Systematic Management of Simulation State for Multi-Branch Simulations in Simulink

Zhi Han, Pieter J. Mosterman, Justyna Zander, and Fu Zhang

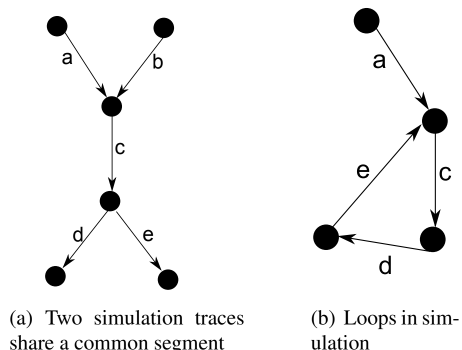
MathWorks, Inc
3 Apple Hill Dr, Natick MA
Zhi.Han|Pieter.Mosterman|Justyna.Zander|Fu.Zhang@mathworks.com

ABSTRACT

Systematic simulation is a technique related and motivated by the formal analysis of hybrid dynamic systems. It combines the exhaustive and conservative nature of traditional model checking with numerical simulation for providing efficient algorithms to manage simulations. Multi-branch simulation is the concept advancing simulation efficiency by reducing the number of state transitions. This paper introduces an approach to implement multi-branch simulation into a popular industrial modeling and simulation tool, Simulink®. The notion of *simulation state* which is distinctly different from the *dynamic system state*, is introduced for Simulink models. From this, a novel semantics based on transition systems is then developed. In a prototype implementation, these semantics are encoded in the current architecture of the Simulink engine and enable demonstrating the benefit of such type of a simulation by three case studies.

1. INTRODUCTION

Systematic simulation is a method that incorporates techniques of formal verification and numerical analysis [9]. Instead of viewing sets of simulation results as traces of state snapshots, it employs a transition system semantics. Thus, the simulation traces are considered as different paths in a possibly infinite state that ultimately results in nondeterministic transition systems. By organizing simulations as a nondeterministic transition system, the efficiency of the simulator can be improved, that is, common segments of simulation traces can be recognized and infinite loops can be detected. Consider the transition system as shown in Fig. 1(a), a trace-based simulator must simulate at least two different traces $a \to c \to d$ and $b \to c \to e$ to try to achieve full simulation coverage. A systematic simulator, on the other hand, recognizes that the transition c is common between two traces and avoids the repeated simulation. Consider the transition system as shown in Fig. 1(b), a trace-based simulator simulates the loop multiple times, while a systematic simulator recognizes that $a \to c \to d \to e$ leads the system to an already reached state, and the only out-going transition from that state has been simulated already, therefore no additional simulation is necessary. A further analysis of numerical proximity for state vectors based on set-based methods can improve the robustness of the simulation. In this case, the sets of states are accounted for in the transition systems [12, 13].



(a) Two simulation traces share a common segment

(b) Loops in simulation

Figure 1. Simulation of transition systems

The efficiency achieved in the simulations comes at the expense of having to manage the state of the transition system. For continuous and hybrid systems the state space is infinite, causing the state explosion problem [5]. One way to tackle this problem is to compute abstractions. For example, states that are close to one another against some metric can be grouped into abstract states, and the abstract transition system is computed instead of the concrete transition system. The abstraction provides a conservative over-approximation of the system behavior [2]. The method of systematic simulation is studied in hybrid systems research and serves as a practical alternative to computational more complex formal verification ([6, 8, 12, 13]). However, the resulting methods are still only satisfactory for specific hybrid system models and so the practical applicability is limited.

Another problem of verification methods is that they usually disregard the numerical simulation algorithms implemented in the simulators by simply defining the transition relations based on the continuous trajectories, that is, solutions of the continuous ODE solutions. In practice, however, these ODEs are solved by numerical methods that exhibit more complex numerical behaviors especially in case of behaviors of a hybrid nature (e.g., [21]). By using systematic simulation the discrete behavior of the simulation engine is part of the analysis, and, therefore, the eventual results. Consequently, practitioners gain increasing confidence that the re-

sults hold for a computational model as it is used throughout the various stages in Model-Based Design (MBD) [17]. This tendency is observed when applied to models that rely on a tool such as Simulink® [15], k, which is a popular industrial tool for design of embedded control systems. In particular, because validation and verification methods are available in the MBD process and can effectively be used. The idea of analyzing the behavior of a simulator together with the model it simulates has been pursued in [18, 19, 21].

In this paper, a general class of Simulink models is considered, and, therefore, the Simulink code base constitutes the core for the implementation. The notion of simulation state is introduced and a method to algorithmically identify the simulation state from a Simulink block diagram model is presented. Based on the simulation state, a transition system semantics is developed for Simulink models. Of particular importance is that the transition system takes into consideration the detection of so-called zero-crossing events in Simulink. The transition system is defined based on the current implementation of the solver and execution engine of Simulink. An architecture overview of a system that implements systematic simulation is then proposed. The system to perform systematic simulations has been implemented in several different ways and based on the collected experience an architectural design has been derived.

Other than areas of formal verification, engineers have studied systematic ways of testing embedded systems (*e.g.*, [4, 11, 20, 22, 23]). These methods study the problem of systematic methodology to design test cases for embedded systems. The main difference of the approach provided in this paper is that here the focus is on the execution parts rather than the design parts, such as the test behavior design, test oracle design, or test input generation. As such, the work in this paper contributes a complement to the existing literature on testing of software for embedded systems. In particular, it is advancing the efficiency of the test execution, bringing the test control of test suites execution to the next level (*cf.*, [**?** 11, 22]). The execution can now be quicker with the same execution coverage and it can be even further automated without the design modification necessity.

This paper is organized as follows. Section 2. introduces the background information. Section 3. then provides a technical overview of the simulation state and introduces the transition system semantics of Simulink models. Section 4. presents an objected-oriented design to implement systematic simulation such that it integrates with the existing Simulink simulation engine architecture. Section 5. then presents three case studies after which Section 6. concludes the paper.

2. BACKGROUND

Simulink is an environment for modeling systems with continuous-time and discrete-time dynamics. A Simulink model is a block diagram consisting of blocks that model dynamic systems. The dynamic systems modeled by Simulink blocks include:

1. Algebraic equations, *e.g.*, $y = g(u)$
2. Continuous-time ordinary differential equations (ODEs), *e.g.*, $\dot{x} = f(x,u)$
3. Discrete-time difference equations, *e.g.*, $x_{t_{k+1}} = f(x_{t_k}, u_{t_k})$ where $\{t_k, k = 1, \cdots\}$ is a set of monotonically increasing time points called *sample hit times*.

The dynamics of a Simulink block involves *input*, *state*, and *output* variables. The dynamic equations are implemented as *methods* of the Simulink blocks including:

`Output` method:	$y := f_O(t,x,u)$
`Update` method:	$x_d := f_U(t,x,u)$
`Derivative` method:	$dx_c := f_D(t,x,u)$
`Zero-crossing` method:	$z := f_Z(t,x,u)$

Methods that implement Simulink blocks furthermore enable the declaration of 'work' variables. When a work variable is declared, it is also specified whether the work variable is reusable. Those work variables that are nonreusable are called *discrete state variables* in this paper. Examples include the state of a state transition diagram that is represented by a Stateflow® [16] Chart block, the internal state of Hysteresis block, and the enable/disable state of Enabled Subsystems [15]. The *continuous state variables* are the variables of which derivatives are defined in the *Derivative* method.

For blocks with discrete state variables, an implicit *function separation* requirement is often necessary for Simulink execution engine to operate properly as illustrated below.

- The block has an `Output` method and an `Update` method.
- The `Update` method can write to the state variables but not the output variables.
- The `Output` method can write to the output variables but not the state variables.

A general dependency template of Simulink execution relations between variables and methods has been presented in previous work [7].

Simulink allows a block instance to specify a *sample time* for its outputs. The sample time defines a set of periodic sample hit times $\{t_0, t_0 + h, t_0 + 2h, \cdots\}$ for the execution of the `Update` and `Output` methods. When the simulation time t is equal to one of the sample hit times of the block, its `Output` and `Update` methods are executed in the current time step. In this case the block is said to have a sample hit at time t.

An implicit assumption made by the Simulink engine is that blocks satisfy a *variable definition* requirement:

- At sample hit times of a block output, the corresponding output variable and state variable must be defined in the block `Output` method.

In this paper, it is assumed that the function separation requirement and variable definition requirement are satisfied by all blocks under consideration. In practice these requirements are sometimes not satisfied in some implementations of blocks. For example, there are a number of blocks that combine the `Output` and `Update` function into a single `Step` function. The violation of the function separation requirement requires special treatment in the implementations of the method, which is implemented in Simulink to preserve data integrity. However, this special treatment in the Simulink engine is out of the scope of this paper.

The simulation of a dynamic system model in Simulink consists of multiple phases, such as, model compilation, linking, and the main simulation loop phase [15]. In the compilation phase, Simulink models are compiled into an operational form. In the linking phase, the resources necessary for simulation are allocated. The simulation loop phase depicted in Fig. 2. is the step where the dynamic equations of the model are solved by computing state and output signal values at each consecutive time step. At each time step, Simulink first invokes the `Output` method to compute the output signals of the blocks for which a hit at the current simulation time occurs. Then Simulink invokes the *ODE solver* to compute the next time step and the state values at the next time step. For zero-crossing functions, Simulink then checks if the signs of the zero-crossing functions have changed for the new state values. If so, Simulink invokes the *zero-crossing detector* to possible reduce the time step to locate the time instances of the zero-crossing events. Once an appropriate time step has been found and the corresponding state vector values are computed, Simulink advances to the next time step. This loop is repeated till the simulation is completed.

The ODE solver and zero-crossing detector of Simulink invoke the `Output` and `Derivative` methods of the model to compute signal values and the state derivative vector, respectively. In variable step solvers, the `Output` method and `Derivative` method may be invoked multiple times in one step. Some results from these method calls may be discarded based on estimations of the approximation error. To distinguish the `Output` calls from the simulation loop and the calls originating from the solver and the zero-crossing detector, the former is referred to as the *major* step and the latter is called the *minor* step [15]. For purely discrete-time models, there are no `Derivative` functions in the model, thus, the minor step is skipped in the simulation loop. The `Zero-Crossing` functions, if any, will be computed in the major step instead.

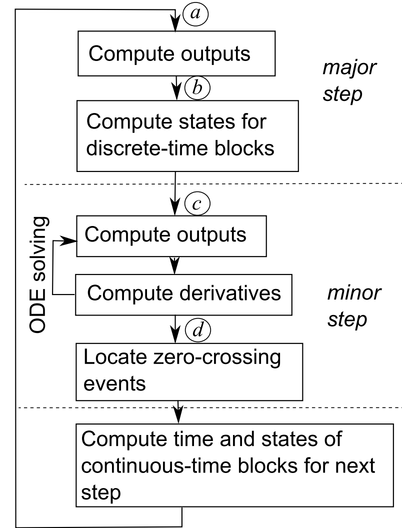


Figure 2. Simulation loop of Simulink®

3. SIMULATION STATE

Before deriving the state of simulation, first the simulation loop of the Simulink engine is modeled. Figure 3 shows a finite-state machine model of the simulation loop where each loop iteration is represented by a single state with self transition. The states of the finite-state machines are called *locations* as they represent the current location of the execution process in the simulation loop. The location of the single-step model is chosen to represent the top of the simulation loop labeled as *a* in Fig. 2.. The execution of the complete loop is modeled by a transition labeled ‘Step’, which represents all the computations performed in the simulation loop.

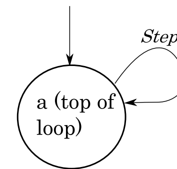


Figure 3. Finite-state model of a single-step simulation loop

The single-step model may be too coarse and it may be necessary to distinguish the various different computations performed in the simulation loop. Figure 4 shows a multi-state model of the simulation loop. The model consists of four locations corresponding to different points in the simulation loop. The locations are marked in Fig. 2. at the corresponding locations in the simulation loop. Note that location *a* corresponds to the same location in the simulation loop as location *a* in Fig. 3.

The variables used by the model include variables declared by the blocks. Following the convention of program analysis (*e.g.*, [1]), a variable is called *live* at a location if it holds a previously computed value that is to be used in later computations. The set of live variables at a location in the simulation loop consists of values that are computed in the loop and will

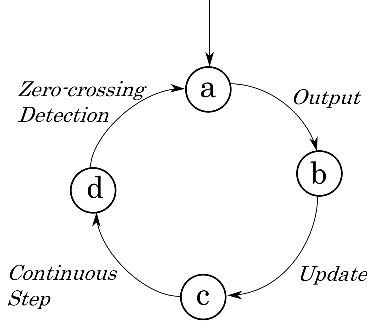


Figure 4. Finite state model of a multi-step simulation loop

be used at later steps. For a Simulink model, define a location variable l where $l \in \{a,b,c,d\}$ for multi-step semantics or $l = a$ for single-step semantics. Let $\mathbf{x}_l \in X_l$ denote the set of live variables of a Simulink model at the program location l. Define the *simulation state* as $\chi = (l, x_l)$.

For sequential programs, the set of live variables can be determined by translating the program to an intermediate representation to perform dataflow analysis on the intermediate representation [1]. For Simulink models, the dataflow analysis is not applicable. Instead, the set of live variables is determined from the information that the Simulink engine computed from the model structure and block execution information [15].

A block output variable is *persistent* if either of the following two conditions are met:

- The source and destination block of the variable have different sample times.
- The `Output` method of the source of the variable is conditionally executed, that is, the block is a child of a conditionally executed subsystem.

Persistent block outputs are live at all locations in the simulation loop since their value may be defined at one time point and used in the computations of another, different, sample time.

A block output variable is *global* if it is not persistent and the following condition is met:

- The variable is used in either the *Update* method, the *Derivative* method, or the *Zero-crossing* method of a different block.

Global output variables are live at location b, c, and d.

At location d, the ODE solver has computed the continuous state values of the next time step assuming there is no zero-crossing event, which is used in the zero-crossing detection algorithms. The computation is performed using the discontinuity locking or lie-in-minor-time-step mechanism [24]. Therefore, the simulation state at d must include these internal variables used by the solver, denoted by x_c', which stores the continuous state values for the next step as if there were no zero-crossing events.

Table 1 lists the simulation state (l, x^l) of a Simulink model at different locations for the multi-step semantics. For single-step semantics, the simulation state is the one corresponding to location a.

Location	Live variables at the location x^l
a	continuous state, discrete state and persistent outputs
b	continuous state, discrete state, persistent and global outputs
c	continuous state, discrete state and non-reusable outputs
d	continuous state, discrete state, non-reusable outputs and continuous state values of next step (x_c')

Table 1. Simulation state for each location

Given a Simulink model with initial values x_0^a for the live variables at a. The semantics of a Simulink model is described as the corresponding transition system created for single-step or multi-step simulation loops. For a set of variables x, use $\mathcal{V}_x$ to denote the set of possible values for x. The transition systems corresponding to a Simulink model are defined in terms of the simulation states in the following ways:

- The single-step transition system is $T_S = (S_S, \rightarrow_S, \chi_{S0})$ consisting of a set of simulation states $S_S = (a, \mathcal{V}_{x^a})$ and a set of transitions $(a, x_1^a) \rightarrow_S (a, x_2^a)$ iff x_2^a are the values of the live variables after performing the computations of a simulation step. The initial state is $\chi_{S0} = (a, x_0)$.
- The multi-step transition system is $T_M = (S_M, \rightarrow_M, \chi_{M0})$ consisting of a set of simulation states $S_M = (a, \mathcal{V}_{x^a}) \cup (b, \mathcal{V}_{x^b}) \cup (c, \mathcal{V}_{x^c}) \cup (d, \mathcal{V}_{x^d})$ and a set of transitions $(l_1, x_1^{l_1}) \rightarrow_M (l_2, x_2^{l_2})$ where $l_1 \rightarrow l_2$ is a transition of the finite state machine shown in Fig. 4 and $x_2^{l_2}$ are the values of the live variables after performing the corresponding computations in $l_1 \rightarrow l_2$. The initial state is $\chi_{M0} = (a, x_0)$.

The transition system semantics is similar to the stream-based formulation of previous work [19] with a difference in that the minor steps of the ODE solver is not considered in this paper.

4. SYSTEMATIC MULTI-BRANCH SIMULATION

In this section, an object-oriented (OO) approach of implementing systematic multi-branch simulation (SMBS) in the Simulink environment is presented. OO approach is preferred to other ways of describing the algorithm because the algorithm for SMBS is simplified to emphasize the focus on how to design the architecture so that the new functionality fits into the existing software architecture of the Simulink simulation engine.

Figure 5 illustrates an architecture design of implementing SMBS for Simulink models. The main object the SMBS is implemented in is the `SimulationManager` class, which is responsible for creating and storing the resulted transition system and managing simulation states. The behavioral analysis determines whether the simulation must continue and computes the input to the system if necessary. Additional analysis such as on-the-fly test case generation can be employed by the behavioral analysis to generate inputs to the model, as simulations of Simulink models are performed for systems with either no input or a given set of input signals. The `SimulationManager` object owns an instance of the `SimulationStepper` object, which has a *Simulator* object attached to it. The `SimulationStepper` object is implemented to manage the locations of the `Simulator`, while the `Simulator` object is a thin wrapper over the existing simulation algorithm of Simulink. The `SimulationStepper` is responsible for starting and stopping the simulation. It also manages a list of simulation states when the simulation is running. The `Simulator` object has an associated Simulink model and the allocated resources to simulate the model.

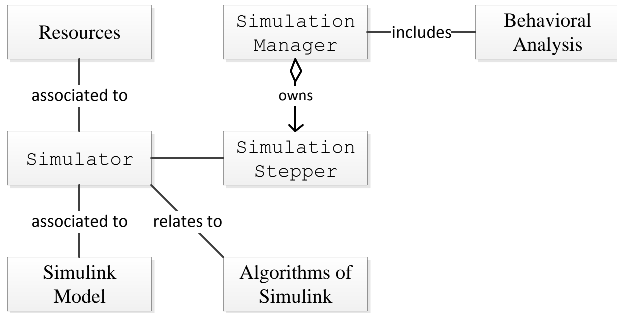


Figure 5. Instance graph of objects implementing the systematic simulation method

A straightforward implementation of the transition system would store all the reachable simulation states for all computation steps. For large-scale Simulink models, the space required for storing these data can be prohibitively large. In addition, storing a simulation state requires copying all the live variables from the model runtime environment to a permanent memory location, which can potentially slow down the simulation when the model is large. To improve memory and computation time efficiency, simple paths are compressed: if a set of transitions $\chi_1 \to \chi_2 \to \cdots \to \chi_m$ forms a path and has no branches, the transition system only contains a transition $\chi_1 \to \chi_m$ together with a log of a set of signals of interests during the simulation. The set of logged signals are specified by the Simulink user.

Figure 6 shows an implementation of the main loop of the `SimulationManager` that builds the transition system and traverses the transition system in breadth-first order. The algorithm breaks out of the loop when the analysis has reached a time limit or the analysis has exhausted all reachable states.

```
initialize_graph;
initialize_state;
while ~isempty(queue)
  [init_state, in] = dequeue;

  % Invoke SimStepper to simulate the model
  final_state = simulate(init_state, in);
  update_transition_system;

  decide_whether_to_stop;
  if (not_stop)
    % invoke behavioral analysis
    in = analyze_result (final_state);
    if (final_state_is_new)
      % inputs may be multiple
      % for non-deterministic simulation
      enqueue (final_state, in);
    end
  end
end
```

Figure 6. Main loop of the simulation manager

In order to build the transition system, it is necessary to compare one simulation state with another. A variable in Simulink can be used to store either numeric values stored in either floating-point or fixed-point data types, or non-numeric values, such as an enumerated data for the state of a Stateflow chart. The live variables of a model consist of a set of non-numeric-valued variables $\mathbf{x}_e$ and a set of numeric-valued variables $\mathbf{x}_n$, *i.e.*, $\mathbf{x} = (\mathbf{x}_e, \mathbf{x}_n)$. For each numeric-valued variable $v \in \mathbf{x}_n$ of the simulation state, define a non-negative value δ_v as a *tolerance* of the corresponding state variable v.

Define a proximate relation between simulation state $\chi_1 = (l_1, \mathbf{x}_1)$ and $\chi_2 = (l_2, \mathbf{x}_2)$ as $\chi_1 \approx \chi_2$ iff all of the following conditions are met:

- $l_1 = l_2$
- $v_1 = v_2$ for all non-numeric state variables $v \in \mathbf{x}_e$
- $|v_1 - v_2| \le \delta_v$ for all numeric variable of the simulation state $v \in \mathbf{x}_n$
- if $l = d$, $sign(g_i(\mathbf{x}_{1c})) = sign(g_i(\mathbf{x}_{2c}))$ and $sign(g_i(\mathbf{x}'_{1c})) = sign(g_i(\mathbf{x}'_{2c}))$ for all zero-crossing functions

The last condition is defined to ensure that the zero-crossing detection algorithm of Simulink behaves the same for the two simulation states [24]. The choice of δ_v can be a tricky problem. If $\delta_v = 0$, any two different simulation states are not proximate of each other, thus simulation needs to be performed for both, which then leads to many simulations. If $\delta_v = \infty$, all simulation states with the same location are proximate of one another, which is likely to cause the algorithm to ignore states that might lead to different behaviors.

The `SimulationStepper` simulates the model for a given initial simulation state χ_i, once simulation state χ_y is returned from the `SimulationStepper`, it is compared

to the existing simulation states, if the numerical part is close and the non-numerical part is identical to an existing simulation state χ_x, it is merged with χ_x by adding transitions $\chi_i \to \chi_x$. Otherwise, the simulation state χ_y is added to the transition system and the transition $\chi_i \to \chi_y$ is added.

The SMBS method works with both the single-step model (Fig. 3) and the multi-step model (Fig. 4). An implementation of `SimulationStepper` based on the single-step model is available in the shipping Simulink software [15].

5. CASE STUDIES

This section demonstrates the feasibility of SMBS using a prototype implementation of SMBS in Simulink. The results in this paper are produced using an implementation of the `SimulationStepper` for single-step semantics.

5.1. Galton Board

A Galton board is a device used to conduct statistical experiment on Binomial distributions [3]. It consists of an upright board with evenly spaced nails (or pegs) driven into its upper half, where the nails are arranged in staggered order, and a lower half divided into a number of evenly-spaced rectangular slots. In the middle of the upper edge, there is a funnel into which balls can be inserted, where the diameter of the balls must be smaller than the distance between the nails. The funnel is located precisely above the central nail of the second row so that each ball, if perfectly centered, would fall vertically and directly onto this nail.

To model the behavior of Galton board, two state variables are introduced for the ball, the horizontal position and the vertical position. The magnitude of the speed is assumed to be constant during the drop. The vertical speed is assumed to be constant and the horizontal speed can only change in direction. The behavior of the system can be simplified as follows. At each grid point (*i.e.*, when the vertical position reaches a grid point) the ball may change its horizontal direction. The model can be simplified by avoiding modeling the grid point explicitly. Every time that the vertical position is at a grid point, $\dot{x}$ can change sign. The tolerance of state for proximity is chosen as $\delta_x = \delta_y = 1 \times 10^{-3}$.

Figure 7 shows the phase plane plot of the simulation results with three intermediate levels. The simulation states at the grid points are stored for the transition system. SMBS avoids repeating simulations by merging simulation states at the grid points. As the number of levels increases, the number of all possible traces are 2^n, each trace would involve simulating the system for $n+1$ seconds. However, the total number of simulation segments is $1 + 2\sum_{i=1}^{n} i = n^2 + n + 1$. and each segment requires simulating the system for 1 second.

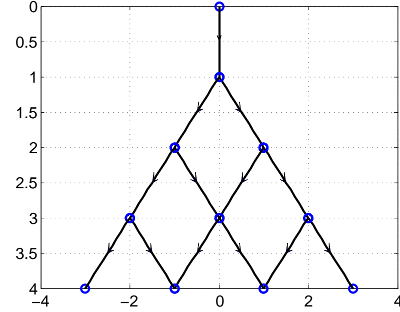


Figure 7. SMBS results of a Galton board

5.2. Event Tree Simulation

Where the Galton board is a contrived example, this section considers application of SMBS in event tree analysis which is applied in, for example, *Failure Mode and Effect Analysis* (FMEA) [10]. The focus is to analyze the behavior of the system when subject to a series of faulty events. Consider, for example, the following series of simulation scenarios.

1. Start-up: A system starts up and runs until it reaches a steady state.
2. Fault 1: At 100s, the system may be subject to a fault indicated by a step up in its input signal.
3. Fault 2: At 100s, the system may be subject to a fault indicated by a step down in its input signal.
4. Keep running: Because the controller reacts to the signal (presumably 2 seconds after the fault), it may decide to keep the system running
5. Shut down: the controller may decide to shut down the system by switching the input signal to 0.
6. Second fault: when fault 2 occurs, it is possible that a second fault occurs whereas when fault 1 occurs, a second fault cannot occur.

Figure 8 shows the event tree of the faulty scenarios for the analysis.

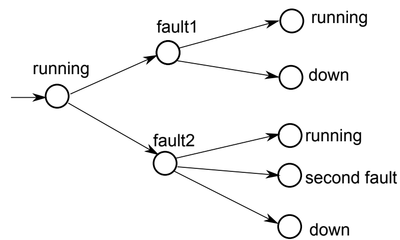


Figure 8. An event tree for fault analysis

In this case study the Signal Builder block of Simulink is employed to introduce the input signals under faulty scenarios. The complete set of input signals for each scenario is partitioned into signal segments. Figure 9 shows the signals

created in the Signal Builder block. Each signal tab specifies a segment of the entire input sequence.

SMBS was performed by modifying the existing MATLAB® [14] implementation of the Signal Builder block. The tolerance of state for proximity is $\delta = 0$.

Applying SMBS, the common transition segments of the input signals are grouped and factored in the `SimulationManager`. After each individual segment is simulated and the graph of the transition system is created, it is possible to traverse the graph to generate all the simulation traces for the scenarios in the analysis. Figure 10 shows the results of all the simulation scenarios for the given event tree. The common segments of simulation traces are simulated only once, thereby improving the efficiency of the simulations.

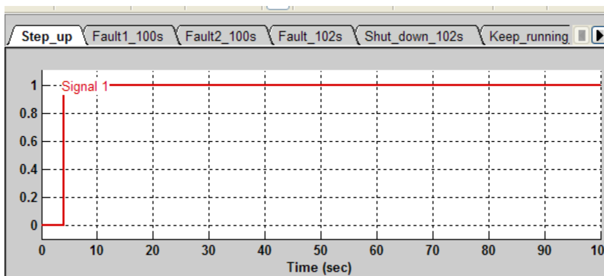


Figure 9. A Signal Builder block to generate the segments of the input signals

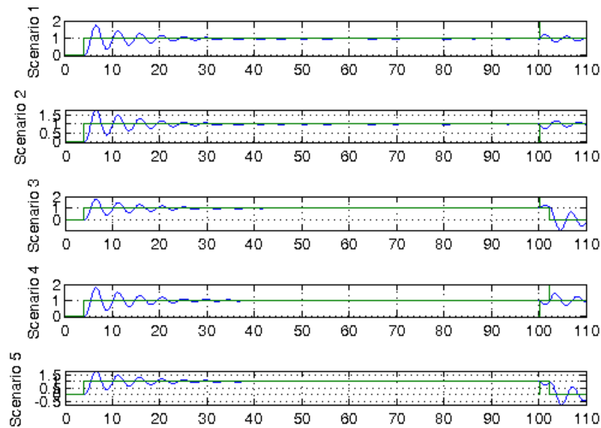


Figure 10. Simulated faulty scenarios for all cases

5.3. Simulating a Transmission Controller

Consider an automatic transmission control system implemented as the Simulink `autotrans` model (Fig. 11), where the input is the command from the driver.

Traditionally engineers simulate the behavior of step-up, step-down, and particularly designed test cases as separate test cases. A time-partition testing method is used [11]: a finite state machine is used to generate user inputs, where each mode of the machine generates an input signal for a certain period of time. The test input generation is implemented in a Stateflow block with logic over time as shown in Fig. 12. The Stateflow model used in this case study implements a nondeterministic state machine.

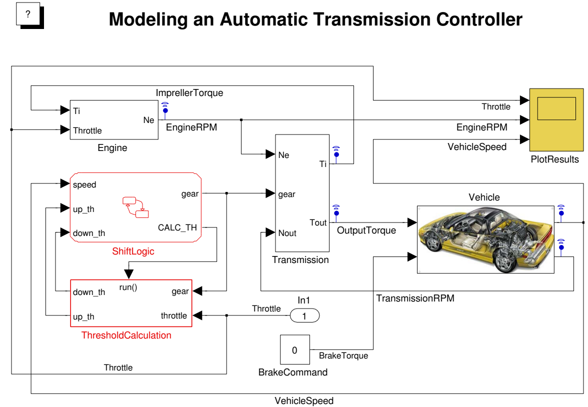


Figure 11. Automatic transmission system model

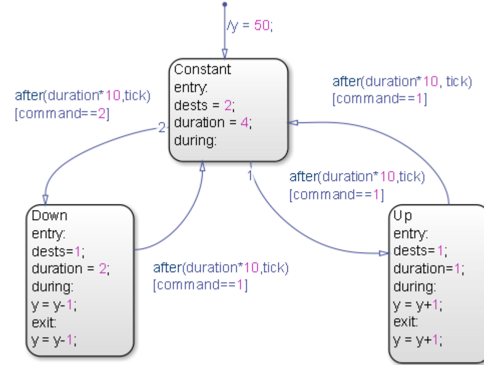


Figure 12. Stateflow® model of the user input to the system

Figure 13 shows the phase plane plot of SMBS for the automatic transmission model. Figure 14 shows the time plots of the SMBS results for the automatic transmission system. The simulation states were stored in the transition system for each nondeterministic transition in the Stateflow model. Consequently, SMBS involves no repetitions while covering all Stateflow transitions for this example.

6. DISCUSSION

In this paper, an SMBS method has been presented. A definition of simulation state was provided and a transition system semantics for general Simulink models has been defined. The design of an architecture for SMBS in Simulink was developed based on the code basis. The case studies served as a validation of the prototypical implementation.

There are a few areas where the presented approach can be enhanced. For example, one of the potential enhancements of the method is to employ *coarse-gained* parallelization to further accelerate the simulation. As the simulation

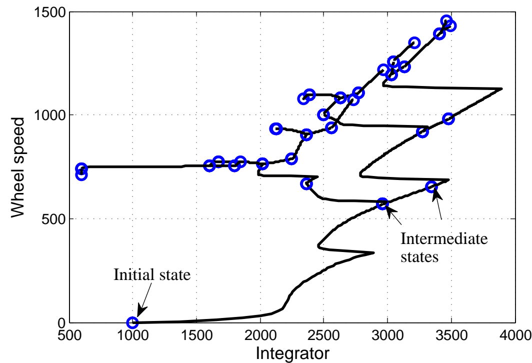


Figure 13. Phase plane plot of SMBS results for the automatic transmission system

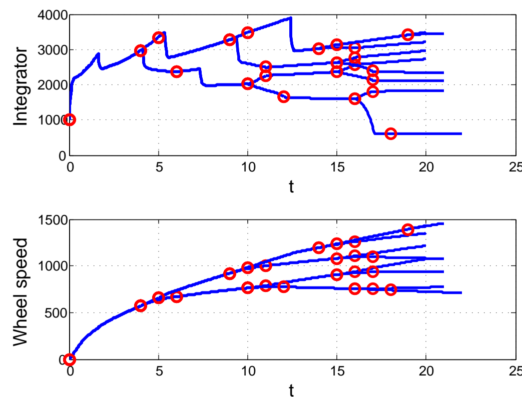


Figure 14. Time plots of systematic multi-branch simulation results for the automatic transmission system

of a model is executed by the `SimulationStepper`, one can design a system that creates multiple instances of `SimulationStepper` objects and thereby perform multiple simulations of the model simultaneously in parallel execution environments. The applications of the mutli-step model also needs to be explored further.

The focus of this paper was on the simulator and the simulation state. The design of other parts of the system (*e.g.*, the test input, test behavior) was not discussed in detail. The design of the test cases, however, requires substantial effort. The design methodologies proposed in other work [11, 20, 22, 23] can be applied to facilitate the entire test engineering process. In addition, the logic in the `SimulationManager` can also be improved by employing advanced model-based analysis techniques such as on-the-fly model checking [5, 13].

Acknowledgments The authors thank Dr. Murali Yeddanapudi and Dr. Rajesh Pavan Sunkari for their help and support on this work.

REFERENCES

[1] Andrew W. Appel. *Modern Compiler Implementation in JAVA*. Cambridge University Press, 1998.

[2] Eugene Asarin, Thao Dang, Goran Frehse, Antoine Girard, Colas Le Guernic, and Oded Maler. Recent progress in continuous and hybrid reachability analysis. In *Proceedings of IEEE Conference on Computer Aided Control System Design (CACSD)*, pages 1582–1587, 2006.

[3] Margherita Barile and Eric W. Weisstein. Galton board. From MathWorld—A Wolfram Web Resource. Last visited on 13/4/2012.

[4] Eckard Bringmann and Andreas Krämer. Systematic testing of the continuous behavior of automotive systems. In *Proceedings of the 2006 international workshop on Software engineering for automotive systems*, SEAS '06, pages 13–20, New York, NY, USA, 2006. ACM.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[6] Thao Dang, Alex Donzé, Oded Maler, and Noa Shalev. Sensitive state-space exploration. In *CDC*, pages 4049 – 4054, 2008.

[7] Ben Denckla and Pieter J. Mosterman. An intermediate representation and its application to the analysis of block diagram execution. In *Proceedings of the 2004 Summer Computer Simulation Conference (SCSC'04)*, pages 167–172, San Jose, CA, July 2004.

[8] Alexandre Donzé and Oded Maler. Systematic simulation using sensitivity analysis. In Alberto Bemporad, Antonio Bicchi, and Giorgio C. Buttazzo, editors, *HSCC*, volume 4416 of *Lecture Notes in Computer Science*, pages 174–189. Springer, 2007.

[9] Juergen Grossmann, Ina Schieferdecker, and Hans Werner Wiesbrock. Modeling property based stream templates with TTCN-3. In *Proceedings of the IFIP 20th International Conference on Testing Communicating Systems (TestCom 2008)*, pages 70–85, Tokyo, Japan, June 2008.

[10] James Kapinski, Bruce H. Krogh, Oded Maler, and Olaf Stursberg. On systematic simulation of open continuous systems. In Oded Maler and Amir Pnueli, editors, *HSCC*, volume 2623 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2003.

[11] J. D. Lazor. Failure Mode and Effects Analysis (FMEA) and Fault Tree Analysis (FTA) (Success Tree Analysis (STA)). In William Grant Ireson, Clyde F Coombs, and Richard Y Moss, editors, *Handbook of reliability engineering and management*, pages 6.1–6.46. McGraw Hill, 1996.

[12] Eckard Lehmann. Time parition testing: a method for testing dynamic function behavior. In *Proceedings of TEST2000*, 2000.

[13] Flavio Lerda, James P. Kapinski, Edmund M. Clarke, and Bruce H. Krogh. Verification of supervisory control software using state proximity and merging. In *Proc. of the 11th International Workshop on Hybrid Systems: Computation and Control (HSCC)*, pages 344–357, 2008.

[14] Flavio Lerda, James P. Kapinski, Hitashyam Maka, Edmund M. Clarke, and Bruce H. Krogh. Model checking in-the-loop. In *Proc. of the 27th American Control Conference (ACC)*, 2008.

[15] MathWorks. *MATLAB User's Guide*. The MathWorks, Inc., Natick, MA, September 2012.

[16] MathWorks. *Simulink User's Guide*. The MathWorks, Inc., Natick, MA, September 2012.

[17] MathWorks. *Stateflow User's Guide*. The MathWorks, Inc., Natick, MA, September 2012.

[18] Pieter J. Mosterman, Sameer Prabhu, and Tom Erkkinen. An industrial embedded control system design process. In *Proceedings of The Inaugural CDEN Design Conference (CDEN'04)*, Montreal, Canada, July 2004. CD-ROM: 02B6.

[19] Pieter J. Mosterman and Justyna Zander. Advancing model-based design by modeling approximations of computational semantics. In *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 3–7, Zürich, Switzerland, September 2011. keynote paper.

[20] Pieter J. Mosterman, Justyna Zander, Grégoire Hamon, and Ben Denkla. A computational model of time for stiff hybrid systems applied to control synthesis. *Control Engineering Practice*, 20(1):2–13, 2012.

[21] Wei-Tek Tsai, Lian Yu, Feng Zhu, and Ray Paul. Rapid embedded system testing using verification patterns. *IEEE Softw.*, 22(4):68–75, July 2005.

[22] Justyna Zander, Pieter Mosterman, Grégoire Hamon, and Ben Denckla. On the structure of time in computational semantics of a variable-step solver for hybrid behavior analysis. In *Proceedings of the 17th IFAC World Congress*, pages 9419–9424, 2011.

[23] Justyna Zander-Nowicka. *Model Based Testing of Embedded Systems in the Automotive Domain*. PhD dissertation, Technical University Berlin, Computer Science and Electrical Engineering Department, March 2009.

[24] Justyna Zander-Nowicka, Abel Marrero Pérez, Ina Schieferdecker, and Zhen Du Dai. Test design patterns for embedded systems. In *10th International Conference on Quality Engineering in Software Technology*, pages 183–200, 2007.

[25] Fu Zhang, Murali Yeddanapudi, and Pieter J. Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. In *Proceedings of the 17th IFAC World Congress*, pages 7967–7972, 2008.