Reduction of Automated Test Generation for Simulink/Stateflow to Reachability and its Novel Resolution

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 a test generation approach for Simulink/Stateflow by reduction to reachability in a Hybrid Automaton, with its locations representing the computations of the Simulink/Stateflow model, and edges representing the computation-succession. A novel reachability resolution method is presented based on the refinement of the hybrid automaton such that the reachability is reduced to the reachability in the underlying graph (without the dynamics), whenever the refinement step terminates. The approach yields a technique that is effective in terms of achieving test coverage and efficient in terms of test generation time.

I. INTRODUCTION

Simulink/Stateflow [1] is a model-based development tool, which is extensively used in many industrial domains. Automated test generation is an essential step in the model-based development using Simulink/Stateflow, that aims to catch errors early in the design process.

Several authors have proposed different ways of test generation and verification for Simulink/Stateflow diagrams (see [2], [3], [4], [5], [6], [7]).

In our previous paper [8], we derive the test suite based on our recursive and automated translation method from Simulink/Stateflow to an Input-Output Extended Finite Automata (I/O-EFA) (see [9], [10]), which preserves the behaviors observed at sampling instances. Each path from initial to final location in the I/O-EFA, corresponds to one possible computation step of the Simulink/Stateflow, and to execute this computation step, some other computation steps may need to be executed first. In [8], we examined two methods, model-checking and constraint-solving, for determining which computation steps can be eventually executed, equivalently "reachable", by a priori bounding number of steps to be explored, thus only checking the sufficiency for reachability. In this paper we propose a new approach for reachability resolution that is both necessary and sufficient.

Our contributions are:

1. We reduce the problem of eventual executability of a computation graph to that of reachability in a discrete-time computation-succession hybrid automaton (CS-HA);

2. A recursive refinement method of CS-HA that preserves reachability, and upon termination reduces reachability to graph connectivity;

3. An algorithm to obtain a test (an input sequence) that eventually executes a reachable computation path. Note the eventual executability problem is in general undecidable, and so the refinement step need not always terminate, but the approach itself is both necessary and sufficient (unlike [8]).

II. PRELIMINARIES

In our prior works [9] [10], we modeled a Simulink/Stateflow diagram as an Input/Output Extended Finite Automaton (I/O-EFA) model preserving its discrete behaviors. In this paper we reduce the test generation problem to the reachability in a discrete-time Input/Output Hybrid Automaton (I/O-HA) model, which is more general than an I/O-EFA model, and is defined as follows.

Definition 1: An I/O-HA 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), and each \( l \in L \) is a 3-tuple, \( l = (G_l, f_l, h_l) \), where
  - \( G_l \subseteq D \times U \) is location-invariant,
  - \( f_l : D \times U \rightarrow D \) is data-update function, and
  - \( h_l : D \times U \rightarrow Y \) is output-assignment function.
- \( D = D_1 \times \cdots \times D_n \) 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 origin location,
  - \( t_e \in L \) is terminal location,
  - \( \sigma_e \in \Sigma \) is symbolic-input,
  - \( \delta_e \in \Delta \) is symbolic-output,
  - \( G_e \subseteq D \times U \) is enabling guard (a predicate),
  - \( f_e : D \times U \rightarrow D \) is data-update function, and
  - \( h_e : D \times U \rightarrow Y \) is output-assignment function.

An I/O-HA \( P \) starts from an initial location \( l_0 \in L_0 \) with initial data \( d_0 \in D_0 \). Within a location \( l \), \( P \) evolves over discrete-time steps as long as the data satisfies the invariant guard condition \( G_l \), and at each time step uses the data update function \( f_l \) and the output assignment function \( h_l \) to modify the data and the output. 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. In what follows below, the data update and output assignments are performed together in a single action.

An I/O-EFA is a specialized I/O-HA with location-invariant as $\text{True}$, location update and assignment functions as the identity maps.

**Example 1:** Consider the Simulink diagram of a bounded counter shown in Fig. 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 100000. The translated I/O-EFA $P$ using the method of [9] is shown in Fig. 2.

![Simulink Diagram of a Counter System](image)

**III. COMPUTATION-SUCCESSION HYBRID AUTOMATON**

Each single-input computation of a Simulink/Stateflow diagram is represented as a computation-path of an I/O-EFA model. Thereby the test generation problem reduces to finding for each computation-path an input-sequence, that eventually executes that c-path. Formally, a computation path of an I/O-EFA is defined as follows.

**Definition 2:** A computation path (or simply a c-path) $\pi$ in an I/O-EFA $P = (L, D, U, \Sigma, \Delta, L_0, D_0, L_m, E)$ is a finite sequence of edges $\pi = \{e_0^\pi \ldots e_{|\pi|-1}^\pi \in E\}^{\omega}$ such that $l_0^\pi \in L_0, h_{|\pi|-1}^\pi \in L_m$.

For each c-path sequence $\omega = \pi_0 \ldots \pi_{|\omega|-1}$, it is possible to compute its enabling guard $G_{\omega} \subseteq D \times U$ recursively backwards, and its data-update function $f_{\omega} : D \times U \rightarrow D$ and its output-assignment function $h_{\omega} : D \times U \rightarrow Y$ recursively forward using [8, Algorithm 2-3]. Then a c-path $\pi$ is immediately executable (equivalently, feasible) if and only if its enabling guard $G_{\pi}(d,u)$ is satisfiable, and $\pi$ is eventually executable (equivalently, reachable) if there is a path-sequence $\omega$ ending in $\pi$ (i.e., $\pi_{|\omega|-1} = \pi$), such that $G_{\omega}(d,\{u_0 \ldots u_{|\omega|-1}\})$ is satisfiable.

**Example 2:** Consider I/O-EFA model of Fig. 2. Then as can be seen it has 18 different feasible c-paths starting and ending in the initial location, and going around the loop once (which is exactly the computation of one time-step). It turns out that only 5 out of 18 feasible c-paths are feasible, as analyzed by [8, Algorithm 2], and only 4 of 5 are reachable. These are listed in Table I.

Since the reachability of a c-path depends on the succession of computations, we introduce the notion of a Computation-Succession Hybrid Automaton to characterize the reachability of the c-paths.

**Algorithm 1:** Given a set of feasible c-paths $\Pi_P$ of an I/O-EFA model $P = (L, D, U, \Sigma, \Delta, L_0, D_0, L_m, E)$, its Computation-Succession Hybrid Automaton (CS-HA) is obtained as, $P^{\Pi} = (L^{\Pi}, D, U, \Sigma, \Delta, L_0^{\Pi}, D_0^{\Pi}, L_m^{\Pi}, E^{\Pi})$, where

- $L^{\Pi} = \bigcup_{\pi \in \Pi_P} \{l^\pi := (G_{\pi}(d,u), f_{\pi}(d,u), h_{\pi}(d,u))\}$ is its set of locations, 1-to-1 mapped to $\Pi_P$. ($P^{\Pi}$ has one location $l^\pi$ for each feasible c-path $\pi$ of $P$, and $l^\pi$’s invariant/data-update/output-assignment are the same as the guard/data-update/output-assignment of $\pi$).

- $L_0^{\Pi} = \bigcup_{\pi \in \Pi_P \cap G_{\pi}(d,u) : D_0 \neq False \{l^\pi\}$ is its set of initial locations. (Initial locations of $P^{\Pi}$ are the initially executable c-paths of $P$.)

- $E^{\Pi} = \bigcup_{e,\pi \in \Pi_P \cap \pi \neq \pi'} \{(l^\pi, l^{\pi'}, \delta, \delta, G_{\pi'}(d,u), \delta, \delta)\}$ is its set of transitions. (Each feasible c-path of $P$ may
be succeeded by each another feasible c-path of \( P \), and so \( P^{\Pi_l} \) has an edge-set that makes its graph completely connected, with each edge guarded by the invariant (equivalently, guard) of its successor location.)

Note, by definition, the CS-HA is a completely connected graph over the set of feasible c-paths acting as nodes (locations), with each incoming edge to a c-path node guarded by that c-path’s guard condition, and the set of initially reachable c-paths serving as the set of initial nodes. For Simulink/Stateflow diagrams with deterministic runs, the corresponding I/O-EFA model is deterministic, and as a result the enabling guards of the c-paths are pair-wise disjoint, meaning the set \( \{ G_\pi(d, u) \mid \pi \in \Pi^P \} \) defines a partition of the set \( D \times U \), implying that the CS-HA \( P^{\Pi_l} \) is also deterministic. Finally note that the CS-HA \( P^{\Pi_l} \) does not possess any self-loops, rather the repeated execution of a c-path is captured through the semantics of a hybrid-automaton that allows evolution in the same location for multiple time-steps, tantamount to executing a self-loop.

The following result is clear from construction.

**Theorem 1:** Given an I/O-EFA \( P \) modeling a Simulink/Stateflow diagram, a feasible c-path \( \pi \in \Pi^P \) is reachable if and only if the location \( l_\pi \) is reachable in the corresponding CS-HA \( P^{\Pi_l} \).

**Example 3:** Given the feasible paths in Table I, the corresponding CS-HA is shown in Fig. 3.

**IV. Reachability Resolution for CS-HA**

To aid the reachability analysis, we present a novel reachability resolution technique that refines the CS-HA such that location reachability is equivalent to reachability in the underlying graph, ignoring the dynamics, whenever the refinement terminates.

For this, the locations are split according to the preconditions to reach their successors. The precondition of the transition from one location to another is defined in the algorithm below. It requires the computation of the guard condition that allows the N steps of evolution in location \( l \), along with the corresponding data-updates and output-assignments.

**Algorithm 2:** Given a CS-HA \( P^{\Pi_l} \), for each \( l \in L^{\Pi_l} \), do the following:

Base step:

\[
\begin{align*}
& j = 0, i = (N - 1) - j; \\
& G^0_l(d, u_{k+1}) := G_l(d, u_{k+1}); \\
& f^j_l(d, u_{k+j}) := f_l(d, u_{k+j}); \\
& h^j_l(d, u_{k+j}) := h_l(d, u_{k+j}).
\end{align*}
\]

Let \( \pi_l \) be the feasible c-path of \( P \) represented by a location \( l \) of \( P^{\Pi_l} \). Then the base step computes the guard for the last, i.e., \( N \)th execution of \( \pi_l \),
where \( \text{l} \) is inherited as is. The refinement of the CS-HA is defined as

while its data-update and output-assignment functions are

satisfies the precondition to reach a subset of successors,

and split the locations accordingly, so each split

to a successor \( \text{l} \) together with the data-update and output-assignment of the

Recursion step:

\[
G_{l,i-1}(d, \{u_{k,i-1}, \ldots, u_{k+N-1}\}) := G_l(d, u_{k+i-1}) \land G_l(f_l(d, u_{k+i-1}), \{u_{k+i-1}, \ldots, u_{k+N-1}\}, h_l(d, u_{k+i-1}));
\]

\[
f_{l,i-1}(d, \{u_{k}, \ldots, u_{k+j+1}\}) := f_l(f_l(d, \{u_{k}, \ldots, u_{k+j}\}), \{u_{k+j+1}, h_l(d, u_{k}, \ldots, u_{k+j})\});
\]

\[
h_{l,i}^1(d, \{u_{k}, \ldots, u_{k+j+1}\}) := h_l(f_l(d, u_{0}, \ldots, u_{k+j})), \{u_{k+j+1}, h_l(d, u_{k}, \ldots, u_{k+j})\}). \] (Recursion step computes the guard of the last \( N - (i - 1) \) executions of \( \pi_l \), together with the data-update and output-assignment of the first \( j + 1 \) executions of \( \pi_l \). Note the former calculation uses the guard for last \( N - i \) executions of \( \pi_l \) (a backward recursion), whereas the latter calculation uses the data-update/output-assignment of the first \( j \) executions of \( \pi_l \) (a forward recursion).

Termination step:

\( j \neq N - 1 \), then increment \( j \) and return to recursion step; else stop, and define the precondition \( G_{l,i}(d) \) to transit from \( l \) to a successor \( l' \in \text{succ}(l) \), as:

\[
G_{l,i}(d) := \bigvee_{N \geq 1} \left[ \exists u\{u_{k}, \ldots, u_{k+N} \} : G_{l,N}(d, \{u_{k}, \ldots, u_{k+N}\}) \land G_{l,i}(f_{l,i-1}(d, \{u_{k}, \ldots, u_{k+N}\}, u_{k+N}) } \right].
\]

(Upon termination, the precondition to transit from \( l \) to successor \( l' \) by evolving at \( l \) for one or more steps is computed. Note that \( G_{l,i}(d) \) is solvable whenever \( G_{l,0}(d) \) and \( f_{l,0} \) can be analytically computed.)

Next these preconditions of the transitions from the locations to their successors are used to partition the location-invariants, and split the locations accordingly, so each split location is endowed with its own stronger invariant, which satisfies the precondition to reach a subset of successors, while its data-update and output-assignment functions are inherited as is. The refinement of the CS-HA is defined as follows.

**Algorithm 3:** Given a CS-HA \( P^{\Pi} = (L^{\Pi}, D, U, Y, \Sigma, \Delta, D^0, D^0, L^{\Pi}, E) \), the refinement algorithm iteratively computes for each iteration \( n \), a refined hybrid automaton \( P^n = (L^n, D, U, Y, \Sigma, \Delta, L^n, D^n, L^n, E^n) \), where \( L^n_0 := \{ l \in L^n \mid G_l \land D_0 \neq \text{False} \} \), and \( E^n := \{ (l, l', \ldots, l', G_{l'}, \ldots, \) \} \) \( l, l' \in L^n, l \neq l' \), \( G_{l'}(d) \neq \text{False} \) \} as follows (note for each \( n \geq 0 \), only \( L^n \) needs to be iteratively computed since definitions of \( L^n_0 \) and \( E^n_0 \) are derived from that of \( L^0 \)):

Base step: \( L^0 = \{ (G_l(d, u), f_l(d, u), h_l(d, u)) \mid l \in L^{\Pi} \}. \) (Locations of \( P^0 \) are the same as those of \( P^{\Pi} \).)

Recursion step: \( L^{n+1} = \bigcup_{l \in L^n} G_l(d) \cap G_l(f_l(d, u), h_l(d, u)), \) where for each \( l \in L^n \), \( G_l := \bigcup_{l' \in \text{succ}(l)} \{ (G_l(d) \land \neg G_{l'}(d)) \} \) is the partition induced by \( [ G_{l'}(d) \mid l' \in \text{succ}(l) ] \). (To obtain the locations \( L^{n+1} \), each location \( l \) of \( L^n \) is split into a number of locations, one per subset of the successors of \( l \). The guard condition of a split location is the precondition to reach a certain subset of successors of the original location, while the data-update and output-assignment are preserved after the split.)

Termination step: If \( L^{n+1} = L^n \) or step-limit, stop, 

and set \( P^{\Pi} := P^n \); else, increment \( n \) and return to recursion step. (Termination occurs when splitting does not introduce additional locations since the extra ones turn out to have False guards.)

**Example 4:** Consider the CS-HA shown in Fig. 4. The CS-HA is refined according to Algorithm 3. Firstly, since \( l^{1,1} \) has three successors with three different edge guards, there are eight different subsets of successors, but only three of them have non-False preconditions, and so \( l^{1,1} \) is split into three locations \( l^{1,1}_0, l^{1,1}_1, \) and \( l^{1,1}_2 \) as in Fig. 5. This requires the application of Algorithm 2 to find the guards \( G_{l^{1,1}_0} = \{ d(k) = 2 \}, \) \( G_{l^{1,1}_1} = \{ d(k) = 3 \}, \) \( G_{l^{1,1}_2} = \{ 2 < d(k) < 3 \lor 3 < d(k) \leq 4 \}, \) and then performing the refinement as in Algorithm 3 that splits \( l^{1,1} \) into \( l^{1,1}_0, l^{1,1}_1, l^{1,1}_2 \) with the invariants \( G_{l^{1,1}_0}, G_{l^{1,1}_1}, G_{l^{1,1}_2} \), respectively, and with the same data-update and output-assignment functions as \( l^{1,1} \). Next, since \( l^{1,0} \) has three successors with three different edge guards, there are eight different subsets of successors, but only three of them have non-False preconditions, and so \( l^{1,0} \) is split into three locations \( l^{1,0}_0, l^{1,0}_1, \) and \( l^{1,0}_2 \) as in Fig. 6. Again this requires applying Algorithm 2 to find the guards \( G_{l^{1,0}_0} = \{ d(k) = 1 \}, \) \( G_{l^{1,0}_1} = \{ 0 < d(k) \leq 1 \}, \) and then performing the refinement as in Algorithm 3 that splits \( l^{1,0} \) into \( l^{1,0}_1, l^{1,0}_1, l^{1,0}_2 \) with the invariants \( G_{l^{1,0}_1}, G_{l^{1,0}_1}, G_{l^{1,0}_2} \), respectively, and with the same data-update and output-assignment functions as \( l^{1,0} \). Note only the node \( l^{1,0}_0 \) remains an initial node since the invariant condition for \( l^{1,0}_0 \) and \( l^{1,0}_2 \) are \( [ d(k) = 1 ] \) and \( [ 0 < d(k) < 1 ] \), which are disjoint from the initial condition \( [ d(k) = 0 ] \). Also only the node \( l^{1,0}_1 \) remains reachable from \( l^{1,2} \), whose outgoing edge guard \( [ d(k) = 1 ] \) has nonempty overlap with only the invariant of \( l^{1,0}_1 \). At this point, each node has at most one successor, and so refinement introduces no additional locations (meaning \( L^{n+1} = L^n \)), causing Algorithm 3 to terminate and yielding the refined CS-HA of Fig. 6.

![Fig. 4. CS-HA with a cycle-location possessing more than one successor](image-url)

The following theorem establishes that the refinement step indeed resolves the reachability. The proof is omitted due to space limitations.

**Theorem 2:** When Algorithm 3 terminates in finite steps with \( L^{n+1} = L^n \), then the refined PS-HA \( P^{\Pi} \) from Algorithm 3 has the property that, if there exists a path from the initial locations to a target location, then the target location is reachable.
The following theorem provides a condition for the termination of Algorithm 3. It employs the notion of late-bisimilarity and late-bismulation quotient, which can be found in [11] in the context of I/O-EFA. Since discrete-time I/O-HA can be straightforwardly translated to I/O-EFA (by removing the guards/data-updates/output-assignments from the locations and introducing self-loop edges with the same guards/data-updates/output-assignments), the definition of [11] also apply to I/O-HA. The proof is omitted due to space limitations.

**Theorem 3:** Algorithm 3 terminates if and only if the CS-HA of the Simulink/Stateflow model possesses a finite late-bisimilar quotient.

**Remark 1:** Example 5 shows a drastic improvement compared to the approach of [8], since to reach the c-path $\pi_2$, a prefix of length 100000 must be executed first. (The guard condition for $\pi_2$ requires a variable to exceed 100000, while that variable has an initial value 0, and is incremented by just one, each time a prefix $\pi_1$ is executed.) Finding such a path using the search employed in [8] is impossible since it has the complexity of $5^{100000}$, which is prohibitive. In contrast, the new reachability and its resolution based approach presented here succeeds in establishing the reachability of all reachable c-paths.

**Remark 2:** Note that the reachability resolution approach can also be applied to general hybrid automata. If a hybrid automaton satisfies the property in Theorem 3, Algorithm 3 terminates within finite steps, yielding a refined hybrid automaton that is a finite late-bisimilar quotient, and for which the reachability is decidable.

V. **Test Generation Based on CS-HA**

Once the reachability of a c-path is resolved using the refinement of the CS-HA proposed in the previous section, the following algorithm can be used to generate a test case for the c-path, i.e., an input sequence that ensures the eventual execution of the c-path.

**Algorithm 4:** A c-path $\pi \in \Pi^T$ is reachable if there exists a location $l \in L^n = L^{n+1}$ in the refined CS-HA, with $G_1 \Rightarrow$
where \( G_\pi \), and a path \( \omega = l_0 \ldots l_{|\omega|-1} \) starting at an initial location \( l_0 \in L_0^P \) and ending with the location \( l_{|\omega|-1} = l \). For the path \( \omega \), its test case can be computed iteratively forward as:

**Base step:**

\[
\text{if } j = k = 0, \text{ and solve for } d_j, N_j, \{u_k, \ldots, u_{k+N_j}\} \text{ such that the following holds:}
\]

\[
D_0 \wedge G_0^j(d_j, \{u_k, \ldots, u_{k+N_j-1}\}) \wedge G_{l_j+1}(f_{N_j}^{-1}(d_j, \{u_k, \ldots, u_{k+N_j-1}\}), u_{k+N_j}).
\]

(The base step finds an initial \((j = 0)\) data \( d_j \), an initial sequence of \( N_j \) inputs that execute the initial \( c \)-path \( l_j \) a total \( N_j \) number of times, so that the resulting data \( d_{l_j+1} \) possesses a next input that can execute the next \( c \)-path \( l_{j+1} \). The base step also finds this next input \( u_{N_j} \).

**Recursion step:**

If \( j = |\omega| - 1 \), then go to termination step, else set

\[
d_{l_j+1} := f_{N_j}^{-1}(d_j, \{u_k, \ldots, u_{k+N_j-1}\}), \{y_{k+i} := h_{l_j}^{-1}(d_j, \{u_k, \ldots, u_{k+i}\}) | 0 \leq i \leq N_j - 1\}, k := k + N_j, j := j + 1, \text{ and solve for } N_j \text{ and } \{u_{k+1}, \ldots, u_{k+N_j}\} \text{ such that the following holds:}
\]

\[
G_0^j(d_j, \{u_k, \ldots, u_{k+N_j-1}\}) \wedge G_{l_j+1}(f_{N_j}^{-1}(d_j, \{u_k, \ldots, u_{k+N_j-1}\}), u_{k+N_j}).
\]

and return to recursion step. (Similar to the base step, the recursion finds \( j \)-th sequence of \( N_j \) inputs so that the \( j \)-th \( c \)-path \( l_j \) can be executed \( N_j \) number of times, so that the resulting data \( d_{l_j+1} \) possesses a next input that can execute the next \( c \)-path \( l_{j+1} \). The recursion step also finds this next input \( u_{N_j} \).

**Termination step:**

Return \( d_0 \) and the input/output-sequence \( \{(u_0, y_0), \ldots, (u_k, y_k)\} \) as the test case. (The recursion stops when \( j = |\omega| - 1 \) at which point each \( c \)-path in \( \omega \) has been executed a certain number of times in the order as appearing in \( \omega \).

**Remark 3:** In order to compute a test case for a reachable \( c \)-path, Algorithm 4 requires an analytical solution of all \( \{f_j, h_j : l \in \omega\} \), and a solver that can solve for the constraints \( G_l : l \in \omega \).

In summary, the overall algorithm of our proposed test generation approach for a Simulink/Stateflow model is as follows.

**Algorithm 5:**

1. Obtain the I/O-EFA model \( P \) of a given Simulink/Stateflow diagram according to [9] [10].
2. Apply [8, Algorithm 2] to enumerate the feasible \( c \)-paths \( \Pi^P \) of \( P \).
3. Apply Algorithm 1 on \( \Pi^P \) to obtain the CS-HA \( \Pi^P \) of I/O-EFA \( P \).
4. Apply Algorithm 3 to refine \( \Pi^P \) and obtain the refined CS-HA \( \Pi^P \).
5. Apply Algorithm 4 on \( \Pi^P \) to identify reachable \( c \)-paths of \( P \), and to generate their test cases.

**Example 6:** Consider the refined CS-HA in Fig. 7 of the counter in Fig. 1. By applying Algorithm 4 on the refined CS-HA, the test cases to reach the reachable \( c \)-paths are obtained in Table II. As can be noted, one of the test cases has a length \( > 100K \), which, as discussed in Remark 1, could not be generated using the search-based method of [8]. This illustrates the effectiveness of the new approach proposed here in terms of providing a better test coverage, and also its efficiency in terms of the time needed for automated test generation.

**VI. Conclusion**

We presented an improved test generation approach for Simulink/Stateflow extending and enhancing our prior work [8]. A discrete-time hybrid automaton called a computation-succession hybrid automaton (CS-HA) was introduced to capture the feasible computation-succession among the feasible \( c \)-paths. The test generation problem was then reduced to a reachability analysis problem of the CS-HA. A novel reachability resolution method was introduced to refine the CS-HA, such that the reachability is reduced to the reachability within its underlying graph, ignoring the dynamics. Test generation was then performed over the refined CS-HA by selecting a path from the initial locations to a target location and finding an input sequence to activate the path. The overall algorithm for the test generation approach was shown to be decidable for the class of Simulink/Stateflow diagrams possessing a finite late-bisimilar quotient.

**References**