SRI International

CSL Technical Note ● January 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.

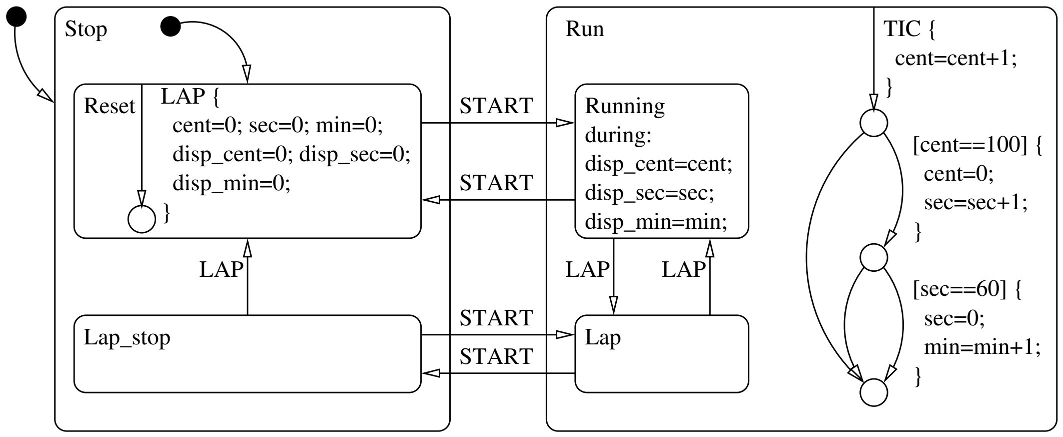

Figure 1: A simple stopwatch in Stateflow

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.

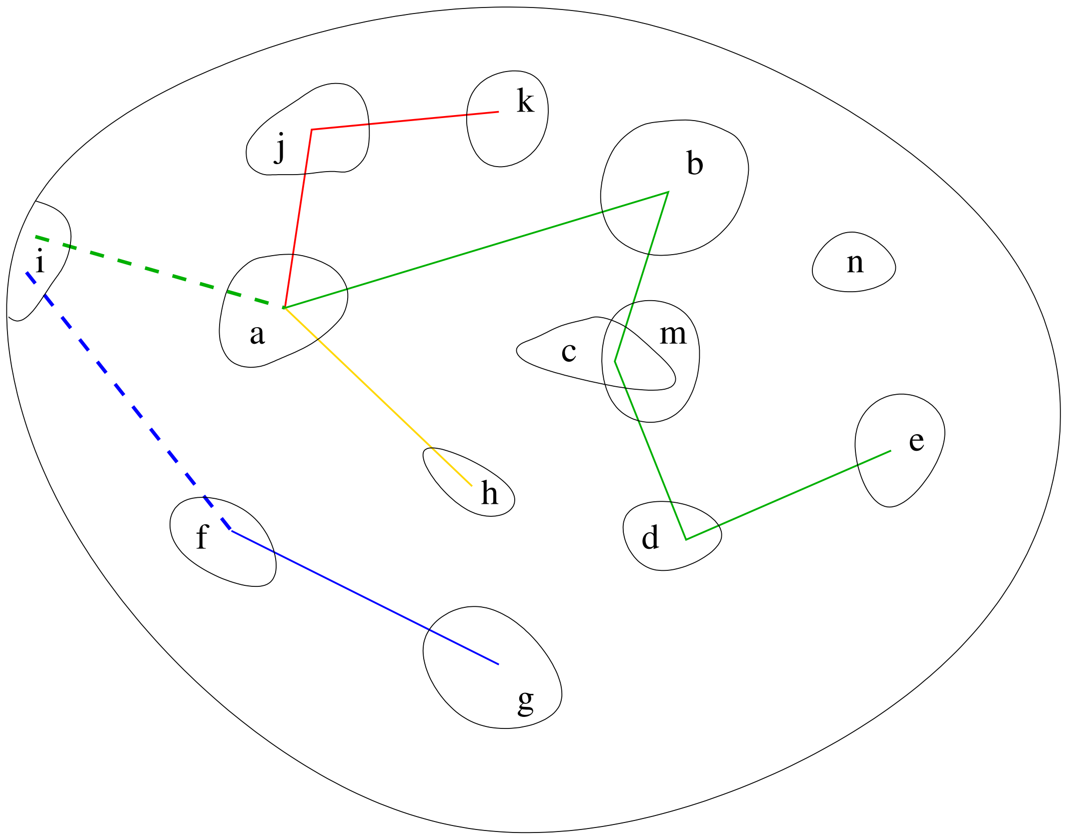

Figure 5: Operation of `sal-atg`

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.

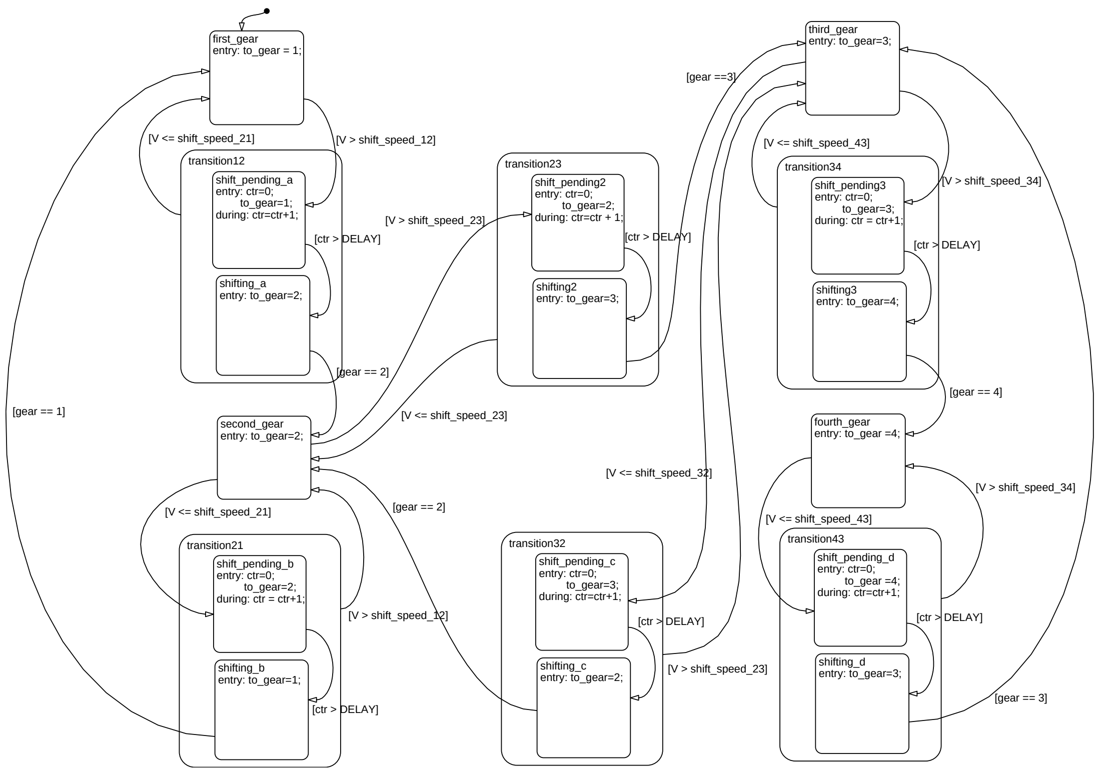

Figure 6: Stateflow model for four-speed shift scheduler

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 $\Psi$, and we have a fault hypothesis that this might incorrectly be implemented as the predicate $\Phi$. We want a test that will discriminate these two cases: that is, an assignment to the state variables of $\Psi$ 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, $\Psi$ is `x <= 6` and $\Phi$ 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 $\Psi$ in the specification, apply some heuristic or table to generate a hypothesized faulty implementation $\Phi$, then generate a variable to trap satisfaction of $\Psi$ `XOR` $\Phi$ (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 $\Psi$ 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 $\Phi$, then $\Phi^p_e$ denotes the modified expression in which $e$ (an arbitrary expression) replaces $p$ in $\Phi$. A test $t$ (i.e., an assignment of truth values to the atomic formulas of $\Phi$) manifests the *meaningful impact* of $p$ in $\Phi$ 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 $\Phi$. This will be so for any $t$ that satisfies $\Phi$ `XOR` $\Phi^p_{\neg p}$. Since one of $p$ or $\neg p$ must be `TRUE` in $\Phi$ 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 $\Phi$ with respect to $p$) and is denoted $\frac{d}{dp}\Phi$. To generate meaningful impact tests for $\Phi$, 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 $\Phi$ (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 $\{\frac{d}{dp}\Phi \mid p$ is an atomic formula of $\Phi\}$ of all the derivatives of $\Phi$: 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 inadequacy 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
```