similarly enumerate instances of that pose difficulties in identification of decisions. Next, for each program we generate a set of test cases the explore the different interpretations of MC/DC that coverage analyzers might implement. We describe technology to support these two stages of automated generation in the following sections. We anticipate that development of a finished benchmark suite may be only semi-automated because both these stages are likely to require some human intervention.

4.1 Automated Generation of Programs

There are several methods for generating inputs to test a given program. The simplest takes a specification of the type of the desired input, and then generates random instances of that type. This approach tends to generate many very similar tests when the input type is unstructured (e.g., three integers), but becomes quite effective when the inputs have complex structure: for example, when they are instances of an abstract data type described by axioms, or are sentences in a language described by a grammar. These generative approaches were first advocated by Gaudel and her colleagues [BGM91] and were later refined using “regularity” and “uniformity” hypotheses [Gau01]. Generative grammar-based testing is widely used in testing language processors such as compilers, and there is even an associated notion of coverage [LJLG04].

Using these methods, it would be fairly straightforward to generate random C programs, say, that cover the full syntax of the language. However, we particularly desire to generate programs in which determination of decisions is challenging. One way to do this is to build it in to the generative process, by elaborating the grammar or biasing the randomization process. Another way is to filter randomly generated instances, selecting only those that are "interesting." The latter method has been implemented in the development environments of some programming languages, starting with the QuickCheck tool of Haskell [CH00] and has been found to be very effective. More recently, these capabilities have been added to theorem proving environments, such as Isabelle [BN04]. To gain experience with these methods, and because they seem so useful, we developed a capability for random test generation in PVS as part of this project; this is described in Chapter 5.

A problem with the "filtering" approach is that it is not very effective when few instances satisfy the filter: much of the work in generating random instances is wasted as most of these are rejected by the filter. It would be better if the generative process could be biased towards creating instances satisfying the filter.

For example, Khurshid *et al* [KPV03] report that when they tried to generate test inputs for a Rockwell "altitude switch" example [HCW02] that operated on linked-list inputs (of timestamped altitude readings with accuracy indicators) they were unable to achieve coverage before exhausting memory when the input sequence was limited to length 3 with altitude allowed to range from 0 to 20,000. Based on methods from Korat [BKM02], Khurshid *et al* extend the Java Pathfinder explicit-state model checker to more of a symbolic execution engine (by replacing calls to arithmetic functions by calls to a symbolic algebra package and using the Omega library for arithmetic constraint satisfaction); this allows them to maintain states symbolically until they find an interesting path, whereupon they instantiate input variables "lazily." Using their approach, Khurshid *et al* were able to generate test inputs for the altitude switch example in 22 minutes (on a 2.2 GHz machine with 2GB of memory). The tool AGATHE [BFG$^+$03] similarly does symbolic execution, using Omega and the rewriter from CafeOBJ to simplify arithmetic constraints.

Although experiments will be required to validate it, our expectation is that generating a suite of programs that are interesting from the perspective of coverage analysis will not be difficult, and that it can be accomplished by suitably tuned grammar-based testing.

4.2 Automated Generation of Test Suites

Given a suite of benchmark programs, we next want to generate test sets for each program that achieve various interpretations of MC/DC. In particular, we would like to generate tests that can distinguish an analyzer that uses one interpretation from another.

Whereas in testing flight software, the tests are generated from the requirements for the software and measured by a coverage analyzer, in testing a coverage tool, it is reasonable to generate tests directly from the source code of the program that the tool is being applied to. Automated generation of structural tests has recently become a well-studied topic [Rus05]. The basic technique is to construct a finite-state abstraction of the program concerned and add Boolean "trap variables" that are set `TRUE` when the target of each test case is accomplished (e.g., when execution reaches a particular decision point with a particular assignment of truth values to its conditions). A model checker is then used to find test cases that set the trap variables `TRUE`: for each trap variable `p`, we challenge the model checker to verify the formula `G(not p)` (i.e., "always `not p`") and the counterexample produced by the model checker, when concretized from the terms of the abstraction back into those of the program, is the required test case for that trap variable.

MC/DC requires tests that come in pairs and this complicates the construction of trap variables. The construction depends on the precise interpretation of MC/DC that is being considered, so we present the basic approach in terms of a generic "meaningful impact" strategy. Let p be a condition (i.e., a Boolean variable or a simple predicate such as $x < y$) that appears in the Boolean expression Φ, then Φ^p_e denotes the modified expression in which e (an arbitrary expression) replaces p in Φ. A test t (i.e., an assignment of truth values to the conditions of Φ) manifests the *meaningful impact* of p in Φ if the test case t' formed by changing the valuation of p (but not of any other condition) in t, also changes the valuation of Φ. This will be so for any t that satisfies $\Phi \texttt{ XOR } \Phi^p_{\neg p}$. Since one of p or $\neg p$ must be `TRUE` in Φ and vice-versa in $\Phi^p_{\neg p}$, this is equivalent to saying that t satisfies $\Phi^p_{\texttt{TRUE}} \texttt{ XOR } \Phi^p_{\texttt{FALSE}}$; this expression is called the *Boolean derivative* (of Φ with respect to p) and is denoted $\frac{d}{dp}\Phi$. To generate meaningful impact tests for Φ, we set pairs of variables that trap satisfaction of $p \wedge \frac{d}{dp}\Phi$ and $\neg p \wedge \frac{d}{dp}\Phi$ for each of the atomic formulas p in Φ (we follow [Kuh99] and [OBY04] in using Boolean derivatives to describe these methods).

Inserting trap variables that distinguish the different interpretations of MC/DC could be quite difficult for an arbitrary program, as also could be construction of the abstraction that is submitted to the model checker (this can be particularly challenging for airborne software because it often contains complex arithmetic that standard decision procedures, which are used in constructing the abstractions, are unable to handle [XDM05]). Fortunately, in generating test suites for a coverage benchmark, we are not confronted by arbitrary programs but by programs generated under our control by the methods suggested in the previous section. Thus, we can instrument the generative process that produces the test programs so that it also generates the corresponding abstractions for the model checker and inserts the appropriate trap variables.

As part of this project we developed an automated test generator based on the model checkers of SAL. This is described in Chapter 6 and greater detail in the manual attached as an appendix.

Chapter 5

Random Testing in PVS

TBD. Here are notes from Sam.

```
(random-test &optional
 ((fnum *) (count 10) (size 100) (dtsize 10) all? ver-
bose? instance
  (subtype-gen-bound 1000))):
    Runs a random test on the given FNUMs, by creating random val-
ues for the
skolem constants and running the ground evaluator on those val-
ues.  This is
useful for checking if the given sequent is worth proving -
if it comes back
with a counter example, then it may not be worth try-
ing to prove.  Of
course, it may just be that a lemma is needed, or relevant formu-
las were
hidden, and that it isn't really a counter exam-
ple.  COUNT tests are run,
and SIZE controls how the random data is generated by provid-
ing a bound, if
necessary (e.g., an integer will be generated between -
SIZE and SIZE).
Normally the test stops when a counter exam-
ple is found; if ALL? is t then
it stops only after COUNT tests.  VERBOSE?, if t, causes each test to be
printed before evaluation.  INSTANCE allows formals and uninter-
preted types
and constants to be given as a theory instance with actu-
als and mappings.
The current theory may also be instantiated this way.
```

~owre/pvs3.2/pvs has the latest random tester in it.

Can be run in two ways:

>From the ground evaluator, type

```
(test expr &optional (count 10) (size 100) all? verbose?)
```

expr should be a FORALL expression, in which case random val-
ues are
generated based on the types of the variables in-
volved. The test values
are then substituted into expr, which is then evalu-
ated. If it returns
false, the falsifying instance is printed, and control re-
turned. If it
returns true, the process is tried again, for count at-
tempts. Size
controls how the values are generated; for example, an inte-
ger will be
generated between -size and size, inclusive, and a list will be
generated of length at most size. The all? flag indi-
cates that the test
should keep going until count is reached, rather than stop-
ping with the
first counter example. The verbose? flag says to print out all values
generated as the tests are run.

>From the prover, use the rule

```
(random-test (fnums *) &optional (count 10) (size 100) all? ver-
bose? instance)
```

An expression is generated from the specified fnums, skolem constants
are then universally quantified and the resulting FORALL expres-
sion is
treated as above. The rule has an extra instance argu-
ment, that allows
instances for uninterpreted types and constants to be given. It takes
the form of a theory name, e.g., "th[int, 0]T := bool, c := true"

Notes:
- The instance argument will be added to the ground evaluator test
  command

```
- nat uses random(0..size), int uses random(-
size..size).  rat creates
  two integers, the second nonzero, and returns the quo-
tient.  Real and
  above just use the rat values.
- All other subtypes create a random value for the super-
type, and then
  check if it satisfies the subtype predicate.  It stops (ungracefully)
  after 1000 attempts (This should be user-defined).
- Functions generate a lazy function, so that
    FORALL (f: [int -> int], x, y, z: int): f(x) + f(f(y)) > f(f(f(z)))
  creates a function that memoizes its val-
ues.  More work needs to be
  done to handle function subtypes (e.g., surjec-
tive? or continuous?)
- Dependent types are only partially supported

Future work:
- Allow users to define random test generators for spe-
cific types.  This
  should be done in PVS specs, rather than Lisp.
- Allow for different random distributions
- Provide handling for inductive definitions (a la Nipkow)
- Check for and remove duplicate tests
- allow different sizes to be associated with different types
    (e.g., 100000 for int, 10 for list[int])
- generate random types for type parameters/uninterpreted types
- higher-order (e.g. A ⊂ B U C)
```

Chapter 6

The SAL-ATG Automated Test Generator

It is well know that model checkers can be used as test generators: the basic idea is to formulate a test goal (say, to reach a particular control state, or to take a particular transition) as a property ϕ in the property language of the model checker and then model check for the assertion "always not ϕ." If the test goal is reachable, the assertion will be FALSE, and the sequence of inputs in the counterexample generated by the model checker serves as the desired test. Sometimes, the test goals can be formulated as properties over the existing state variables of the specification, but in other cases it may be desirable or necessary to instrument the specification with *trap variables*: these are new Boolean variables that are initially FALSE and are set TRUE when their corresponding test goal is satisfied. In model-based design, the model is generally constructed in an industrial notation such as Simulink/Stateflow, Esterel/SCADE, or UML; for model checking, we insert SAL into the tool chain by translating the model into a specification in the SAL language. For test generation, the translator can be extended to insert trap variables automatically for test goals based on structural coverage criteria (e.g., all transitions, or MC/DC); our prototype translator for Stateflow [HR04] does this.

There are several practical drawbacks to the simple method just described for test generation by model checking. First, each test goal is tackled separately, and so the model checker is called repeatedly and performs much redundant work. Some model checkers can ameliorate the cost of checking many assertions (e.g., the SAL symbolic model checker can first compute the reachable states and then "read off" the results for each assertion) but this may not help here (e.g., test generation is often applied to specifications that are too big for symbolic model checking). Second, the set of tests produced is generally inefficient: there is a separate test for each test goal (and there is often a high startup cost in performing each test), and there is often much redundancy among the tests (e.g., some are prefixes of others) so that although each test is generally short, the total length of the test set is large (and each step incurs some cost in performing the tests). Third, some test goals may require tests that are too long for the model checker to find when searching unaided from the start states. Finally, the tests generated may not be very good: that is, they may achieve strong structural coverage yet expose few erroneous implementations—for example, Heimdahl, George, and Weber [HGW04] found that tests generated in this way had less ability to detect mutants in a Flight Guidance System (FGS) benchmark than randomly generated tests. Part of the explanation for this surprising observation is that the model checker is "too clever": it generally finds the *shortest* test to discharge any given goal, and these short tests often exploit some special case and never reach the interesting parts of the state space.

6.1 Design and Implementation

`SAL-ATG` attempts to overcome these drawbacks by generating tests in a novel way [HdMR04]: instead of tackling each test goal separately, `SAL-ATG` works on the whole set of goals, and instead of starting each test anew from a start state, it attempts to *extend* the test found so far to reach some new goal. The basic algorithm employed by `SAL-ATG` is the following. It begins by seeking an *initial test segment* from a start state to any undischarged test goal; if it is successful, it then attempts to construct an *extension test segment* from the last state of that segment to any other undischarged goal, and repeats this process until all goals are discharged or it is unable to construct an extension; in the latter case it returns to the start and attempts to construct an initial test segment for the remaining goals. Either symbolic or bounded model checking can be used to construct initial test segments; extensions are constructed using bounded model checking. Maximum lengths (i.e., search depths) can be specified for both initial and extension test segments, and minimum lengths for extensions. The specification may optionally be sliced (i.e., the cone of influence reduction may be applied) prior to constructing each segment, or prior to each initial segment. Slicing creates a "virtuous circle": as the remaining test goals become increasingly hard to reach (the easy ones are discharged early), so the specification becomes smaller (as the transitions and state variables for discharged goals are removed), potentially easing the difficulty. The incremental character of extensions also eases the difficulty of reaching deep goals: it is generally easier to perform 10 bounded model checks to depth 10 than 1 to depth 100.

`SAL-ATG` is implemented as a 400-line Scheme script on the API of SAL. SAL provides a suite of powerful model checkers (e.g., symbolic, bounded, infinite-bounded, witness) but these are actually scripts on an API that provides the basic functions. `SAL-SIM` is a `read-eval-print` loop on the API that can be used to prototype new scripts, and `SAL-ATG` was initially developed in this way. The SAL API had to be extended in minor ways to support `SAL-ATG` (principally by allowing the bounded model checker to extend an existing path), and some optimizations were later added to improve its performance (e.g., caching the k-fold composition of the transition relation used as part of the input to the SAT solver in bounded model checking).

Test goals are represented by trap variables; the names of trap variables for which tests are required are currently communicated to `SAL-ATG` as a list in Scheme, but we plan to extend the SAL language so that it can specify these.

6.2 Performance

We have evaluated `SAL-STG` on several examples of embedded systems (see the manual attached as an appendix). Here, we report here only the FGS benchmark mentioned earlier [HGW04]. The SAL version of this example was kindly provided by Jimin Gao of the University of Minnesota who is developing an RSML^{-e} to SAL translator. This example has 490 state variables, which translate to 2,332 state bits, 196 reachable test goals for state coverage, and 313 for transition coverage. On a 2 GHz, 1 GByte Pentium 4, the Linux version of `SAL-ATG` takes 61 seconds to generate a single test case of length 45 that covers all the state coverage goals. When extensions are disabled (so that `SAL-ATG` operates like an ordinary model checker used for test generation), it covers all the test goals in 48 tests of total length 65 (17 tests are of length 2 and the rest of length 1). For transition coverage, `SAL-ATG` takes 98 seconds to generate a single test of length 55 that discharges all 313 goals. Without extensions, it required 212 seconds to generate 73 tests having a total length of 84 (1 of length 3, 9 of length 2, and the rest of length 1). We are in the process of evaluating the mutant-rejection quality of these tests on the University of Minnesota's testbed. To produce tests in the format required to drive a given test harness requires customizing the output routines of `SAL-ATG`; the manual provided as an appendix describes how to do this.

6.3 Extended Capabilities

The very short tests generated for the FGS example by an ordinary model checker exploit a special case in this example: an FGS in `active` mode exhibits interesting behavior, whereas one in `inactive` mode does not. For example, in `active` mode, there are complex rules that determine when to enter `ROLL` mode, whereas in `inactive` mode the FGS will enter `ROLL` on receipt of a simple command to do so. A model checker always finds the short path that first makes the system `inactive`, then tells it to enter `ROLL` mode. We might get better tests if we could tell the test generator the *kinds* of tests it should look for (e.g., those that keep the system in `active` mode).

We previously indicated that each test goal is communicated to `SAL-ATG` as the name of a trap variable; in fact, it is communicated as a *list* of trap variables, which is interpreted as their conjunction. A simple way to exploit this capability is to conjoin each ordinary trap variable with one that reports a desired state of the system (e.g., whether it is in `active` mode). A more elaborate exploitation uses an "observer" module to record whether the path seen so far satisfies some "test purpose." The observer merely has to *recognize* tests satisfying its purpose, and `SAL-ATG` effectively performs constraint satisfaction to generate tests satisfying that purpose. Yet more elaborate exploitations use one set of trap variables for structural coverage in the code, and another for "boundary coverage" in certain data structures, then targets the *product* of these two sets; another targets the product of structural coverage goals in two separate components of the system; these product constructions can generate large and thorough test sets (details can be found in Chapter 3 of the manual attached as an appendix).

`SAL-ATG` is an efficient generator of efficient test sets that scales to large examples. It employs a novel method that can reach deep into the state space and through use of conjunctive goals it allows the test engineer to steer it to desired test purposes or to products of goal sets. By making `SAL-ATG` available now, we hope to encourage others to use it and to evaluate the quality of the tests generated. `SAL-ATG` is implemented as a Scheme script on the SAL API and demonstrates the utility of providing a scriptable interface to model checkers. It should be quite simple for others to modify our script or to develop their own to explore alternative strategies.

The next release of SAL will provide a high-performance explicit state model checker and we plan to allow `SAL-ATG` to make use of it; we will consider use of ATPG techniques in the bounded model checker. We will also explore different strategies when a test cannot be extended: currently `SAL-ATG` returns to the start states, or to the end of the initial segment, but it might be preferable to return to some state nearer the end of the current test. We are also developing a notation to allow test goals to be specified in SAL; we wish these to have a logical status, rather than simply comprise an extralogical list of trap variables.

Chapter 7

Beyond Unit Tests

The previous chapters have been concerned with unit tests: how to evaluate coverage analyzers for unit tests, how to generate unit tests, and the automation of these in the random test generator of PVS and the `SAL-ATG` tool of SAL. However, we believe that the transition of industry to model-based development methods, and the power of modern methods of formal analysis, changes the landscape and creates new opportunities. First, some model-based methods such as SCADE come with qualified code generators and MC/DC test coverage is not required (see `http://www.esterel-technologies.com/products/scade-suite/do-178b-code-generation.html`). Second, some powerful methods of static analysis (e.g., the ASTREE tools `http://www.astree.ens.fr/`) may be considered equivalent to MC/DC testing. Both these approaches are being employed in development an assurance for the Airbus A380.

A new opportunity created by these developments is to focus testing activity, and test automation, on higher levels of the system development lifecycle, such as *integration* testing.

For integration testing in reactive systems, a test is no longer a static assignment of values to input variables, but an active entity that engages in continuous interaction with the system under test (SUT). The problem of test generation for reactive systems is similar to program synthesis: the idea is we take the SUT and then synthesize an environment (the test program) that will drive it through the behaviors that we wish to examine. The behaviors can be selected by coverage criteria just as for sequential programs, and we can use test purposes or test observers to further refine test coverage, or to substitute for missing specifications. The synthesized tester will interact with the SUT and attempt to drive it along a path that satisfies the chosen criterion, while monitoring its operation to check that it satisfies its specification. The literature on testing reactive systems is mostly formulated in terms of process algebras [BT00] and devotes considerable attention to the precise notion of correctness that should be employed in the presence of "refusals" (where the SUT may deadlock while its specification does not). Coverage is defined in this theory in a way that is similar to those employed in unit testing: the idea behind "ioco" based testing, for example, is to generate scenarios that explore the structure of the labeled transition system that is the specification.

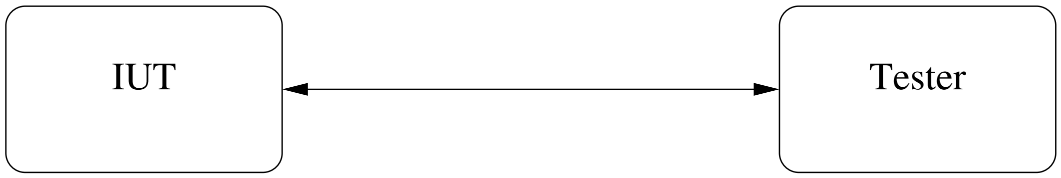

Figure 7.1: Tester Provides Environment for Reactive System SUT

The SUT may be nondeterministic (particularly if we are actually testing several components at the same time, so that the SUT comprises interacting components). Thus, it may not be possible for the tester to force the SUT along a desired path. Instead, the tester must observe the steps made by the SUT and then respond by steps of its own that should move the SUT back toward the desired path. In some test runs, the SUT may take a direction from which the tester is unable to steer it back toward a desired scenario—and in these cases the tester must record an "inconclusive" outcome and start again. Failure outcomes are recorded when the observed behavior departs from that specified—either by exhibiting different outputs than those specified, or by refusing to accept input when the specification remains live.

The basic operation of an "online" tester for reactive systems is the following: we run a model checker from an initial state and choose the first step made by the tester to be one for which some possible behaviors of the SUT and tester lead to the desired scenario; we execute that step and then observe the corresponding step of the SUT; we then run the model checker again from the current state and choose the next step of the tester to be one that preserves the possibility of achieving the desired scenario; this process is repeated until either the desired scenario is executed (and we record the outcome), or the SUT makes a move that excludes the desired scenario—in which case we declare the test inconclusive and try again.

There are obvious optimizations possible here: such as preferring tester steps that minimize the possibility of an inconclusive outcome. And if the SUT takes a path different than the one desired for the current test scenario, we need not declare it inconclusive if it matches some other test scenario. There is no need to run the model checker "online" as described above: we can use it to precompute the tester's response to all possible behaviors by the SUT and save the result as a state machine. The tools TorX [TB99] and TGV [JM99] operate similarly to this description and can also output the tester strategy as a TTCN script.

The methods described above for reactive systems are appropriate for control-dominated systems (e.g., communications protocols) but are defeated by the state explosion problem when the program uses significant amounts of data. In these cases, it is necessary to avoid simple-minded elaboration of the statespace in a way that considers all possible data values in an indiscriminate way. The methods of Rusu *et al* [RdBJ00] do this: the specification and a test purpose are composed and simplified to yield a “symbolic test case” that is then instantiated to yield actual test cases.

The SUT may comprise several components. In some circumstances we will target the tests toward one component and the others will be present simply to provide (part of) its environment. In this case, the tests can be driven by coverage criteria for the target component just as in unit testing. In other circumstances, we may be interested in examining the interaction between two or more components and unit test criteria will be less appropriate. To test interactions, we need a specification or test purpose that specifically describes the joint behaviors of the target components. Suitable specifications include Message Sequence Charts and Use Cases. A typical approach (e.g., in Lucent’s UBET toolset) is to construct a finite state machine from the given joint specification, then generate tests from a transition tour in the state machine.

The arrangement described above is an ideal one: the tester provides the complete environment for the SUT as portrayed in Figure 7.1. In other circumstances (e.g., hardware-in-the-loop testing), part of the environment will be “given” and the tester will synthesize the rest, as portrayed in Figure 7.2.

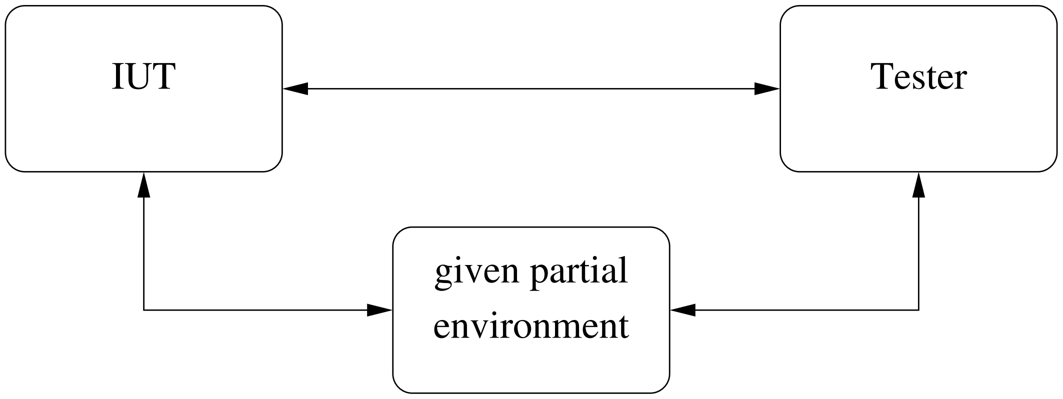

Figure 7.2: Tester Provides Only Part of the Environment

The presence of the given partial environment (GPE) raises two complications. First, it reduces the amount of control that the tester can exert, possibly increasing the proportion of inconclusive outcomes, and increasing the length of tests. Suppose, for example, that the SUT is a participant in a communication protocol and the test purpose is to examine its behavior when the network drops messages. If the tester provides the entire environment, including the (simulated) network, then it can obviously choose to drop messages. If, however, it is necessary or desirable to use a real network as a GPE, then the tester may not be able to control when it will drop messages and will need to generate many messages and wait until one is dropped.

The second complication raised by the presence of a GPE concerns the accuracy of our model for it. From the tester’s point of view, the GPE is just part of the SUT, differing only in that the coverage criterion is not applied to the GPE. Thus, the behavior of the tester is partially determined by its model of the GPE and things may not go as intended if this model is abstracted or inaccurate. If the modeled GPE has more behaviors than the real thing, then a calculated test scenario may degenerate into one already considered (or it may be infeasible and the real GPE may deadlock). On the other hand, if the modeled GPE has fewer behaviors than the real one, then the real GPE may perform an action that takes the test scenario away from the one intended. In both cases, differences between the modeled and real GPE may cause an actual test to play out differently than calculated by the tester. Since the tester is a program that must cope with uncertainty on the part of the SUT, its control strategy may allow it to compensate for some modeling inaccuracies, but in general we are likely to find that many tests degenerate into cases encountered previously or are marked as inconclusive, and that some coverage goals prove difficult to achieve.

If the model of the GPE is an overapproximation, then it is possible that synthesized tests repeatedly attempt to exploit a modeled behavior that the real system does not possess. If the tester is suitably "complete" (i.e., if it explores all paths that could accomplish its goal) it must eventually try a behavior that does exist in the real GPE (since its model is an overapproximation). Even if nondeterminism conspires against us, we must eventually succeed if the system is fair. Alternatively, some kind of learning behavior might be appropriate here: modeled transitions could be removed if they repeatedly lead to scenarios that play out differently than predicted. Ideas and algorithms from automata learning theory, such as L*, may be worth investigating.

If the model of the GPE is an underapproximation, then there is the danger that unmodeled behavior will conspire against us and our tests will always take an undesired path (e.g., if the only way we know to reach the state where the transmission should change gear is to take the engine to 5,000 RPM, but there is an unmodeled governor that limits engine speed to 4,500 RPM, we will never be able to generate a successful test).

Some approaches do not attempt to take GPE behavior into account, nor to steer the SUT behavior. Instead, the tester is simply a replacement for some part(s) of the environment and is driven by a model of the part it replaces, not the SUT. In these cases, testing is performed by simple random generation of behavior from a finitization of the model concerned. The methods of Peleska seem to work in this way [Pel02] and appear to be well-received. We can think of Peleska's method as constructing the tester that would be synthesized for a completely nondeterministic GPE. Or, from another perspective, it is the tester appropriate for the crudest overapproximation to the behavior of the GPE.

7.1 Test Generation for Timed and Hybrid Systems

When the system of interest is not merely reactive but real-time, then delays—and the passage of time in general—are among the properties we would like to observe and test. If we have explicit *clock* variables, then the methods described in the previous section can be used to generate tests that assign to these variables; of course, "assignment" to clock variables is accomplished by real-time delays (some of which will be under control of the tester, others will be determined by the SUT). As with interaction tests of reactive systems, test generation for timed systems is best driven from a specification that describes the intended timed interactions, such as a timed-automaton specification.

Nielsen and Skou [NS01] describe an approach of this kind: they use event recording automata (a special class of timed automata that is closed under determinization) and a coverage criterion similar to ioco-based methods. The method of Salva *et al* [SPF01] is similar. These methods are complicated by region and zone constructions of timed automata reachability computations (as used in Uppaal [LPY97]). It is possible that methods based on calendar automata [DS04] could do better using infinite-bounded model checkers.

If the SUT is a control system, then integration tests must examine its operation in conjunction with the plant that it controls (or a simulation thereof). A test case for this kind of integrated system is a scenario in which plant inputs are varied over time—to examine the throttle control system of a car, for example, a test will take the car for a "drive" along a road with hills: the test inputs will include a profile of the hill, and the driver's operation of the accelerator and brake pedals.

As with testing other kinds of system, the goal of the tester will generally be to exercise selected paths in the SUT, but other test purposes may be stated as properties of the plant (e.g., try to overspeed the engine, or bring the two aircraft very close together). And in addition to examining behavior of the SUT, testing may also be used to validate accuracy of the plant models: this is accomplished by comparing real plant output variables with modeled ones during tests.

Plant dynamics will have an impact on the behavior of the SUT and may complicate the tester's ability to construct a test case that steers the SUT toward the states or transitions targeted for test. If the plant dynamics are described as a hybrid system, then the test generation problem becomes one of controller synthesis for hybrid systems. This problem is notoriously difficult because it requires the ability to perform reachability analysis—which traditionally has been possible only for simple kinds of hybrid system. In general, it will be necessary for the tester to use an abstraction of the plant model. Thus, even when the testing is performed with the modeled (rather than real) plant, the test configuration resembles Figure 7.2 because the tester uses an abstraction of the plant model. More particularly, the test configuration for a hybrid system is portrayed in Figure 7.3

As with testing reactive systems with a GPE, in testing hybrid systems we must consider the possibility that the abstraction of the plant model used in guiding the tester is not fully accurate with respect to the detailed model, or to the actual plant, that is used to close the loop. As discussed earlier, the best choice seems to be for the abstraction used in test synthesis to overapproximate the actual or simulated plant.

Hahn *et al* [HPPS03] consider test generation in exactly this context; their abstracted plant model is a discrete system constructed by ad-hoc manual means based on qualitative abstraction. The tester is constructed similar to one for reactive systems—except that we will have to "concretize" the abstracted variables. That is, test generation will take tran-

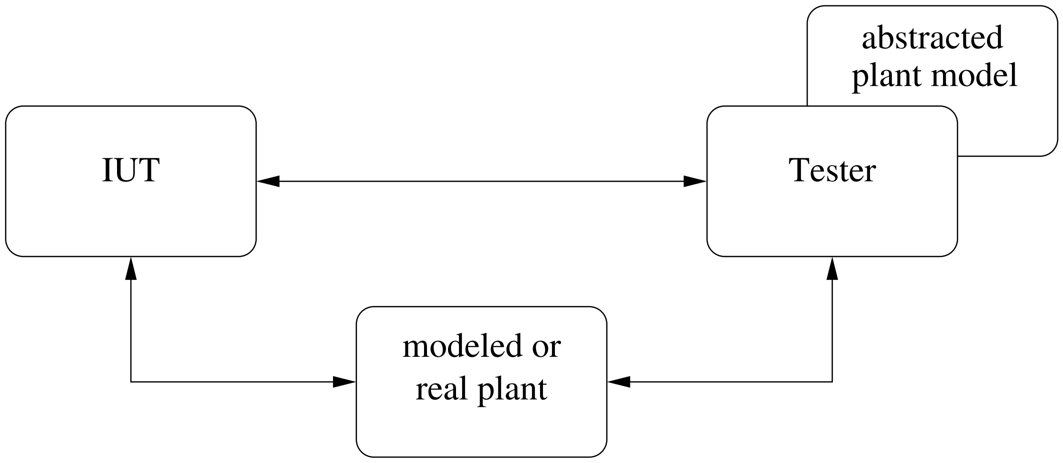

Figure 7.3: Test Configuration for Hybrid System

sitions through the statespace of the abstracted plant model—for example, from a state in which velocity is zero to one where it is positive—and we need to choose a specific concrete velocity (and possibly an acceleration) to induce the corresponding transition in the real (or modeled) plant. Hahn *et al* supply suitable constraints for concrete variables (e.g., for a car, the plausible accelerations, velocities, and road inclines will be specified) and then generate random values within those constraints.

Tiwari's approach to modeling hybrid system [Tiw03, TK02] provides a way to construct the abstracted discrete plant model used in Hahn *et al*'s approach in an automatic way. The abstract states that Tiwari constructs represent the qualitative signs (i.e., negative, zero, or positive) of polynomials over continuous variables (which may include derivatives and higher order derivatives). In this approach, it may be that the transition we wish to follow affects many qualitative values so that more elaborate constraint satisfaction may be needed to generate suitable concrete values than in the simpler models of Hahn *et al*. The abstracted model constructed by Tiwari's method is an overapproximation to the original hybrid plant model, and may therefore generate distinct scenarios that collapse to the same behavior in the presence of the real or simulated plant.

A more primitive approach to test generation in hybrid systems is proposed by Ciarlini and Frühwirth [CF00]: they use discrete time steps, so that the differential equations of the hybrid system become difference equations. This reduces the problem to that of test generation for reactive systems, where some variables range over the reals. Ciarlini and Frühwirth use symbolic simulation and constraint logic programming to construct test cases that will take the SUT along chosen paths. The disadvantage of this method seems to be that the discretization of time means that test sequences become very long, and the constraint satisfaction problems that generate them are correspondingly hard to solve.

Bibliography

[AOH03] Paul Ammann, Jeff Offutt, and Hong Huang. Coverage criteria for logical expressions. In *14th International Symposium on Software Reliability Engineering (ISSRE'03)*, pages 99–107, Denver, CO, 2003. IEEE Computer Society. 9

[BFG$^+$03] Céline Bigot, Alain Faivre, Jean-Pierre Gallois, Arnault Lapitre, David Lugato, Jean-Yves Pierron, and Nicolas Rapin. Automatic test generation with AGATHA. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 591–596, Warsaw, Poland, April 2003. Springer-Verlag. 12

[BGM91] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: A theory and a tool. *IEE/BCS Software Engineering Journal*, 6(6):387–405, November 1991. 11

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–122, Rome, Italy, July 2002. Association for Computing Machinery. 12

[Bla05] Paul E. Black. Software assurance metrics and tool evaluation. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP'05)*, Las Vegas, NV, June 2005. CSREA Press. 4

[BN04] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *2nd International Conference on Software Engineering and Formal Methods*, pages 230–239, Beijing, China, September 2004. IEEE Computer Society. 12

[BT00] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *Modelling and Verification of Parallel Processes: MOVEP 2000*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195, Nantes, France, June 2000. Springer-Verlag. 23

[CAS01] Certification Authorities Software Team (CAST). *Rationale for Accepting Masking MC/DC in Cetificaiton Projects*, August 2001. Position Paper CAST-6, available from http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/. 9

[CAS02] Certification Authorities Software Team (CAST). *What is a "Decision" in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)?*, June 2002. Position Paper CAST-10, available from http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/. 8

[CF00] Angelo E. M. Ciarlini and Thom Frühwirth. Automatic derivation of meaningful experiments for hybrid systems. In *ACM SIGSIM Conference on AI, Simulation and Planning (AIS'2000)*, Tucson, AZ, March 2000. Association for Computing Machinery. 28

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Progamming*, pages 268–279, Montreal, Canada, September 2000. Association for Computing Machinery. 12

[CM94] John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *IEE/BCS Software Engineering Journal*, 9(5):193–200, September 1994. 9

[Cro03] Judith Crow. Software verification tools assessment study. Task 7 Report for Contract NAS1-00079 with NASA Langley Research Center, Computer Science Laboratory, SRI International, Menlo Park, CA, June 2003. 3

[DS04] Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, Grenoble, France, September 2004. Springer-Verlag. 27

[Fed03] Federal Aviation Administration. *Order 8110.49: Software Approval Guidelines*, June 2003. Available via http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/software_regs. 3

[Gau01] Marie-Claude Gaudel. Testing from formal specifications, a generic approach. In *Reliable Software Technologies: Ada-Europe 2001*, volume 2043 of *Lec-*

ture Notes in Computer Science, pages 35–48, Leuven, Belgium, May 2001. Springer-Verlag. 11

[HCW02] Mats P.E. Heimdahl, Yunja Choi, and Mike Whalen. Deviation analysis through model checking. In *17th IEEE International Conference on Automated Software Engineering (ASE'02)*, pages 37–46, Edinburgh, Scotland, September 2002. IEEE Computer Society. 12

[HdMR04] Grégoire Hamon, Leonardo de Moura, and John Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, September 2004. IEEE Computer Society. 20

[HGW04] Mats P.E. Heimdahl, Devaraj George, and Robert Weber. Specification test coverage adequacy criteria = specification test generation *In*adequacy criteria? In *High-Assurance Systems Engineering Symposium*, pages 178–186, Tampa, FL, March 2004. IEEE Computer Society. 20, 21

[HPPS03] G. Hahn, J. Philipps, A. Pretschner, and T. Stauner. Prototype-based tests for hybrid reactive systems. In *14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, pages 78–85, San Diego CA, June 2003. IEEE Computer Society. 27

[HR04] Grégoire Hamon and John Rushby. An operational semantics for Stateflow. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering: 7th International Conference (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 229–243, Barcelona, Spain, 2004. Springer-Verlag. 19

[HVCR01] Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. A practical tutorial on modified condition/decision coverage. NASA Technical Memorandum TM-2001-210876, NASA Langley Research Center, Hampton, VA, May 2001. Available at `http://www.faa.gov/certification/aircraft/av-info/software/Research/MCDC%20Tutorial.pdf`. 9

[JM99] Thierry Jéron and Pierre Morel. Test generation derived from model-checking. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–121, Trento, Italy, July 1999. Springer-Verlag. 24

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of*

Systems (TACAS '03), volume 2619 of *Lecture Notes in Computer Science*, pages 553–568, Warsaw, Poland, April 2003. Springer-Verlag. 12

[Kuh99] D. Richard Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411–424, 1999. 13

[LJLG04] Hu Li, Maozhong Jin, Chao Liu, and Zhongyi Gao. Test criteria for context-free grammars. In *28th International Computer Software and Applications Conference (COMPSAC)*, pages 300–305, Hong Kong, September 2004. IEEE Computer Society. 11

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997. 27

[Mot04] The Motor Industry Software Reliability Association, Nuneaton, UK. *MISRA-C:2004 – Guidelines for the use of the C language in critical systems*, 2004. 8

[NS01] Brian Nielsen and Arne Skou. Automated test generation from timed automata. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 343–357, Genova, Italy, April 2001. Springer-Verlag. 27

[OBY04] Vadim Okun, Paul E. Black, and Yaacov Yesha. Comparison of fault classes in specification-based testing. *Information and Software Technology*, 46(8):525–533, June 2004. 13

[Pel02] Jan Peleska. Hardware/software integration testing for the new Airbus aircraft families. In Ina Schieferdecker, Hartmut König, and Adam Wolisz, editors, *Testing of Communicating Systems XIV (Testcom'02): Application to Internet Technologies and Services*, pages 335–351, Berlin, Germany, March 2002. Kluwer. 26

[RdBJ00] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *2nd International Workshop on Integrated Formal Method (IFM'00)*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357, Dagstuhl, Germany, November 2000. Springer-Verlag. 24

[Rus05] John Rushby. Automated test generation and verified software. In *Verified Software: Theories, Tools, Experiments (IFIP Working Conference)*, Zurich, Switzerland, October 2005. To appear. 13

[San03] Vdot Santhanam. *Study of Qualification Criteria for Software Verification Tools*. Boeing, 2003. 9

[SPF01] Sébastien Salva, Eric Petitjean, and Hacène Fouchal. A simple approach to testing timed systems. In Ed Brinksma and Jan Tretmans, editors, *Formal Approaches to Testing of Software (FATES'01)*, pages 93–107. BRICS, Aalborg, Denmark, August 2001. 27

[TB99] Jan Tretmans and Axel Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 1999. EuroStar Conferences, Galway, Ireland. 24

[Tiw03] Ashish Tiwari. Approximate reachability for linear systems. In O. Maler and A. Pnueli, editors, *Hybrid Systems: Computation and Control, 6th International Workshop, HSCC 2003*, volume 2623 of *Lecture Notes in Computer Science*, pages 514–525, Prague, Czech Republic, April 2003. Springer-Verlag. 28

[TK02] Ashish Tiwari and Gaurav Khanna. Series of abstractions for hybrid automata. In C.J. Tomlin and M.R. Greenstreet, editors, *Hybrid Systems: Computation and Control, 5th International Workshop, HSCC 2002*, volume 2289 of *Lecture Notes in Computer Science*, pages 465–478, Stanford, CA, March 2002. Springer-Verlag. 28

[WGS94] Elaine Weyuker, Tarak Goradia, and Ashutosh Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994. 9

[XDM05] Songtao Xia, Ben Di Vito, and César Muñoz. Toward automated test generation for engineering applications via predicate abstraction. In *20th ACM/IEEE International Conference on Automated Software Engineering (ASE'05)*, Long Beach, CA, November 2005. IEEE Computer Society. To appear. 13

Appendices

Two documents are provided as appendices.

The first is a paper on automated test generation for the IFIP Working Conference on *Verified Software: Theories, Tools, Experiments* (see `http://vstte.ethz.ch/`). This conference is part a program initiated by Sir Tony Hoare to establish verified software as the topic for a Grand Challenge in science. The attached paper makes the case that testing must be included among the topics considered.

The second is the manual for `SAL-ATG`. Some of this work was performed as part of a NASA cooperative agreement NCC-1-377 with Honeywell and was published as its Report 36b. However, that report, and the work it documents, were significantly extended by the present project, which has brought `SAL-ATG` to public release. In particular, Chapter 3 of the attached manual, titled “Test Engineering and Automation” describes relationships between boundary testing and meaningful impact strategies such as MC/DC, and presents novel methods for generating high-quality tests based on conjunctive goals and test purposes.

Invited paper for VSTTE, Zurich, Switzerland, October 2005.

Automated Test Generation And Verified Software [⋆]

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA

Abstract. Testing remains the principal means of verification in many certification regimes. Formal methods of verification will coexist with testing and should be developed in ways that improve, supplement, and exploit the value of testing. I describe automated test generation, which uses technology from formal methods to mechanize the construction of test cases, and discuss some of the research challenges in this area.

1 Introduction

By *testing* I mean observation of a program in execution under controlled conditions. Observations are compared against an explicit or informal oracle to detect bugs or confirm correctness. Much of the testing process is automated in modern development environments, but construction of test cases (i.e., the specific experiments to be performed) remains a largely manual process.

Testing is the method by which most software is verified today. This is true for safety critical software as well as the commodity variety: the highest level of flight critical software (DO-178B Level A) is required to be tested to a structural code coverage criterion known as MC/DC (Modified Condition/Decision Coverage) [1]. And although formal methods of analysis and verification are becoming sanctioned, even desired, by some certification regimes, testing continues to be required also—because it can expose different kinds of problems (e.g., compiler bugs), can examine the program in its system context, and increases the diversity of evidence available.

The weakness of testing is well-known to the formal methods and verification communities—it can only show the presence of bugs—but those communities are now beginning to recognize its strength: it *can* show the presence of bugs—often, very effectively. It is a great advantage in verification if the software to be verified is actually correct, so inexpensive methods for revealing incorrectness early in the development and verification process are necessary for verified software to be economically viable.

[⋆] This work was partially supported by NASA Langley Research Center through contract NAS1-00079, and by SRI International.

Thus, testing is not a rival to formal methods of verification, but a valuable and complementary adjunct. It is worthwhile to study how each can support the other, both in the technology that they employ, and in their contribution to the overall goal of cost-effective verification.

In this regard, the most significant recent development in testing has been the application of technologies from verification (notably, model-checking, SAT solving, and constraint satisfaction) to automate the generation of test cases. Automated test generation poses urgent opportunities and challenges: there are many technical challenges in achieving effective automation, there is a wealth of opportunity in the different ways that automated testing can be used, and there are serious implications for traditional certification regimes, and opportunities for innovative ones; there are also opportunities for theoretical research in the relationship between testing and verification, and for empirical inquiry into their pragmatic combination.

In this position paper, I briefly survey the topics mentioned above, and suggest research directions for the development and use of automated test generation in verification.

2 Technology for Automated Test Generation

Much of the process of test execution and monitoring is automated in modern software development practice. But the generation of test cases has remained a labor-intensive manual task. Methods are now becoming available that can automate this process.

A simple test-generation goal is to find an input that will drive execution of a (deterministic, loop-free) program along a particular path in its control flow graph. By performing symbolic execution along the desired path and conjoining the predicates that guard its branch points, we can calculate the condition that the desired test input must satisfy. Then, by constraint satisfaction, we can find a specific input that provides the desired test case. This method generalizes to find tests for other structural coverage criteria, and for programs with loops, and for those that are reactive systems (i.e., that take an input at each step). A major impetus for practical application of this approach was the realization that (for finite state systems) it can be performed by an off-the-shelf model checker: we simply check the property "always not P," where P is a formula that specifies the desired structural criterion, and the counterexample produced by the model checker is then the test case desired [2]. Different kinds of structural or specification-based tests can be generated by choosing suitable P.

Using a model checker to generate tests in this way can be very straightforward in model-based development, where we have an executable specification for the program that is in, or is easily translated to, the language of a model checker: the tests are generated from the executable specification, which then provides the oracle when these are applied to the generated program. There are many pragmatic issues in the selection of explicit-state, symbolic, or bounded model checkers for this task [3] and it is, of course, possible to construct specialized test generators that use the technology of model checking but customize it appropriately for this application.

The test generation task becomes more challenging when tests are to be generated directly from a low-level program description, such as C code, when the the path required is very long (e.g., when it is necessary to exhaust a loop counter), when the program is not finite state, and when nondeterminism is present.

When tests are to be generated directly from C code, or similar, it is natural to adopt techniques from software model checking. These seldom translate the program directly into the language of the model checker but usually first abstract it in some way. Predicate abstraction [4] is the most common approach, and discovery of suitable predicates is automated very effectively in the lazy-abstraction approach [5]. Abstractions for test generation are not necessarily the same as those used for verification. For the latter, the abstraction needs to be conservative (i.e., it should have more behaviors than the concrete program), whereas in the former case we generally desire that any test generated from the abstraction should be feasible in the concrete program (i.e., the abstraction may have fewer behaviors than the concrete program) [6]. This impacts the method for constructing the abstraction, and the choice of theorem proving or constraint satisfaction methods employed [7].

When very long test sequences are needed to reach a desired test target, it is sometimes possible to generate them using specialized model checking methods (e.g., those based on an ATPG engine [8]), or by generating the test incrementally (so that each subproblem is within reach of the model checker). Some of the most effective current approaches for generating long test sequences use combinations of methods. For example, random test generation rapidly produces many long paths through the program; to reach an uncovered test target, we find a location "nearby" (e.g., measured by Hamming distance on the state variables) that has been reached by random testing and then use model checking or constraint satisfaction to extend the path from that nearby location to the one desired [9]. An alternative approach is to reduce the size of the model that represents the program (it is easier to find longer paths in smaller models): this can be done by standard model checking reductions such as slicing and cone of influence reduction, and also by the predicate abstraction techniques mentioned above.

Traditional model checking technology must be extended or adapted when the program is not finite state. In some cases, an infinite state bounded model checker can be used (i.e., a bounded model checker that uses a decision procedure for satisfiability modulo theories (SMT) [10] rather than a Boolean SAT solver) [11]. In other cases, such as those where inputs to the program are complex data structures (e.g., trees represented as linked lists), we can randomly or exhaustively generate all inputs up to some specified size. Straightforward approaches can be very inefficient (e.g., very few randomly generated list structures represent a valid red-black tree) and redundant (i.e., they generate many inputs that are structurally "isomorphic" to each other), so that it is best to view the search as a constraint satisfaction problem and to use technology from that domain [12].

The test generation problem changes significantly when the program under test is nondeterministic, or when part of the testing environment is not under the control of the tester (e.g., testing an embedded system in its operational environment). In these cases, we cannot generate test sequences independently of their actual execution: it is necessary to observe the behavior of the system in response to the test generated so far and to generate the next input in a way that advances the purpose of the test. Thus, test generation becomes a problem of controller synthesis; methods for solving this problem can use technology similar to model checking but can seldom use an off-the-shelf model checker [13].

The problem becomes yet more difficult when the test environment includes mechanical systems: for example, testing the shift controller of an automatic gearbox in its full system context with a (real or simulated) gearbox attached. Here, the test generation problem is escalated to one of controller synthesis in a hybrid system (i.e., one whose description includes differential equations). This is a challenging problem, but a plausible approach is to replace the hybrid system elements of the modeled environment by conservative discrete approximations, and then use methods for test generation in nondeterministic systems [14]. As in the case of predicate abstraction, the notion of "conservative" that is suitable for test generation may differ from that used in verification.

3 Selection of Test Targets

The previous section has sketched how test cases can be generated automatically; the next problem is to determine how to make good use of this capability. One approach uses test generation to help developers explore their emerging designs [15]: a designer might say "show me a run that puts control at this point with $x \leq 0$." This approach is very well-suited to model-based design environments (i.e., those where the design is executable), but is less so for traditional programming. An approach that has proven useful in traditional programming is random test generation at the unit level. In some programming environments, each unit is automatically subjected to random testing against desired properties if these have been specified, or generic ones (e.g., no exceptions) as it is checked in (Haskell QuickCheck [16] is the progenitor of this approach). A similar approach can be used in theorem proving environments: before attempting to prove a putative theorem, first try to refute it by random test generation [17] (in PVS, this can also be tried during an interactive proof, if the current proof goal looks intractable). These simple approaches are highly effective in practice. More challenging tests can be achieved by exhaustive generation of inputs up to some bounded size [18]. In Extreme Programming, tests take on much of the rôle played by specifications in more traditional development methods [19], and automated, incremental test generation can support this approach [20].

More traditional uses of testing are for systematic debugging, and for validation and verification. In tests developed by humans, the first of these is generally driven by some explicit or implicit hypotheses about likely kinds of bugs, while the others are driven by systematic "coverage" of requirements and code.

One simple fault hypothesis is that errors are often made at the boundaries of conditions (e.g., the substitution of $<$ for $\leq$) and some automated test generators target these cases [21]. Another hypothesis is that compound decisions (e.g., $A \wedge B \vee C$) may be constructed incorrectly so tests should target the "meaningful impact" [22] of each condition within the decision (i.e., each must be shown able to independently affect the outcome).[1] It turns out that these ideas are related: boundary testing for $x \leq y$ is equivalent to rewriting the decision as $x < y \vee x = y$ and then testing for meaningful impact of the two conditions. The classes of faults detected by popular test criteria for compound decisions have been analyzed by Kuhn [23] and extended by others [24,25].

Requirements- or specification-based testing is most easily automated when the requirements or specification are provided in executable form—as is commonly done in model based development. Here, we can use the methods sketched in Section 2 to generate tests that explore portions of the specified behavior. The usual idea is that a good set of tests should thoroughly explore the control structure of the specification; typical criteria for such structural coverage are to reach every control state, to take every transition between control states, and more elaborate variants that explore the conditions within the decisions that control selection of transitions (as in the meaningful impact criteria mentioned earlier). Structural coverage criteria can be augmented by "test purposes" [26] that describe the kind of tests we want to generate (e.g., those in which the `gear` input to a gearbox shift selector changes at each step, but only to an adjacent value), or by predicates that describe relationships that should be explored (e.g., a queue is empty, full, or in between) [27]. Test purposes and predicates are related to predicate abstraction and can be used to reduce the statespace of the model, and thereby ease the model checking task underlying the test generation. Generating a separate test for each coverage target produces inefficient test sets that contain many short tests and much redundancy, so recent methods attempt to construct more efficient "tours" that visit many targets in each test [3,27].

Requirements-based testing is more difficult when requirements are specified as properties. One approach is to translate the properties into automata (i.e., synchronous observers), then target structural coverage in the automata.

4 Testing for Verification

Certification regimes for which testing is an important component generally require evidence that the testing has been thorough. DO-178B Level A (which applies to the highest level of flight-critical software in civil aircraft) is typical: it requires MC/DC code coverage. The expectation is that tests are generated by consideration of requirements and their execution is monitored to measure coverage of the code. As the industry moves toward model-based development,

[1] This use of *decision* and *condition* is the one employed in MC/DC, which is a testing criterion of this kind.

it can be argued that the requirements are represented by the models, and hence that automated test generation from the model is a form of requirements-based testing. One way to do this is by targeting MC/DC coverage in the model. Heimdahl, George, and Weber did this for a model of a flight guidance system developed by Rockwell, and then executed the tests on implementations that had been seeded with errors [28]. They found that the autogenerated tests detected relatively few bugs, and generally performed worse than random testing. Part of the explanation for this distressing observation is that the model checking technology underpinning the test generation is "too clever": it generally finds the *shortest* test to discharge any given goal, and these short tests often exploit some special case and never reach the interesting parts of the state space. There is hope that methods that generate tours through many test goals will do better than those that target the goals individually, or that suitable test purposes may guide the test generator into more productive areas of the state space, but these ideas need to be validated in practice.

Another way in which testing has been employed for verification is in "conformance testing," which is generally applied to distributed systems and protocols. Given a formal specification and an implementation that purports to satisfy it, conformance testing generates a series of tests such that any departure from the specification will eventually be revealed (subject to various technical caveats) [29]. Only a relatively small number of tests can be performed in practice, so the eventuality guarantee is of mainly theoretical interest, and the more pragmatic concern is to try and arrange things so that tests generated early in the series are effective at finding bugs.

There is relatively little work that combines automated testing with formal verification. One attractive approach developed by Rusu uses test generation to decompose the classical formal verification problem into smaller components [30].

5 Research Challenges

Testing is the dominant means of verification used today. Any research agenda in software verification must include testing as a topic, and its roadmap must suggest how the proposed research will improve testing, and how it can use it, as well as how it may replace it in selected areas.

Automated test generation is an attractive topic in this area: it can reduce the cost of testing and may improve its quality. And it is an "invisible" application of formal methods and thus provides a good opportunity to introduce this technology to new communities. Among the most eager adopters of this capability are those in regulated industries where onerous testing requirements constitute a significant part of overall development costs. As mentioned above, there is some evidence that simply using the test coverage requirements as a target for automated test generation may be a flawed strategy: coverage metrics are intended to measure the thoroughness of human-generated tests, and do not necessarily lead to good test sets when used in an inverted role as a specification for the tests required.

Thus, an urgent research topic is development of techniques for specifying good test sets. There are two subtopics here: the role of the human tester will change from construction of tests to *specification* of tests (the tests will be generated automatically from the specification), so we need ideas and techniques for specifying tests (e.g., an extended notion of test purpose); second, we need empirical data on what kinds of test specification produce good tests (i.e., those that are effective in revealing errors). Because automated test generation performs constraint satisfaction (either explicitly, or implicitly via model checking), it is possible to specify test purposes using a recognizer rather than a generator, and this creates attractive possibilities [31].

Most current methods and tools for automated test generation are limited to unit tests. A second general research area is development of methods and technology for other (arguably more important) testing tasks, such as integration and system tests. At these levels, tests become interactive programs, and the formal context becomes that of controller synthesis for nondeterministic, timed, and hybrid systems. Abstraction is likely to be necessary, both for the system under test and for its environment, and there are interesting questions regarding the appropriate kinds of abstractions to use, and the theorem proving and model checking methods that are most suitable for constructing and using them.

A third suggested general research area is the integration of testing with formal methods of analysis and verification. Again, there are two subtopics: one is technical integration—for example, how can testing help in formal specification and proof (cf. QuickCheck-like methods for rapid refutation)—while the other focuses on how the overall verification process can be decomposed into elements that are effectively tackled by different means. There are proposals, for example, to replace some unit test requirements in avionics by static analysis; yet testing can address some issues (such as compiler bugs, which are a genuine problem) that static analysis does not (unless applied to machine code), so the overall web of argument in support of verification may become interestingly complex. A companion paper in these proceedings outlines some of the issues in technical integration of verification components [32], while the larger issues of "compositional assurance," in which the assurance case for a system is composed from different kinds of verification evidence for its components, is only just beginning to receive attention.

References

1. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A practical tutorial on modified condition/decision coverage. NASA Technical Memorandum TM-2001-210876, NASA Langley Research Center, Hampton, VA (2001) Available at `http://www.faa.gov/certification/aircraft/av-info/software/Research/MCDC%20Tutorial.pdf`.
2. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In Nierstrasz, O., Lemoine, M., eds.: Software Engineering—ESEC/FSE '99: Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering. Volume

1687 of Lecture Notes in Computer Science., Toulouse, France, Springer-Verlag (1999) 146–162
3. Hamon, G., de Moura, L., Rushby, J.: Generating efficient test sets with a model checker. In: 2nd International Conference on Software Engineering and Formal Methods, Beijing, China, IEEE Computer Society (2004) 261–270
4. Saïdi, H., Graf, S.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: Computer-Aided Verification, CAV '97. Volume 1254 of Lecture Notes in Computer Science., Haifa, Israel, Springer-Verlag (1997) 72–83
5. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with Blast. In: Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN). Volume 2648 of Lecture Notes in Computer Science., Springer-Verlag (2003) 235–239
6. Ball, T., Kupferman, O., Yorsh, G.: Abstraction for falsification. [33] 67–81
7. Xia, S., Di Vito, B., Muñoz, C.: Toward automated test generation for engineering applications via predicate abstraction. In: 20th IEEE International Conference on Automated Software Engineering (ASE'02), Long Beach, CA, IEEE Computer Society (2005) To appear.
8. Boppana, V., Rajan, S.P., Takayama, K., Fujita, M.: Model checking based on sequential ATPG. [34] 418–430
9. Ho, P.H., Shiple, T., Harer, K., Kukula, J., Damiano, R., Bertacco, V., Taylor, J., Long, J.: Smart simulation using collaborative formal simulation engines. In: International Conference on Computer Aided Design (ICCAD), Jan Jose, CA, Association for Computing Machinery (2000) 120–126
10. Barrett, C., de Moura, L., Stump, A.: SMT-COMP: Satisfiability modulo theories competition. [33] 20–23
11. de Moura, L., Rueß, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In Voronkov, A., ed.: 18th International Conference on Automated Deduction (CADE). Volume 2392 of Lecture Notes in Computer Science., Copenhagen, Denmark, Springer-Verlag (2002) 438–455
12. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, Association for Computing Machinery (2002) 123–122
13. Jéron, T., Morel, P.: Test generation derived from model-checking. [34] 108–121
14. Tiwari, A.: Abstractions for Hybrid Systems, Computer Science Laboratory, SRI International, Menlo Park, CA. (2004) Combines several conference papers: available at http://www.csl.sri.com/~tiwari/new.pdf.
15. Ben-David, S., Gringauze, A., Sterin, B., Wolfsthal, Y.: PathFinder: A tool for design exploration. In: Computer-Aided Verification, CAV '2002. Volume 2404 of Lecture Notes in Computer Science., Copenhagen, Denmark, Springer-Verlag (2002) 510–514
16. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: International Conference on Functional Progamming, Montreal, Canada, Association for Computing Machinery (2000) 268–279
17. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: 2nd International Conference on Software Engineering and Formal Methods, Beijing, China, IEEE Computer Society (2004) 230–239
18. Sullivan, K., Yang, J., Coppit, D., Khurshid, S., Jackson, D.: Software assurance by bounded exhaustive testing. In: International Symposium on Software Testing and Analysis (ISSTA), Boston, MA, Association for Computing Machinery (2004) 133–142

19. Beck, K.: Test Driven Development: By Example. Addison-Wesley (2002)
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.: Extreme model checking. In: Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday. Volume 2772 of Lecture Notes in Computer Science., Springer-Verlag (2004) 332–358
21. Kosmatov, N., Legeard, B., Peureux, F., Utting, M.: Boundary coverage criteria for test generation from formal models. In: 15th International Symposium on Software Reliability Engineering (ISSRE'04), Saint-Malo, France, IEEE Computer Society (2004) 139–150
22. Weyuker, E., Goradia, T., Singh, A.: Automatically generating test data from a Boolean specification. IEEE Transactions on Software Engineering **20** (1994) 353–363
23. Kuhn, D.R.: Fault classes and error detection capability of specification-based testing. ACM Transactions on Software Engineering and Methodology **8** (1999) 411–424
24. Tsuchiya, T., Kikuno, T.: On fault classes and error detection capability of specification-based testing. ACM Transactions on Software Engineering and Methodology **11** (2002) 58–62
25. Okun, V., Black, P.E., Yesha, Y.: Comparison of fault classes in specification-based testing. Information and Software Technology **46** (2004) 525–533
26. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: STG: a symbolic test generation tool. In Katoen, J.P., Stevens, P., eds.: Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference, TACAS 2002. Volume 2280 of Lecture Notes in Computer Science., Grenoble, France, Springer-Verlag (2002) 470–475
27. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, Association for Computing Machinery (2002) 112–122
28. Heimdahl, M.P., George, D., Weber, R.: Specification test coverage adequacy criteria = specification test generation *In*adequacy criteria? In: High-Assurance Systems Engineering Symposium, Tampa, FL, IEEE Computer Society (2004) 178–186
29. Tretmans, J., Belinfante, A.: Automatic testing with formal methods. In: EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review, Barcelona, Spain, EuroStar Conferences, Galway, Ireland (1999)
30. Rusu, V.: Verification using test generation techniques. In Eriksson, L.H., Lindsay, P., eds.: Formal Methods Europe (FME'02). Volume 2391 of Lecture Notes in Computer Science., Copenhagen, Denmark, Springer-Verlag (2002) 252–271
31. Hamon, G., de Moura, L., Rushby, J.: Automated test generation with SAL. Technical note, Computer Science Laboratory, SRI International, Menlo Park, CA (2004) Available at http://www.csl.sri.com/users/rushby/abstracts/sal-atg.
32. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N.: Integrating verification components. These proceedings (2005)
33. Etessami, K., Rajamani, S.K., eds.: Computer-Aided Verification, CAV '2005. Volume 3576 of Lecture Notes in Computer Science., Edinburgh, Scotland, Springer-Verlag (2005)
34. Halbwachs, N., Peled, D., eds.: Computer-Aided Verification, CAV '99. Volume 1633 of Lecture Notes in Computer Science., Trento, Italy, Springer-Verlag (1999)

SRI International

CSL Technical Note ● September 2005

Automated Test Generation with SAL

Grégoire Hamon, Leonardo de Moura and John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

This research was supported by NASA Langley Research Center under contract NAS1-00079 and Cooperative Agreement NCC-1-377 with Honeywell Tucson.

Abstract

We describe `sal-atg`, a tool for automated test generation that will be distributed as part of the next release of SAL. Given a SAL specification augmented with Boolean *trap variables* representing test goals, `sal-atg` generates an efficient set of tests to drive the trap variables to `TRUE`; SAL specifications are typically instrumented with trap variables representing structural coverage criteria during automatic translation from a higher-level source notation, such as RSML^{-e} or Stateflow.

We describe extensions to the method of test generation that use *conjunctions* of trap variables; we show how these can be used to provide boundary coverage and to encode *test purposes*. We also describe how the output of the tool can be customized to the requirements of the test harness concerned. We describe experiments with `sal-atg` on realistic examples and preparations for evaluating the quality of tests generated using the experimental framework of Heimdahl, George and Weber [HGW04].

Contents

List of Figures

1 Introduction

We describe the prototype version of `sal-atg`, a new member of the SAL tool suite [dMOR+04b] that performs automated generation of efficient test sets using the method described in [HdMR04]. The prototype is available for beta testing now from `http://sal.csl.sri.com/pre-release`; following some planned refinements and feedback from users, it will be released as a standard part of the SAL tool suite.

In this introductory section, we illustrate straightforward use of `sal-atg`; in Section 2 we describe its operation in more detail and present some examples; in Section 3, we describe extensions that allow a test engineer to specify more complex tests, including conjunctions of goals and "test purposes," and in Section 4 we describe how the output of `sal-atg` can be customized to suit the requirements of the test environment in which it is to be used.

1.1 Basic Test Generation with `sal-atg`

The idea of automated test generation is to construct a sequence of inputs that will cause the system under test (SUT) to exhibit some behaviors of interest, called the *test goals*. The test goals may be derived from requirements, from the domains of input variables (e.g., just inside or just outside the boundary of acceptable values), from the structure of the SUT or its specification (e.g., branch coverage in the specification), or from other considerations. The `sal-atg` tool has no set of test goals built in, but instead generates test sequences from a SAL specification for the SUT that has been augmented with *trap variables* that encode the chosen test goals. Trap variables are Boolean state variables that are initially `FALSE` and are set `TRUE` when some test goal is satisfied. For example, if the test goals are to achieve state and transition coverage, then each state and transition in the specification will have a trap variable associated with it, and these will be set `TRUE` whenever their associated state or transition is encountered or taken. Trap variables may be *latching* or *nonlatching*: in the former case, they remain `TRUE` once their associated test goal has been satisfied, while in the latter they remain `TRUE` only as long as the current state satisfies the goal. For example, if a test goal requires a certain control state to be visited, a latching trap variable will be set and remain `TRUE` once that control state is encountered, whereas a nonlatching trap variable will be set `TRUE` when that control state is encountered, but will return to `FALSE` when the control state takes another value. It is sometimes easier to program latching trap variables, and the test generator can check for them more efficiently (it only need look in the final state of a putative test sequence), but when conjunctions of trap variables are employed (see Section 3) they should generally be nonlatching.

As SAL is an intermediate language, we expect that generation and manipulation of trap variables will be part of the automated translation to SAL from the source notation. For example, our prototype translator from Stateflow to SAL [HR04] can automatically insert trap variables for state and transition coverage in the Stateflow specification.

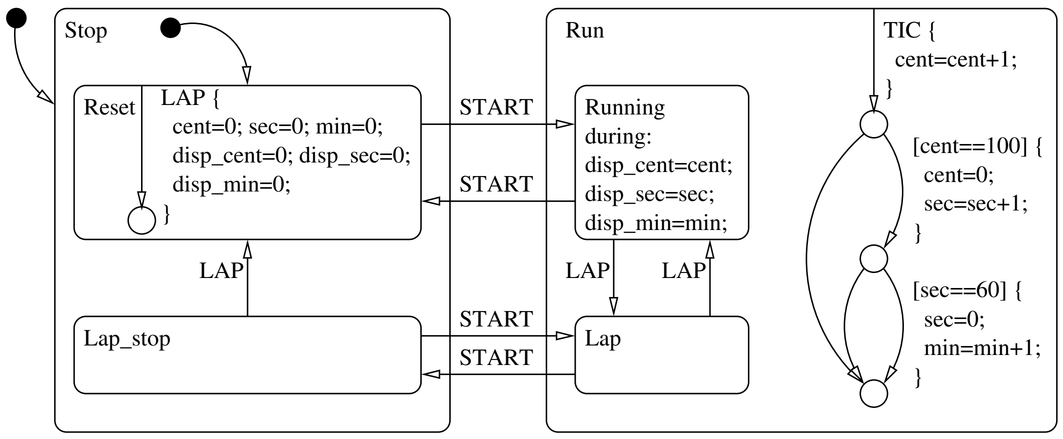

Figure 1: A simple stopwatch in Stateflow

As an illustration, Figure 1 shows the Stateflow specification for a stopwatch with lap time measurement; state coverage corresponds to visiting each of the states (i.e., boxes) and each of the junctions (i.e., small circles), and transition coverage corresponds to taking each of the arcs. To generate test cases using `sal-atg`, we first translate this example into the SAL language. We use the simplified hand-translated specification shown in Figures 2 and 3, rather than the specification generated by our Stateflow translator. This hand-translated SAL specification does not preserve the hierarchical states of the original Stateflow, and is therefore an unfaithful translation, but it is correspondingly much simpler and better suited for exposition here than the faithful mechanical translations.

The first part of the SAL translation, shown in Figure 2, begins by introducing the types needed for the specification. The stopwatch itself is specified in the `clock` module; this has three local variables (`min`, `sec`, and `cent`) that record the state of its counter, and one (`pc`) that records the currently active state. The stopwatch is driven by `events` at its `ev` input variable (where the values `TIC`, `START`, and `LAP` respectively represent occurrence of a timer tick, or pressing the start or lap button), while the output of the module is given by the three variables (`disp_min`, `disp_sec`, and `disp_cent`) that represent its display. The Boolean variables `s1, s2, s3, t0, ...t10` are trap variables added for test generation purposes. Notice that these declarations and other SAL code added for test generation are shown in blue.

The behavior of the `clock` module is specified by the transition relation specified in Figure 3 by means of a series of guarded commands. For example, in the `reset` state, a `LAP` event sets the display variables to zero, while a `START` event causes the state to change to `running`. The six following guarded commands similarly enumerate the behavior of the stopwatch for each combination of the `LAP` and `START` events in its other three states.

```
stopwatch: CONTEXT =
BEGIN
  ncount: NATURAL = 99;
  nsec: NATURAL = 59;
  counts: TYPE = [0..ncount];
  secs: TYPE = [0..nsec];
  states: TYPE = {running, lap, reset, lap_stop};
  event: TYPE = {TIC, LAP, START};

clock: MODULE =
BEGIN
INPUT
  ev: event
LOCAL
  cent, min: counts,
  sec: secs,
  pc: states,
  s1, s2, s3: BOOLEAN,
  t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10: BOOLEAN
OUTPUT
  disp_cent, disp_min: counts,
  disp_sec: secs
INITIALIZATION
  cent = 0;
  sec = 0;
  min = 0;
  disp_cent = 0;
  disp_sec = 0;
  disp_min = 0;
  pc = reset;
  s1 = FALSE;  s2 = FALSE;  s3 = FALSE;

  t0 = FALSE;  t1 = FALSE;  t2 = FALSE;  t3 = FALSE;  t4 = FALSE;
  t5 = FALSE;  t6 = FALSE;  t7 = FALSE;  t8 = FALSE;  t9 = FALSE;
  t10 = FALSE;

...continued
```

Figure 2: First part of SAL translation of the stopwatch

```
TRANSITION
[
  pc = reset AND ev = LAP -->
    disp_cent' = 0; disp_sec' = 0;  disp_min' = 0;
    pc' = pc;  t0' = TRUE;
[]
  pc = reset AND ev = START -->
    pc' = running;  s1' = TRUE; t1' = TRUE;
[]
  pc = running AND ev = LAP -->
    pc' = lap;  s2' = TRUE; t2' = TRUE;
[]
  pc = running AND ev = START -->
    pc' = reset;  t3' = TRUE;
[]
  pc = lap AND ev = LAP -->
    pc' = running;  s1' = TRUE; t4' = TRUE;
[]
  pc = lap AND ev = START -->
    pc' = lap_stop;  s3' = TRUE; t5' = TRUE;
[]
  pc = lap_stop AND ev = LAP -->
    pc' = reset;  t6' = TRUE;
[]
  pc = lap_stop AND ev = START -->
    pc' = lap;  s2' = TRUE; t7' = TRUE;
[]
  ev = TIC AND (pc = running OR pc = lap) -->
    cent' = IF cent /= ncount THEN cent+1 ELSE 0 ENDIF;
    t8' = IF cent' /= cent THEN TRUE ELSE t8 ENDIF;
    sec' =  IF cent /= ncount THEN sec
              ELSIF sec /= nsec THEN sec+1 ELSE 0 ENDIF;
    t9' = IF sec' /= sec THEN TRUE ELSE t9 ENDIF;
    min' =  IF cent /= ncount OR sec /= nsec THEN min
              ELSIF min /= ncount THEN min+1 ELSE 0 ENDIF;
    t10' = IF min' /= min THEN TRUE ELSE t10 ENDIF;
    disp_cent' = IF pc = running THEN cent' ELSE disp_cent ENDIF;
    disp_sec' = IF pc = running THEN sec' ELSE disp_sec ENDIF;
    disp_min' = IF pc = running THEN min' ELSE disp_min ENDIF;
[]
ELSE -->
]
END;

END
```

Figure 3: Final part of SAL translation of the stopwatch

The final guarded command specifies the behavior of the variables representing the counter in response to `TIC` events (corresponding to the flowchart at the right of Figure 1).

The Boolean variables `s1`, `s2`, and `s3` are latching trap variables for state coverage and are set `TRUE` when execution reaches the `running`, `lap`, and `lap_stop` states, respectively. The variables `t0` ... `t10` are likewise latching trap variables for the various transitions in the program. Trap variables obviously increase the size of the representations manipulated by the model checkers (requiring additional BDD or SAT variables), but they add no real complexity to the transition relation and their impact on overall performance seems negligible.

We can generate tests for this specification using the following command.[1]

```
sal-atg stopwatch clock stopwatch_goals.scm -ed 5 --incremental
```

Here `stopwatch` is the name of the SAL context concerned, `clock` is the name of the module, and the test goals (i.e., trap variables) are specified in the Scheme source file[2] `stopwatch_goals.scm` whose contents are as follows.[3]

```
(define goal-list '(
"s1" "s2" "s3"
"t0" "t1" "t2" "t3" "t4" "t5" "t6" "t7" "t8" "t9" ; "t10"
))
```

The items in black define a list in the Scheme language, while the items in blue enumerate the trap variables for the test goals concerned. Note that a semicolon introduces comments in Scheme, so the trap variable `t10` is actually excluded from this list. The argument `-ed 5` instructs the tool to use a maximum search depth of 5 when seeking to extend a test to reach a new goal, while the flag `--incremental` instructs it to undertake the search incrementally (i.e., first to depth 1, then 2, and so on).

The inputs described produce the 17-step test case shown in Figure 4 in under 5 seconds (without the `--incremental` parameter, the test case is 19 steps long). In this case, `sal-atg` fails to generate a test to satisfy the test goal represented by the trap variable `t9`. This goal corresponds to the middle junction in the flowchart to the right of Figure 1 and it requires a test containing 100 `TIC` inputs to satisfy it. The parameters supplied to `sal-atg` do not allow a search to this depth. If, instead, we supply the following parameters, then all goals are satisfied.

```
sal-atg stopwatch clock stopwatch_goals.scm -ed 5 -id 101 --latching
```

Here, the parameter -id 101 allows `sal-atg` to search to a depth of 101 in seeking the initial segment to a path. With this modification, a test set comprising two tests is

[1]A platform-independent GUI will soon be available for all SAL tools.

[2]Scheme is the implementation language for the SAL toolkit; the reference for this language is [KCe98].

[3]In a forthcoming release of SAL, we will extend the SAL language to allow specification of test goals, but for the present, these must be supplied in a Scheme file in the manner shown here.

```
1 tests generated; total length 17
1 unreached test goals:(t9)
========================
Path
========================
Step 0:
 ev = LAP
Step 1:
 ev = TIC
Step 2:
 ev = START
Step 3:
 ev = TIC
Step 4:
 ev = LAP
Step 5:
 ev = LAP
Step 6:
 ev = LAP
Step 7:
 ev = START
Step 8:
 ev = LAP
Step 9:
 ev = TIC
Step 10:
 ev = START
Step 11:
 ev = START
Step 12:
 ev = LAP
Step 13:
 ev = START
Step 14:
 ev = LAP
Step 15:
 ev = START
Step 16:
 ev = START
Step 17:
 ev = TIC
```

Figure 4: Test inputs produced for the stopwatch

generated (in less than 10 seconds); the first is a test of length 19 similar to that of Figure 4, and the second is sequence of length 101 that discharges `t9`. Notice that we have added the `--latching` parameter, which allows `sal-atg` to apply a minor optimization for latching trap variables, but have dropped the `--incremental` parameter: this option is expensive when large search depths are involved.[4]

In the next section, we describe these and other options to `sal-atg`, and illustrateatics of their use.

2 Using `sal-atg`

Test generation using model checkers is well-known; the traditional method generates a separate test for each test goal and results in inefficient test sets having much redundancy and many short tests. Not only are short tests inefficient, there is some evidence that they are poor at revealing bugs [HUZ99, HGW04]. The method of `sal-atg` uses the SAL model checkers, but in a novel way [HdMR04] that generates test sets that are efficient, and that we hope will also be more revealing.

The technique used to generate test cases in `sal-atg` is illustrated in Figure 5. Here, the large oval represents the statespace of the SUT, while the smaller ovals represent test goals or, more concretely, regions of the statespace where the corresponding trap variables become `TRUE`. Test cases are sequences of inputs that drive the SUT along paths, represented by the colored lines, that visit the test goals.

The method of operation of `sal-atg` is to start at an initial state (indicated by **i**), then use model checking to construct an *initial test segment* to any unvisited test goal: for example, that represented by the dashed green line to **a**. From there, it uses further model checking to construct an *extension test segment* to any other unreached goal: for example, that represented by the first part of the solid green line from **a** to **b**. This extension process is repeated until the model checker is unable to reach any further goals; the sequence of inputs that drives the system along the path (i.e., concatenation of segments) is a test case. For example, the input sequence that drives the SUT along the path represented by concatenation of the dashed and solid green lines is a test case that discharges the test goals **a**, **b**, **c**, **m**, **d**, and **e**. Notice that a single segment may discharge multiple goals if it happens to reach a state that is in their intersection (e.g., **c** and **m** here).

Next, `sal-atg` returns to some earlier state and attempts to construct additional paths to visit any remaining test goals. There is a tension between returning to a recent state (e.g., that associated with discharging the test goal **d**), from which it might be feasible to reach a nearby goal (e.g., **n**), and the desire to avoid redundancy in the tests (e.g., a segment from **d** to **n** would come at the expense of repeating the part of the test that drives the SUT along the path from **i** through **a**, **b**, and **c**, to **d**). Currently, `sal-atg` offers two options when it is no longer able to extend the current path. By default, it returns to a start state and attempts

[4]We could use the `--incrext` option, which restricts incremental construction to extension segments.

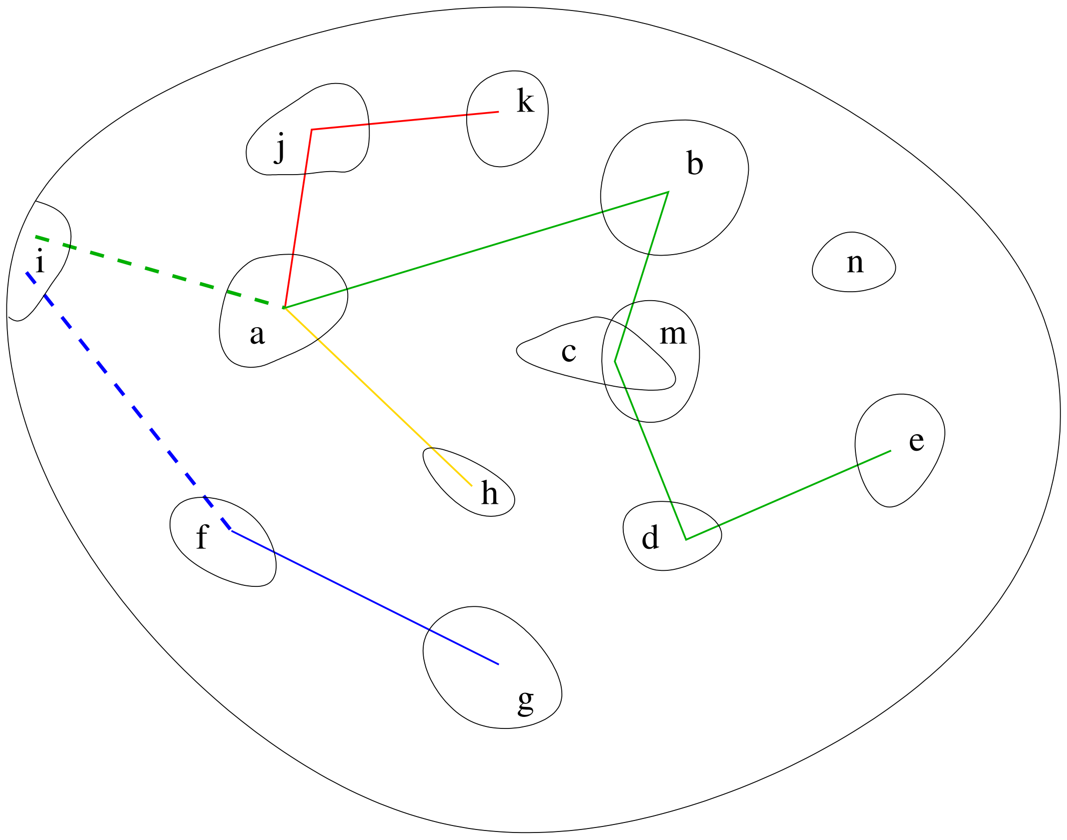

Figure 5: Operation of `sal-atg`

to construct a new initial segment (e.g., that represented by the dashed blue line to **f**), and then attempts to extend that (represented by the blue line to **g**). However, if the `--branch` option is enabled, it first returns to the end of the current initial segment and attempts to find new extensions from there (represented by the red line to **j** and **k**, and the yellow line to **h**).

The idea behind the branching option is that considerable resources may be expended constructing an initial segment (e.g., using symbolic model checking or a large search depth) to some remote part of the state space, and this hard-won access should then be "mined" for as many test goals as possible. The argument against using this option is that returning to a starting state might produce shorter or less redundant tests. In the future we may explore using the branching option only as a last resort. Although none of the examples we have seen so far require more sophisticated path search strategies, we may in the future also consider using arbitrary known paths as initial segments in attempts to reach otherwise unattainable test goals: for example, once all the other goals have been removed from consideration, reaching goal **n** from an initial state may be beyond the capacity of the model checker, whereas it might be within range from the path to **b**.

The following parameters are used to control the search performed by `sal-atg`

`--branch`: when it is no longer possible to extend the current path, `sal-atg` returns by default to an initial state and attempts to construct a new initial segment; this option instructs it instead to return to the current initial segment and attempt to extend that.

`--smcinit`: by default, `sal-atg` uses bounded model checking to construct the initial segments; this option instructs it to use symbolic model checking instead. Bounded model checking is generally faster for specifications with large state spaces, but symbolic can often search deeper in smaller specifications.

`-id` n: the search for an initial segment is limited to depth n (default 8). When `--smcinit` is specified, the special value `-id 0` indicates unlimited search depth.

`-ed` n: the search for extension segments is limited to depth n (default 8). Extensions always use bounded model checking; `-ed 0` means that no extensions will be generated (so each test is generated as an initial segment—this is how test generation with model checkers is conventionally performed).

`-md` n: the search for extension segments always considers extensions of depth at least n (default 0). This option can be used to force longer tests (though some goals may then become unreachable because the test is forced to extend beyond the state that gives access to them).

`--incremental`: this option causes bounded model checking to operate incrementally: it divides the initial and extension depths by 10 and steps the corresponding search depth by those amounts. This option generally generates shorter tests and is often able to reach more test goals. It may reduce generation time (if many tests are found at small depths) or increase it (if many unsuccessful searches are performed at large intermediate depths).

`--incrext`: this is similar to the `--incremental` option but applies only to extension segments.

`--incrinit`: this is similar to the `--incremental` option but applies only to initial segments.

`--noprune`: a segment may happen to reach a state that discharges several test goals simultaneously (e.g., **c** and **m** in Figure 5). By default, `sal-atg` removes all such goals from further consideration; this option will remove only one. This option can therefore be used to increase the number, or the length, of tests generated. The minimum search depth (`-md` n) should be nonzero when this option is used—otherwise, the model checker will find extensions of length zero and vitiate its purpose.

`--latching`: by default, trap variables are assumed to be nonlatching and so `sal-atg` must scan the whole of the most recently added segment to check for any trap variables that have become `TRUE`. This option informs `sal-atg` that trap variables are latching, so that it need inspect only the final state of each segment. This is a minor optimization.

`--noslice`: by default, `sal-atg` slices the specification each time it starts a new initial segment; this option disables this behavior. Slicing reduces the size of the specification and can speed the search.

`--innerslice`: this option causes `sal-atg` to slice the specification each time it constructs an extension.

`-s` *`solver`*: by default `sal-atg` uses ICS as its SAT solver. This option allows use of other solvers; the *siege* solver (see `http://www.cs.sfu.ca/~loryan/personal/`) is often particularly fast.

`--fullpath`: by default, `sal-atg` prints the values of only the input variables that drive the tests; this option causes `sal-atg` to print the full state at each step. Note that slicing causes variables that are irrelevant to the generation of the tests to receive arbitrary values, so it is necessary to specify `--noslice` if the values of these variables are of interest.

`--testpurpose`: this is described in Section 9 on page 29.

`--fullpath`: by default, `sal-atg` prints the

`-v` n: sets the verbosity level to n. This controls the level of output from many SAL functions; `sal-atg` provides a running commentary on its operations when $n \geq 1$.

In addition to the parameters listed above, which are specific to test generation, `sal-atg` also takes other parameters, such as those concerning variable ordering in symbolic model checking; the command `sal-atg --help` will provide a list.

Next, we describe three examples that illustrate some of the parameters to `sal-atg` and give an indication of its performance.

2.1 Stopwatch

In the introduction, we saw that `sal-atg` can generate test cases for all test goals (with `t10` excluded) for the stopwatch using the parameters `-id 101 -ed 5`. By default, `sal-atg` uses bounded model checking to construct both the initial segment and the extensions. We can instruct it to use symbolic model checking for the initial segments by adding the parameter `--smcinit` as follows.

```
sal-atg stopwatch clock stopwatch_goals.scm -ed 5 -id 101 --smcinit
```

With these parameters, a test set comprising two tests is generated in less than 5 seconds; the first is a test of length 17 similar to that of Figure 4, and the second is test of length 101 that discharges `t9`. Symbolic model checking is naturally incremental (i.e., it constructs the shortest counterexample) and is unaffected by the `--incremental` parameter; however, if this parameter is restored (recall, it was removed when we increased the initial depth to 101 while using bounded model checking) it will affect the way extensions are produced (these are always generated by bounded model checking) and the length of the first test case is thereby reduced from 17 to 16.

Observant readers may wonder why, if the incremental flag was used, this and the test case of Figure 4 contain superfluous `TIC` events, which cannot contribute to satisfaction of the test goals in the Statechart on the left of Figure 1. The explanation is that the guarded commands in the transition relation of Figure 1 use the *current* inputs to determine the *next* values of local and output values. Thus, in constructing a segment in which a certain trap variable becomes `TRUE` in its last step, the model checker must exhibit *some* input value for that last step. This input value is chosen arbitrarily and may fortuitously start the next segment on a good or a bad path, or it may be irrelevant (e.g., a TIC). SAL allows specifications in which the *next* values of input variables are used (i.e., the primed forms of input variables appear in guards and on the right hand side of assignments). This capability must be used with great caution and understanding of its consequences (we have seen many specifications rendered meaningless in this way), but it does eliminate the need for an arbitrary input at the end of each segment. If this example is changed to use `ev'` in the guards, then incremental search will generate a test case for the statechart of length only 12

When symbolic model checking is used, it is not necessary to set a bound on the initial search depth (other than to limit the resources that may be used). The special value `-id 0` is used to indicate this. Thus, the following command achieves the same result as the previous one.

```
sal-atg stopwatch clock stopwatch_goals.scm -ed 5 -id 0 --incremental --smcinit
```

If the comment that excludes `t10` from the list in `stopwatch_goals.scm` is now removed, then this same invocation of `sal-atg` will discharge that goal also (in a total time of 106 seconds), adding a third test of length 6,001 to the existing two.

If the extension depth, as opposed to the initial depth, is set to zero, then this has the effect of disabling extensions. For example, the following command generates 9 tests, with total length 37, that discharge all goals except `t9` and `t10`.

```
sal-atg stopwatch clock stopwatch_goals.scm -ed 0 --incremental
```

2.2 Shift Scheduler

Our next example is a shift scheduler for a four-speed automatic transmission that was made available by Ford as part of the DARPA MoBIES project.[5] The example was provided as a model in Matlab Simulink/Stateflow; the Stateflow component is shown in Figure 6: it has 23 states and 25 transitions.

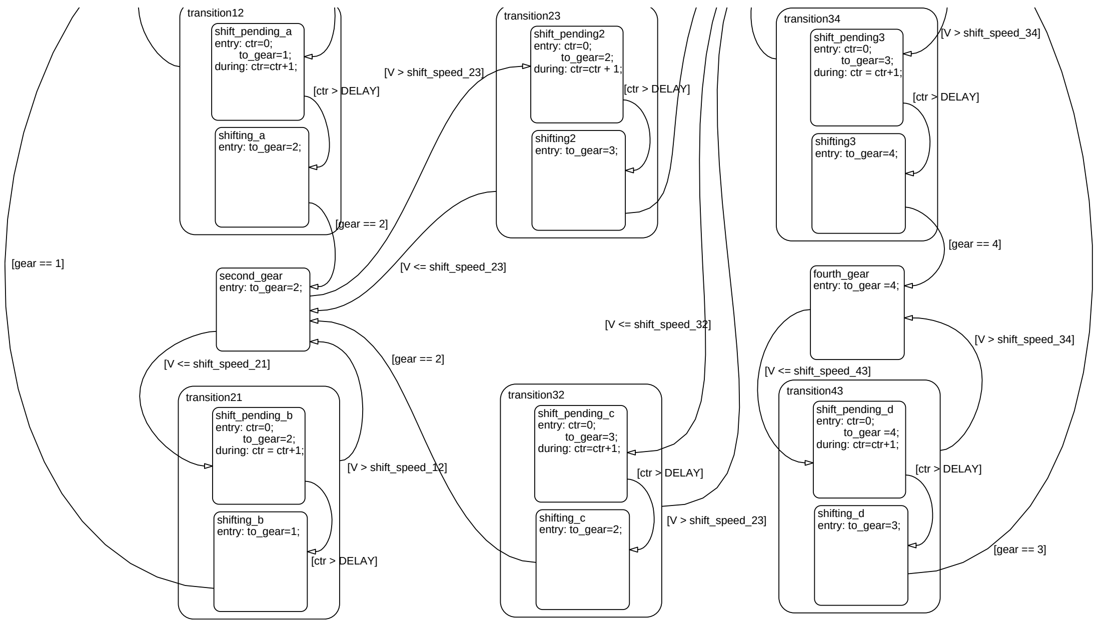

Figure 6: Stateflow model for four-speed shift scheduler

We converted the Stateflow component of the Matlab `.mdl` file for this example into a SAL file `trans_ga.sal` using a prototype translator based on the Stateflow semantics presented in [HR04]. Several of the inputs to this example are real numbers; we redefined

[5]See `vehicle.me.berkeley.edu/mobies/`.

the REAL type to a small integer range for the purpose of test generation by adding the following declaration to the SAL file.

```
 REAL: TYPE = [-10..100];
```

The Stateflow component of the Matlab model is not really self-contained in this example: it has several open inputs that are constrained by the surrounding Simulink blocks. In particular, the `shift_speed_ij` inputs that determine when a shift from gear i to j should be scheduled are driven from a single `torque` input and therefore cannot change independently in the actual context of use. As we do not have a translator from Simulink to SAL, we wrote the `constraints` module shown below by hand and composed it synchronously with the module `main` that was produced by the Stateflow translator to yield a `system` module. The composition simply drives all the `shift_speed_ij` inputs from a common `torque` input, which is constrained to be positive; the gear input is also constrained to take only values `1..4`.

```
constraints: MODULE =
BEGIN
OUTPUT
      shift_speed_21_54 : REAL,
      shift_speed_34_59 : REAL,
      V_52 : REAL,
      shift_speed_23_57 : REAL,
      shift_speed_12_53 : REAL,
      gear_51 : [-128..127],
      shift_speed_43_60 : REAL,
      shift_speed_32_58 : REAL
INPUT
 torque: [0..127],
 velocity: [0..127],
 gear: [1..4]
TRANSITION
      shift_speed_21_54' = torque;
      shift_speed_34_59' = torque;
      V_52' = velocity;
      shift_speed_23_57' = torque;
      shift_speed_12_53' = torque;
      gear_51' = gear;
      shift_speed_43_60' = torque;
      shift_speed_32_58' = torque;
END;

system: MODULE = main || constraints;
```

Our Stateflow translator optionally adds trap variables for state and transition coverage to the SAL module that it generates, and also provides the content for the file `trans_ga_goals.scm`, which is shown below, that communicates these to `sal-atg`.

```
(define goal-list '(
  "latch_shifting_24" "latch_shifting_23" "latch_shifting_22"
  "latch_shift_pending_21" "latch_shift_pending_20"
  "latch_shift_pending_19" "latch_transition21_18"
  "latch_transition32_17" "latch_transition43_16"
  "latch_second_gear_15" "latch_fourth_gear_14" "latch_shifting3_13"
  "latch_shifting2_12" "latch_shifting_11" "latch_shift_pending3_10"
  "latch_shift_pending2_9" "latch_shift_pending_8"
  "latch_transition34_7" "latch_transition23_6" "latch_transition12_5"
  "latch_first_gear_4" "latch_third_gear_3"
  "latch_trans_v1_1_trans_slow_trans_slow_torques_shift_scheduler_shift_schedule_2"
  "latch_transition_30" "latch_transition_29" "latch_transition_28"
  "latch_transition_27" "latch_transition_26" "latch_transition_25"
  "latch_transition_49" "latch_transition_48" "latch_transition_47"
  "latch_transition_46" "latch_transition_45" "latch_transition_44"
  "latch_transition_43" "latch_transition_42" "latch_transition_41"
  "latch_transition_40" "latch_transition_39" "latch_transition_38"
  "latch_transition_37" "latch_transition_36" "latch_transition_35"
  "latch_transition_34" "latch_transition_33" "latch_transition_32"
  "latch_transition_31"
))
```

There are 48 test goals identified in this file. Using the default settings for its parameters, `sal-atg` generates a single test of length 29 that leaves 11 of these goals undischarged. If we increase either the initial depth or the extension depth to 15 then all goals are discharged. For example, the following command line generates two tests of total length 47 that achieve full coverage.

```
sal-atg trans_ga system trans_ga_goals.scm -id 15 --incremental
```

Using the parameter `-ed 15` instead of `-id 15`, full coverage is achieved with a single test of length 47. The generation time in both cases is 38 seconds on a 2 GHz Pentium (there are 311 state bits in the representation sent to the model checker).

2.3 Flight Guidance System

Our third example is a model of an aircraft flight guidance system developed by Rockwell Collins under contract to NASA to support experiments such as this [HRV+03]. This example has previously been used in experiments in test generation at the University of Minnesota [HRV+03, HGW04, HD04] and we were able to build on that work.

The model was originally developed in RSML^{-e}, with test generation performed using nuSMV. We used a SAL version of the specification kindly provided by Jimin Gao of the University of Minnesota who is developing an RSML^{-e} to SAL translator. There is actually a collection of increasingly complex models, but we used only the final and most complex model, whose SAL specification, `FGS05`, has more than 490 state variables. For test goals we worked from the nuSMV input files provided by the University of Minnesota. The test goals for this example target notions of state and transition coverage that differ from the usual control flow interpretation. In this example, a state is interpreted as a specific value assignment to a specific state variable (e.g., `ROLL = Selected`), and a transition is interpreted as a specific guard in one of the tables that constitutes an RSML^{-e} specification taking a specific truth value. There are 246 state coverage goals and 344 transition coverage goals of this kind; in the University of Minnesota experiments, they are represented by CTL formulas in nuSMV syntax such as the following (the first is a state coverage goal, the second a transition coverage goal).

```
G(!(ALTSEL_Selected = Active))

G(((X((!Is_This_Side_Active)))) ->
    X(!ALTSEL_Active = Offside_ALT SEL_Active))
```

To use `sal-atg`, we need to convert CTL formulas such as these into manipulation of trap variables. For state coverage goals, this translation is fairly simple. A CTL formula such as the first one shown above is intended to generate a test for the negation of the property within the `G` operator; that is, in this case, one in which the state variable `ALTSEL_Selected` takes the value `Active`. In SAL, we need to introduce a trap variable that goes `TRUE` when this assignment occurs. We call this trap variable `state5` (because this happens to be the fifth in the list of state coverage goals), initialize it to `FALSE`, and add the following to the `TRANSITION` section of the SAL specification.

```
state5' = state5 OR ALTSEL_Selected = Active;
```

Note that this is added to the `TRANSITION` section *before* the guarded commands; assignments appearing here are executed on every state transition. Hence, `state5` will be set to `TRUE` in the state following the first one in which `ALTSEL_Selected` takes the value `Active`; the disjunction with the previous value of `state5` then causes this value to be latched (i.e., once set `TRUE`, it stays `TRUE`). A minor complication to this encoding is that the SAL and nuSMV translations of the original RSML^{-e} specification are not in straightforward correspondence: they sometimes use different names for the same RSML^{-e} variable or constant. In this case, the SAL value that corresponds to `Active` in the nuSMV translation is `Selected_State_Active`, and so the correct form for this trap variable assignment is the following.

```
state5' = state5 OR ALTSEL_Selected = Selected_State_Active;
```

We have written an awk script (reproduced in the Appendix) that reads the nuSMV formulas for the state coverage goals and produces SAL text defining the corresponding trap variables, their initialization and assignment, and the Scheme file encoding the goal list needed by `sal-atg`. The fragments of SAL text produced by this script are inserted at appropriate points in the file `FGS05.sal` that contains the translation of the RSML^{-e}

model, and the goal list is saved in the file `stategoals.scm`. The following command then invokes `sal-atg` on these inputs.

```
sal-atg FGS05 main stategoals.scm -ed 5 -id 5 --incremental
```

In 61 seconds, this command generates a single test case of length 45 that discharges all but 50 of the 246 state coverage goals. Of the 50 unreached goals, 48 have names ending in `_Undefined` and seem to be present for error detection and are not intended to become activated. The two remaining unreached goals are `When_Selected_Nav_Frequency_Changed = TRUE` and `When_Selected_Nav_Source_Changed = TRUE`, and the University of Minnesota researchers confirm that these are unreachable in the original model also.

There are, therefore, 196 reachable state goals in this example—but a single test of length 45 covers them all. Obviously, many test segments are discharging multiple goals. We can modify the `sal-atg` command so that a separate segment is used for each goal as follows.

```
sal-atg FGS05 main stategoals.scm -ed 5 -id 5 --incremental --noprune -md 1
```

This yields a single test of length 217 in 334 seconds. Of course, each segment may still discharge multiple goals, but only one is removed from the list each time: this provides a simple way to force longer (and possibly more effective) tests.

Another variant is the following, which prevents use of extension segments (by setting the extension depth to zero), and corresponds to the conventional use of model checking to generate test cases.

```
sal-atg FGS05 main stategoals.scm -ed 0 -id 5 --incremental
```

This yields 48 tests of total length 65 in 99 seconds that discharge all 196 reachable goals. Of these tests, 17 are of length 2 and the remaining 31 are of length 1. The very short length of these tests raises doubts about their likely effectiveness (i.e., their ability to detect bugs in an implementation). Heimdahl, George, and Weber at the University of Minnesota found these doubts to be justified [HGW04]: short test cases generated in this way had less ability to detect mutants than randomly generated tests. They suggested three possible explanations for this distressing result: short tests, the structure of the model, and inadequate coverage criteria. Our method (unless hobbled by setting the extension depth to zero) generates long tests, and we hope this leads it into deeper parts of the statespace and provides superior bug detection (there is some prior evidence that long tests are more effective than short [HUZ99]). To investigate this hope, we are collaborating with the researchers at the University of Minnesota to run our tests through their simulator for FGS. This requires modifying the output of `sal-atg` to match the requirements of their simulator and its test harness, and the method for doing this is described in Section 4. The concerns about model structure and inadequate coverage criteria are considered in Section 3.

For the transition coverage goals, the translation from the CTL properties used with nuSMV to the trap variables used in SAL is a little more complicated. A transition goal needs to refer to values in both the current state and the next, and the CTL `X` (next) operator is used for this purpose in nuSMV, as illustrated by the following example where, as before, the overall formula is intended to generate a test case for the negation of the expression within the `G` operator.

```
G(X(!Is_This_Side_Active) ->
    X(!ALTSEL_Active = Offside_ALT SEL_Active))
```

It is easy to represent this as a SAL property (although the native language of most of the SAL model checkers is LTL, they automatically translate the fragment of CTL that is common to LTL), but we need to encode it in the SAL specification language (as operations on a trap variable), not the property language. A natural way to reflect the intent of the `X` operator within a specification is by reference to primed variables, and this suggests the following definition for the trap variable `trans1` associated with this test goal.

```
trans1’ = trans1 OR NOT ((NOT Is_This_Side_Active’)
      => (NOT ALTSEL_Active’ = Offside_ALTSEL_Active’));
```

Automation of this translation is a little tricky and we use the combination of an awk script followed by a sed script to accomplish it. These scripts are shown in the Appendix. The resulting `goal-list` is placed in the file `transgoals.scm`. The following command then invokes `sal-atg` on these inputs.

```
sal-atg FGS05 main transgoals.scm -ed 5 -id 5 --incremental
```

In 98 seconds,[6] this command generates a single test case of length 55 that discharges all but 31 of the 344 transition coverage goals. All of the undischarged goals are of the following form and are plainly unreachable.

```
trans94’ = trans94 OR NOT (NOT (TRUE)
   => (Is_AP_Engaged’ = (AP’ = AP_State_Engaged)));
```

As before, we can increase the length of the test using the `--noprune` option.

```
sal-atg FGS05 main transgoals.scm -ed 5 -id 5 --incremental --noprune -md 1
```

In 857 seconds this generated a single test of length 318 that discharged the same 313 goals as before.

And we can set the extension depth to zero so that `sal-atg` operates like an ordinary model checker.

[6]These experiments were performed on a SAL specification that also included the trap variables for state coverage, making for 1,167 state variables in all. Slicing removed 302 variables (presumably including all the trap variables for state coverage since these were not selected as goals) right away.

```
sal-atg FGS05 main transgoals.scm -ed 0 -id 5 --incremental
```

In 212 seconds, this generated 73 tests having a total length of 84 (63 of length 1, 9 of length 2, and 1 of length 3) that discharge the same 313 goals as before.

3 Test Engineering and Automation

In current practice, test cases are developed by test engineers: they use experience, coverage monitors, and some theory to help them write effective tests. With an automated test generator such as `sal-atg`, the task of the test engineer changes from writing tests to writing *specifications* for tests: the test generator then produces the actual tests. For `sal-atg`, tests are specified through trap variables, and in this section we illustrate ways in which trap variables can be used to specify more searching tests, and we describe how *conjunctions* of trap variables can be used to specify very potent tests.

3.1 Boundary Coverage and Meaningful Impact Tests

The examples we have seen so far use test goals derived from structural coverage criteria. However, simple structural criteria such as state or transition coverage can produce unrevealing tests. Consider, for example, a simple arithmetic comparison such as the following.

```
IF x <= 6 THEN ... ELSE ... ENDIF
```

Transition coverage requires only that we generate one test that takes the `THEN` branch and another that takes the `ELSE` branch, and a test generator that targets this requirement might propose the test set `x = 0` and `x = 8` (assuming the type of `x` includes these values). A human tester, on the other hand, might be concerned that the implementation of this specification could mistakenly substitute < for <= and would therefore propose the test set `x = 6` and `x = 7` to discriminate between the correct implementation and the hypothesized faulty one. Notice that both test sets satisfy transition coverage, but only the second will reveal the hypothesized fault.

There is an extensive literature on methods for generating tests that are likely, or in some cases guaranteed, to detect various kinds of hypothesized faults, but rather few of these methods have been automated. One that has is the "boundary coverage" method [KLPU04], which is implemented in the BZ-Testing-Tool suite and its commercial derivative (see `www.leirios.com`). The boundary coverage method collects the various constraints implied by postconditions, preconditions, and guards, then calculates their "boundary" and selects some test points from (just) inside the boundary, and some from (just) outside. The method of `www.t-vec.com` is similar, although it apparently calculates just the test "vectors" (the input/output cases for each transition), and not the full ("preamble") sequence of inputs needed to bring the system to the point where the vectors are applied. There is evidence that tests generated by boundary coverage are quite revealing.

Now `sal-atg` operates by driving trap variables to `TRUE` and it is used in conjunction with a translator or preprocessor that inserts those trap variables and their associated code into the SAL specification concerned. We will speak generically of the component that inserts the trap variables as the *preprocessor* (while recognizing that it may actually be a translator from some other notation rather than a true SAL-to-SAL preprocessor). We are interested in generating more revealing tests, such as those for boundary coverage, so here we examine whether this can be accomplished by augmenting the preprocessor to set trap variables appropriately, or whether additional capabilities are needed in `sal-atg` itself. We note that it is certainly feasible to augment `sal-atg` to explore the boundaries of arithmetic constraints by substituting the infinite bounded model checker of SAL [dMRS02] for the (ordinary) bounded model checker that is currently employed. The infinite bounded model checker uses the ICS decision procedures [dMOR+04a], whose arithmetic solver has the latent ability to maximize an expression subject to arbitrary constraints.

The motivation for the boundary coverage method is the hypothesis that implementation faults often arise at such boundaries. Boundaries are constructed by predicates appearing in the specification (such as the `x <= 6` example above), so a plausible way to generate tests for a boundary fault hypothesis might be to probe the predicates at branch points and in guards. Suppose we have a branch guarded by a predicate Ψ, and we have a fault hypothesis that this might incorrectly be implemented as the predicate Φ. We want a test that will discriminate these two cases: that is, an assignment to the state variables of Ψ satisfying the formula $\Psi \neq \Phi$. We can cause `sal-atg` to generate a suitable test by simply inserting a trap variable that is set `TRUE` if $\Psi \neq \Phi$ at the point where the guard is evaluated. Disequality on the Boolean type is the same as the `XOR` operator, and we will use this notation henceforth (in the literature on testing this is often written as $\oplus$). In the example above, Ψ is `x <= 6` and Φ is `x < 6`, so our variable will trap the expression `x <= 6 XOR x < 6`; this simplifies to `x = 6` and hence this is the test that will be generated by this trap variable.

There are two ways in which a `sal-atg` preprocessor might automate the generation of trap variables for boundary cases using the ideas presented above. The first method directly automates the approach as presented: for each guard Ψ in the specification, apply some heuristic or table to generate a hypothesized faulty implementation Φ, then generate a variable to trap satisfaction of Ψ `XOR` Φ (we may add this to the tests for transition coverage rather than have a separate test: in this case we would trap $\Psi \wedge (\Psi$ `XOR` $\Phi)$ and $\neg\Psi \wedge (\Psi$ `XOR` $\Phi)$, which simplify to $\Psi \wedge \neg\Phi$ and $\neg\Psi \wedge \Phi$).

The second method expands Ψ into some Boolean combination of more primitive cases, hypothesizes simple faults (e.g., "stuck-at" faults) in each of these, and then generates trap variables whose tests will expose these simple faults. For example, `x <= 6` can, as suggested in [DF93], be expanded to `x < 6 OR x = 6` and we may then set trap variables whose tests will show if either of these disjuncts has an implementation that is invariantly `TRUE` or `FALSE`. Such tests will work by setting things up so that changing the value of one of the disjuncts, while holding the others constant, changes the value of the whole expression (if the implementation has a “stuck at” fault, then the value of the expression will not change, thereby revealing the fault).

The idea that a good testing strategy should check that the subexpressions of complex Boolean expressions independently can affect the outcome is a popular one: it underlies the coverage criterion called MC/DC (Modified Condition/Decision Coverage) [CM94] that the FAA requires for “Level A” airborne software [RTCA92], and it is the basis for the influential “meaningful impact” class of test generation strategies [WGS94]. Let p be an atomic formula (i.e., a Boolean variable or a simple predicate such as $x < y$) that appears in the Boolean expression Φ, then Φ^p_e denotes the modified expression in which e (an arbitrary expression) replaces p in Φ. A test t (i.e., an assignment of truth values to the atomic formulas of Φ) manifests the *meaningful impact* of p in Φ if the test case t' formed by changing the valuation of p (but not of any other atomic formula) in t, also changes the valuation of Φ. This will be so for any t that satisfies Φ `XOR` $\Phi^p_{\neg p}$. Since one of p or $\neg p$ must be `TRUE` in Φ and vice-versa in $\Phi^p_{\neg p}$, this is equivalent to saying that t satisfies $\Phi^p_{\mathtt{TRUE}}$ `XOR` $\Phi^p_{\mathtt{FALSE}}$; this expression is called the *Boolean derivative* (of Φ with respect to p) and is denoted $\frac{d}{dp}\Phi$. To generate meaningful impact tests for Φ, we set pairs of variables that trap satisfaction of $p \wedge \frac{d}{dp}\Phi$ and $\neg p \wedge \frac{d}{dp}\Phi$ for each of the atomic formulas p in Φ (we follow [Kuh99] and [OBY04] in using Boolean derivatives to describe these methods).

Returning to our example $\Phi =$ `x < 6 OR x = 6`, the derivative with respect to the first atomic formula is the negation of the second, and vice-versa. Hence, we require variables that trap each of the following four cases.

- `x < 6 AND NOT x = 6` (this simplifies to `x < 6`)
- `NOT x < 6 AND NOT x = 6` (this simplifies to `x > 6`)
- `x = 6 AND NOT x < 6` (this simplifies to `x = 6`)
- `NOT x = 6 AND NOT x < 6` (this simplifies to the redundant case `x > 6`)

Notice that this second method for testing boundary cases is really comprised of two separate submethods: one that expands predicates into interesting cases, and another that generates trap variables to test for meaningful impact of these cases. The latter mechanism is independently useful, since it will generate trap variables for meaningful impact tests of any compound Boolean expression, not just those formed by expanding expressions to expose boundary conditions.

To summarize this discussion: we believe that it is feasible to generate effective tests for boundary coverage by exploring the predicates appearing in guards and conditions, and we consider this approach more attractive than constraint solving because of its wider applicability (it should work for any predicates, not just arithmetic ones). A preprocessor can set trap variables to explore predicates with respect to explicit fault hypotheses, or it can expand predicates into more primitive combinations and generate tests for meaningful impact in those. We favor the second approach because it seems to have wider utility. Empirical investigation is required to validate these beliefs.

Note, however, that the test sets that will be generated by applying `sal-atg` to a specification with trap variables for meaningful impact differ from those suggested in the literature on this topic. The proposals for testing strategies derived from [WGS94] (e.g., [KTK01]) are concerned with heuristics for making "good" choices among the possibly many ways to satisfy $\frac{d}{dp}\Phi$ or, more particularly, among the many satisfying instances to the set $\{\frac{d}{dp}\Phi \mid p \text{ is an atomic formula of } \Phi\}$ of all the derivatives of Φ: the ideas are that "good" choices may generate larger, or more varied, or more effective sets of tests. In contrast, the trap variables of our approach latch *any* assignments of state variables that satisfy $p \wedge \frac{d}{dp}\Phi$ and $\neg p \wedge \frac{d}{dp}\Phi$ for each atomic formula p. Empirical investigation is needed to determine whether our indifference to the precise satisfying assignment produces test sets of reduced effectiveness. Since our approach is underpinned by the counterexample generation of standard model checking methods, it is not practical to constrain the choice of satisfying assignments; instead, the process is controlled by the selection of the expressions that are monitored by trap variables, and we suspect that the selection heuristics of the meaningful impact strategies can be reproduced, if necessary, by simply trapping more expressions, or by specifying those expressions more precisely.

Observe also that the meaningful impact strategies are ways to *generate* tests, whereas the similarly motivated MC/DC notion is a way to *measure* the coverage of tests. In certification of airborne software, the idea is that tests should be generated by consideration of *requirements* and *specifications* (perhaps using meaningful impact strategies), and their thoroughness is evaluated by monitoring MC/DC coverage on the *code* [HVCR01]. A potential difficulty in our approach to automated test generation for meaningful impact is that our trap variables separately latch $p \wedge \frac{d}{dp}\Phi$ and $\neg p \wedge \frac{d}{dp}\Phi$ and could therefore use *different* assignments to the variables in $\frac{d}{dp}\Phi$, whereas some interpretations of MC/DC require the *same* assignment for each test of the pair [AOH03]. If empirical investigation reveals that it is necessary to force the same assignment to the variables of $\frac{d}{dp}\Phi$ for each test of the pair, then `sal-atg` can be modified to accomplish this.

3.2 Conjunctions of Trap Variables: A Process Scheduler Example

The intuitive interpretation of the "boundaries" in boundary testing is clear for state variables of numeric types, but what about variables of other types? Legeard, Peureux and Utting [LPU02] suggest applying the method to some numeric function of the variables concerned: for example, if a variable is of a set type, then the boundary method could be applied to the cardinality of the set. They illustrate this approach on a scheduler for a set of processes that was originally introduced by Dick and Faivre [DF93]. The scheduler maintains sets of `ready`, `waiting`, and `active` processes, and provides three operations: `new`, which introduces a new process into the `waiting` set; `makeready`, which moves a process from `waiting` to `active` if the system is idle (i.e., no process is currently `ac-`

```
sched: CONTEXT =
BEGIN

n: NATURAL = 4;
pid: TYPE = [1..n];
ops: TYPE  = {new, makeready, swap};

pidset: CONTEXT = sets{pid};

scheduler: MODULE =
BEGIN
INPUT
 op: ops,
 id: pid
OUTPUT
 active, ready, waiting: pidset!set
INITIALIZATION
 active = pidset!emptyset;
 ready = pidset!emptyset;
 waiting = pidset!emptyset;
TRANSITION
[
 op = new AND NOT active(id) AND NOT ready(id) AND NOT waiting(id) -->
   waiting’ = pidset!insert(waiting, id);
[]
 op = makeready AND waiting(id) -->
   waiting’ = pidset!remove(waiting, id);
   active’ = IF pidset!empty?(active)
          THEN pidset!singleton(id)
          ELSE active ENDIF;
   ready’ = IF pidset!empty?(active)
          THEN ready
          ELSE pidset!insert(ready, id)  ENDIF;
[]
([] (q:pid): op = swap AND ready(q)-->
   waiting’ = pidset!union(waiting, active);
   active’ = pidset!singleton(q);
   ready’ = pidset!remove(ready, q);
)
[]
op = swap AND pidset!empty?(ready) -->
   waiting’ = pidset!union(waiting, active);
   active’ = pidset!emptyset;
[]
ELSE -->
]
END;
END
```

Figure 7: The Scheduler Example

```
set_states: TYPE = {empty, single, partfull, full};

set_monitor: MODULE =
BEGIN
INPUT
 mset: pidset!set
LOCAL
 traps: ARRAY set_states OF BOOLEAN
INITIALIZATION
 traps = [[x: set_states] FALSE];
TRANSITION
[
  pidset!empty?(mset) --> traps' = [[x: set_states] x = empty]
[]
  pidset!single?(mset) --> traps' = [[x: set_states] x = single]
[]
  pidset!partfull?(mset) --> traps' = [[x: set_states] x = partfull]
[]
  pidset!full?(mset) --> traps' = [[x: set_states] x = full]
[]
  ELSE -->
]
END;
```

Figure 8: The Set Monitor Module

`tive`), or to `ready` otherwise; and `swap`, which exchanges the currently `active` process for a `ready` one, or leaves the system idle if none were `ready`. A SAL rendition of the example is shown in Figure 7. Processes are identified by small integers of the type `pid` ("process id"), and sets of these (the type `pidset`) are defined by importing the generic `sets` context instantiated for the type `pid`.

To generate thorough tests from this specification, intuition suggests that we should explore not only its local control structure (i.e., transition coverage), but also the special cases for each of the sets that it maintains: for example, the cases where they are empty, contain a single element, are full, or none of these. Legeard, Peureux and Utting [LPU02] approach this through boundary testing on the sum of the cardinalities of the sets, while Dick and Faivre [DF93] expand guards and conditions to distinguish empty from nonempty sets and then probe the compound expressions in a manner related to meaningful impact tests. We could, as suggested in the previous section, imagine a preprocessor that inserts trap variables to automate the second approach.

But given a technology based on model checking, we also have an opportunity to do something quite different: we can add a "monitor" module that observes the state of the various sets and traps the interesting special cases. A suitable module is shown in Figure 8; unlike previous examples, we use here an array of trap variables, rather than a collection of separate variables and, for reasons that will be explained later, these variables are nonlatching (e.g., `traps[empty]` is set `TRUE` in a state in which the `mset` is empty, but it will become `FALSE` in any subsequent state in which the set is nonempty).

The original scheduler module may then be composed with three instances of the `set_monitor` module as shown below; each instance monitors one of the sets in the specification.

```
mon_sched: MODULE =
   (RENAME traps TO atraps, mset TO active  IN set_monitor) ||
   (RENAME traps TO rtraps, mset TO ready   IN set_monitor) ||
   (RENAME traps TO wtraps, mset TO waiting IN set_monitor) ||
   scheduler;
```

We then define the `goal-list` in the file `schedgoals.scm` as follows.[7] The trap variables `atraps[sched!partfull]`, `atraps[sched!full]`, and `rtraps[sched!full]` are absent from the `goal-list` because they are unreachable (it is easy to confirm by model checking that there is at most one active process, and the system will not be idle if there is a ready process).

```
(define goal-list '(
 "atraps[sched!empty]" "atraps[sched!single]"
 "rtraps[sched!empty]" "rtraps[sched!single]" "rtraps[sched!partfull]"
 "wtraps[sched!empty]" "wtraps[sched!single]" "wtraps[sched!partfull]"
 "wtraps[sched!full]"))
```

The following command generates a single test of length 17 that discharges all nine test goals defined by these trap variables.

```
sal-atg sched.sal mon_sched -v 3 --incremental schedgoals.scm
```

Although it targets interesting cases only in the states of the three set variables, this test also achieves transition coverage on the `scheduler` module. [8] In contrast, the test (of length 10) that is generated by targeting simple transition coverage in the `scheduler` module fails to drive any of the sets beyond the `empty` and `singleton` states. The method of Dick and Faivre [DF93] does drive the sets into more interesting states, but Dick and Faivre generated their test (of length 19) by hand.

The boundary coverage method of Legeard, Peureux and Utting generates many more tests than any of these methods because "its aim is to test *each* transition with both minimum and maximum boundary values" [LPU02, page 35, emphasis ours], and this suggests

[7]Note that if we did not rename the `traps` variables, then the variable `atraps` would be referred to as `traps.1.1`, `rtraps` would be `traps.1.2`, and `wtraps` would be `traps.2`. This is because synchronous compositions are represented internally as nested 2-tuples.

[8]To see the coverage achieved on the traps for transition coverage, add the arguments `--fullpath` and `--noslice`; the former allows the values of the trap variables to be inspected in the trace output by `sal-atg`, while the latter prevents these from being sliced away.

an interesting combination of the two approaches used in `sal-atg`: instead of targeting structural coverage in the `scheduler`, or "state coverage" in its set variables, we could target both. We do not mean "both" in the sense of simply taking the union of the two sets of trap variables, but in the sense of taking their *product*: that is, for each structural target in the `scheduler`, we seek tests in which the three sets are taken through their full range of states. This is easily done, because the elements in the `goal-list` targeted by `sal-atg` may either be the names of trap variables (which is the only case we have seen so far), or a *list* of such names, which is interpreted as a goal requiring the *conjunction* of the trap variables appearing in the list.

Figure 9 presents a preprocessed version of the `scheduler` module (the preprocessing was done by hand and the module is called `ischeduler` for instrumented `scheduler`) in which the `IF...THEN...ELSE` expressions have been "lifted" into the guards, and an array of (nonlatching) trap variables for structural coverage has been added.

The test case of length 10 for transition coverage that was previously mentioned was generated by the following command

```
sal-atg sched.sal ischeduler --incremental ischedgoals.scm
```

where `ischedgoals.scm` contains the following definition for `goal-list` (the goal `s[6]` is absent because the transition that it traps is, correctly, unreachable).

```
(define goal-list '(
 "s[1]" "s[2]" "s[3]" "s[4]" "s[5]" "s[7]" "s[8]"))
```

To generate tests from the product of the transition and set coverage goals, we place the following definitions in the file `prodgoals.scm` (`goal-product` is a function defined by `sal-atg` that constructs a list of lists in which each of the inner lists contains one member of the list specified as its first argument, and one member of the list specified as its second argument).

```
(define goal-list1 '(
 "s[1]" "s[2]" "s[3]" "s[4]" "s[5]" "s[7]" "s[8]"))

(define goal-list2 '(
 "atraps[sched!empty]" "atraps[sched!single]"
 "rtraps[sched!empty]" "rtraps[sched!single]" "rtraps[sched!partfull]"
 "wtraps[sched!empty]" "wtraps[sched!single]" "wtraps[sched!partfull]"
 "wtraps[sched!full]"))

(define goal-list (goal-product goal-list1 goal-list2))
```

Then, with `mon_isched` defined follows

```
mon_isched: MODULE =
   (RENAME traps TO atraps, mset TO active  IN set_monitor) ||
   (RENAME traps TO rtraps, mset TO ready   IN set_monitor) ||
   (RENAME traps TO wtraps, mset TO waiting IN set_monitor) ||
   ischeduler;
```

```
ischeduler: MODULE =
BEGIN
INPUT
 op: ops, id: pid
OUTPUT
 active, ready, waiting: pidset!set
LOCAL
 s: ARRAY [1..8] OF BOOLEAN
INITIALIZATION
 active = pidset!emptyset; ready = pidset!emptyset;
 waiting = pidset!emptyset; s = [[x: [1..8]] FALSE];
TRANSITION
[
 op = new AND NOT active(id) AND NOT ready(id) AND NOT waiting(id) -->
   waiting’ = pidset!insert(waiting, id);
   s’ = [[x: [1..8]] x = 1];
[]
 op = new AND (active(id) OR ready(id) OR waiting(id)) -->
   s’ = [[x: [1..8]] x = 7];
[]
 op = makeready AND waiting(id) AND pidset!empty?(active) -->
   waiting’ = pidset!remove(waiting, id);
   active’ = pidset!singleton(id);
   s’ = [[x: [1..8]] x = 2];
[]
 op = makeready AND waiting(id) AND NOT pidset!empty?(active) -->
   waiting’ = pidset!remove(waiting, id);
   ready’ = pidset!insert(ready, id);
   s’ = [[x: [1..8]] x = 3];
[]
 op = makeready AND NOT waiting(id) -->
   s’ = [[x: [1..8]] x = 8];
[]
([] (q:pid): op = swap AND ready(q)-->
   waiting’ = pidset!union(waiting, active);
   active’ = pidset!singleton(q);
   ready’ = pidset!remove(ready, q);
   s’ = [[x: [1..8]] x = 4];
)
[]
op = swap AND pidset!empty?(ready) -->
   waiting’ = pidset!union(waiting, active);
   active’ = pidset!emptyset;
   s’ = [[x: [1..8]] x = 5];
[]
ELSE -->
   s’ = [[x: [1..8]] x = 6];
]
END;
```

Figure 9: The Scheduler Example Augmented with Trap Variables

we generate tests for the "product" case with the following command.

```
sal-atg sched.sal mon_isched --incremental -ed 4 prodgoals.scm
```

In 39 seconds, this generates two tests of length 10 and 39 that discharge 49 of the $7 \times 9 = 63$ test goals. The 14 undischarged goals are genuinely unreachable (one way to confirm this is to add `--smcinit -id 0` to the `sal-atg` command).

It should now be apparent why the trap variables are nonlatching in this example: latching is an optimization that is useful when each trap variable is required to participate in only a single test, but in the product construction, each trap variable participates in multiple tests and we need to know that the conditions trapped by conjoined trap variables are `TRUE` simultaneously.

Conjunctions of test goals allow rather powerful testing strategies to be specified in a succinct way. The "product" construction that it supports is particularly effective. It can be used, as in the scheduler example, to generate tests for two different notions of coverage used in combination. It is also useful in distributed systems, where it allows structural coverage of two different components to be explored in combination. Another potential application is to derive tests from a model in combination with its requirements or properties (expressed as monitors).

Yet another use for conjunctive goals is to support a form of "test purpose." This technique is so useful that `sal-atg` provides specific support for it, and this is described in the following section.

3.3 Test Purposes

Test goals specify what the generated tests should accomplish in terms of visiting certain states, transitions, and combinations of these, but they do not otherwise constrain the tests that are generated. In particular, when powerful technology such as model checking is employed for test generation, it will invariably find the *shortest* test to satisfy any given goal. The shortest tests may exploit some special cases and can satisfy the test goals while leaving much of the behavior of the system unexplored. The flight guidance system, for example, is intended to be one of a pair, one of which is active while the other, inactive, one operates as a hot spare. An active system exhibits interesting behavior: for example, there are some complex rules that determine when it should enter `ROLL` mode. An inactive system, on the other hand, just follows inputs it receives from its active counterpart: so all it needs to enter `ROLL` mode is an input commanding it to do so. A model checker will discover this and will satisfy a test goal to put the system in `ROLL` mode in just two steps: by first making the system inactive, then commanding it to enter `ROLL` mode. This is what Heimdahl, George, and Weber mean when they say that *model structure* may allow generation of unrevealing tests [HGW04]. In the case of the flight guidance system, we might hope to generate better tests by closing the "loophole" described above and requiring that tests should keep the system in the active state. This is an example of what is sometimes called a *test purpose*: a test purpose augments the test goals by describing what *kind* of tests we want to generate.

If we want to restrict the search for tests to those satisfying a particular purpose, it may seem natural to write a module that generates inputs satisfying that purpose: for example, we could write a module that generates arbitrary inputs to the flight guidance system, *except* those that would make it inactive. The composition of the flight guidance system and its generator will then be a closed system (i.e., one that has no free inputs—the inputs of the flight guidance system are closed by linking them to the outputs of the generator). This is standard practice in model checking; for example, in analyzing the OM(1) fault tolerant algorithm described in the SAL tutorial [Rus04], we have a `controller` module that injects faults into other components, while taking care not to inject more faults than we know the algorithm can handle. This is reasonable because the model checker will close the system anyway (i.e., it will synthesize an environment to drive any open inputs in a totally nondeterministic manner), and the distinction between those inputs that are closed by our generator and those that are closed by the synthesized environment is not important in traditional model checking, where the objective is simply to drive the modeled system through all its behaviors of interest while checking satisfaction of given properties.

In test generation, however, we do care which inputs are left open and which are closed, because test cases are the sequences of values supplied to just the open inputs, and these need to comprise exactly those needed to drive the real SUT. Thus, it is not feasible to implement test purposes by writing generator modules that close off some of the inputs.[9] Instead, we implement test purposes by writing *recognizer* modules (sometimes called "synchronous observers"). A recognizer module is one that takes (some of) the same inputs as the modeled SUT (and, if necessary, observes some of its internal variables) and produces a Boolean output that remains `TRUE` as long as the input seen so far satisfies the test purpose concerned; then we modify the test generation process so that it considers only those paths on which the test purpose output remains true.

In environments where the specification is translated to SAL from some other source language (e.g., Stateflow), the monitors can be written in the source language also and translated in the same way. Observe that it is often easier to write monitors to check that a test purpose is being satisfied than it is to construct a generator for that purpose: in effect, the model checker performs constraint satisfaction and automates construction of a suitable generator.

The automation is accomplished by restricting tests to those paths that satisfy the test purpose, and in `sal-atg` this is easily done using conjunctive goals: we simply conjoin the Boolean variable that defines the test purpose with each of the trap variables that define the test goals. This can be specified in Scheme using the `goal-product` function introduced in the previous section, but the construction is so useful that it is supported directly by

[9]Although one could modify the operation of the underlying model checker by telling it that certain local variables should be treated as if they are inputs when constructing counterexamples.

`sal-atg`, where it is invoked by the `--testpurpose` parameter, whose operation is described as follows.

`--testpurpose`: tests are constrained to those paths on which the conjunction of goals defined in the Scheme variable `purpose-list` is `TRUE`.

We illustrate simple use of a test purposes by revisiting the Flight Guidance System example. Later, we describe more ambitious test purposes using the shift scheduler.

Recall, from the introduction to this section, that Heimdahl, George, and Weber proposed that better test cases can be obtained for the flight guidance system by requiring it to remain in the `active` state [HGW04]. To formulate this as a test purpose, we need to introduce a variable that is `TRUE` as long as the FGS has been `active` throughout the test seen so far. The state variable that records whether the FGS is active is `Is_This_Side_Active`; this is a local variable and is not visible outside the module. There are two ways to proceed: we could change this variable to an output, or we could introduce a new output variable that tracks its value. Here, we use the second option and introduce a new Boolean output variable called `Report_Is_This_Side_Active`, and then add the following equation to the `DEFINITION` section.

```
  Report_Is_This_Side_Active = Is_This_Side_Active;
```

We then specify a `monitor` module that takes this variable as an input and produces an output variable `ok` that remains `TRUE` as long as the input is `TRUE`. (Obviously, we could have added `ok` and its associated transition to the `main` module, but we do it this way to illustrate the general method.)

```
monitor: MODULE =
BEGIN
INPUT
  Report_Is_This_Side_Active: BOOLEAN
OUTPUT
  ok: BOOLEAN
INITIALIZATION
  ok = Report_Is_This_Side_Active
TRANSITION
  ok’ = ok AND Report_Is_This_Side_Active
END;

system: MODULE = main || monitor;
```

We synchronously compose `monitor` with the original `main` module, and add the following definition for `purpose-list` to the file `stategoals.scm`.

```
(define purpose-list ’("
   ok"
))
```

We then invoke `sal-atg` as before, but adding the `--testpurpose` parameter.

```
sal-atg FGS05 system stategoals.scm -ed 5 -id 5 --incremental --testpurpose
```

In 65 seconds, we obtain a single test case of length 46 (compared with 45 previously) that discharges the same test goals as before (minus the one to drive `Is_This_Side_Active` to `FALSE`). If we disable extensions, then we see a slightly bigger change.

```
sal-atg FGS05 system stategoals.scm -ed 0 -id 5 --incremental --testpurpose
```

In 114 seconds, this generates 49 tests with total length 72 (31 of length 1, 14 of length 2, 3 of length 3, 1 and of length 4); this compares with 48 tests with total length 65 when the test purpose was not used. Similar results are obtained when we substitute `transgoals.scm` for `stategoals.scm`.

The test purpose illustrated by this example is very simple and has only a modest effect. Test engineers who have a good understanding of the SUT can construct more elaborate test purposes, as we illustrate in the following section.

3.4 Test Purposes: Shift Scheduler Example

We first illustrate use of test purposes on the shift scheduler example. The inputs to the shift scheduler are `torque`, `velocity`, and `gear`: the first of these indicates the current power output of the engine, the second gives the road speed, and the third indicates the gear currently selected by the gearbox; the shift scheduler drives actuators that change clutch pressures and thereby influence the gearbox to select a different gear. The test cases generated by the commands in Section 2.2 have many "discontinuities" in the `gear` input: that is, the currently selected gear may go from 2 to 4 to 1 in successive inputs. We might suppose that a more realistic test sequence would not have these discontinuities, and might therefore propose a test purpose in which the `gear` input changes by at most one at each step. We can implement this purpose by adding the following to the SAL specification of the shift scheduler.

```
monitor: MODULE =
BEGIN
INPUT
 gear: [1..4]
OUTPUT
 continuous: BOOLEAN
INITIALIZATION
 continuous = (gear=1);
TRANSITION
 continuous' = continuous AND (gear - gear' <= 1)
                          AND (gear' - gear <= 1);
END;

monitored_system: MODULE = system || monitor;
```

Here, the `monitor` module takes `gear` as input and produces the Boolean output `continuous`: this output remains `TRUE` as long as the sequence of inputs changes by at most 1 at each step (and starts at 1). The `monitor` is then synchronously composed with the previous `system` to yield the `monitored_system`. We specify that `continuous` is a test purpose by adding the following to the `trans_ga_goals.scm` file.

```
(define purpose-list '(
     "continuous"
))
```

Then we perform test generation with this purpose by the following command.

```
sal-atg trans_ga monitored_system trans_ga_goals.scm -ed 15 --incremental --testpurpose
```

In 45 seconds, this generates a single test of length 49 that discharges all coverage goals. Inspection of the test confirms that the `gear` input changes by at most 1 at each step.

We observe that this test holds the `gear` input constant for long periods (e.g., the first ten inputs are 1, 1, 1, 1, 1, 1, 2, 3, 3, 3) and we might also be interested in a test purpose that requires the `gear` input always to change value from one step to the next. We can add a Boolean output `moving`, initially `TRUE`, to the `monitor` module, and then add the following to the `TRANSITION` section.

```
 moving' = moving AND (gear /= gear');
```

We add `moving` to the purpose list

```
(define purpose-list '(
     "continuous"
     "moving"
))
```

and then invoke `sal-atg` as before.

In 46 seconds, we obtain a single test of length 51 that discharges all the goals. Inspection of the test confirms that it satisfies both our test purposes (the first ten `gear` inputs are 1, 2, 3, 2, 3, 2, 3, 2, 3, 2). When multiple test purposes are specified, `sal-atg` conjoins them. Additional purposes that can be conjoined in this example include those that force the `torque` and `velocity` inputs to be nonzero. If a disjunction is required, simply introduce a new Boolean variable and specify it as the disjunction of existing variables in the `DEFINITION` section of the monitor module, and name only that new variable in the `purpose-list`.

The two test purposes we have seen so far are invariants; suppose, now, that we want a test in which the `gear` input takes the value 4 at least 13 times. We can easily encode this test purpose by adding the Boolean output `enough` to the `monitor` module and expanding the body of the module as follows.

```
n4: NATURAL = 13;

monitor: MODULE =
BEGIN
INPUT
  gear: [1..4]
OUTPUT
  continuous, moving, enough: BOOLEAN
LOCAL
  fours: [0..n4]
INITIALIZATION
  continuous = (gear=1);
  moving = TRUE;
  fours = 0;
  enough = FALSE;
TRANSITION
  continuous’ = continuous AND (gear - gear’ < 2)
                           AND (gear’ - gear < 2);
  moving’ = moving AND (gear /= gear’);
  enough’ = (fours >= n4);
[
  gear = 4 AND fours < n4 --> fours’ = fours +1;
[]
  ELSE -->
]
END;
```

If we add the variable `enough` to the `purpose-list`, then `sal-atg` has to construct an initial test segment in which the `gear` input takes the value 4 at least 13 times before any of its structural coverage goals are eligible for consideration. In 57 seconds, the following command succeeds in constructing a single test of length 85 that discharges all the goals and purposes (and in which `gear` takes the value 4 exactly 23 times).

```
sal-atg trans_ga monitored_system trans_ga_goals.scm -id 30 -ed 15 --testpurpose --incrext
```

An alternative is to add the variable `enough` to the `goal-list` (rather than the `purpose-list`). This is a less demanding test specification and the invocation used earlier causes `sal-atg` to generate in 76 seconds a single test case of length 65 that discharges all the goals (and in which `gear` takes the value 4 exactly 14 times).

4 Producing Tests in the Form Required by a Test Harness

By default, `sal-atg` prints test cases in the form shown in Figure 4 on page 6; this is useful for evaluating the operation of the tool, but the real need is for test cases in the form needed to drive the test harness for the SUT. Customizing the presentation of test cases is accomplished by reprogramming the `sal-atg` output routines. We illustrate this using the "Intermediate Trace Representation" (ITR) format used by the University of Minnesota [ITR]; an example test case in ITR format is shown in Figure 10.

There actually are two issues in generating output from `sal-atg` that can drive the University of Minnesota test harness: the first is generating ITR format, the second is to deal with variable renamings and changed data representations. The second of these arises because the SAL specification and the simulator in the test harness are separately generated by translation from an RSML^{-e} source specification, and the two translations use slightly different naming conventions and data representations. For example, the SAL specification uses `Switch_OFF` where the test harness requires simply `OFF`, and SAL uses `TRUE` and `FALSE` as the Boolean constants, whereas the test harness uses 1 and 0.[10] Also, the SAL trap variables are an artifact of test generation and should be excluded from the test cases sent to the test harness. We need to decide which of the second set of issues should be handled in the translation to ITR, and which are best left to postprocessing by an awk script or other text processor. In this example, we postpone all issues from the second set to postprocessing, including deletion of the trap variables: although it is easy to identify trap variables as those named in `goal-list`, it is possible that some of the trap variables are absent from `goal-list` (as when we wish to generate only a partial set of tests) or that in other applications some of the trap variables are also genuine state variables, so we prefer to leave their deletion to the postprocessing stage. Our ITR output script also omits the ITR `DECLARATION` section: this is easily constructed by cut-and-paste or text processing from the SAL specification.

The Scheme function that prints test cases is called `test-path/pp` and it takes two arguments: the SAL `path` that encodes the test case concerned, and an integer giving the sequence number of this test case. A Scheme function to print the path as an ITR test case is shown here. The individual steps of the test case will be produced by the function

[10]The translation from the CTL representation of the test goals into SAL trap variables faced these issues in reverse.

```
DECLARATION
This_Input:BOOLEAN;
AltPreRefChanged:BOOLEAN;

TESTCASE 1
    STEP 1
      INPUT
        INTERFACE  Other_Input: READER
          AltSel = 0;
          AltselAct = 0;
        ENDINTERFACE
        INTERFACE  This_Input: READER
          AltPreRefChanged = 0;
          AltselCaptureCondMet = 0;
        ENDINTERFACE
      ENDINPUT
      OUTPUT
        INTERFACE  This_Output: SENDER
          AltLamp=OFF;
          AltSel=0;
        ENDINTERFACE
      ENDOUTPUT
      STATE
        This_Output=1;
        FD_Switch=OFF;
      ENDSTATE
    ENDSTEP 1
    STEP 2
      INPUT
        INTERFACE  Other_Input: READER
          AltSel = 0;
          AltselAct = 0;
        ENDINTERFACE
        INTERFACE  This_Input: READER
          AltPreRefChanged = 0;
          AltselCaptureCondMet = 1;
        ENDINTERFACE
      ENDINPUT
      OUTPUT
        INTERFACE  This_Output: SENDER
          AltselAct=1;
          FdOn=1;
        ENDINTERFACE
      ENDOUTPUT
      STATE
        Is_ALTSEL_Capture_Cond_Met=1;
      ENDSTATE
    ENDSTEP 2
ENDTESTCASE 1
```

Figure 10: Example of a test case in ITR format

display-step that is developed below. The other functions that appear here are part of the standard API to SAL.

```
(define (test-path/pp path n)
  ;; Assuming it is not a cyclic path
  (let* ((flat-module (slot-value path :flat-module))
         (state-vars (slot-value flat-module :state-vars))
         (step-list (slot-value path :step-info-list)))
    (print "TESTCASE " n)
    (let loop ((step-list step-list)
               (i 1))
      (unless (null? step-list)
              (let* ((step (car step-list))
                     (assignment-table (slot-value step :assignment-table)))
                (display-step i state-vars assignment-table 4))
              (print "")
              (loop (cdr step-list) (+ i 1))))
    (print "ENDTESTCASE " n)))
```

In the let*, we first extract the flat-module from the path, and then use this to extract the state-vars; we also extract the step-list. We then print the enclosing text for this test case and in between we loop through the individual steps of the test case and call display-step on each. The named let construction used here is one of the most convenient ways to program a loop in Scheme: the loop variables are initialized in the heading of the let and the loop is reentered by using its name (here, this is loop) as a function call whose arguments are the updated values of the loop variables. The first argument to display-step is the index of this step within the test case, and the last (here, 4) is the amount by which the output text should be indented. The second and third arguments are the state variables and the table giving the values assigned to them, respectively. The call to print with an empty string produces a blank line.

```
(define (display-step idx state-vars assignment-table n)
  (indent n) (print "STEP " idx)
  (let ((n (+ n 4))) ;; indent the body of the step
    (indent n) (print "/* Input Variables */")
    (indent n) (print "INPUT")
    (display-input-variable-values state-vars assignment-table (+ n 4))
    (indent n) (print "ENDINPUT")
    (print "")
    (indent n) (print "/* Output Variables */")
    (indent n) (print "OUTPUT")
    (display-output-variable-values state-vars assignment-table (+ n 4))
    (indent n) (print "ENDOUTPUT"))
    (indent n) (print "/* State Variables */")
    (indent n) (print "STATE")
    (display-local-variable-values state-vars assignment-table (+ n 4))
    (indent n) (print "ENDSTATE")
    (print "")
  (indent n) (print "End Step " idx))
```

The function `display-step` prints the input, output, and (local) state variables in that order; it does so by calling the appropriate function from the family `display-xxx-variable-values`. These take the same `state-vars` and `assignment-table` arguments as `display-step` itself, and the indent is increased by 4.

```
(define (display-input-variable-values state-vars assignment-table n)
  (display-variable-values state-vars assignment-table n
     (lambda (var) (and (instance-of? var <sal-input-state-var-decl>)
       (not (instance-of? var <sal-choice-input-state-var-decl>)))) "READER"))

(define (display-output-variable-values state-vars assignment-table n)
  (display-variable-values state-vars assignment-table n
       (lambda (var) (instance-of? var <sal-output-state-var-decl>)) "SENDER"))

(define (display-local-variable-values state-vars assignment-table n)
  (display-variable-values state-vars assignment-table n
       (lambda (var) (instance-of? var <sal-local-state-var-decl>)) "LOCAL"))
```

The `display-xxx-variable-values` functions invoke the function `display-variable-values` whose first three arguments are the same `state-vars` `assignment-table` and indent `n` arguments as were passed in, and whose fifth argument is a string that will be printed to describe the kind of ITR "interface" concerned. The fourth argument is a predicate that takes a state variable as its single argument and returns whether this variable's value should be printed in this list; these predicates check whether the variable is of the kind required for the function concerned; additionally, in the case of input variables, it checks that this variable is not one of the "choice" variables that SAL uses to keep track of which guarded commands were executed so that it can provide additional information in detailed counterexamples.

```
(define (display-variable-values state-vars assignment-table n pred? if-string)
  (for-each (lambda (state-var)
      (when (pred? state-var)
            (display-variable-value state-var assignment-table n if-string)))
         state-vars))
```

The function `display-variable-values` simply steps down the list of `state-vars` and checks the predicate to see if this variable should be printed; if so, it calls the function `display-variable-value` to do the work

```
(define (display-variable-value state-var assignment-table n if-string)
  (let* ((lhs
          (make-ast-instance <sal-name-expr> state-var :decl state-var))
         (value (cond
                 ((eq-hash-table/get assignment-table state-var) => cdr)
                 (else #f)))
         (assignments (make-queue))
         (rname #f))
    (when value
          (sal-value->assignments-core value #f lhs assignments))
    (for-each (lambda (assignment)
       (indent n)
       (if (instance-of? (sal-binary-application/arg1 assignment)
                         <sal-record-selection>)
          (begin
           (unless
            (eq? rname (sal-selection/target
                        (sal-binary-application/arg1 assignment)))
            (set! rname (sal-selection/target
                         (sal-binary-application/arg1 assignment)))
            (display "INTERFACE ") (sal-value/pp rname)
            (print ": " if-string) (indent n))
           (indent 4)
           (sal-value/pp
            (make-sal-equality
             (sal-selection/idx (sal-binary-application/arg1 assignment))
             (sal-binary-application/arg2 assignment)))
           (print ";"))
          (if rname
              (begin
                (set! rname #f)
                (print "END INTERFACE")
                (sal-value/pp assignment))
              (begin
                (sal-value/pp assignment)
                (print ";")))))
              (queue->list assignments))
    (when rname (indent n) (print "END INTERFACE"))))
```

The function `display-variable-value` prints the value of the state variable supplied as its first argument according to the `assignment-table` given in its second argument. It begins by setting `lhs` to the name of the state variable concerned and `value` to its value. Now that value could be structured (e.g., a record or a tuple) and we want to separately display the value of each component; the variable `assignments` is used to record these component assignments and it is initialized to an empty queue. The function `sal-value->assignments-core` breaks the structured value down into separate assignments to its components, which are stored in `assignments` (unstructured values store a single value in this queue). We then iterate through the `assignments` and print each one.

Now, the translator from RSML^{-e} to SAL actually uses only one kind of structured object, and for a special purpose: *records* are used to group input and output variables into *interfaces*. Thus, we first check if the assignment is to a record field and, if it is, we check whether this is to the same record as that whose name is saved in `rname`. If not, we must be starting to print the values of a new record, so we output the ITR `INTERFACE` text containing the record name; the `if-string` argument will provide the additional string `READER` or `SENDER` as appropriate. Otherwise (if we are in a record) we construct a new SAL assignment consisting of the field name and its value and call the basic `sal-value/pp` function to print that. Otherwise (if we are not in a record) we check whether we need to end the previous `INTERFACE` and then print the assignment.

All the Scheme functions described above are placed (in reverse order) in a file `itr.scm` and we then invoke `sal-atg` as before, but with `itr.scm` added to the command line, as in the following example (we have dropped the `--testpurpose` parameter to make it fit on one line).

```
sal-atg FGS05 system stategoals.scm -ed 5 -id 5 --incremental itr.scm
```

This produces the test cases in ITR format on standard output. Some postprocessing is then necessary to add the `DECLARATION` text, to remove assignments to trap variables, and to rename variables and values to the forms required by the test harness. Scripts to perform this are described in the appendix.

5 Conclusion

We have described use of the `sal-atg` test case generator, and reported some experiments using it. We believe that its ability to satisfy test goals with relatively few, relatively long tests is not only efficient (both in generating the tests and in executing them), but is likely to be more revealing: that is, its tests will expose more bugs than the large numbers of short tests generated by traditional methods for test generation using model checkers. Its ability to use conjunctions of test goals and to augment test goals with test purposes (and to set minimum as well as maximum lengths for test segments) should improve the quality of tests still further.

Our most urgent task is to validate these claims, and we hope to do so using the example of the flight guidance system with the help of researchers at the University of Minnesota. We used the example of the input format to their simulator to illustrate how the output of `sal-atg` can be customized to suit the needs of a give test harness.

Test goals are communicated to `sal-atg` through trap variables and we described how a preprocessor can set these to explore boundary values and to demonstrate the "meaningful impact" of subexpressions within Boolean decisions. We plan to augment our translator from Stateflow to SAL so that it generates trap variables for these kinds of tests. We showed how conjunctions of test goals and test purposes allow a test engineer to specify rather complex sets of tests and we plan to validate the effectiveness of these tests on suitable examples.

Currently, test goals are specified by listing the corresponding trap variables in Scheme files, but we are developing a notation to allow these to be specified directly in SAL; we hope to do so in a way that gives a logical status to tests and their goals, so that these can contribute to larger analyses managed by a SAL "tool bus."

The next release of SAL will provide a high-performance explicit state model checker and we plan to allow `SAL-ATG` to make use of it; we will consider use of ATPG techniques in the bounded model checker. We will also explore different strategies when a test cannot be extended: currently `SAL-ATG` returns to the start states, or to the end of the initial segment, but it might be preferable to return to some state nearer the end of the current test.

Acknowledgement

Thanks to Rance DeLong for comments on an earlier version.

References

[AOH03] Paul Ammann, Jeff Offutt, and Hong Huang. Coverage criteria for logical expressions. In *14th International Symposium on Software Reliability Engineering (ISSRE'03)*, pages 99–107, Denver, CO, 2003. IEEE Computer Society. 21

[CM94] John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *IEE/BCS Software Engineering Journal*, 9(5):193–200, September 1994. 20

[DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, pages 268–284, Odense, Denmark, April 1993. Volume 670 of *Lecture Notes in Computer Science*, Springer-Verlag. 19, 21, 23, 24

[dMOR$^+$04a] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In David Basin and Michaël Rusinowitch, editors, *2nd International Joint Conference on Automated Reasoning (IJCAR)*, pages 218–222, Cork, Ireland, July 2004. Volume 3097 of *Lecture Notes in Computer Science*, Springer-Verlag. 19

[dMOR$^+$04b] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV '2004*, pages 496–500, Boston, MA, July 2004. Volume 3114 of *Lecture Notes in Computer Science*, Springer-Verlag. 1

[dMRS02] Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *18th International Conference on Automated Deduction (CADE)*, pages 438–455, Copenhagen, Denmark, July 2002. Volume 2392 of *Lecture Notes in Computer Science*, Springer-Verlag. 19

[HD04] Mats P.E. Heimdahl and George Devaraj. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, Linz, Austria, September 2004. 14

[HdMR04] Grégoire Hamon, Leonardo de Moura, and John Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, September 2004. IEEE Computer Society. 1, 7

[HGW04] Mats P.E. Heimdahl, Devaraj George, and Robert Weber. Specification test coverage adequacy criteria = specification test generation *In*adequacy criteria? In *High-Assurance Systems Engineering Symposium*, pages 178–186, Tampa, FL, March 2004. IEEE Computer Society. i, 7, 14, 16, 27, 29, 43

[HR04] Grégoire Hamon and John Rushby. An operational semantics for Stateflow. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering: 7th International Conference (FASE)*, pages 229–243, Barcelona, Spain, 2004. Volume 2984 of *Lecture Notes in Computer Science*, Springer-Verlag. 1, 12

[HRV$^+$03] Mats P.E. Heimdahl, Sanjai Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-generating test sequences using model checkers: A case study. In *Third International Workshop on Formal Approaches to Software Testing (FATES)*, pages 42–59, Montreal, Canada, October 2003. Volume 2931 of *Lecture Notes in Computer Science*, Springer-Verlag. 14

[HUZ99] Alan Hartman, Shmuel Ur, and Avi Ziv. Short vs. long—size does make a difference. In *IEEE International Workshop on High Level Design Validation and Test (HLDVT)*, pages 23–28, San Diego, CA, November 1999. IEEE Computer Society. Available at `http://www.haifa.il.ibm.com/projects/verification/mdt/papers/testlength.pdf`. 7, 16

[HVCR01] Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. A practical tutorial on modified condition/decision coverage. NASA Technical Memorandum TM-2001-210876, NASA Langley Research Center, Hampton, VA, May 2001. Available at `http://www.faa.gov/certification/aircraft/av-info/software/Research/MCDC%20Tutorial.pdf`. 21

[ITR] *Intermediate Trace Representation Format*. Software Engineering Center, University of Minnesota. Undated. 33

[KCe98] Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised^5 report on the algorithmic language Scheme. *Higher Order and Symbolic Compututation*, 11(1):7–105, 1998. Available from `http://www.schemers.org/Documents/Standards/R5RS/`. 5

[KLPU04] Nikolai Kosmatov, Bruno Legeard, Fabien Peureux, and Mark Utting. Boundary coverage criteria for test generation from formal models. In *15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 139–150, Saint-Malo, France, November 2004. IEEE Computer Society. 18

[KTK01] Noritaka Kobayashi, Tatsuhiro Tsuchiya, and Tohru Kikuno. A new test data selection strategy for testing Boolean specifications. In *Twelfth International Conference on Software Reliability Engineering (ISSRE '01), Fast Abstracts*, Hong Kong, November 2001. IEEE Computer Society. Available from `http://www.chillarege.com/fastabstracts/issre2001/2001117.pdf`. 21

[Kuh99] D. Richard Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411–424, 1999. 20

[LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from Z and B. In Peter Lindsay, editor, *FME 2002: Formal Methods–Getting IT Right*, pages 21–40, Copenhagen, Denmark, July 2002. Volume 2391 of *Lecture Notes in Computer Science*, Springer-Verlag. 21, 23, 24

[OBY04] Vadim Okun, Paul E. Black, and Yaacov Yesha. Comparison of fault classes in specification-based testing. *Information and Software Technology*, 46(8):525–533, June 2004. 20

[RTCA92] *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. Requirements and Technical Concepts for Aviation, Washington, DC, December 1992. This document is known as EUROCAE ED-12B in Europe. 20

[Rus04] John Rushby. SAL tutorial: Analyzing the fault-tolerant algorithm OM(1). CSL technical note, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2004. Available at `http://www.csl.sri.com/users/rushby/abstracts/om1`. 28

[WGS94] Elaine Weyuker, Tarak Goradia, and Ashutosh Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994. 20, 21

A Scripts to Convert From/To Formats Used by University of Minnesota

We are in the process of connecting `sal-atg` to the tool chain developed at the University of Minnesota; our goal is to evaluate the quality of the tests generated by `sal-atg` using the mutant implementations they have developed for the Flight Guidance System example [HGW04]. Here we document the awk and sed scripts that we use to translate between the various representations used.

The SAL specification we employ was provided by Jimin Gao of the University of Minnesota who is developing an RSML^{-e} to SAL translator. He also provided lists of CTL properties for nuSMV that generate tests for state and transition coverage. The translation from RSML^{-e} to nuSMV uses slightly different variable and constant names than the translation to SAL, so the following scripts not only have to transform CTL properties into assignments to SAL trap variables, but must do some renaming as well.

The following awk script, `states2sal`, takes a file of CTL properties for state coverage and generates output containing assignments to the trap variables, their declarations, their initialization, and the Scheme definition for the `goal-list`.

```
#!/bin/awk -f
#
# Translate UMN state coverage properties to SAL
#

/add_property/
    gsub("add_property -l ", "");
    gsub("G\\(!\\(", "");
    gsub("\\)\\)\"", "");
    gsub("= Cleared", "= Base_State_Cleared");
    gsub("= Selected", "= Base_State_Selected");
    gsub("= Armed", "= Selected_State_Armed");
    gsub("= Active", "= Selected_State_Active");
    gsub("= Capture", "= Active_State_Capture");
    gsub("= Track", "= Active_State_Track");
    gsub(" = Un_defined", "_Undefined");
    gsub("Lamp = OFF", "Lamp = Lamp_OFF");
    gsub("Lamp = ON", "Lamp = Lamp_ON");
    gsub("= Disengaged", "= AP_State_Disengaged");
    gsub("= Engaged", "= AP_State_Engaged");
    gsub("= On", "= On_Off_On");
    gsub("= Off", "= On_Off_Off");
    gsub("= LEFT", "= Side_LEFT");
    gsub("= RIGHT", "= Side_RIGHT");
    gsub("= 0", "= FALSE");
    gsub("= 1", "= TRUE");
    gsub("&", "AND");
    gsub("->", "=>");
#     print "state" NR ": LEMMA main |- " $0 ";"
    print "state" NR "’ = state" NR " OR " $0 ";";
    decl = decl ", state" NR;
    init = init "state" NR " = FALSE;\n";
    gl = gl "\"state" NR "\" ";


END
print decl ": BOOLEAN";
print init;
print "(define goal-list ’(" gl "))";
```

Given a file `example-states.ctl` with the following contents

```
add_property -l "G(!(ALTSEL_Active = Capture))"
add_property -l "G(!(When_Selected_Nav_Source_Changed = 1))"
```

the command

```
./states2sal example-states.ctl
```

generates the following output

```
state1’ = state1 OR ALTSEL_Active = Active_State_Capture;
state2’ = state2 OR When_Selected_Nav_Source_Changed = TRUE;

, state1, state2: BOOLEAN

state1 = FALSE;
state2 = FALSE;

(define goal-list ’("state1" "state2" ))
```

The following awk script, `trans2sal`, similarly takes a file of CTL properties for transition coverage and generates intermediate output containing assignments to the trap variables, their declarations, their initialization, and the Scheme definition for the `goal-list`. The translation for the nested `X` properties is a little difficult, and this is completed by the sed script `trans2sal.sed` that further transforms the intermediate output and which is shown following the awk script.

```
#!/bin/awk -f
#
# Translate UMN transition coverage properties to SAL
#

/add_property/
    gsub("add_property -l \"", "");
    gsub("!", "NOT ");
    gsub("\"", "");
    gsub("= Cleared", "= Base_State_Cleared");
    gsub("= Selected", "= Base_State_Selected");
    gsub("= Armed", "= Selected_State_Armed");
    gsub("= Active", "= Selected_State_Active");
    gsub("= Capture", "= Active_State_Capture");
    gsub("= Track", "= Active_State_Track");
    gsub(" = Un_defined", "_Undefined");
    gsub("Lamp = OFF", "Lamp = Lamp_OFF");
    gsub("Lamp = ON", "Lamp = Lamp_ON");
    gsub("= Disengaged", "= AP_State_Disengaged");
    gsub("= Engaged", "= AP_State_Engaged");
    gsub("= On ", "= On_Off_On ");
    gsub("= Off ", "= On_Off_Off ");
    gsub("= On)", "= On_Off_On)");
    gsub("= Off)", "= On_Off_Off)");
    gsub("= LEFT", "= Side_LEFT");
    gsub("= RIGHT", "= Side_RIGHT");
    gsub("= 0", "= FALSE");
    gsub("= 1", "= TRUE");
    gsub("0 & ", "FALSE AND ");
    gsub("1 & ", "");
    gsub("(1)", "TRUE");
    gsub("&", "AND");
    gsub(" \\| ", " OR ");
    gsub("->", "=>");
#    print "trans" NR ": LEMMA main |- " $0 ";"
    print "trans" NR "' = trans" NR " OR " $0 ";";
    decl = decl ", trans" NR;
    init = init "trans" NR " = FALSE;\n";
    gl = gl "\"trans" NR "\" ";


END
print decl ": BOOLEAN,";
print init;
print "(define goal-list '(" gl "))";
```

```
/G(/s//NOT (/g
/\.result/s///g
/((/s/((\([^(]*\)))/(\1)/g
/X/s/X(\([^=)]*\))/(\1')/g
/X/s/X(\([^= )]*\) = \([^=)]*\))/(\1' = \2')/g
/X/s/X(NOT \([^= )]*\) = \([^=)]*\))/(NOT \1' = \2')/g
/X/s/X(\([A-Za-z0-9_]*\) = (\([A-Za-z0-9_]*\) = \([A-Za-z0-9_]*\)))/
 (\1' = (\2' = \3'))/g
/X/s/X(NOT \([A-Za-z0-9_]*\) = (\([A-Za-z0-9_]*\) = \([A-Za-z0-9_]*\)))/
 (NOT \1' = (\2' = \3'))/g
/Active_State_Capture'/s//Active_State_Capture/g
/Selected_State_Armed'/s//Selected_State_Armed/g
/Lamp_ON'/s//Lamp_ON/g
/Lamp_OFF'/s//Lamp_OFF/g
/Selected_State_Active'/s//Selected_State_Active/g
/Active_State_Track'/s//Active_State_Track/g
/Base_State_Cleared'/s//Base_State_Cleared/g
/Base_State_Selected'/s//Base_State_Selected/g
/AP_State_Engaged'/s//AP_State_Engaged/g
/THIS_SIDE'/s//THIS_SIDE/g
/On_Off_On'/s//On_Off_On/g
/On_Off_Off'/s//On_Off_Off/g
/TRUE'/s//TRUE/g
/FALSE'/s//FALSE/g
/AP_State_Disengaged'/s//AP_State_Disengaged/g
/AP_State_Engaged'/s//AP_State_Engaged/g
/Side_RIGHT'/s//Side_RIGHT/g
/Side_LEFT'/s//Side_LEFT/g
```

Given a file `example-transitions.ctl` with the following contents (actually, the properties must each be on a single line)

```
add_property -l "G(((X((!Is_This_Side_Active))))
     -> X(!ALTSEL_Active = Offside_ALTSEL_Active))"
add_property -l "G(!((X(m_Deselect_VS.result)
   & X(Is_This_Side_Active) & (VS = Selected)))
     -> X(VS = Cleared))"
```

the command

```
./trans2sal example-transitions.ctl | sed -f trans2sal.sed
```

generates the following output

```
trans1’ = trans1 OR NOT ((((NOT Is_This_Side_Active’)))
    => (NOT ALTSEL_Active’ = Offside_ALTSEL_Active’));
trans2’ = trans2 OR NOT (NOT (((m_Deselect_VS’)
    AND (Is_This_Side_Active’) AND (VS = Base_State_Selected)))
         => (VS’ = Base_State_Cleared));

, trans1, trans2: BOOLEAN,

trans1 = FALSE;
trans2 = FALSE;

(define goal-list ’("trans1" "trans2" ))
```

The following sed script `sal2itr.sed` is applied to the ITR output described in Section 4 to perform variable renaming and deletion of irrelevant states variables.

```
/state[0-9]/d
/trans[0-9]/d
/NimbusSystemClockReceiver_Receive/d
/Report_Is_This_Side_Active/d
/_Undefined/d
/_Random/d
/false/s//0/
/true/s//1/
/Input_Msg/s//Input/
/Switch_OFF/s//OFF/
/Switch_ON/s//ON/
/Output_Msg/s//Output/
/Lamp_OFF/s//OFF/
/Lamp_ON/s//ON/
/Side_LEFT/s//LEFT/
/Side_RIGHT/s//RIGHT/
/Base_State_Cleared/s//Cleared/
/Base_State_Selected/s//Selected/
/Selected_State_Armed/s//Armed/
/Selected_State_Active/s//Active/
/Active_State_Capture/s//Capture/
/Active_State_Track/s//Track/
/AP_State_Engaged/s//Engaged/
/AP_State_Disengaged/s//Disengaged/
/On_Off_On/s//On/
/On_Off_Off/s//Off/
```

A typical command is the following.

```
sed -f sal2itr.sed FGS05.tests
```

The following sed script `declarations.sed` is used to extract the declarations needed at the start of an ITR file from the SAL specification concerned.

```
1i\
DECLARATION
/^\(.*\)_Msg_Type.*\[\#/s//\1: BOOLEAN;\
/
/\#\];/s//;/
/, /s//;\
/g
$a\
ENDDECLARATIONS
```

A typical command using this file is the following.

```
sed -n ’/#/p’ FGS05.sal | sed -f declarations.sed
```