Model-Based Automatic Test Generation for Simulink/Stateflow using Extended Finite Automaton

Meng Li, Member IEEE, and Ratnesh Kumar, Fellow IEEE
Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA 50010, USA

Abstract—Simulink/Stateflow is a popular commercial model-based development tool for many industrial domains. For safety and security concerns, verification and testing must be performed on the Simulink/Stateflow designs and the generated code. In this paper, we present an automatic test generation approach for Simulink/Stateflow based on its translation to Input/Output Extended Finite Automata (I/O-EFA) that we have developed in our prior works. The test generation problem requires identifying the executable paths of the I/O-EFA model and also generating a test input for those paths. Note in order to execute a path, a certain sequence of other paths must be executed first, which we automatically identify. The approach is implemented by applying two different techniques, model checking and constraint solving. Both test generation implementations are validated by a case study. The results show that both implementations can generate test cases as expected and the implementation based on constraint solving is faster.

I. INTRODUCTION

Simulink/Stateflow [1] is a model-based development tool, which is widely used in many industrial domains, such as power systems, nuclear plants, aircraft, and automotives. Code generators are used within the Simulink/Stateflow to automatically generate the embedded software for the target system from the Simulink/Stateflow diagram, and thereby considerably increasing the productivity.

The existing code generators cannot guarantee that the generated code complies correctly with the functional behaviors specified in the design. Verification and testing of the generated code is necessary to find errors in the code generation process and thus avoid software faults in any future use of the system.

Model-based test generation is an essential step in the model-based development process. It aims to validate that the object code to be implemented in the target processor complies with the design requirements. For Simulink/Stateflow, model-based test generation intends to validate whether the generated code (for example ANSI C) preserves the functional behaviors of the Simulink/Stateflow diagram.

Several type of errors may occur in the implementation process from the Simulink/Stateflow diagram to the target code, such as:

• Errors in the Simulink/Stateflow diagram models will get carried over.
• Errors in the automatic code generator for the Simulink/Stateflow diagram caused for example by finite precision arithmetic or timing constraints.
• Any human errors in the selection of code generation options, library naming/inclusion, and others.

A model-based approach to reveal these errors is to create a set of test cases from Simulink/Stateflow and then execute them on the generated code to see if the test passes. Any failed test cases can be used to find the errors introduced during the code generation process.

Since Simulink/Stateflow has originally been designed for the simulation purposes, automated test generation for Simulink/Stateflow diagram is greatly needed to identify the errors. Several authors have tried different ways of test generation and verification for Simulink/Stateflow diagram. Scaife et al. [2] are able to translate a subset of Simulink/Stateflow into Lustre and verify the model using a model checking tool called Lesar. Gadkari et al. [3] have translated Simulink/Stateflow to a formal language, called Symbolic Analysis Laboratory (SAL), and they generate test cases based on SAL model checking. Reactis [4] and T-VEC [5] are two popular commercial tools for automated test generation for Simulink/Stateflow models. In our case, we derive the test suite based on the translation from Simulink/Stateflow to an automaton, which preserves the discrete behaviors (behaviors observed at discrete time steps when the inputs are sampled and the outputs are computed).

In our previous works [6] [7] [8], we introduced a recursive method to translate a Simulink/Stateflow diagram to an Input/Output Extended Finite Automata (I/O-EFA), which is a formal model of reactive untimed infinite state system, amenable to formal analysis. It captures each computation cycle of Simulink/Stateflow in form of an automata extended with data-variables to capture internal states and also the input and output variables. This paper discusses the method to generate test cases for the Simulink/Stateflow diagram based on the corresponding I/O-EFA derived using the approach of [6] [7] [8]. To provide coverage for all computation flows of a Simulink/Stateflow diagram which corresponds to the execution paths in the translated I/O-EFA model, each execution path is analyzed for feasibility and reachability, and test cases are generated accordingly. The test generation approach
is implemented by using two techniques, model-checking and constraint solving using mathematical optimization. The model-checking based implementation abstracts the I/O-EFA and checks each execution path for eventual reachability (note in order to execute a path some other sequence of paths may have to be executed in earlier cycles and hence the requirement of eventual reachability); while the constraint solving based implementation recursively evaluates the longer and longer path-sequences and the associated predicate for reachability. The test cases are generated from the counterexamples (resp. path-sequence predicates) for the case of model-checking (resp. constraint solving) process.

We have integrated the translation tool along with both the test generation implementations into an automated test generation tool, written in Matlab script. A simple example of a counter has been used as the case study to validate and compare the test generation implementations. The test generation results show that both of the implementation methods can generate the expected test cases while the constraint solving based approach is in general faster. The contributions of the paper are:

- We have developed a systematic test generation method for I/O-EFA models that representing the computations of a Simulink/Stateflow diagram.
- Two implementation techniques, model-checking and constraint solving, are implemented and compared.
- We have developed an automated test generation tool based on the above two techniques within the Matlab environment.

II. INTRODUCTION TO I/O-EFA [8]

An I/O-EFA is a symbolic description of a reactive untimed infinite state system in form of an automaton, extended with discrete variables of inputs, outputs and data.

**Definition 1:** An I/O-EFA is a tuple \( P = (L, D, U, Y, \Sigma, \Delta, L_0, D_0, L_m, E) \), where

- \( L \) is the set of locations (symbolic-states),
- \( D = D_1 \times \cdots \times D_n \) is the set of data (numeric-states),
- \( U = U_1 \times \cdots \times U_m \) is the set of numeric inputs,
- \( Y = Y_1 \times \cdots \times Y_p \) is the set of numeric outputs,
- \( \Sigma \) is the set of symbolic-inputs,
- \( \Delta \) is the set of symbolic-outputs,
- \( L_0 \subseteq L \) is the set of initial locations,
- \( D_0 \subseteq D \) is the set of initial-data values,
- \( L_m \subseteq L \) is the set of final locations,
- \( E \) is the set of edges, and each \( e \in E \) is a 7-tuple, \( e = (o_e, t_e, \sigma_e, \delta_e, G_e, f_e, h_e) \), where
  - \( o_e \in L \) is the origin location,
  - \( t_e \in L \) is the terminal location,
  - \( \sigma_e \in \Sigma \cup \{ \varepsilon \} \) is the symbolic-input,
  - \( \delta_e \in \Delta \cup \{ \varepsilon \} \) is the symbolic-output,
  - \( G_e \subseteq D \times U \) is the enabling guard (a predicate),
  - \( f_e: D \times U \to D \) is the data-update function, and
  - \( h_e: D \times U \to Y \) is the output-assignment function.

I/O-EFA \( P \) starts from an initial location \( l_0 \in L_0 \) with initial data \( d_0 \in D_0 \). When at a state \((l, d)\), a transition \( e \in E \) with \( o_e = l \) is enabled, if the input \( \sigma_e \) arrives, and the data \( d \) and input \( u \) are such that the guard \( G_e(d, u) \) holds. \( P \) transitions from location \( o_e \) to location \( t_e \) through the execution of the enabled transition \( e \) and at the same time the data value is updated to \( f_e(d, u) \), whereas the output variable is assigned the value \( h_e(d, u) \) and a discrete output \( \delta_e \) is emitted. For notational convenience, if \( d \in D \) and \( u \in U \) is such that for an edge \( e \in E \), \( G_e(d, u) \) hold, then letting \( d' := f_e(d, u) \) and \( y := h_e(d, u) \), we write \((o_e, d) \xrightarrow{\sigma_e, u, \delta_e, y} (l_e, d')\) to denote the state transition from \((o_e, d)\) to \((l_e, d')\) under the input \((\sigma_e, u)\) and producing the output \((\delta_e, y)\).

III. TEST GENERATION BASED ON I/O-EFA

Our previous works [6] [7] [8] presented a translation approach from a Simulink/Stateflow diagram to an I/O-EFA model while preserving the discrete behaviors observed at sample times. In such an I/O-EFA model, each transition sequence from the initial location \( l_0 \) back to the initial location \( l_0 \) through the time advancement edge \( e \in (l_0, l_0, -1, -1, -1, \{ k := k + 1 \}) \) represents a computation sequence of the Simulink/Stateflow diagram at a sampling time. Note for the time advancement edge \( e \), it holds that \( h_e \equiv \{ k := k + 1 \} \) that advances the discrete time counter by a single step. Such a transition sequence is called a computation path as defined next.

**Definition 2:** A computation path (or simply a c-path) \( \pi \) in an I/O-EFA \( P = (L, D, U, Y, \Sigma, \Delta, L_0, D_0, L_m, E) \) is a finite sequence of edges \( \pi \in \{ e_1^{\ast} \cdots e_{|\pi|}^{\ast} \in E \ast | o_{e_1}, t_{e_{|\pi|-1}} \in L_0, h_{e_{|\pi|-1}} \equiv \{ k := k + 1 \}, \forall i \in [1, |\pi| - 1]: o_{e_{i+1}} = t_{e_{i-1}} \} \).

Our test generation approach is to find a set of input sequences, also called test cases, which execute a certain set of computation sequences specified by a desired coverage criterion. For this, first the paths, representing those computation sequences, are located in the I/O-EFA model, and next the input sequences which activate those paths are obtained. Our previous Simulink/Stateflow to I/O-EFA translation approach formalizes and automates this mapping from computation sequences of Simulink/Stateflow diagram to the c-paths in the translated I/O-EFA model.

**Example 1:** Consider the Simulink diagram \( \Psi \) of a bounded counter shown in Figure 1, consisting of an enabled subsystem block and a saturation block. The output \( y_5 \) increases by 1 at each sample-period when the control input \( u \) is positive, and \( y_5 \) resets to its initial value when the control input \( u \) is not positive. The saturation block limits the value of \( y_5 \) in the range between \(-0.5\) and 7. The translated I/O-EFA \( P \) using the method of [6] is shown in Figure 2. Each c-path in \( P \) represents a possible computation of the counter at a sampling instant. For example, the path \( \pi_3 = e_2 e_8 e_{10} e_{12} e_{13} e_{19} e_{20} e_{21} \) in I/O-EFA \( P \) represents the “reset” behavior, which is the computation sequence of the Simulink diagram \( \Psi \) in which the input is zero so that the subsystem is disabled and its output remains as the initial value and hence the saturation is not triggered in the saturation block. There are totally 18 c-paths in the I/O-EFA \( P \), representing all 18 computation sequences in the Simulink diagram \( \Psi \).
Some of the computation sequences involving certain sequence of Simulink/Stateflow computations may not be possible. This property is made transparent in our I/O-EFA by showing conflict among the conditions along the corresponding $c$-paths. In Example 1, five out of 18 computation sequences are possible and the corresponding five $c$-paths in I/O-EFA are valid. As an example consider an invalid computation sequence “subsystem disabled” and “saturation reaches upper limit”. Since the disabled subsystem generates an initial output 2, which is within the saturator’s limit, the saturator cannot reach its upper limit. This conflict also shows up in the corresponding $c$-path $\pi_5 = e_2 e_8 e_9 e_12 e_13 e_19 e_20 e_21$ over the edges $e_2$ and $e_9$, where $y_5(k) := 2$ on edge $e_2$, whereas $y_5(k) > 7$ on edge $e_9$.

Besides the conflict among the conditions along the edges of a path, some of the impossibilities of certain computation sequences are caused by the initial condition of the system. Consider the saturation condition $y_5(k) < -0.5$ in Example 1. None of the computation sequences with this saturation condition can be executed, since the counter output starts from zero and increments by one each time it counts, and thus the count can never be less than zero. The I/O-EFA model also captures these impossible computation sequences by showing the corresponding $c$-paths as unreachable from the initial conditions.

Based on the above discussion, the test generation problem for Simulink/Stateflow can be converted to finding the input sequences that execute the corresponding $c$-paths in the I/O-EFA. We obtain the feasible and reachable paths and choose a subset of these paths satisfying a desired coverage criterion.

In summary, our I/O-EFA based test generation for Simulink/Stateflow has the following steps.

- Translate the Simulink/Stateflow diagram to I/O-EFA.
- Find all the paths in I/O-EFA.
- Analyze the paths in I/O-EFA for feasibility and reachability.
- Invalid paths are reported for model soundness and robustness analysis.
- Valid paths satisfying the coverage criterion are used to generate a set of test cases for activating them.

The translation method is implemented in our previous work. The remaining challenges to implement this test generation approach are listed as follows.

- How to identify the valid paths. The feasibility of these paths relies on not only itself but also the initial condition and other paths that may be executed as prefixes.
- How to obtain the input sequences activating the valid paths. Some of the valid paths cannot be activated at the very first time step. These paths require some prefix before they can be activated.

In the next section, we discuss the implementations of our I/O-EFA based test generation approach to deal with these challenges.

IV. IMPLEMENTATION OF TEST GENERATION APPROACH

The proposed test generation approach for Simulink/Stateflow has been implemented by applying two different methods. Our previous translation tool SS2EFA has been integrated with these two implementations to support the translation from Simulink/Stateflow diagram to I/O-EFA. The following discussion focuses on the part of test generation to be executed following the translation step.
A. Implementation using Model-Checking

Model-checking is a method to check automatically whether a model of a system meets a given specification. NuSMV [9] is an open source symbolic model checker, which supports the CTL and LTL expressed specification analysis and provides interface to Matlab, so that the test generation tool (written in Matlab script) can call NuSMV for model-checking.

In this implementation, paths in I/O-EFA are checked against the abstracted I/O-EFA model in NuSMV for feasibility and reachability. Since NuSMV only allows for the representation of finite state systems, the translated I/O-EFA is first converted into a finitely abstracted transition system as defined in Definition 3 below.

The finite abstraction of the model is based on the implementation requirements. Most of the real world systems have finite data space and running time. The finite abstraction is implemented in NuSMV input language as described below.

- Variable “steplimit” is set to a value to limit the number of time steps the system can evolve, i.e. to upper bound the finite data space and running time. The finite abstraction is as defined in Definition 3 below.
- The system evolves exceeding the value “deadend” and has no further outgoing transitions.
- Variable “precision” is the limit for the number of significant digits. Since NuSMV can only verify integer values, “precision” determines how the non-integer value in the I/O-EFA model can be transformed into integer. Each non-integer value is transformed as follows: \( \text{value}_{\text{new}} = \text{round}(\text{value}_{\text{old}} \times 10^\text{precision}) \), where \( \text{value}_{\text{old}} \) is the value in the I/O-EFA model, and \( \text{value}_{\text{new}} \) is the value in NuSMV file.

- Each variable \( d_j \) is converted to an integer with upper limit \( d_{\text{max}}^j \times 10^\text{precision} \) and lower limit \( d_{\text{min}}^j \times 10^\text{precision} \), so that data space is finite. \( d_{\text{max}} \) and \( d_{\text{min}} \) are determined by the requirements on the system.

**Definition 3:** Given an I/O-EFA \( P = (L, D, U, Y, \Sigma, \Delta, L_0, D_0, L_m, E) \), its finite abstracted transition system \( P^f \) is a tuple \( P^f = (S, U^f, Y^f, \Sigma, E^f, S_0) \), where

\[
S = L \times D^f \quad \text{is the set of its states, where } D^f \text{ is the finite abstraction of } D, \\
E^f = \{((l_1, d_1^f), (l_2, d_2^f)) | \exists d_1 \in d_1^f, u \in u^f, y \in y^f, d_2 \in d_2^f : (l_1, d_1) \xrightarrow{\sigma, u, y, \delta, g} (l_2, d_2) \} \text{ is its set of transitions,} \\
S_0 = L_0 \times D_0^f \text{ is the set of its initial states, where } D_0^f \text{ is the finite abstraction of } D_0, \\
U^f \text{ is the finite abstraction of } U, \\
Y^f \text{ is the finite abstraction of } Y.
\]

The finite abstracted transition system is implemented in the NuSMV input language, where:

- The locations \( L \) of the I/O-EFA model is set as a variable and each location \( l_i \) is a value for the variable “locations”.
- Each discrete variable \( d_j \) in the I/O-EFA has its corresponding variable in NuSMV file. Data update functions \( f_c : D \times U \rightarrow D \) are expressed by the “next(\( l_i \))” functions in the assignment part of NuSMV file.

- Each input variable \( u_k \) is defined in the NuSMV model as a nondeterministic variable. It can choose any value in its range at the beginning of each time-step.
- Edges \( E \) in I/O-EFA model are mapped to a variable “edgeNum”, and each edge \( e_i \) corresponds to an integer value of variable “edgeNum”. This integer value is determined by the edge number in the I/O-EFA model. Thus, a sequence of “edgeNum” value in NuSMV file represents a sequence of edges, i.e. a path, in the I/O-EFA model.

The corresponding NuSMV file is used to check if the \( c \)-paths in the I/O-EFA model are reachable. This is done as an instance of finding a counterexample as prescribed in the following algorithm.

**Algorithm 1:** A \( c \)-path \( \pi = e_0 \ldots e_{n-1} \) of an I/O-EFA \( P = (L, D, U, Y, \Sigma, \Delta, L_0, D_0, L_m, E) \) is determined to be reachable if in the finite abstraction \( P^f \models \phi \) holds, where \( \phi \) is the CTL formula \( E F (e_0^f \land E X (e_1^f \cdots \land E X e_{n-1}^f) \ldots) \), meaning path \( \pi \) can eventually be activated in the finite abstraction \( P^f \). An input sequence that makes \( \pi \) eventually executable is found as a counterexample to the model-checking problem \( P^f \models \neg \phi \).

If a counterexample for \( P^f \models \neg \phi \) is found, then \( P^f \models \phi \) holds, and the sequence of inputs within the counterexample is a test case activating the path \( \pi \). The final test suite is the set of input sequences obtained from a subset of reachable paths satisfying a desired coverage criterion.

In summary, the model-checking based test generation implementation generates the test cases by the following steps.

- Translate the Simulink/Stateflow diagram into I/O-EFA model;
- Map the I/O-EFA model to the corresponding NuSMV file;
- Extract all the paths from the I/O-EFA model and translate them into corresponding CTL specifications;
- Check the CTL representations of the paths against the NuSMV model. Select the reachable paths satisfying the coverage criterion and the set of input sequences activating those paths as the test suite. Report the unreachable paths for the analysis of model soundness and robustness.

The above implementation utilizes the existing model checker NuSMV and automates the test generation for Simulink/Stateflow. However, model-checking process is time-consuming as the state space explodes and the finite abstraction may also cause problems in the test generation. So we investigate another approach as described next.

B. Implementation using Constraint Solving

Mathematical optimization is used to check feasibility of a set of constraints and to select a best element from a set of available alternatives. The standard form of an optimization problem is:

\[
\begin{align*}
\text{minimize}_x & \quad f(x) \\
\text{subject to} & \quad g_i(x) \leq 0, \ i = 1, \ldots, m \\
& \quad h_i(x) = 0, \ i = 1, \ldots, p
\end{align*}
\]

where

- \( f(x) : R^n \rightarrow R \) is the objective function to be minimized over the variable \( x \),
Constraint Solving based Test Generator:

For a path \( P \) of an I/O-EFA, if the path predicate \( P^e(x) \) is satisfiable (does not equate to False), the path predicate \( P^e(x) \) is reachable, \( \pi \). The above constraint solving problem has solution if the path predicates \( P^e(x) \) is satisfiable (does not equate to False). The path predicate \( P^e(x) \) along with its data \( d^e \) and output \( y^e \) can be obtained as follows.

Algorithm 2: For a path \( \pi = e_0...e_{|\pi|-1} \), its path-predicate \( P^e(x) \) can be computed recursively backward, and data \( d^e \) and output \( y^e \) can be computed recursively forward as:

Base step:
\[ P^e_j(d, u, y) := \text{True}; \]
\[ d^e_0(d, u, y) := d; \]
\[ y^e_0(d, u, y) := y. \]

Recursion step:
\[ P^e_{j+1}(d, u, y) := G_{e_{j-1}}(d, u, y) \land P^e_j(d, u, y), u, h_{e_{j-1}}(d, u, y); \]
\[ d^e_{j+1}(d, u, y) := f^e_{j-1}(d^e_{j-1}(d, u, y), u, y^e_j(d, u, y)); \]
\[ y^e_{j+1}(d, u, y) := h^e_{j-1}(d^e_{j-1}(d, u, y), u, y^e_j(d, u, y)). \]

Termination step:
\[ P^e(d, u, y) := P^e_0(d, u, y); \]
\[ d^e(d, u, y) := d^e_0(d, u, y); \]
\[ y^e(d, u, y) := y^e_0(d, u, y). \]

Note: If any of \( G_e \) is undefined, it is simply assumed true, i.e., \( G_e(d, u, y) = \text{True} \), and similarly if any of \( h_e \) is undefined, then it is simply assumed to be the same as identity, i.e., \( h_e(d, u, y) = y \).

Constraint solving problem is constructed to check if \( P^e(d, u, y) \neq \text{False} \), in which case, the path \( \pi \) is feasible. The feasible paths obtained in Algorithm 2 are the candidate paths for test generation. They are further checked to see if they can be reached from the initial condition, i.e., if there exists a feasible path-sequence \( \Pi \) starting at the initial condition and ending with the path under evaluation for reachability. The algorithm to determine the feasibility and reachability of a path-sequence \( \Pi \) is as follows.

Algorithm 3: Given a path-sequence \( \Pi = \pi_0...\pi \) starting at the initial condition \( I(d, u, y) \) ending with the path \( \pi \), the feasibility/reachability of \( \Pi \) can be checked recursively backward as:

Base step:
\[ P^{\Pi}_0(d, u, y) := P^\pi; \]

Recursion step:
\[ P^{\Pi}_{j+1}(d, u, y) := P^\Pi_{j-1} \land P^{\Pi}_j(d^\Pi_j, u_j, y^\Pi_j); \]

Termination condition:
- If \( P^{\Pi}_{j-1}(d, u, y) = \text{False} \), then \( \Pi \) is infeasible and also unreachable, stop;
- If \( P^{\Pi}_{j-1}(d, u, y) \neq \text{False} \) and \( j \neq 0 \), then decrement \( j \) and return to recursion step;

If \( j = 0 \), then \( \Pi \) is reachable iff \( P^{\Pi}_j(d, u, y) \land I(d, u, y) \neq \text{False} \), stop.

Given a feasible path \( \pi \), if none of the path-sequence with \(|\Pi| \leq \text{steplimit}\) ending with \( \pi \) is reachable, \( \pi \) is unreachable within the steplimit. Otherwise, \( \pi \) is reachable. Note steplimit is the test case length requirement of the system.

Given a feasible path-sequence \( \Pi = \pi_0...\pi \) ending with path \( \pi \), a test input sequence \( t_\pi = u_0...u_{|\Pi|-1} \) activating the reachable path \( \pi \) is obtained by letting \( u_j \in P^\Pi_j, j = 0...|\Pi| - 1 \). This input sequence \( t_\pi \) is the test case for path \( \pi \).

The constraint solving based test suite is derived with an open source optimization tool CVX [10], written in Matlab. Our test generation tool calls the CVX tool to check the feasibility of the problem.

In summary, this constraint solving based test generation implementation generates the test cases using the following steps:

- Translate the Simulink/Stateflow diagram into I/O-EFA model;
- Extract all the paths from the I/O-EFA;
- Apply Algorithm 2 to obtain the feasible paths;
- Apply Algorithm 3 to obtain the reachable paths (the test cases are generated based on the reachable paths satisfying the coverage criterion);
- Report the unreachable paths identified in the previous two steps for the analysis of model soundness and robustness.

The above implementation applies the constraint solving to solve for the recursively obtained path predicates. This method does not require finite abstraction of the data space and loading of the model in another tool. This implementation is thus exact (requiring no abstraction) and is able to generate test cases faster than the implementation based on the model checker.

V. VALIDATION OF TEST GENERATION IMPLEMENTATIONS

Both of the test generation implementations described above, as well as the Simulink/Stateflow to I/O-EFA translation tool, have been incorporated in an automated test generation tool. Upon specifying a source Simulink/Stateflow model file, both of our implementation methods can be executed to output the test suite for the corresponding Simulink/Stateflow diagram.

Example 2: Model Checker based Test Generator: Consider the I/O-EFA model (see Figure 2) of the counter system (see Figure 1). By specifying \( \text{steplimit} = 10, \text{precision} = 0 \), all variables within \([-2, 10], u \in \{0, 1\} \), and path-covered criterion (all paths be covered), the model checker based test generator generates four reachable paths and the corresponding test cases are shown in Table I.

The test generation time (using Intel Core 2 Duo P8400 2.27GHz, 2GB RAM) is 349.3 seconds and the results are as expected.

Example 3: Constraint Solving based Test Generator: Consider the same I/O-EFA model (see Figure 2) of the counter system (see Figure 1). By specifying \( \text{steplimit} = 10, \text{precision} = 0, \text{termination step} = 50 \), all variables within \([-2, 10], u \in \{0, 1\} \), and path-covered criterion (all paths be covered), the model checker based test generator generates four reachable paths and the corresponding test cases are shown in Table I.
TABLE I
REACHABLE PATHS AND TEST CASES FROM IMPLEMENTATION WITH MODEL-CHECKING

<table>
<thead>
<tr>
<th>Path</th>
<th>Test Case (u at each sample time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0e7e2e6e2e6e10e12e14e15</td>
<td>1</td>
</tr>
<tr>
<td>e1e6e7e1e6e19e20e21</td>
<td>1, 1</td>
</tr>
<tr>
<td>e1e3e4e5e6e7e8e9e12e14e15</td>
<td>1, 1, 1, 1, 1, 1, 1, 1, 1</td>
</tr>
<tr>
<td>e2e6e1e0e12e13e20e21</td>
<td>0</td>
</tr>
</tbody>
</table>

TABLE II
FEASIBLE PATHS FROM IMPLEMENTATION WITH CONSTRAINT SOLVING

<table>
<thead>
<tr>
<th>Path No.</th>
<th>Path Predicate</th>
<th>Path Data</th>
<th>Path Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>π0</td>
<td>u(k) &gt; 0 ∧ d’ = 0</td>
<td>d(k) := 0, y3 := 1, y5(k) := 0, y4(k) := 1, d(k + 1) := 1, k := k + 1</td>
<td>y2(k) := 0</td>
</tr>
<tr>
<td>π1</td>
<td>u(k) &gt; 0 ∧ d’ = 1 ∧ -0.5 ≤ d(k) ≤ 7</td>
<td>y3(k) := 1, y5(k) := d(k), y4(k) := d(k) + 1, d(k + 1) := d(k) + 1, k := k + 1</td>
<td>y2(k) := d(k)</td>
</tr>
<tr>
<td>π2</td>
<td>u(k) &gt; 0 ∧ d’ = 1 ∧ d(k) &gt; 7</td>
<td>y3(k) := 1, y5(k) := d(k), y4(k) := d(k) + 1, d(k + 1) := d(k) + 1, k := k + 1</td>
<td>y2(k) := 7</td>
</tr>
<tr>
<td>π3</td>
<td>u(k) &gt; 0 ∧ d’ = 1 ∧ d(k) &lt; -0.5</td>
<td>y3(k) := 1, y5(k) := d(k), y4(k) := d(k) + 1, d(k + 1) := d(k) + 1, k := k + 1</td>
<td>y2(k) := -0.5</td>
</tr>
<tr>
<td>π4</td>
<td>u(k) ≤ 0</td>
<td>y5(k) := 2, d’ := 0, d(k + 1) := d(k), k := k + 1</td>
<td>y2(k) := 2</td>
</tr>
</tbody>
</table>

VI. CONCLUSION

We presented an Input/Output Extended Finite Automata (I/O-EFA) based test generation approach for Simulink/Stateflow. While preserving the discrete behaviors, a Simulink/Stateflow diagram is translated to an I/O-EFA model, with each path of the I/O-EFA model representing a computation sequence of the Simulink/Stateflow diagram. Paths are inspected for feasibility and reachability. They are further used for test generation and model soundness analysis. Two techniques, model-checking and constraint solving, are applied to implement this approach. Model-checker based implementation maps I/O-EFA to a finite abstracted transition system modeled in NuSMV file. Test cases are generated by checking each path in I/O-EFA against the model in NuSMV. Constraint solving based implementation utilizes two algorithms to recursively compute the path and path-sequence predicate respectively for capturing the feasibility problems. Test cases are obtained from the predicates of the reachable paths. The performance of both implementations was evaluated with a case study. The results showed that both implementations can generate the expected results and the implementation based on constraint solving is superior to the implementation based on model checker with respect to the speed of test generation.

REFERENCES


10, u ∈ {0, 1}, and path-covered criterion (all paths be covered), the constraint solving based test generator provides five feasible paths as shown in Table II and four of them are reachable (π3 is identified as unreachable). The test cases are generated as shown in Table III.

The test generation time (using Intel Core 2 Duo P8400 2.27GHz, 2GB RAM) is 102.7 seconds and the results are as expected.

The two test generators provide identical test cases regarding the same Simulink/Stateflow diagram and specifications. Constraint solving based implementation is able to obtain the result about two times faster than model checker based implementation.