Automatic Generation of Test-Cases Using Model Checking for SL/SF Models

Ambar A. Gadkari, Swarup Mohalik, K.C. Shashidhar, Anand Yeolekar, J. Suresh, and S. Ramesh

General Motors R&D - India Science Lab, Bangalore.
{ambar.gadkari, swarup.mohalik, shashidhar.kc, anand.yeolekar, suresh.jeyaraman, ramesh.s}@gm.com

Abstract. Model based test-case generation methods use different techniques, viz., random data generation, guided simulation, constraint solving, etc., for automatically deriving test-cases from reference models. Recently, automatic test generation using model checking, a formal verification technique, has been reported to produce more efficient test-suites compared to conventional test generation techniques. In this paper, we describe our experience in test generation using model checking for the Simulink/Stateflow (SL/SF) models of two automotive controller examples. Model checking based test generation is non-trivial since the SL/SF models have to be first translated into a formal language to serve as an input for the model checker tool. Moreover, to handle the size and complexities of the industrial designs the translation has to make use of various abstractions yet preserve the semantics of the original model relevant for test generation. We provide an outline of the translation scheme used for translating the SL/SF models into a formal language called SAL. Preliminary results indicate that model checking based test generation, in conjunction with suitable model abstractions, can yield better results in terms of coverage and efficiency of test-cases as compared to the conventional approaches based on simulation and random data generation.

1 Introduction

Model-based development of embedded control software is gaining wide acceptance for design of automotive systems. Automotive controllers are usually designed using the Simulink/Stateflow [7] (SL/SF) language. The SL/SF models serve as reference models for testing the software implementation of the controllers. The methods and tools addressing model based test generation can be classified based on various parameters such as modeling paradigm, test generation technology, supported coverage criteria and so on. A good description of various model based testing techniques can be found in [3, 16, 17]. There is a wide range of commercial, academic and proprietary tools for model based

* The opinions expressed in this article are those of the authors, and do not necessarily reflect the opinions or positions of their employers, or other organizations.
test generation (MBT); a good number of them have been listed in [1] with brief
descriptions. The choice of methods and tools for test generation depends on var-
ious factors, primary being the effectiveness (capability of test to detect bugs)
of the generated tests for the specific application domain. Some of the other
factors influencing the choice of the right method or tool in an industrial set-
ting are scalability, ease-of-use, degree of automation, portability of test-suites,
etc., as discussed in [9, 13]. Some of the commercially available MBT tools, es-
pecially those available for control software applications such as Reactis [10]
and STB [14] are based upon the random test generation and guided simulation
based methods. These methods seem to be good in practice and are scalable to
large industrial designs. However, these methods suffer from a limitation that if
the tests do not cover certain conditions then one cannot infer that those condi-
tions are indeed unreachable. The unreachable conditions could be due to either
some logical errors or due to redundant elements. In safety critical applications
it is a must that some conclusive evidence for such unreachable conditions be
available. Hence, there is a need to complement the random based tools with
those that provide such evidence. Formal verification techniques such as model
checking can provide such evidence through counter-examples or witnesses.

Model checking has been shown to provide an efficient technique to automatic-
ically derive test sequences from transition system models [2,4,6]. This approach
for test generation relies on capabilities of the model checkers to generate traces
for counter-examples of properties that do not hold in the model. Test sets are
usually derived to satisfy certain coverage criteria of a model. The coverage crite-
ria are mostly based on structural coverage of the transition system model such
as state and transition coverage. In the model checking based test generation
scheme, structural elements (state and transitions) are associated with boolean
variables called *trap* variables. Structural coverage of a model element is then
reduced to model checking the reachability of the state where the associated
trap variable is true. The path which witnesses the reachability is said to cover
the trap variable. For a given set of coverage criteria, there can be more than
one set of test cases that can satisfy the criteria. One can associate a notion of
cost of a test-suite (a function of the number and length of test cases) [6]. Model
checking based test generation strives to find the most efficient test-suite, i.e.,
the one with minimal cost.

The main motivation for the work presented here is to evaluate the model
checking based approach for test generation in the context of automotive con-
trol systems. This paper describes our experience in generation of test-cases for
medium-sized automotive controller examples using the model checking based
approach. The application of model checking based technique for test-case gen-
eration from SL/SF models involves translating the models into a formal language
amenable for formal verification. The translation scheme in order to cater to the
size and complexities of the industrial designs has to employ non-trivial abstrac-
tion techniques ensuring the semantics of the model relevant for the test-case
generation is preserved. We describe the key aspects of such a translation scheme
for translating SL/SF models into a formal language known as SAL (Symbolic

We have compared the results with a commercial tool based on widely used test generation techniques, namely, the random test generation and simulation-based approach. In summary, the main contributions of this paper include:

- Test generation flow for SL/SF models based on model checking approach including the translation scheme for translating SL/SF models to SAL for purpose of test generation.
- Experimental results for evaluation of model checking-based test generation approach for SL/SF models based on two medium-sized case study examples from automotive domain.

Our experimental results indicate that the model checking-based test generation, in conjunction with suitable model abstractions, can yield better results in terms of coverage and efficiency of test-cases as compared to the conventional approaches based on simulations and random data generation.

We would like to note that MathWorks has recently released a Simulink Design Verifier [7] tool-kit which uses SAT-solver-based technology for test-case generation. Although the work described here has been initiated earlier and carried out independently, it would be interesting to take up a comparative study between the two tools at a later stage.

The paper is organized as follows: Section 2 describes the test generation flow for SL/SF models based on model checking. Section 3 describes the implementation of the test generation flow using SAL. Section 4 presents the experimental evaluation of this approach with experimental results and discussion for the case study. Section 5 concludes the paper with pointers for future research directions.

2 Automatic Test Generation Flow

We describe here the proposed flow for automatic test-case generation, hereafter referred to as ATG in this paper, from SL/SF models based on model checking approach. Figure 1 shows the steps in this flow. There are three inputs to ATG, namely, SL/SF models, high-level requirements and test specifications including different coverage goals. The output of ATG is a test-suite of timed input-output sequences that can be used for testing the implementations.

The test specification includes different coverage goals based on various structural criteria defined over models such as block coverage, condition coverage, decision coverage, MC/DC coverage, lookup-table coverage, states and transitions coverage and so on. The test specification and high-level requirements are encoded in form of formal properties using a subset of Linear Temporal Logic (LTL) by the property generator module. The SL/SF model is translated into a formal language which can be fed to the model checking engine. The translation of SL/SF model to formal model is non-trivial and involves various steps such as time discretization, type abstractions and capture of simulation semantics of SL/SF. Thus the translated model retains all the necessary elements needed for test generation meeting various structural and behavioral criteria. The model checking engine then verifies the formal model using the formal properties. Most
of the model checkers are capable of generating either a counter-example trace (if the property fails) or a witness trace (if the property is verified). These traces are converted into test-cases consisting of the timed input-output sequences.

3 ATG Implementation

For implementation of the ATG we chose SAL as the formal modeling language. SAL provides a wide range of tools including different model checking engines such symbolic model checker (sal-smc), bounded model checker (sal-bmc) and also recently introduced test generator (sal-atg). In the following, we give an outline of the algorithm implemented for automatic translation of SL/SF models into SAL. The translation algorithm currently supports a subset of SL/SF blocks which includes arithmetic and logical blocks, abs, gain, saturation, zero-delay, memory, integration, if block, general expression block, switches, constant, discrete pulse generator, lookup-table and a subset of Stateflow.

The aim of translating a given SL/SF model and a coverage criterion to SAL is to produce a model in SAL and a goal-list such that a model entity is covered by some test-case in the SL/SF model iff a corresponding trap variable in the goal-list is covered in the SAL model. Our translation algorithm achieves the above objective by preserving the discrete-step semantics of SL/SF in SAL and ensuring that all the computations of a time-discretized transformation of the given SL/SF model are present in the SAL model. Since information about lookup-tables and calibration variables are available only in the MATLAB workspace, we first use a MATLAB script to extract the following information about the blocks and ports of the model: the list of blocks, sample times, input and output signals.
and block-specific default values for each of the blocks, the list of signals, their
types and the connection information (source block, destination block).

Each block in the discrete SL/SF model has a sample time, which can be
default, user-defined or inherited from their input blocks. From all these sample
times, we calculate the fundamental sample time of the model and the frequency
at which a block executes. The exact type information of the signals (the default
is double) is inferred as much as possible from the type of the blocks (e.g. logical
blocks like AND, OR blocks must have boolean inputs and outputs). Since we
use a version of sal-atg with only bounded integer data types, we require the
user to supply a mapping for all unbounded signal values to bounded integers.
This requires the ATG-generated test inputs to be post-processed and cast to
the original types suited for injecting in the SL/SF model for generation of ex-
pected outputs. Each basic SL/SF block is translated into an equivalent SAL
module. Each SAL module has a counter which controls the firing of the module
at the block’s frequency. The connector signals are captured by the common
input/output variables of the SAL modules. Hierarchy of blocks is modeled by
synchronous composition of the corresponding SAL modules. Thus, the hierar-
chical structure of the model is reflected in the structure of the SAL model to
ease the instrumentation for criteria like block and sub-system coverage. The
data-flow dependency among the inputs and outputs of SL/SF blocks within
an execution cycle is modeled by exploiting the fact that SAL treats the global
transition relation as a constraint between the inputs and outputs.

In order to generate test-cases for structural coverage criteria, we instrument
the SAL modules with trap variables corresponding to the coverage criterion. For
example, for block coverage criteria, the first time the corresponding SAL module
for a block is activated, a trap variable is set to true. For other coverage criteria
like MC/DC, the corresponding SAL modules are more complex because they
have to take into account all the combination of changes in the conditions and
decisions to capture the semantics of the criteria. In order to generate tests that
cover a set of high-level requirements given as LTL formulae of form $G(\phi \rightarrow \psi)$
we generate trap monitors (SAL modules) automatically corresponding to the
formulae $F(\phi)$ and $F(\neg \psi)$ (following [5]), where each trap variable is set to true
if the corresponding LTL formula is true. These modules are integrated in the
main SAL model.

Discussion: The translation procedure outlined above is faithful to the struc-
ture and discrete-time semantics of the original SL/SF model. However, the re-
quirement for translation does not mandate such faithfulness and has scope for
aggressive abstractions preserving only the coverage of selected entities. Trans-
lation methods for subsets of SL/SF to formal languages have been previously
reported in the literature [8,12,15]. These methods preserve the exact behavior
because their purpose is formal verification. On the other hand, since our goal
is test generation, we can achieve our results using under-approximated models
where the abstraction scheme is guided by the type of coverage needed for a test
generation session. Within such a restricted scope, say for state coverage, further
optimizations are possible by slicing techniques provided in the SAL framework.
to target specific coverage (e.g., only a subset of interesting states). This can lead to small abstracted models and hence to better scalability in contrast to the earlier reported translation schemes.

4 Experimental Evaluation of ATG

One of the primary aims of this work is to experimentally evaluate the effectiveness of the above-described approach on automotive controller examples and compare it with other test generation techniques such as random test generation. We carried out the evaluation of the ATG in comparison with a commercial test generation tool for two automotive controller examples, namely, ATC (automatic transmission controller) and ACC (adaptive cruise controller). The generated tests are executed on SL/SF model with MATLAB’s Verification and Validation tool-box. This provides a common comparison framework for evaluating the efficacy of test-suites with respect to different coverage metrics.

4.1 Application 1: Automatic Transmission Controller (ATC)

The ATC model is available as a demo example in the MATLAB tool distribution. This system implements a controller for a 4-gear automatic transmission based on speed, brake and throttle inputs (see Fig. 2). The controller (gear and selection logic) is modeled in a Stateflow chart and the vehicle, transmission and engine subsystems are modeled in Simulink blocks. The threshold calculation block decides the thresholds for gear shift, depending on the current gear and throttle inputs. The Selection logic block detects if the speed threshold is violated, and causes a gear shift in the Gear block, by means of events. The engine subsystem models the engine behavior and outputs the rpm, based on the external throttle input. The transmission subsystem outputs torque based on current gear value and the rpm. The vehicle subsystem models the calculation for vehicle speed, based on throttle and torque inputs, and various physical factors such as aerodynamic drag, friction, drive ratio, vehicle inertia etc. The Stateflow chart is hierarchical, with a total of 9 states. States are composed in
both parallel-and and exclusive-or modes, with event broadcasting for communication. Temporal operator after is used to measure ticks. The Simulink portion includes 1-D and 2-D lookup-tables, gains, integrators (2 continuous variables), non-linear multiplication operation and feedback loops.

4.2 Application 2: Adaptive Cruise Controller (ACC)

The ACC model (see Fig. 3) uses driver inputs, environment information such as vehicle’s current speed, leader’s absence/presence and separation distance etc., through the ACC radar sub-system and outputs the host vehicle’s speed at any given instant of the model’s execution. From the environment inputs, the Mode decider block calculates the major mode (CC mode or the ACC mode) of the controller. In the ACC mode, the corresponding Stateflow chart can be in either of the five substates, viz., off, init, inactive, active and ACC. The value of the current substate is used to calculate the actuator’s outputs that are fed into a plant model. Speed and braking torque are used in the plant model to regulate the speed according to the driver’s set speed. The full ACC model has over sixty five Simulink blocks and three Stateflow diagrams. The model also consists of two lookup-tables; one for calculating the safe clearance that the host must maintain with respect to the leader and the other for calculating the effective braking torque to be applied if the separation distance reduces below the prescribed limit.

4.3 Experimental Results and Discussion

The test-case generation using ATG for both the case study examples involved experimenting with various options provided by sal-atg, such as, the use of sal-smc in combination with sal-bmc, bounding depth of sal-bmc, lengths of initial search segment and extended search segments used by sal-smc and sal-bmc, model slicing options, incremental test sequence generation, and so on. Choosing the optimal combination of these options for a particular application is a highly empirical process. Some of the options used during experimentation along with the resulting effects on test lengths, numbers and respective coverages achieved by them are listed in Table 1.

Observe that 100% coverage could not be achieved for conditions and decisions. These uncovered test goals were traced back to the trap variables in SAL model. In most of the cases, the unreachability of these goals could be proved using combination of tools such as deadlock-checker, sal-smc, path-finder, etc., available in the SAL tool suite. This capability is a distinguishing feature of the formal verification based test-case generation methods. Few of the coverage goals, especially related to the computation intensive portions of the Simulink example were found difficult to be covered. In fact, even proving the unreachability of these goals failed due the current limitations on the size of state space handling capability of sal-smc.
<table>
<thead>
<tr>
<th>Test suite</th>
<th>sal-atg options</th>
<th>No. of test-cases</th>
<th>MathWorks’ % Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Conditions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Application 1: The ATC System**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>10</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>5</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>5</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>67</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>67</td>
<td>91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>67</td>
<td>91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>17</td>
<td>67</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>17</td>
<td>67</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>17</td>
<td>67</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>67</td>
<td>91</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Application 2: The ACC System**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50</td>
<td>8</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td>70</td>
<td>8</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>50</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>70</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>32</td>
<td>80</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>45</td>
<td>80</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>314</td>
<td>80</td>
<td>88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>591</td>
<td>86</td>
<td>92</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Coverage obtained from test-cases generated using ATG for ATC and ACC.

The commercial test generation tool used as the comparison benchmark also provides various user options. Various combinations of random search and targeted (or guided search) options were exploited for flexing the tool to produce best possible results on both the examples. Starting with lowest values for the test steps for both random and targeted phases, we increased them gradually to achieve better coverage. The minimal length test-cases achieving the maximum coverage in a category (with a given set of options) were selected for comparison purpose. A sample of different options used with the respective coverages are shown in Table 2 It may be noted that in many cases the percentage coverage values indicated by the commercial tool differed from the percentage coverage values indicated by the MATLAB’s Verification and Validation toolbox. For the uniformity of comparison we have considered the coverage data provided by the latter alone.

Observe that although similar coverages for condition and decision criteria were obtained by the test-suites produced by this tool there was no clear indication that the further coverage could not be reached. The decision for terminating the experiments with random test generation were most often driven by the heuristics. Also note from both the tables that the number of total steps is significantly less in case of ATG compared to those generated by commercial tool. This was due to one of the main points of distinction between the two approaches, namely, that the ATG allowed the use of abstractions in the formal model. The sampling-time approximations were done during the SL/SF to SAL translation allowing the tests to be generated which were capable of simulating longer behavioral paths with less number of input injection points. This helped in reaching deeper coverage goals even with less input data injections.

In the above study, the test generation time was found to be much less in the case of the commercial tool (in order of few seconds) compared to that of model
<table>
<thead>
<tr>
<th>Test suite</th>
<th>Commercial Tool Options</th>
<th>No. of test-suite</th>
<th>MathWorks’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>test-cases length</td>
<td>% Coverage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Conditions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decisions</td>
</tr>
</tbody>
</table>

### Application 1: The ATC System

<table>
<thead>
<tr>
<th></th>
<th>Random Phase</th>
<th>Targeted Phase</th>
<th>No. of tests</th>
<th>steps/test</th>
<th>steps</th>
<th>% Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>3</td>
<td>15</td>
<td>33</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>1000</td>
<td>3</td>
<td>21</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>3000</td>
<td>4</td>
<td>24</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>15000</td>
<td>5</td>
<td>153</td>
<td>75</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>50000</td>
<td>2</td>
<td>102</td>
<td>67</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>100000</td>
<td>5</td>
<td>1707</td>
<td>75</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>100</td>
<td>1000</td>
<td>1</td>
<td>74</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>100</td>
<td>1000</td>
<td>5</td>
<td>165</td>
<td>83</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>1000</td>
<td>20000</td>
<td>4</td>
<td>1820</td>
<td>83</td>
</tr>
</tbody>
</table>

### Application 2: The ACC System

<table>
<thead>
<tr>
<th></th>
<th>Random Phase</th>
<th>Targeted Phase</th>
<th>No. of tests</th>
<th>steps/test</th>
<th>steps</th>
<th>% Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>4</td>
<td>64</td>
<td>52</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>500</td>
<td>6</td>
<td>275</td>
<td>46</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>6000</td>
<td>10</td>
<td>230</td>
<td>74</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>100</td>
<td>1000</td>
<td>5</td>
<td>276</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>100</td>
<td>20000</td>
<td>11</td>
<td>456</td>
<td>77</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>100</td>
<td>100000</td>
<td>9</td>
<td>2529</td>
<td>79</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>1000</td>
<td>1000</td>
<td>5</td>
<td>1407</td>
<td>66</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>1000</td>
<td>100000</td>
<td>12</td>
<td>1971</td>
<td>83</td>
</tr>
</tbody>
</table>

**Table 2.** Coverage obtained from test-cases generated using a commercial tool.

Checking based ATG (in order of tens of minutes), while the test execution time is directly proportional to the test steps indicated in the above tables. Since the tests are generated once and used multiple times the significant difference in test execution times off-sets the drawback due to longer test generation time in model checking based ATG.

### 5 Conclusions and Future Work

This paper discusses our findings on applying model checking based test generation for two automotive controller examples designed using SL/SF models. We have developed a flow for automatic test-case generation (ATG) from SL/SF models using model checking. This includes the translation of SL/SF models into formal models implemented in SAL and test-case generation using the model checking engine. Few abstractions such as time discretization, type abstraction and sampling-time approximations were experimented during translation. Two automotive application examples viz., ATC and ACC, have been used for the case study. The results of our ATG were compared with a commercial test generation tool which uses random and guided simulation based techniques. The comparison reveals that the model checking based test generation approach allows generation of test data to achieve the desired coverages in a controlled
manner. The possibility of using abstractions on the formal model while retaining the necessary information in the model for test generation purpose leads to efficient tests with less number of input injection steps.

In summary, the model checking based ATG for SL/SF models, when combined with the abstractions suited for test generation, seems more promising than the conventional test generation techniques based on random data generation and simulations. However, the model checking based approach suffers from limitations due to scalability issues to industrial scale designs, which we plan to address in future through integration of various techniques such as predicate abstraction and counter-example guided abstraction for the purpose of test-case generation.

6 Acknowledgments

The authors thank their colleagues Sathyaraja H. Nandugudi, Manoj Dixit and A. C. Rajeev for valuable discussions that helped improve the quality of this work.

References
A  An Example SAL Module with Instrumentation

We give a generic SAL module for a sum block in the following.

```
sumblock : MODULE =
BEGIN
  INPUT in1, in2 : inputType
  OUTPUT out : outputType
  LOCAL freq : FREQRANGE
  LOCAL trap : boolean

  INITIALIZATION
    out = default;
    trap = false;
    freq = mfreq;

  TRANSITION
    [ freq = 0 -->
      out' = in1 + in2;
      freq' = mfreq;
      trap' = true;
    ]
    [ else -->
      freq' = freq - 1;
    ]
END;
```

B  Experimental Setup

The experimental set up used for evaluating the effectiveness of the test generation techniques is shown in Fig 4.

```
Fig. 4. Experimental setup.
```
C The SL/SF Models of ATC and ACC Systems

Fig. 5. The complete SL/SF model of the ATC system.

Fig. 6. The Simulink component in the ATC system.
Fig. 7. The Stateflow component in the ATC system.

Fig. 8. The complete model of the ACC system.
Fig. 9. The SL/SF model of the ACC mode subsystem.

Fig. 10. Mode decider Stateflow component of the ACC system.