Automatic test case generation from Simulink/Stateflow models using model checking

Swarup Mohalik‡, Ambar A. Gadkari*,§, Anand Yeolekar¶, K. C. Shashidhar| and S. Ramesh

India Science Lab, General Motors Global Research and Development, GM Technical Centre India Pvt Ltd., India

SUMMARY

Model-based test generation techniques based on random input generation and guided simulation do not satisfy the demands of high test coverage and completeness guarantees as required by safety-critical applications. Recently, test generation techniques based on model checking have been reported to bridge this gap. To evaluate the effectiveness of these techniques, an in-house tool suite, AutoMOTGen, has been developed for Simulink/Stateflow and applied on real-life case studies at General Motors. This paper outlines the test generation methodology of AutoMOTGen and gives a comparative study with a commercial, primarily random input-based, test generation tool on the same set of examples. The results indicate that in terms of coverage, model checking-based techniques complement the random input-based techniques. In addition, they provide proofs for unreachability that can aid in debugging the models. Therefore, it is recommended that model checking-based tools be utilized to complement and enhance the effectiveness of model-based testing methods in safety-critical systems engineering. Copyright © 2013 John Wiley & Sons, Ltd.

Received 5 October 2009; Revised 6 October 2012; Accepted 18 November 2012

KEY WORDS: model-based testing; automatic test generation; model checking; model translation; test coverage; Simulink/Stateflow

1. INTRODUCTION

Model-based development (MBD) is gaining increasing acceptance in software and systems engineering practice. This is especially true in the domain of automotive embedded control software design and development, where harsh time-to-market constraints have expedited its adoption. In this methodology, a set of context-specific models are built at the outset of the system development process. These models are executed and tested to validate the requirements, to check the fidelity of the model (e.g. no unreachable elements) and to verify platform-independent design choices. The debugged models serve as reference during the design, implementation and testing of the downstream artefacts in the system development process. Methods and tools to support the aforementioned testing-related activities in MBD are studied under the purview of model-based testing (MBT) [1].

A core enabler for MBT is the generation (either manual or automatic) of tests from the models for different test purposes. This is usually based upon certain metrics that measure the adequacy of the test suites towards addressing a given test purpose. For example, metrics for statement,
condition and decision coverage are related to detection of different types of errors in the software [2]. Similar metrics can be defined on the model elements and can be used not only to evaluate and compare different test suites but also to synthesize them automatically from the given models. Automatic generation of tests from models to achieve specified coverage metrics is the focus of the current paper.

Every automatic test generation method must take into account the underlying modelling language and its semantics. A variety of modelling languages are currently in use in MBD methodologies across different application domains. In the automotive and aerospace industries, the Simulink/Stateflow (SL/SF) [3] language is a popular choice for the development of embedded control software. The distinguishing aspect of this language is that it allows modelling both the continuous and discrete dynamics of the system and simulating it in a number of ways (e.g. fixed-step vs variable-step and discrete vs continuous solvers). It comes with a rich suite of toolboxes and design block sets providing a wide range of model analyses, making it a particularly compelling platform for system engineering.

Existing automatic test generation tools for SL/SF models primarily use a combination of random input generation and guided simulation. The scalability of these techniques has led to the use of these tools in many large industrial applications. However, random input-based techniques may not always provide the high coverage guarantees required in safety-critical applications. As a case in point, avionic standard DO-178B [4] requires complete Modified Condition Decision (MCDC) [5] coverage for the safety level A software. MCDC is also highly recommended for Automotive Safety Integrity Level D software by the upcoming ISO 26262 functional safety standard [6] in the automotive domain. Further, to certify that a test suite provides complete coverage, one must give a formal guarantee that the elements that are not covered by the test suite are indeed unreachable. This is not possible by tools employing primarily random input-based techniques.

Recently, it has been shown that the formal verification technique of model checking [7] can be used to automatically derive test sequences [8–11]. This approach for test generation relies on the capability of model checkers to exhaustively explore the state space of the system and generate traces or counter-examples of properties specified over the model. Thus, it is expected that model checking-based test generation can achieve complete coverage and provide formal guarantees of unreachability of model elements. However, there is no industrial strength, model checking-based test generation tool for SL/SF models. Therefore, an in-house tool, AutoMOTGen, has been developed, which automatically generates tests from SL/SF models and test specifications using

---

**Figure 1. Model-based testing using the AutoMOTGen tool.**
model checking. Furthermore, in case of incomplete coverage, the tool tries to prove the unreachability of the uncovered goals. Figure 1 shows one of the use cases of AutoMOTGen in a MBT environment, where the test cases generated by AutoMOTGen can be used to test the software code corresponding to the original SL/SF model. Thus, when the software code is independently developed, its conformance with the model can be verified. On the other hand, when the code is automatically generated (e.g. by Simulink Coder [12]), this can validate the translation mechanism itself.

Several issues arise when a model checking-based method is applied to large industrial models: (i) One must be able generate formal models preserving the behaviour of the original model, (ii) model checkers should be able to analyse all the constructs of the generated formal model and (iii) model checkers should scale for the large models avoiding the state space explosion. Although truthful translations to formal models are possible, approximations may be necessary to work around the limitations of model checkers. Moreover, for large models, complete exploration of the state space may not be possible. In this paper, an overview of the test generation methodology of AutoMOTGen is given to illustrate the designs incorporated in the tool to address these issues. These include the translation of SL/SF models to SAL [13] (a formal language and tool suite from SRI), encoding of the coverage specifications in the formal model, introducing some non-trivial approximation schemes employed during translation and using the bounded model checking-based test case generation capability of the sal-atg tool.

Because of the difficulties mentioned earlier, model checking-based approaches also may not be able to provide complete coverage and guarantee for unreachability. In such situations, the pertinent question is how do the model-checking techniques compare vis-a-vis the random input-based techniques in terms of coverage and scalability? In this paper, this question is addressed empirically by comparing the results from AutoMOTGen, the in-house model checking-based tool and Reactis [14], a popular tool in the industry employing random input-based techniques. The results have been obtained from a number of industrial case studies from General Motors (GM). Coverage results have also been collected from Simulink Design Verifier (SDV) [15], a recent tool from MathWorks, which generates test cases using formal verification.** The experimental results indicate that AutoMOTGen and SDV can achieve coverage comparable with that of Reactis. It is also observed that the coverages by Reactis and AutoMOTGen are complementary in nature. Thus, the test suites from the tools can be combined to obtain better coverage than individual test suites separately. Moreover, AutoMOTGen provides proofs of unreachability of uncovered goals that can aid in debugging the models. Therefore, it seems profitable to use model checking-based tools to complement and enhance the effectiveness of MBT processes.

Preliminary results of our work have appeared in Gadkari et al. [16]. Also, a demo of AutoMOTGen tool has been presented in Gadkari et al. [17]. The work presented here describes the basis for the algorithms implemented in the tool. It also confirms the earlier experimental results through significantly larger case studies carried out on real examples from industrial controller designs. In summary, the main contributions of this paper include the following:

- Implementation of an industrial strength automatic test generation tool for SL/SF based on model checking-based techniques.
- Description of the design considerations in AutoMOTGen for automatic translation of SL/SF to SAL, showing the instrumentation of the coverage specifications.
- Discussion of the approximations introduced to handle the limitations of the SAL model checkers and their impact on test generation.
- Experimental results of the application of AutoMOTGen, Reactis and SDV on medium to large size models from automotive controller designs within GM.

The paper is organized as follows. Section 2 discusses related work in the literature. Section 3 gives an overview of SL/SF, the methodology of model checking-based test generation, the SAL suite of model-checking techniques and the architecture and usage of the AutoMOTGen tool. The

**The results obtained from Simulink Design Verifier are partial because the tool did not provide support for some blocks used in the selected case studies.
translation algorithm and the approximation schemes employed in AutoMOTGen are described in Section 4. Experimental results on the comparative study are tabled and discussed in Section 5. Section 6 concludes the paper with a summary and some directions for future research.

2. RELATED WORK

A host of methods address the problem of model-based test generation. The distinction between them lies in, among other things, the addressed modelling paradigm, the test generation method and the supported coverage criteria. Owing to the large body of work on the theory and practice of MBT in the recent literature alone (cf. [18–24]), the inclined reader is referred to the existing surveys and expository literature (cf. [20, 25–29]). In this work, the discussion of related work is restricted to automatic test generation techniques for SL/SF models.

The choice of methods and tools for test generation depends on various factors, primary being the effectiveness (capability to detect bugs) of the generated test cases for the specific application domain. Some of the other factors influencing the choice of the right method or tool in an industrial setting are scalability, ease-of-use, degree of automation, portability of test suites and so on, as discussed in Pretschner et al. [30] and Sinha et al. [31]. Commercially available automatic test generation tools such as BEACON [32], BTC Embedded Tester [33], Reactis [14] and STB [34] take as input SL/SF models of control software applications and automatically generate test cases. Among these tools, Reactis is being widely used in the automotive domain, and it is compared with AutoMOTGen using the case studies in our experiments. There is little or no information available in the public domain related to the actual test generation algorithms that these tools implement. The information that one could glean from the data sheets and user guides of some of these tools, in conjunction with the in-house tool evaluation reports, suggests that these tools predominantly use random input generation methods coupled with some constraint solving for guided simulation to enhance the coverage; but as discussed in the introduction, they are found wanting in the context of development of safety-critical applications. The safety standard DO-178B imposes coverage of code and MCDC conditions. However, if a segment of code or MCDC condition is unreachable, the percentage of coverage will be less than 100%. In this case, the test generator should prove the unreachability of code/conditions and generate tests that can cover all the reachable code/conditions.

At present, development of such symbolic analysis algorithms to aid automatic test generation methods for expressive models as captured by SL/SF is still an active area of research (cf. [35–39]). Recently, a tool known as SDV has been introduced by MathWorks [3], capable of formal property checking and test case generation for SL/SF. SDV has been evaluated using the models from our case study. SDV is still evolving in terms of its capability in handling the variety of SL/SF blocks and the size of industrial models.

Our focus is on developing an MBT tool suite based on a mature formal verification technique such as model checking. It is expected that such a tool suite complements the random method-based tools in realizing a rigorous testing process in the development of safety-critical systems. The main issue in the application of formal verification techniques to SL/SF models is the need to have a formal semantics of SL/SF. Because there is no published formal semantics of SL/SF, one has to first develop a formal semantics of SL/SF models suited for the analysis (verification or test generation) at hand and implement it in a translation tool. In one of the proposals [40], a subset of Simulink models is translated to NuSMV for formal verification of LTL or CTL specifications, but the translation does not handle Stateflow charts. Further, because of the scope of the target language NuSMV, the translation output is constrained to be a finite state model. This constraint forces one to approximate the signal values by fixed precision and bounded ranges. However, the large number of signals and large ranges result in extremely large state spaces making it less amenable to finite state verification. Indeed, there is no reported result on the application of this translation method to realistic case studies. Some verification techniques for infinite state transition systems have been developed (cf. [41]), but they have not yet been applied to SL/SF models. In another proposal [42], an intermediate format called Hybrid Systems Interchange Format has been given for Simulink and a very restricted subset of Stateflow. There are not many analysis tools supporting this format. Currently, only a simulator called HyVisual [43] is available in the Hybrid Systems Interchange Format toolset.
The most comprehensive translation mechanism so far has been implemented in the ss2lus tool [44,45] in the SCADE tool suite. The method translates a large subset of the SL/SF language to the Lustre language [46]. However, the methods target formal property verification of SL/SF models and not test generation for which certain aggressive approximation methods can be employed. There exists an automatic test case generator for Lustre called Lurette [47], but it has not been evaluated for test case generation from SL/SF models (in conjunction with the ss2lus translator).††

3. MODEL CHECKING-BASED TEST GENERATION FOR SL/SF MODELS

In this section, an overview of the SL/SF language and the simulation semantics, the test generation methodology based on model checking and the supporting modules of the SAL framework is given.

3.1. Overview of Simulink/Stateflow

Simulink is a part of the MATLAB suite of tools widely used for modelling, simulating and analysing dynamical systems [3] such as automotive control systems. It provides a graphical user interface for building models as block diagrams. The blocks model signals (real-valued functions of time) and transfer functions over both continuous and discrete sampled time. Each block has a number of parameters, for example, initial value and ranges, which control the operation of the block. The input and output ports of a model represent the input and output signals, respectively. The connectors across the blocks represent a dataflow relation to compose smaller blocks into a larger system. Block parameters and signals have types that can be explicitly specified or are derived by a set of inheritance rules by SL/SF. Objects for which a type cannot be derived are assigned the float type by default.

Simulink provides a comprehensive block library including complex numerical and control-theoretic computations. The block library can also be extended by users to include customized block sets. These blocks can be broadly categorized into function and stateful blocks. The output of the function blocks represents transformations of only the current input signal. Examples of function blocks are logical and mathematical operations, user-defined functions and switches and so on. On the other hand, output of the stateful blocks is dependent upon both its current state and the current input signal. Delay, integrator blocks and transformations are examples of stateful blocks. The stateful blocks can further be categorized into discrete and continuous blocks. In the discrete stateful blocks, the state does not change between time steps, whereas the continuous stateful blocks update the state continuously according to specified differential equations. For example, a continuous integrator block represents the differential equation $\dot{y} = x$, where the output signal $y$ evolves according to a differential equation that is a function of the signal $x$. To solve the differential equations used to specify dynamics of signals, Simulink provides continuous and discrete solvers. Solvers can be configured for either variable or fixed-step sampling. Figure 11 shows a small Simulink model consisting of input and output ports, an arithmetic design, a two-dimensional (2D) lookup table, a gain block and an integrator block.

Stateflow [3] is an independent graphical design tool that works with Simulink to model and simulate event-driven or reactive systems. Event-driven systems transition from one operating mode to another in response to events and conditions. Stateflow charts model the mode changes as a transition system. Stateflow allows input and output data, events for triggering other Stateflow charts, and actions and conditions that can be attached to states and transitions. Other modelling abilities include hierarchical states and charts, parallel states, temporal operators, flowcharts using junctions and so on. From the point of view of Simulink, a Stateflow chart can be seen as a discrete stateful block and can be composed with other Simulink blocks in the usual manner. Figure 15 of Section 5.1.3 shows a Stateflow chart with an AND-state decomposed into four OR states.

The basic SL/SF blocks can be composed together to design subsystems. A complete model is built from basic SL/SF blocks and subsystems in a hierarchical fashion. Simulink provides special

††There have been some more reports [48–50] on formal verification and test case generation from Simulink/Stateflow after the submission of this paper.
blocks such as subsystem and enabled and triggered subsystem for encapsulating composition of smaller subsystems.

The simulation of an SL/SF model is governed by (i) the type of solver selected, (ii) the simulation step size and (iii) the mode (normal, accelerator and rapid accelerator). The simulation of a model proceeds in a series of steps. At each step, the selected solver updates the states and then computes the output of the blocks according to an order satisfying the dataflow relation among the blocks. SL/SF models may include blocks that execute only when triggered by certain events (e.g. triggered subsystem). Such blocks may remain inactive in a given simulation step depending upon the trigger conditions.

3.2. Tests and coverage criteria

A test for an SL/SF model is a timed sequence of input and internal/output signal values. The adequacy of a test suite for an SL/SF model is measured through different coverage criteria. The following coverage criteria and their descriptions are as given by the Verification and Validation (V&V) toolbox from MathWorks.

1. Block and subsystem coverage: A test covers a block or subsystem, if the simulation run on the test inputs causes the block/subsystem to be triggered.
2. Decision coverage: Decision coverage analyses elements that represent decisions in a model, such as a Switch block or Stateflow states. A decision is covered by a test if the simulation path for the test produces the decision.
3. Condition coverage: A test suite covers a condition if it causes each input to each instance of a logic block in the model and each condition on a transition to be true at least once during the simulation and false at least once during the simulation (on possibly another test).
4. MCDC coverage: It analyses blocks that output the logical combination of their inputs and Stateflow transitions to determine the extent to which the test case tests the independence of logical block inputs and transition conditions. A test case achieves full coverage for a block when there is a pair of simulation times, upon changing only one input, that causes a change in each block’s output. A test case achieves full coverage for a transition when there is at least one time when a change in the condition triggers the transition for each condition.

A coverage goal is defined as a set of values for some signals. For example, MCDC goals for an AND block are defined as the values 11, 10 and 01 for the two input signals. A test covers a goal if it occurs in a simulation run on the test inputs. A goal (block, subsystem, condition or decision, MCDC) is unreachable if no simulation run of the system covers the goal. Given a model and a coverage specification, the effectiveness of any method for automatic test case generation is judged from (ii) coverage: the percentage of the goals (among all reachable goals) covered by tests that are generated by the method and (ii) unreachability: the number of goals that are proved to be unreachable.

3.3. Model checking-based test generation

A model checker takes a state transition system and a set of properties as inputs and verifies whether the properties hold true for all runs of the system. A model checker can provide witness traces if a property holds for all the runs of the system and returns a counter-example trace in case of property violation. These traces can be used for test generation and debugging. To apply model checking-based methods to SL/SF models, one translates the models to a state transition model and the coverage specifications to properties that can be verified on the model. This is depicted pictorially in Figure 2.

In model checking-based test case generation, for each coverage goal, a Boolean state variable called a trap or goal is instrumented in the transition system model. A goal \( g \) is set to true in a transition when the values of signals associated with the element are true. For a goal \( g \), the corresponding property is set as \((\square \neg g)\) (read as never \( g \)). Then, if the property is violated, it indicates that \( g \) is reachable and the counter-example trace gives the simulation run leading to the state where \( g \) is true. One can then extract the inputs out of the trace as the test case.
Symbolic model checking using BDDs and bounded model checking (BMC) using constraint solving methods are well-known model-checking techniques. Because of its capability of capturing bugs early, and fast constraint solvers, BMC is becoming increasingly popular for test generation. With the techniques of satisfiability modulo theories [51], BMC is extended to handle infinite state transition systems.

For infinite state systems, because complete state space exploration is not possible using symbolic techniques, a sound technique called $k$-induction [52] has been proposed based upon BMC. During this process, one systematically increases $k$ to either find a proof of unreachability of a coverage element or find a test case covering it. However, because no a priori bound can be put on $k$, the termination of the procedure is usually decided by resource limits.

Check for unreachability is more complicated when an existential abstraction (over-approximation) $A$ of the original system $M$ is used. In this case, the unreachability of a model element $g$ in $A$ implies its unreachability in the original model $M$ as well. However, if there is a reachable trace in the abstract transition system $A$, the trace may be spurious, that is, $g$ may not actually be reachable in the original system. Hence, the following scenarios may occur for unreachability of an element $g$ in an abstract transition system:

1. It is proved to be unreachable in $A$ in which case it is unreachable in $M$.
2. It is reachable in $A$ in which case a trace is produced in $A$. This trace is simulated in the original model. If $g$ is reached through the simulation, then $g$ is reachable. Otherwise, the trace is rejected, the model is refined and the model is further explored for verification of unreachability. This is the well-known counter-example guided abstraction refinement approach [53].
3. It is neither proved to be unreachable nor can $k$-induction show its reachability within available resources. In this case, the status of $g$ is undecided, and $g$ is termed uncovered.

3.4. SAL, sal-bmc and sal-atg

SAL is a language and associated suite of tools, to specify and analyse discrete transition systems [13]. A SAL specification is a context: It contains a number of modules modelling transition systems and a composition of these modules modelling the entire system. In Figure 3, the template of a SAL module and a context are given.

The sal-bmc tool is a bounded model checker for SAL specifications. Given a safety specification $\phi$, sal-bmc checks for the violation of $\phi$ up to a specified depth. In case of violation, it
returns a counter-example trace. This tool is used to test the reachability (within a given bound \(d\)) of specified Boolean conditions \(C\), by checking \(\neg C\) for the bound \(d\). The counter-example trace, if found, witnesses the occurrence of \(C\) within bound \(d\). One can try to prove the unreachability (for all depths) by using \texttt{sal-bmc} for \(k\)-induction. The \texttt{sal-bmc} tool initially had only the facility for model checking of bounded models. An extension of \texttt{sal-bmc}, called \texttt{sal-inf-bmc}, handles unbounded models by using constraint solver engines that employ the technique of satisfiability modulo theories [54].

The \texttt{sal-atg} tool is an automatic test generator built on top of \texttt{sal-bmc}(\texttt{sal-inf-bmc}). It takes a SAL specification and a set of Boolean variables called \textit{traps} or goals. The \texttt{sal-atg} tool uses both symbolic model checking and BMC for given depths to explore the state space and find the states where the traps are set to true. The paths leading to the coverage of traps are stripped to obtain the tests (sequence of inputs). Thus, \texttt{sal-atg} generates a test suite to cover a set of traps. There is set of parameters that can control the way \texttt{sal-atg} traverses the state space.

Initially, \texttt{sal-atg} constructs an initial segment of length \(id\) (initial-depth), which covers at least one goal in the goal list. This goal is deleted from the goal list for further exploration. \texttt{sal-atg} then tries to extend the initial segment as specified by another user-defined parameter \(ed\) (extend-depth). The extension algorithm returns a segment that is assured to cover one or more new goals. If the current segment cannot be extended further to serve more goals, \texttt{sal-atg} starts from the initial state again (with a smaller goal list) to construct a new initial segment. This can result in two scenarios: (i) All goals are served; in this case, \texttt{sal-atg} returns the constructed paths and terminates. (ii) There are some unserved goals; in this case, with the given parameters (\(id\) and \(ed\)), these goals are unreachable from the current path. To cover more goals, one can either increase the initial-depth and extend-depth or use the \texttt{-branch} option. With this option, \texttt{sal-atg} starts a new exploration from the terminal state of the initial segment before falling back to the initial state. A couple of other options of \texttt{sal-atg} that lead to some optimization are the \texttt{--incremental} and \texttt{--slicing} options. In the former, the extension is carried out gradually up to the ed; hence, goals can be detected early. In the slicing option, the models are sliced with respect to the remaining goals leading to smaller models that can be processed faster.

### 4. ARCHITECTURE OF THE AUTOMOTGEN TOOL SUITE

The AutoMOTGen tool suite is built upon the concepts and engines described in the previous section. The tool takes an SL/SF model and a set of coverage criteria as input. The user can select from the provided structural criteria, viz., block coverage, condition coverage, decision coverage, MCDC coverage, lookup table coverage, states and transitions coverage. AutoMOTGen also provides certain options for the underlying test generator (\texttt{sal-atg}) to override the default values. For a given input model, it outputs a test suite in MATLAB’s M-file format, a coverage report and a list of unreachable goals for the model. Some screen shots of its user interface are shown in Figure 4, and its internal architecture is shown in Figure 5. An outline of the main components of the tool suite is given in the following.

```
T_1 : MODULE = BEGIN
   INPUT
   Declare input variables
   LOCAL
   Declare local variables
   OUTPUT
   Declare output variables
   INITIALIZATION
   Initialize local variables
   TRANSITION
   [ Transitions ]
END;

M : CONTEXT = BEGIN
   sampling_step = 0.1;
   T_1 : MODULE = BEGIN
      ...
   END;
M : CONTEXT = BEGIN
   T_2 : MODULE = BEGIN
      ...
   END;
M : CONTEXT = BEGIN
   T_1 | T_2;
END;
```

Figure 3. Schematic of a module and context in SAL.
Parser. The parser is a collection of MATLAB M-scripts that starts a MATLAB session, loads the SL/SF model, parses the model elements using MATLAB APIs and stores them as Java objects. The parser uses MATLAB-provided JAVA API to create a tree of these objects. The ranges for the primary inputs are derived depending upon their types. These can also be overridden by the user from his or her knowledge of the domain and application.

Translator. The object tree output by the parser is processed by the translator to produce a SAL model. Translation includes creating SAL modules and guards, identifying data types of variables corresponding to model blocks, producing assignment statements reflecting block functionality and finally, processing Stateflow to produce equivalent SAL modules. The translator outputs a
file containing a list of trap variables in SAL format based on the coverage criteria provided by the user. The translator can produce both under-approximate and over-approximate translations of non-linear blocks in the model. Details of these are provided in the following section.

**SAL-ATG engine.** sal-atg is the model checking-based test generator that generates input sequences covering the list of trap variables. Execution of sal-atg is controlled through different values for the parameters mentioned in Section 3.4. Note that although sal-atg generates a test sequence containing assignments to inputs and outputs of the SAL model, only the inputs in the test sequence are extracted. Assignments to output and various intermediate variables are discarded, as the SAL model may contain various abstractions. In this sense, sal-atg is used as an input generator, rather than a test case (which includes expected outputs as well) generator.

**Converter scripts and test harness.** The scripts are used to extract the input sequences from the test sequences generated by sal-atg and convert them to a suite of test inputs in the M-file format. These inputs are used to simulate the original SL/SF model by using Simulink’s simulator for deriving actual outputs, thereby generating test cases from the model.

**SL/SF model simulation.** The Simulink model simulator can be invoked using the sim command. It takes various arguments including timed inputs for each input port. In the cases where the SAL inputs are generated at a different (higher) time step, the simulation engine either interpolates the missing inputs or holds the last known input, depending on the input port settings. The root-level input and output ports of the model are logged to obtain the test oracle (expected outputs) via simulation.

**Simulink V&V toolbox.** Mathworks provides the Simulink V&V toolbox that is capable of reporting decision, condition and MCDC coverage obtained during simulation run(s). This toolbox is used to report coverage obtained by sal-atg-generated test cases.

**Unreachability checker.** This module requires a SAL model where non-linear blocks are over-approximated. The over-approximate model is used to check the unreachability of a set of goals using $k$-induction (cf. Section 3.3).

The various components of the AutoMOTGen tool are used in a flow as described in the following. AutoMOTGen first parses the SL/SF model to obtain a tree of Java objects corresponding to the model elements and invokes the translator to generate a SAL model from the object tree, the coverage specification and the time step. At this point, the non-linear blocks in the model are under-approximated. The resulting model is processed by sal-atg to generate a test suite for the coverage specification. If there are uncovered goals, the tool next invokes the translator to generate an over-approximate model with the fundamental sample rate as the time step and tries to prove unreachability by using the unreachability checker. Note that because most of the goals are reachable in practical examples, the initial step of test suite generation can cover many of them. Thus, the number of goals given to the unreachability checker in the second step is reduced, lessening the load on the expensive $k$-induction procedure. After the unreachability check, there may still be uncovered goals. The tool then automatically refines the options (increase id and ed and decrease the time step) and generates test suites to cover more goals until memory constraints prevent further refinement. The refinements can also be specified by the tester overriding the default scheme. At the end of the application of the model checkers, converter scripts translate the test cases generated by sal-atg to the test inputs in M-file format, the SL/SF model is simulated on these inputs and a coverage report is generated using the Simulink V&V toolbox.

5. TRANSLATION OF SL/SF MODELS TO SAL

The core component of AutoMOTGen is the translator that translates an SL/SF model to a transition system captured as a SAL context. In this section, a sketch of the translation mechanism is given along with some approximation schemes.

**Subset of Simulink and Stateflow.** Because the block sets of SL/SF are vast and MathWorks supports a variety of simulation mechanisms, the scope of AutoMOTGen is restricted to the blocks and simu-
Figure 6. Simulink blocks currently supported by AutoMOTGen.

Simulation configurations commonly used in the design of control applications in the automotive domain. Figure 6 gives all the Simulink blocks currently supported by AutoMOTGen. The tool also handles hierarchical Stateflow charts with event-based communication allowed only among the states at the same level. This subset was found to be sufficiently expressive for a wide array of case studies and avoids a number of problems such as event-stack overflow and early return logic, as are found in the general Stateflow charts (cf. [45] for more details).

**Sampling rate and ODE solver.** In addition, the following assumptions are made on the sampling rate and ODE solvers for the simulation semantics of the SL/SF models. Note that these restrictions can be relaxed depending upon the capability of the underlying model-checking engine.

- The models are uni-rate, that is, all the blocks in the model have a fixed sampling rate $\Delta$ called the fundamental sampling rate.
- For the continuous state update, the ODE solver is chosen to be the fixed-step Euler solver. Thus, the solution of an equation $\dot{x} = k$ is given as $x(n + 1) = x(n) + \Delta k$.

**Coverage specifications.** Currently, AutoMOTGen supports the following coverage specifications: block, condition, decision and MCDC. In the case of Simulink, MCDC coverage points are computed for each logical block. This is clearly inadequate to capture the independent effect requirement of MCDC coverage pronounced in the larger Boolean expressions (Kelly et al. [55]). However, at the time of evaluation, Reactis also defined MCDC in a similar manner. In the case of Stateflow, the MCDC coverage points for guards on transitions are pre-computed as in Rayadurgam et al. [56] and are used during translation. The generated test suite does not guarantee minimality. It could result, for example, in multiple-condition coverage. However, at the time of the implementation, the focus of AutoMOTGen was to address only the coverage performance vis-a-vis Reactis. Optimization of test suites was to be taken up only on a need basis.

5.1. Modelling

Signal types *Single* and *Double* in Simulink are mapped to the type *Real* in SAL. Int32, Int16 and Uint are mapped to bounded *Integer* types, and Boolean type is mapped to the Boolean type in SAL. Each signal in the SL/SF model is modelled as a SAL variable with the corresponding type.
Time is modelled using a global parameter $\delta_t$ that has the default value of $\Delta$, the fundamental sampling rate. This can be overridden by the user during the generation of SAL code. The purpose of keeping the sampling rate parametric is to facilitate a timestep approximation scheme, which is explained later in this section. Because of the assumption of fixed-step, continuous state solvers using the Euler method, the only blocks whose translation is affected by time are continuous and discrete integration blocks. The translation of these blocks is shown along with other blocks in the following.

### 5.1.1. Atomic blocks

Each atomic block is translated to a set of statements modelling the functionality and related coverage specifications. The statements use the variables corresponding to the input and output signals of the block and their *primed* versions. The latter is used to capture the value of output signals at the next time step.

The translation scheme distinguishes between direct feedthrough and non-direct feedthrough ports (of blocks). For example, for numerical or logical computation blocks of Simulink, the input signal, after transformation by the block, appears *instantly* at the block’s output port, that is, there is no time delay or state update for these blocks during their execution. Hence, the translation for these blocks uses primed versions of variables on both the LHS and RHS of the update expression in SAL. For stateful blocks, such as integrator and delay, the block’s output is *delayed* compared with the input signal timing, and hence, the non-primed versions of variables are used in the RHS. In addition, for integrator blocks, the evolution of the output during the previous sample step is modelled with the help of the time step $\delta_t$. The translations for some atomic blocks are explained in the following.

A mathematical operation `sum` block named `sum` with input signals `in1` and `in2` and output signal `out` is translated as in Figure 7(a). The Boolean variable `trap_sum'` is the goal variable associated with the sum block. Translations for the integrator block `integrator` and the delay block are given in Figure 7(b) and (c), respectively.

A logical operation `OR` block with input signals `in1` and `in2` and output signal `out` is translated as in the Figure 8. The first line generates the output depending upon the input signals. Lines 2–5 capture the condition coverage goals, lines 6 and 7 captures the decision coverage goals, and lines 8–13 capture the MCDC coverage goals. All the trap variables are set to be *latching*, that is, whenever a trap is true, it remains true forever.

Lookup table blocks are modelled by two SAL functions. The first function captures the lookup table entries (cf. Figure 9 for an example). The second function takes input signals as input parameters and finds the positions of the input (index $i$ such that the input value is between $table[i]$ and

```plaintext
(a) Sum block
out' = in1' + in2';
trap_sum' = TRUE;

(b) Integrator block
out' = out + in * delta_t;
trap_integrator' = TRUE;

(c) Delay block
out' = in;
trap_delay' = TRUE;
```

Figure 7. Modelling atomic blocks in SAL.

```plaintext
1 out' = in1' OR in2';
2 IF in1' THEN trap_cond_in1_true' = TRUE;
3 ELSE trap_cond_in1_false' = TRUE;
4 IF in2' THEN trap_cond_in2' = TRUE;
5 ELSE trap_cond_in2_false' = TRUE;
6 IF out' THEN trap_dec_out_true' = TRUE;
7 ELSE trap_dec_out_false' = TRUE;
8 IF (NOT(in1')) AND (in2') THEN trap_mcdc_or_l_1' = TRUE;
9 IF (in1' AND NOT(in2')) THEN trap_mcdc_or_l_2' = TRUE;
10 IF (NOT(in1')) AND (NOT(in2'))
11 THEN trap_mcdc_or_l_3' = TRUE;
12 IF (trap_mcdc_or_l_1' AND trap_mcdc_or_l_2')
13 AND (trap_mcdc_or_l_3') THEN trap_mcdc_or_l_1' = TRUE;
```

Figure 8. Modelling an OR block.
table[i + 1]). It then computes the output value corresponding to the inputs either by interpolation or by end values, depending upon the properties of the table.

Blocks outside the collection shown in Figure 6 cannot be exactly processed by the current version of AutoMOTGen. They are translated to code that produces all possible values in the range of the output of the block nondeterministically. Because the final coverages are measured on the original model, the check for spuriousness is handled implicitly.

5.1.2. Subsystems. Simulink subsystems are translated as a module in SAL. Inports and outports of the subsystem are translated as input and output variables of the module. A subsystem S with basic blocks connected together (referred to as a base subsystem) is translated as a SAL module with a single transition (cf. Figure 10(a)), where statement(bk) denotes the translation generated for block bk as indicated earlier. The trap, trap_S', is introduced to capture subsystem coverage.

Figures 11 and 12 show an example of a Simulink subsystem and its corresponding translation into a module in the SAL language, respectively. This subsystem is a part of the Automatic Transmission Control (ATC) System, available as a demo example in the MATLAB tool distribution. In the translated module, lines 2 and 7 model the integrator block with the condition to take care of the initialization value. Line 4 is the translation of the sum block, line 5 is a SAL function modelling

```
downth: tab_downth = [
    i: [1..tab_downth_len]
    IF i=1 THEN 0
    ELIF i=2 THEN 5
    ELIF i=3 THEN 40
    ELIF i=4 THEN 50
    ELIF i=5 THEN 90
    ELSE 100
    ENDIF
];
```

Figure 9. Model for a 1D lookup table.
the one-dimensional (1D) lookup table and line 6 corresponds to the gain block with 2.21 as the gain parameter. The output $Ne$ is updated to the new value of the integrator output in line 7.

When a base subsystem is encapsulated as a triggered or enabled subsystem with a trigger as its control input, the transition in the resulting SAL module looks as shown in Figure 10(b). A subsystem comprising of a number of subsystems is translated as a synchronous composition of the SAL modules for the smaller subsystems. Translation of base subsystems and triggered subsystems has been discussed earlier (cf. Figure 10). The control inputs are propagated to the transitions in the lowest level of hierarchy and conjuncted to the guard for all the transitions. This is shown schematically for an example system in Figure 13. The successive steps during the translation of such a system into a SAL model are as shown in Figure 14, where $[X1']$ and $[X2']$ denote the SAL modules corresponding to the subsystems $X1$ and $X2$ under the control input trigger1.

5.1.3. Stateflow. An AND state (irrespective of where it is placed in the state hierarchy) is translated to SAL as a single module. OR state siblings at each level in the state hierarchy are translated as a single SAL module, with a variable to keep track of the current active state. For modelling hierarchy, the parent–child link is simulated using Boolean variables. Finally, all the modules are composed synchronously to yield the complete chart. The state explosion resulting from flattening hierarchy and parallel states at the formal modelling stage is avoided by using synchronous composition of these states. The actual product operation is left for the model checker to handle efficiently.

The only events allowed are the non-nested ones between parallel states in the hierarchy. There is no explicit stack or queue for handling events; these are modelled as Boolean variables. If a sequence of actions share variables or event broadcast causes multiple updates to the same variable in a single execution cycle, the sequence of actions are discovered through data dependency analysis and is represented as multiple transitions in SAL.

Actions that update global variables in the chart may get spread over a number of SAL modules representing the chart. Because global variables cannot be shared in a synchronous composition, a module is created per chart variable. This module maintains the global copy of the variable. Every module that updates this variable (through some action) updates its local copy; all such updates are collected and transferred to the module holding the global state. This necessitates that there be at most a single update to the variable in any step.

Figure 15 shows a part of the Stateflow chart from the ATC example. Its translation in SAL is shown in Figure 16. The module gs models the OR decomposition state (with dotted outline) by just enabling the transitions for the OR states in the gs-child module. This is carried out in gs by setting
5.2. Approximations in the SAL models

Whenever the SL/SF models have bounded signals and linear operations, the aforementioned translation generates SAL models that are equal in behaviour to the original model, that is, for the same sequence of inputs separated by $\delta_t$, the SL/SF model and the SAL translation both have the same sequence of outputs. Even though there is no formal model of SL/SF, semantics of most blocks can be guessed and checked experimentally. For small, linear models, the conformance has been verified through experiments (similar to the approach taken in Tripakis et al. [45]). However, almost all industrial designs are large and have non-linear operations, for example, operations involving multiplication or division of unbounded variables, trigonometric expressions and so on. When the signal ranges are discrete and their ranges are bounded, SAL tools can transform some of the non-linear operations such as multiplication and divisions to a number of linear operations. However, this results in a very large number of transitions in the SAL models; because of which, tools cannot handle the scale. Therefore, there is a need for approximating these to generate SAL models with linear operations without blowing up the models.

Two types of approximations have been implemented in AutoMOTGen. The first type of approximation gives an over-approximate SAL model whose set of runs is larger than the set of runs of
the SL/SF model. This is possible because the SAL models have the provision of non-deterministic assignments to variables. Hence, if a test goal is unreachable in the SAL model, it is assured that the corresponding element is unreachable in the original model. The second type of approximation results in under-approximate SAL models whose set of runs is a restricted subset of the set of runs of the original model. This is created by restricting the input range of the blocks that are to be approximated. Thus, there may not be a behaviour in the SAL model for a sequence of inputs, but if an element is reachable in the SAL model, it is guaranteed to be reachable in the SL/SF model. Under-approximate models address the inability of model checkers in handling non-linear blocks while being conservative about coverage specification.

Note that both types of approximations are implemented in AutoMOTGen with certain default configurations. These configurations can be changed by the user to refine the models, within tolerable limits of computation overhead.

5.2.1. Over-approximation. Over-approximate translation has been provided for the following blocks: (i) multiplication, (ii) division, (iii) 2D lookup tables, (iv) trigonometric and (v) power blocks. The SAL translation for these blocks result in simpler transition systems with non-deterministic outputs. For example, an over-approximation corresponding to a product block with inputs \(x\) and \(y\) and output \(z\) can be translated in SAL as \(z' \in \{r: \text{REAL } | \text{TRUE}\}\), which non-deterministically outputs an arbitrary real number.

To refine the non-determinism, the tool provides custom translations for the blocks (cf. Figure 17). The translation partitions one of the operands \(x\) into ranges (lines 2–6), bounding the multiplication result for each range chosen. The actual result is still non-deterministic, but the degree of non-determinism is less because of the restricted bands of \(x\). The user can refine this model further (e.g. by adding more partitions) so as to be closer to the original computation. Similarly, 2D lookup tables are approximated by non-deterministically choosing all the values from the 2D box about the two operands (a 1D lookup table can be translated to a function with linear expressions). Refinement in this case involves breaking up the 2D boxes specified in the original table into a number of smaller boxes.

The modules are generated for each operation on the basis of the user-specified parameters. For example, the module for multiplication is generated from (i) the range of operands and (ii) a size for the intervals. The module for 2D lookup table requires the size for the smaller boxes.

5.2.2. Under-approximation. An under-approximate translation scheme for non-linear blocks such as multiplication, \(nD\) lookup tables, trigonometric operations, power and polynomials allows only those inputs for which the precise output values are known. An under-approximate SAL model ensures that the runs of the SAL model are a subset of the runs of the original model (opposite to the over-approximation). This is illustrated through the product block \(z = x \times y\), translated to a SAL model in Figure 18. Concrete values to one of the operands \(x\) (lines 2–5) ensure that a run (test case), if found, will have correct computation for \(z\). Line 6 ensures that no other runs can be generated, as the constraint is \textit{false} for any other value of \(x\). If a satisfying run cannot be found, the under-approximate implementation will have to be refined, by adding more concrete values of \(x\).

When precise output values for inputs are not known, the translated modules provide an approximate output. For example, for the multiplication block, one has additional statements of the form

\[
\text{ELSIF } y > 1 \text{ and } y < 10 \text{ then } x \times 5.
\]

Note that this is similar to over-approximation but

<table>
<thead>
<tr>
<th>Line</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(z' \in {r: \text{REAL }</td>
</tr>
<tr>
<td>2</td>
<td>\textbf{IF } x=0 \text{ THEN } {r: \text{REAL }</td>
</tr>
<tr>
<td>3</td>
<td>\textbf{ELSIF } 0&lt;x \text{ AND } x&lt;=10 \text{ THEN } {r: \text{REAL }</td>
</tr>
<tr>
<td>4</td>
<td>\textbf{ELSIF } 10&lt;x \text{ AND } x&lt;=100 \text{ THEN } {r: \text{REAL }</td>
</tr>
<tr>
<td>5</td>
<td>\textbf{ELSIF } 100&lt;x \text{ AND } x&lt;=1000 \text{ THEN } {r: \text{REAL }</td>
</tr>
<tr>
<td>6</td>
<td>\textbf{ELSIF } 1000&lt;x \text{ THEN } {r: \text{REAL }</td>
</tr>
<tr>
<td>7</td>
<td>ENDIF</td>
</tr>
</tbody>
</table>

Figure 17. Over-approximate multiplication in SAL.
without the non-deterministic assignments. It has been observed that this method results in better coverage than in a purely under-approximation scheme described in the previous paragraph. One possible reason is that the approximate output and the actual output may be within tolerable limits, thus not affecting the coverage of subsequent transition sequences. In AutoMOTGen, approximate modules are generated automatically using parameters for size of intervals and one of the *floor*, *median* and *ceiling* to choose the output. The aforementioned statement in the multiplication module is generated for the choice of *median*. For the choice of *floor* and *ceiling*, the statement would be \(\ldots x \times 1\) and \(\ldots x \times 10\), respectively.

### 5.2.3. Under-sampling approximation

Many models involve timers that introduce *wait* for some time (usually in seconds). Because the sampling time is usually of the order of milliseconds for automotive control systems, the model goes through a large number of steps before a response is computed or new state is entered. An exact translation to SAL would then require exploration of very large depths during SAL-ATG search for goals, which could be computationally expensive. To address this problem, an approximation is provided based on the sampling step parameter, which can be controlled by the user. The idea is to under-sample the model by increasing the timestep parameter. With larger timesteps, system states occurring after many transitions can be reached with fewer transitions. One can interpret this as modelling those runs of the system where inputs do not change for long intervals. In case of continuous state variables, the longer sampling steps will result in an inaccurate SAL model, specially because of the linear expression of the Euler equation. However, in practice, under-sampling procedure has been found to be of help in increasing test coverage. A possible reason is that control systems are often modelled around certain modes (steady states), where the dynamics are robust (the models are not extremely sensitive to the input changes). Another reason is as follows: If in the approximation the value of a continuous variable crosses a limit, it is going to cross the limit in the original model as well. As an illustration, the performance of under-sampling approximation on the ATC model was noted, where gear changes occur only after speed of the vehicle crosses a threshold. This can take a large number of transitions with small sampling steps. With a timestep of 0.04 s (the default value), only 19% of the decision goals could be covered. Increasing the timesteps to 0.4 and 1 s, the coverage increased remarkably to 38% and 63%, respectively.

### 6. INDUSTRIAL CASE STUDIES

One of the major aims of this paper is to collect data to compare the performance of model checking-based test generation techniques vis-à-vis random input-based ones. Towards this aim, in this section, the benchmark models for the case studies are described, and a number of different types of data are identified. The method followed for the experiments, and data collection is given next. Finally, the implications of the gathered data are discussed and summarized.

#### 6.1. Benchmarking applications

To explain the experimental setup and evaluation methodology, two representative applications, viz., a model from the Mathworks distribution and a model adapted from the Reactis distribution, have been selected. The main evaluation is carried out using case studies based on different control system applications obtained from engineering groups within GM. Although it is not possible to discuss all

```
1 z' IN
2 IF x=0 THEN (r:REAL | r=0)
3 ELSIF x=10 THEN (r:REAL | r=10*y)
4 ELSIF x=100 THEN (r:REAL | r=100*y)
5 ELSIF x=1000 THEN (z:REAL | z=1000*y)
6 ELSE (r:REAL | r>0 AND r<0)
7 ENDIF;
```
S. MOHALIK ET AL.

the details of the proprietary models because of confidentiality reasons, an overview of these applications along with certain model characteristics, such as model size, number of inputs and outputs, number of coverage goals and prominent block types, are provided. These characteristics give an indication of the size and complexity of the case studies.

**Automatic Transmission Controller (ATC).** The ATC model is available as an example in the MATLAB tool distribution. This model implements a controller for a four-gear automatic transmission based on speed, brake and throttle inputs. The controller (gear and selection logic) is modelled in a Stateflow chart, and the vehicle, transmission and engine subsystems are modelled in Simulink blocks. The threshold calculation block decides the thresholds for gear shift, depending on the current gear position and throttle inputs. The Selection logic block in the controller 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 behaviour and outputs the rpm on the basis of the external throttle input. The transmission subsystem outputs the torque on the basis of the current gear value and the engine speed in rpm. The vehicle subsystem models the calculation for vehicle speed on the basis of throttle and torque inputs, and various physical factors such as aerodynamic drag, friction, drive ratio and vehicle inertia. The Stateflow chart is hierarchical and consists of a total of nine states. It is an AND state with two AND-siblings; each of which is an OR state with event-broadcasting mechanism for communication. The Simulink portion includes 1D and 2D lookup tables, gains, integrators, non-linear multiplication operation and feedback loops.

**Adaptive Cruise Controller (ACC).** The ACC model is derived from the CC model available in the Reactis distribution. It uses driver inputs, environment information such as vehicle’s current speed, leader’s absence/presence and separation distance and so on through the ACC radar subsystem and outputs the host vehicle’s speed at any given instant of the model’s execution. From the environment inputs, the Mode selector 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 one 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 65 Simulink blocks and three Stateflow diagrams. The model also consists of two lookup tables, one for calculating the safe distance 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.

**Real-life Controller Examples.** The Electronic Stability Control (ESC) is designed to provide enhanced stability and manoeuvrability to the vehicle. It integrates various chassis control subsystems to detect and mitigate vehicle directional error. ESC determines the control actions that will help provide stable handling and adjust vehicle direction towards driver intent. Real-time control of the brakes, steering, suspension subsystems and powertrain torque commands is utilized to enhance vehicle stability and manoeuvrability.

The **Performance Traction Control (PTC)** is designed to prevent loss of traction of the driven road wheels, and therefore the control of the vehicle, when excessive throttle is applied by the driver and the condition of the road surface (because of varying factors) is unable to cope with the torque applied. PTC is a feature that manages powertrain output to maximize tractive effort under racetrack conditions. PTC uses inputs from the driver, wheels and vehicle inertial sensors to manage the maximum drive torque delivered by the engine and transmission. It is designed to facilitate driver modulation of engine torque near the tractive limits of the driven wheels.

The **Electronic Throttle Control (ETC)** is responsible for regulating the air flow into the engine based on the driver’s request for power and the current throttle position coming from the Powertrain Control Module. It typically consists of the Throttle Body, Accelerator Pedal Mechanism and Engine Control Module. The ETC implements a number of functions, viz., sensing accelerator pedal position and driver intent, throttle positioning to meet driver and
engine/powertrain/vehicle demands, providing accelerator pedal position feedback to the driver, managing internal diagnostics, controlling emitted energy (audible and electromagnetic) and power consumption.

The Heating, Ventilation and Air Conditioning (HVAC) system mainly controls the heating/cooling, humidity and airflow and orchestrates the different air distribution modes in the passenger cabin. In addition, it provides heating/cooling of the seats, arbitrates between conflicting requirements for resources from different modules and optimizes the energy usage ensuring passenger comfort and safety at the same time.

The Active Safety Control (ASC) system includes various features related to occupant safety. These include functions to sense any imminent collision (front, back and the sides) during the vehicle in motion and take corrective actions to either avoid the collision or prepare the vehicle for minimizing the impact through a combination of braking and steering and in collaboration with other systems such as seat belts and air bags.

The various model characteristics for these applications are shown in Table I. The model sizes vary from as small as 37 blocks to more than 900 blocks. The block types such as constants, inports, outports and action ports are common to all the models and hence are not shown in the table.

6.2. Experimental setup

The experimental setup used for evaluation is shown in Figure 19. For each model in the benchmark, one first generates the test suites for condition, decision and MCDC coverage. These test cases are converted to a common format, namely, the M file format of MATLAB, and coverages are

<table>
<thead>
<tr>
<th>Model name</th>
<th>Size</th>
<th>I</th>
<th>O</th>
<th>D</th>
<th>C</th>
<th>M</th>
<th>Main block types</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETC1</td>
<td>37</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>18</td>
<td>9</td>
<td>Logic, Relation, Switch, IERElseIf</td>
</tr>
<tr>
<td>PTC1</td>
<td>59</td>
<td>7</td>
<td>1</td>
<td>18</td>
<td>22</td>
<td>11</td>
<td>Logic, Relation, Switch, IERElseIf, MinMax</td>
</tr>
<tr>
<td>ESC1</td>
<td>59</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>Merge, Demux, 2D-LUT, IERElseIf, MinMax</td>
</tr>
<tr>
<td>ESC2</td>
<td>107</td>
<td>8</td>
<td>1</td>
<td>21</td>
<td>26</td>
<td>12</td>
<td>Logic, Relation, Switch, Math, Stateflow</td>
</tr>
<tr>
<td>ASC1</td>
<td>124</td>
<td>8</td>
<td>4</td>
<td>126</td>
<td>0</td>
<td>0</td>
<td>Relation, Switch, IERElseIf, MinMax, Sum</td>
</tr>
<tr>
<td>HVAC1</td>
<td>129</td>
<td>9</td>
<td>1</td>
<td>22</td>
<td>46</td>
<td>20</td>
<td>Logic, Relation, Switch, Delay</td>
</tr>
<tr>
<td>ETC2</td>
<td>135</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>48</td>
<td>18</td>
<td>Logic, Relation, If, Merge, LUT</td>
</tr>
<tr>
<td>ASC2</td>
<td>156</td>
<td>8</td>
<td>3</td>
<td>248</td>
<td>94</td>
<td>32</td>
<td>Relation, IERElseIf, Math</td>
</tr>
<tr>
<td>PT3</td>
<td>199</td>
<td>5</td>
<td>4</td>
<td>32</td>
<td>22</td>
<td>10</td>
<td>Logic, Switch, IERElseIf, Merge, Math, Delay</td>
</tr>
<tr>
<td>HVAC2</td>
<td>227</td>
<td>13</td>
<td>1</td>
<td>34</td>
<td>62</td>
<td>29</td>
<td>Logic, Relation, Switch, Math, Delay, Mux</td>
</tr>
<tr>
<td>PTC2</td>
<td>304</td>
<td>19</td>
<td>24</td>
<td>98</td>
<td>98</td>
<td>50</td>
<td>Logic, Relation, Switch, IERElseIf, MinMax, Abs, Math, Delay, Demux, DigitalLowPass</td>
</tr>
<tr>
<td>HVAC3</td>
<td>341</td>
<td>14</td>
<td>7</td>
<td>97</td>
<td>74</td>
<td>36</td>
<td>Logic, Relation, Switch, Math, Demux, Delay, 2D-LUT, Signal Conversion, Mux</td>
</tr>
<tr>
<td>PTC3</td>
<td>347</td>
<td>10</td>
<td>5</td>
<td>32</td>
<td>76</td>
<td>32</td>
<td>Logic, Relation, Switch, IERElseIf, Math, Delay, DigitalLowPass</td>
</tr>
<tr>
<td>ETC4</td>
<td>436</td>
<td>10</td>
<td>6</td>
<td>52</td>
<td>10</td>
<td>4</td>
<td>S-function, Legacy Code, IERElseIf, MinMax, Math, Delay, SwitchCase</td>
</tr>
<tr>
<td>PTC4</td>
<td>462</td>
<td>20</td>
<td>5</td>
<td>105</td>
<td>120</td>
<td>49</td>
<td>Logic, Relation, Switch, Abs, IERElseIf, Merge, Math, Delay, DigitalLowPass, SwitchCase</td>
</tr>
<tr>
<td>ETC5</td>
<td>514</td>
<td>14</td>
<td>7</td>
<td>75</td>
<td>8</td>
<td>2</td>
<td>Switch, Legacy Code, IERElseIf, MinMax, Math, Delay, SwitchCase, Cal1D</td>
</tr>
<tr>
<td>PTC5</td>
<td>584</td>
<td>8</td>
<td>4</td>
<td>100</td>
<td>232</td>
<td>96</td>
<td>Logic, Relation, Switch, IERElseIf, Abs, Merge, Math, Delay, LUT, MinMax</td>
</tr>
<tr>
<td>ESC3</td>
<td>632</td>
<td>9</td>
<td>16</td>
<td>92</td>
<td>32</td>
<td>16</td>
<td>Logic, Relation, Switch, Mux, K112D-LUT, Demux, MinMax</td>
</tr>
<tr>
<td>ESC4</td>
<td>711</td>
<td>20</td>
<td>2</td>
<td>106</td>
<td>82</td>
<td>38</td>
<td>Logic, Relation, Switch, IERElseIf, Merge, LUT, Demux, Math, Delay, Mux</td>
</tr>
<tr>
<td>ESC5</td>
<td>901</td>
<td>34</td>
<td>17</td>
<td>138</td>
<td>104</td>
<td>48</td>
<td>Logic, Relation, Switch, IERElseIf, Merge, Demux, Math, Delay, Mux</td>
</tr>
</tbody>
</table>

Size = #blocks; I = #inputs; O = #outputs; C = #condition; D = #decision; M = #MCDC goals.

measured using the Simulink V&V toolbox provided by MathWorks. For each of the models in the benchmark, along with the generated tests, the following data are collected:

1. Condition, Decision and MCDC coverage as a percentage of the respective coverage elements.
2. Coverage obtained by combining the tests of AutoMOTGen and Reactis.
3. Number of uncovered and provably unreachable elements.
4. Time taken for the generation of the test suite.
5. The size of the test suite, that is, the total number of test steps in all the individual tests.

Although the coverage and unreachability results are primary, test generation time and test suite sizes are important indicators of the usability of the tools.

6.2.1. Experiments with AutoMOTGen. For every model, the ranges and initial values for the input variables are specified in the tool. Then, the coverage specifications are selected and values for the following options provided by sal-atg: lengths of initial segment (id) and extended segment (ed), and options for branching, model slicing and incremental test generation. The tool first generates a test suite for the specified coverage specification, and the size of the test suite and time required for generation are noted. Then, the tool generates an over-approximate model for the same coverage specification and checks the unreachability of the uncovered goals from the test generation phase. For the still uncovered goals, the tool carried out the refinements (increase id and ed and decrease the time step) until either all the goals are covered or the resource limits are encountered.

Different combinations of the sal-atg options result in different coverage. Table II shows some experimental data for the ESC2 model. The trend shows that with a small id (10) and large ed (35), with all the others options set to True, the coverage value saturates. On the other hand, with very large values of ed, sal-atg aborts because of memory limitations.

6.2.2. Experiments with Reactis. For the experiments with the Reactis tool, ranges of the input ports are given first. The two controls for test generation in Reactis are the number of test steps of the random and the targeted (guided) phases, where one can also choose the coverage specification. Starting with lowest values, the number of test steps for both random and targeted phases are increased gradually to achieve better coverage with minimum lengths. The iterations are stopped when increase in the number of test steps did not increase the coverage. At this point, the minimal length test cases achieving the maximum coverage and corresponding data for time of generation and test suite sizes are logged. A sample of different options used on ATC and ACC examples with their respective coverages is shown in Table III.
### Table II. Coverage obtained from test cases generated using AutoMOTGen.

<table>
<thead>
<tr>
<th>Test suite id</th>
<th>ed</th>
<th>branch</th>
<th>incremental</th>
<th>innerslice</th>
<th>No. of test cases</th>
<th>Test suite length</th>
<th>Condition</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>1</td>
<td>20</td>
<td>96</td>
<td>68</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>1</td>
<td>20</td>
<td>93</td>
<td>68</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>20</td>
<td>96</td>
<td>68</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>1</td>
<td>20</td>
<td>96</td>
<td>68</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>35</td>
<td>95</td>
<td>70</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>33</td>
<td>80</td>
<td>62</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>1</td>
<td>37</td>
<td>48</td>
<td>52</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>23</td>
<td>87</td>
<td>62</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>1</td>
<td>19</td>
<td>87</td>
<td>62</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>28</td>
<td>91</td>
<td>72</td>
</tr>
<tr>
<td>11</td>
<td>50</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>1</td>
<td>31</td>
<td>91</td>
<td>64</td>
</tr>
<tr>
<td>12</td>
<td>50</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>1</td>
<td>42</td>
<td>78</td>
<td>67</td>
</tr>
<tr>
<td>13</td>
<td>15</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>1</td>
<td>23</td>
<td>93</td>
<td>68</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>35</td>
<td>95</td>
<td>72</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>1</td>
<td>35</td>
<td>96</td>
<td>73</td>
</tr>
</tbody>
</table>

### Table III. Coverage obtained from test cases generated using Reactis.

<table>
<thead>
<tr>
<th>Test suite No. of tests</th>
<th>Steps/test</th>
<th>Steps test cases</th>
<th>Test suite length</th>
<th>Condition</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application 1: The Automatic Transmission Controller System</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>—</td>
<td>100</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>1000</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>3000</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>15000</td>
<td>5</td>
<td>153</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>50000</td>
<td>2</td>
<td>102</td>
</tr>
<tr>
<td>6</td>
<td>—</td>
<td>Y</td>
<td>100000</td>
<td>5</td>
<td>1707</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>100</td>
<td>1000</td>
<td>1</td>
<td>74</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>100</td>
<td>1000</td>
<td>5</td>
<td>165</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>1000</td>
<td>20000</td>
<td>4</td>
<td>1820</td>
</tr>
<tr>
<td>Application 2: The Adaptive Cruise Controller System</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>—</td>
<td>100</td>
<td>4</td>
<td>64</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>—</td>
<td>500</td>
<td>6</td>
<td>275</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>6000</td>
<td>10</td>
<td>230</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>100</td>
<td>1000</td>
<td>5</td>
<td>276</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>100</td>
<td>20000</td>
<td>11</td>
<td>456</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>100</td>
<td>100000</td>
<td>9</td>
<td>2529</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>1000</td>
<td>1000</td>
<td>5</td>
<td>1407</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>1000</td>
<td>100000</td>
<td>12</td>
<td>1971</td>
</tr>
</tbody>
</table>

It is noted that in many cases even when coverage indicated by the Reactis tool (with respect to the coverage criteria of Reactis) shows 100%, the actual coverage measured by Simulink V&V tool (with respect to its criteria) is less than 100%. In such cases, the values of the control parameters for Reactis (number of steps in random and directed phases) are increased to generate more tests and achieve maximum coverage with respect to Simulink V&V criteria. This is reported in Table III.

### 6.3. Results and discussion

The data obtained from the experiments are analysed and compared in the following.

#### 6.3.1. Test suite sizes

In AutoMOTGen, the test suite sizes are significantly smaller than those in Reactis, although the simulation times are similar. This is due to the time approximation in
AutoMOTGen, which leads to input values being changed at larger intervals. In Reactis, the inputs to the models are (possibly) changed at the fundamental sampling rate. A closer inspection reveals that most of the rows of inputs in the Reactis generated test cases repeat, and hence, they too can be presented in a compact format. With this interpretation, the test suites generated from the two tools are comparable in size.

6.3.2. Test generation times. For each iteration of the Reactis tool, the test generation time is in the order of few seconds, whereas in the case of AutoMOTGen, it is in the order of minutes. It can be argued that tools based on random input data generation are superior in terms of generating long test sequences in shorter time, which is an advantage during activities such as debugging. However, when the test suites are used for conformance testing of the software, at a later stage during the MBD lifecycle, this advantage is not significant.

6.3.3. Coverage and complementarity. The coverage data for the various real-life controller models have been summarized in the bar charts shown in Figure 20. It is observed from the data that AutoMOTGen shows higher coverage compared with Reactis for about one third of the cases, whereas Reactis shows higher coverage for about one other third of the cases. In the rest of the cases, the coverage obtained by both the tools are found to be equal. A careful analysis of this data

![Figure 20](image_url)
by applying different criteria—block sizes, coverage goals, input–output signals and block types—
indicates few trends, namely, when the models have more logic, switches and delay types of blocks,
AutoMOTGen performs better than Reactis, whereas for models with more blocks of mathematical
operations, Reactis performs better compared with AutoMOTGen. It indicates that the exhaustive
nature of BMC, the underlying technique in AutoMOTGen, is better suited for covering paths with
logical constraints. When approximations are applied to handle complex mathematical operators,
the coverage by AutoMOTGen suffers.

6.3.4. Unreachability. Random input-based techniques cannot prove the unreachability of goals.
However, on the basis of the techniques in Section 3.3, AutoMOTGen can prove the unreachabil-
ity of those goals that cannot be covered by the generated tests (cf. Section 4). This is reflected in
the results shown in Table IV. Unreachability cannot be proved for some goals (marked as ‘—’)
because of resource limitations. However, the results still show that in about 50% of the models, all
the undischarged goals are proved to be unreachable through AutoMOTGen. In the absence of this
capability in the random input-based tools, the achievement is quite significant.

6.4. Summary of results

From the study on the benchmark models, it is found that AutoMOTGen and Reactis are at par
in terms of test suite sizes, whereas Reactis has better test generation times. On some models
larger than 2000 blocks, AutoMOTGen failed because of memory constraints, whereas Reactis did
not have any problem in scaling. The results are as expected because of the contrasting under-
lying techniques of the tools and hence are not projected in the tables. An important finding of
the experiments is that the structural coverage achieved by AutoMOTGen and Reactis is com-
plementary in nature. Therefore, use of both the techniques in combination is recommended to
achieve high coverage. The other important result is that model checking-based techniques can
prove the unreachability of a large number of goals in many cases. This is particularly helpful in
the case of safety-critical applications where every improvement in coverage and every discovery of
unreachability are valuable.

In addition to the extensive experiments on comparison of AutoMOTGen with Reactis, a few
experiments are also conducted with a recently introduced tool, the SDV from MathWorks, which is

<table>
<thead>
<tr>
<th>Sl. no.</th>
<th>Model name</th>
<th>Model size</th>
<th>Total no. of goals</th>
<th>Undischarged goals</th>
<th>Unreachable proved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ETC1</td>
<td>37</td>
<td>38</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>PTC1</td>
<td>59</td>
<td>58</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>ESC1</td>
<td>59</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>ESC2</td>
<td>107</td>
<td>73</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>AS1</td>
<td>124</td>
<td>136</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>HVAC1</td>
<td>129</td>
<td>107</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>ETC2</td>
<td>135</td>
<td>104</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>AS2</td>
<td>156</td>
<td>488</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>ETC3</td>
<td>199</td>
<td>90</td>
<td>5</td>
<td>—</td>
</tr>
<tr>
<td>10</td>
<td>HVAC2</td>
<td>227</td>
<td>158</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>11</td>
<td>PTC2</td>
<td>304</td>
<td>328</td>
<td>35</td>
<td>—</td>
</tr>
<tr>
<td>12</td>
<td>HVAC3</td>
<td>341</td>
<td>286</td>
<td>47</td>
<td>—</td>
</tr>
<tr>
<td>13</td>
<td>PTC3</td>
<td>347</td>
<td>243</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>14</td>
<td>ETC4</td>
<td>436</td>
<td>118</td>
<td>62</td>
<td>—</td>
</tr>
<tr>
<td>15</td>
<td>PTC4</td>
<td>462</td>
<td>350</td>
<td>110</td>
<td>—</td>
</tr>
<tr>
<td>16</td>
<td>ETC5</td>
<td>514</td>
<td>141</td>
<td>6</td>
<td>—</td>
</tr>
<tr>
<td>17</td>
<td>PTC5</td>
<td>584</td>
<td>581</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>18</td>
<td>ESC3</td>
<td>632</td>
<td>277</td>
<td>12</td>
<td>—</td>
</tr>
<tr>
<td>19</td>
<td>ESC4</td>
<td>711</td>
<td>282</td>
<td>20</td>
<td>—</td>
</tr>
<tr>
<td>20</td>
<td>ESC5</td>
<td>901</td>
<td>484</td>
<td>35</td>
<td>—</td>
</tr>
</tbody>
</table>
also based on a model-checking approach. The SDV tool is evolving and cannot handle many of the models because of either their large size or the presence of unsupported block types in them. The coverage data obtained from SDV for the same set of models are also presented in Figure 20. It can be noted that for the eight models where SDV can be applied, except for an outlier case (ASC1—an active safety control system), the coverage results show trends similar to that of AutoMOTGen.

7. CONCLUSIONS

Automatic generation of test cases is an important activity during MBD of safety-critical embedded systems. Existing techniques based on random input generation and guided simulation lack the capability to provide high test coverage and proof for unreachability. Therefore, in this paper, a relatively new approach for test generation and unreachability proving, based on the model-checking technique, has been explored. The strengths of these techniques have been studied empirically by comparing the results of their application on a set of real-life industrial controller models.

In the absence of off-the-shelf tools, an in-house model checking-based tool suite called AutoMOTGen has been developed. The tool translates SL/SF models into formal models implemented in SAL, encodes coverage specifications as goals in the model and generates tests via counter-example traces by using the model-checking engine. Model approximations such as sampling time approximations, over-approximations and under-approximations have been implemented in the translation. AutoMOTGen is compared, on a number of scales, with Reactis, a commercial tool that implements a combination of random input-based and guided simulation-based techniques. The comparison of the results leads one to conclude that the techniques are complementary; thus, one can use both the techniques together to obtain better coverage and unreachability guarantees.

One of the main problems faced by model checking-based approaches is their scalability and difficulty in handling non-linear operations. Research in the recent past has shown enormous progress in making constraint solvers, and decision procedures scale to large problem instances observed in practice. Also, there have been a number of attempts at solving large non-linear constraints (cf. [57]). In addition, because pure random input-based techniques are not sufficient, constraint solving-based methods are being implemented in tandem to uncover more goals (or discover unreachability), as is seen in the Reactis tool. Thus, the trend towards integrating different techniques seems to be a positive way of addressing the problem of automatic test case generation for higher coverage and unreachability discovery.

ACKNOWLEDGEMENTS

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

REFERENCES

AUTOMATIC TEST CASE GENERATION FROM SIMULINK/STATEFLOW MODELS


