2013

A model-based approach to automated test generation and error localization for Simulink/ Stateflow

Meng Li
Iowa State University

A model-based approach to

automated test generation and error localization for Simulink/Stateflow

by

Meng Li

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Ratnesh Kumar, Major Professor
Nicola Elia
Joseph Zambreno
Carl Kochao Chang
Atul G. Kelkar

Iowa State University

Ames, Iowa

2013

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this dissertation.

First and foremost, Dr. Kumar for his guidance, patience and support throughout this research and the writing of this dissertation. His insights and words of encouragement have often motivated and inspired me for completing my graduate education. I also thank him for teaching me the art of writing scientific articles and providing me with financial support throughout my graduate studies at Iowa State University. Working with him has been a very rewarding and pleasant experience.

I would also like to express my cordial thanks to Dr. Elia, Dr. Zambreno, Dr. Chang, and Dr. Kelkar for dedicating their time to participate in my committee. Their feedback and comments were very helpful in improving the work presented in this dissertation.

Finally, I would like to thank my parents for their continuous encouragement and support during my graduate study.

The research was supported in part by the National Science Foundation under the grants NSF-CCF-0811541, NSF-ECCS-0801763, NSF-ECCS-0926029, and NSF-CCF-1331390.

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. We present an automatic test generation approach for Simulink/Stateflow based on its translation to a formal model, called Input/Output Extended Finite Automata (I/O-EFA), that is amenable to formal analysis such as test generation. The approach automatically identifies a set of input-output sequences to activate all executable computations in the Simulink/Stateflow diagram by applying three different techniques, model checking, constraint solving and reachability reduction & resolution. These tests (input-output sequences) are then used for validation purposes, and the failed versus passed tests are used to localize the fault to plausible Simulink/Stateflow blocks. The translation and test generation approaches are automated and implemented in a toolbox that can be executed in Matlab that interfaces with NuSMV.

CHAPTER 1 Introduction

Simulink/Stateflow [1] is a popular commercial model-based development tool for many industrial domains, such as power systems, aircraft, automotives and chemical plants. Simulink is much better for handling continuous systems, whereas Stateflow is much better for handling state based problems. 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. Owing to the correctness, safety, security, etc. requirements of such systems, methods to analyze the system designs are needed. Since Simulink/Stateflow has originally been designed for the simulation purposes, automated test generation and verification 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. [4] [5] introduced a mutation-based test generation method for Simulink. [6] proposed a transition coverage testing for Simulink/Stateflow using messy genetic algorithm. [7] integrates different test generation techniques to enhance the test coverage of Simulink/Stateflow models. Various commercial tools have been developed for the verification/testing of Simulink/Stateflow. Simulink Design Verifier [8] is a verification/validation toolbox in Matlab for Simulink/Stateflow based on formal analysis. HiLiTE [9] is a requirements-based verification/testing tool for Simulink/Stateflow developed by Honeywell. Reactis [10] and T-VEC [11] 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 this dissertation, we present an automated test generation and verification approach for Simulink/Stateflow. A recursive method is introduced 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. We then discusses the method to generate test cases for the Simulink/Stateflow diagram based on the corresponding translated I/O-EFA. 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. An improved test generation method is also described to use a compact model, analytically solve the computations, and reduce the test generation problem to a reachability problem, so that the technique is effective in terms of achieving test coverage and efficient in terms of test generation time.

Test generation based on the requirements is also discussed by translating the requirements to an equivalent automaton. Test cases are obtained as acyclic executions accepted by the automaton and are applied to test the requirements. Validation methods are proposed to validate the model-based tests against the requirements and the requirements-based tests against the model for "fail/pass". Finally, we develop an error localization approach that uses the failed versus passed tests to locate the errors within the Simulink/Stateflow blocks.

1.1 Existing Tools for Simulink/Stateflow Testing

Some of the prominent commercially available tools for generating test cases from Simulink/Stateflow models are: Simulink Design Verifier (SDV) [8] from Mathworks, Reactis [10] from Reactive Systems Inc., T-VEC [11] from T-VEC Technologies, and HiLiTE [9] from Honeywell International.

Simulink Design Verifier [8] is a tool for Simulink to perform test case generation from and prove model properties of SL/SF models. This tool can also show un-reachability of certain model elements. Simulink Design Verifier can generate test inputs that satisfy standard coverage objectives as well as user-defined test objectives and requirements. These test inputs can also be combined with tests defined using measured data so that simulations are testing against model coverage, requirements, and real-world scenarios. However, many of the Simulink blocks are not supported, such as "integrator"; some design properties cannot be expressed by the tool, such as true liveness properties; and, a pre-defined upper bound of the test case length is required for the test generation.

Reactis tester [10] uses a combination of random testing and guided simulation. It can generate test cases very fast, however, due to the randomness of the search, high coverage is hard to achieve. Besides, this approach is heuristics without explanation and is limited regarding the length of generating input signals, model size and complexity leads to lower structural coverage.

T-VEC [11] generates test cases automatically from the domain testing theory. It produces unit, integration and system level test vectors and test drivers necessary to verify implementations of models. The test selection process produces the set of test vectors in revealing both decision and computational errors in logical, integer and floating-point domains.

HiLiTE [9] is a requirements-based verification/testing tool for Simulink/Stateflow. Each block is specified with certain requirements (predicates over the signals). It uses a data-flow model of the Simulink/Stateflow and can propagate through the data-flow model to find the range for each signal. Model defects are detected and test cases are automatically generated and executed to cover all requirements in the Simulink/Stateflow model. However, it cannot

handle general feedback loops in the Simulink/Stateflow models.

1.2 Dissertation Contributions

The main contributions of the dissertation are the followings.

1. We have developed a recursive method to translate a Stateflow chart into an I/O-EFA that preserves the discrete behaviors. The overall model of a Stateflow chart has the same structure as the model of a Simulink diagram proposed in our previous work, which makes the two models integrable. The translated model shows different paths to represent all the computational sequences, which makes it easier for formal analysis.

2. We have developed a systematic test generation method for Simulink/Stateflow based on I/O-EFA models that representing the computations of a Simulink/Stateflow diagram. Two model-based test implementation techniques, model-checking and constraint solving, are implemented and compared. A requirements-based test generation approach for requirements expressed as safety LTL formula is proposed.

3. We have developed an automated translation and test generation tool within the Matlab environment that is ready for use. The translated I/O-EFA model can itself be simulated in Matlab (by treating it as a "flat" Stateflow model).

4. Test validation method is introduced and error localization approach is applied to locate the error at the Simulink/Stateflow level.

5. We reduce the problem of test generation to that of a reachability problem by introducing the notion of a Computation-Succession automaton that is a discrete-time hybrid automaton where a location is reached if and only if a target computation-path possesses a test case that can eventually enable the computation-path. A reachability resolution procedure is developed to refine the hybrid automata, so that reachability based on the refined hybrid automata is equivalent to reachability in the underlying graph, ignoring the dynamics. When the results of the multiple execution of the computation-paths are

analytically determined, our approach yields a more effective and efficient technique with higher test coverage and faster test generation time. A condition for the termination of the reachability resolution is provided.

1.3 Dissertation Organization

The remainder of the dissertation is organized as follows: Chapter 2 presents the modeling of Simulink/Stateflow (results also presented in [12] [13] [14]); Chapter 3 proposes the test generation approach based on the translated model (results also presented in [15]); Chapter 4 introduces an improved test generation approach based on a more compact model (results also presented in [16]); Chapter 5 describes the test validation and error localization following the test generation; Chapter 6 summarizes the chapters and concludes with suggestions for further research; and Appendix A includes a detailed description of the definition of bisimulation.

CHAPTER 2 Modeling of Simulink/Stateflow

In this chapter, we present the modeling of Simulink/Stateflow. Both Simulink and Stateflow are translated to Input/Output Extended Finite Automata (I/O-EFA), which is a formal model of a reactive untimed infinite state system, amenable to formal analysis. A brief introduction of I/O-EFA is provided. Our previous Simulink translation work [12] done by Changyan Zhou is described and the translation of Stateflow (event-driven blocks) is presented to complete the modeling approach. We have also implemented our Stateflow translation algorithm along with the Simulink translation approach in [12] into an automated translation tool SS2EFA, written in the Matlab script. A counter and a complex motor control system have been used as the case studies for the proposed translation method and the tool. The simulation results show that the translated model simulates correctly the original Simulink diagram at each time step.

2.1 Introduction to I/O-EFA

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,

- Σ is the set of symbolic-inputs,
- Δ 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 σ_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 δ_e is emitted. In what follows below, the data update and output assignments are performed together in a single *action*.

2.2 Review Modeling of Simulink

In [12], a recursive modeling method is introduced to translate Simulink diagram to I/O-EFA. Blocks in the Simulink library are treated to be "atomic" and two rules, connecting-rule and conditioning-rule, are formulated to build complex blocks by combining the simpler ones.

[12] presented algorithms for (i) modeling an atomic- block as an I/O-EFA, (ii) combining the I/O-EFA models of simpler Simulink diagrams to build the I/O- EFA model of a more complex Simulink diagram, constructed using certain rules of composition. [12] introduced the concept of a step (resp., step-trajectory) of an I/O-EFA to emulate the computation of a Simulink diagram at a sample time (resp., over a sequence of 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 $e = (l_m, l_0, -, -, -, -, \{k := k + 1\})$ 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) π 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 $\pi \in \{e_0^\pi ... e_{|\pi|-1}^\pi \in E^* | o_{e_0^\pi}, t_{e_{|\pi|-1}^\pi} \in L_0, h_{e_{|\pi|-1}^\pi} \equiv \{k := k + 1\}, \forall i \in [1, |\pi| - 1] : o_{e_i^\pi} = t_{e_{i-1}^\pi}\}$.

Example 1 Consider the Simulink diagram Ψ of a bounded counter shown in Figure 2.1, 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 $y5$ resets to its initial value when the control input u is not positive. The saturation block limits the value of $y5$ in the range between -0.5 and 7. The translated I/O-EFA P using the method of [12] is shown in Figure 2.2. 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 Ψ in which the input is zero so that the subsystem is disabled and its output remains as the initial level 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 Ψ.

[12] showed that the modeling approach is sound and complete: The input-output behavior of an I/O-EFA model, as defined in terms of a step- trajectory, preserves the input-output

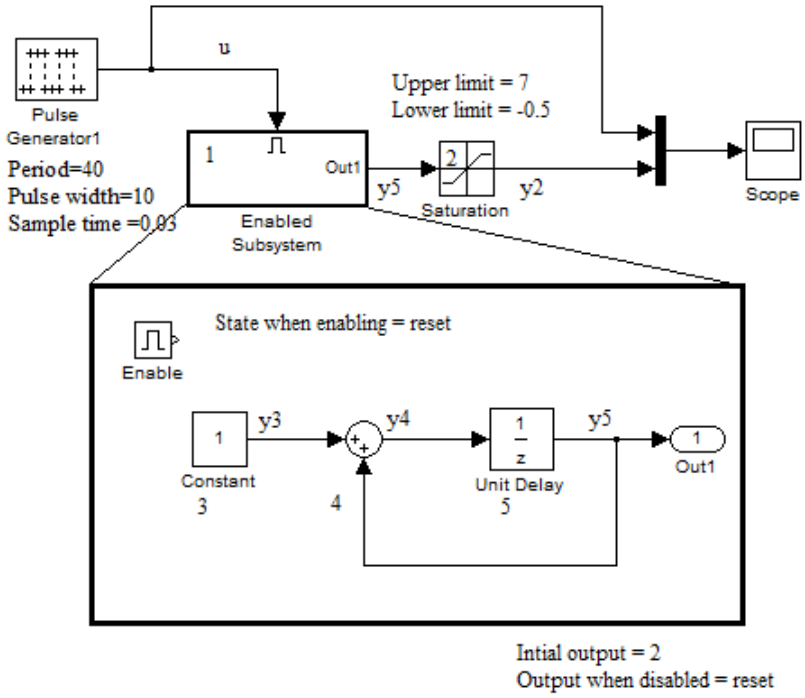

Figure 2.1 Simulink Diagram of a Counter System

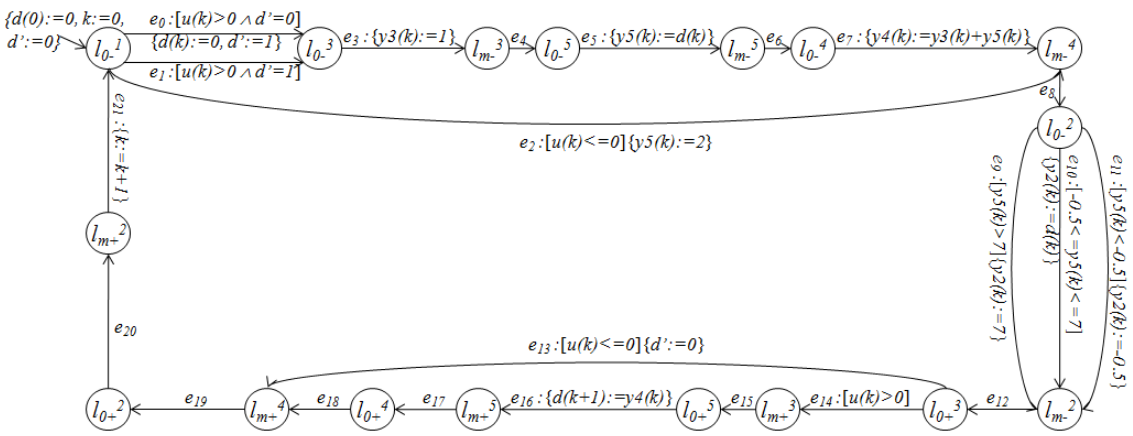

Figure 2.2 I/O-EFA of a Counter System

behavior of the corresponding Simulink diagram at each sample time. Also due to the way the models are composed to obtain the more complex models from the simpler ones, the approach avoids any state-space explosion.

2.3 Modeling of Stateflow

For the modeling of Stateflow, we continue to use I/O-EFA as the target model for translation so as to retain consistency with the modeling of the Simulink diagrams. In order to have our modeling process recursive, we treat the individual states of a Stateflow chart to be the most elementary constructs for modeling, and define the atomic models for them. Next, two composition rules are defined to interconnect the simpler models to form the more complex models for the "AND" versus "OR" states, preserving their state execution and transition behaviors. By viewing the Stateflow chart's hierarchical structure as a tree, we recursively apply the two composition rules in a bottom-up algorithm to obtain the overall I/O-EFA model. Finally, the additional Stateflow features, such as event broadcasting and interlevel transitions, are incorporated by refining the model at locations where the features reside. Furthermore, a composition rule between Stateflow and Simulink models is introduced to combine them into a single complete model.

2.3.1 Atomic Model for States

States are the most basic components in a Stateflow chart in that a simplest Stateflow chart can just be a single state. Stateflow allows states to be organized hierarchically by allowing states to possess substates, and same holds for substates. A state that is down (resp., up) one step in the hierarchy is termed a substate (resp., superstate). We represent the most basic components of a Stateflow chart as atomic models, which are the smallest modules that are interconnected (following the semantics of the hierarchy and other Stateflow features as described in the following sections) to build the model of an overall Stateflow chart.

Consider a Simulink diagram of a counter system shown in Figure 2.3. The counter itself is a Stateflow chart, which gets the input from the signal source "pulse generator" to switch

between the "count" and the "stop" mode. The saturation is a Simulink block that sets the lower and upper bounds for the output values. The Stateflow chart consists of six states with two parallel top-level states and two exclusive substates for each of them. This hierarchical structure of the states is shown in a tree format in Figure 2.4. Each node of the tree can be modeled as an atomic I/O-EFA model, which we describe below in this section.

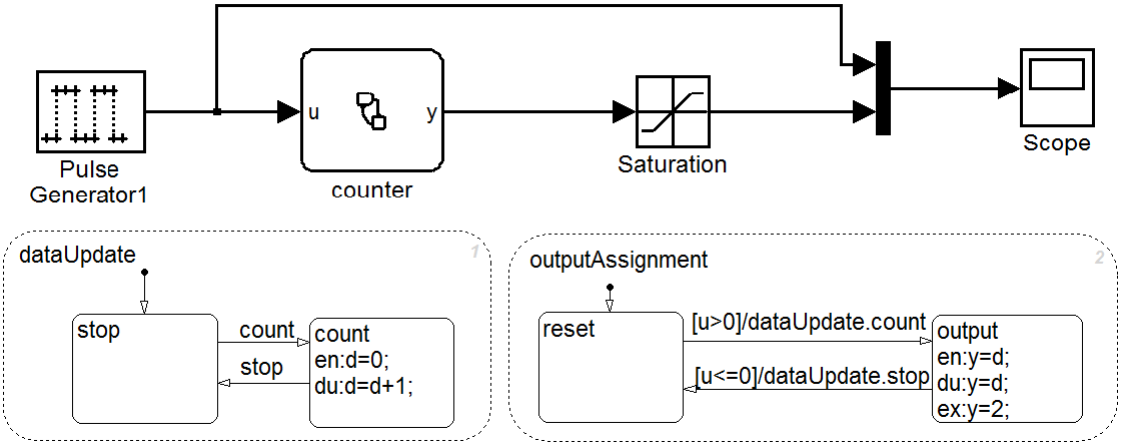

Figure 2.3 Simulink diagram of a Counter system (top) and the Stateflow chart of the counter (below)

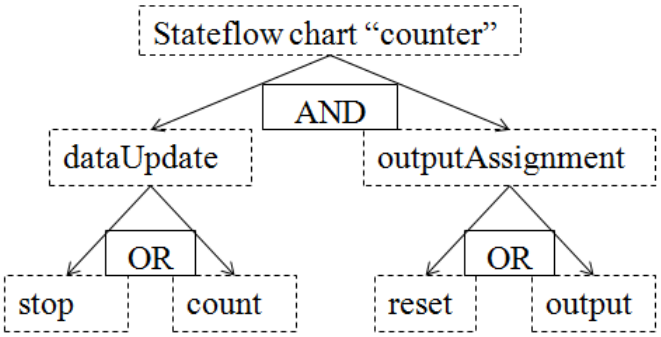

Figure 2.4 Hierarchical Structure of Conuter's Stateflow chart

The behavior of a Stateflow state comprises of three phases: entering, executing and exiting, where

- Entering phase marks the state as active and next performs all the entry actions;
- Executing phase evaluates the entire set of outgoing transitions. If no outgoing transition is enabled, the during actions, along with the enabled on-event actions, are performed;
- Exiting phase performs all the exit actions and marks the state inactive.

According to the above behaviors, an individual state s of a Stateflow can be represented in the form of an I/O-EFA of Figure 2.5.

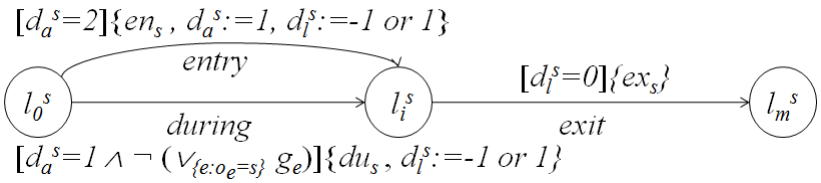

Figure 2.5 Atomic Model for a Stateflow state

As can be seen from the figure, the atomic I/O-EFA model has three locations l_0^s, l_i^s, l_m^s for differentiating the activation versus the deactivation process, where the transition

- $l_0^s \rightarrow l_i^s$ captures the activation process, including the state entry and during actions, and the transition
- $l_i^s \rightarrow l_m^s$ captures the deactivation process, including the state exit actions and the transitions to higher/lower level.

The atomic model has internal data-variables d_a^s, d_l^s, $\{d^e | o_e = s\}$ for controlling the execution flow, where

- d_a^s is to determine if the particular state is inactive/active/newly active as captured by the three values (0/1/2),
- d_l^s is to determine the direction of flow in the hierarchy: down/same/up as captured by the three values (-1/0/1), and
- d^e is to determine if the outgoing transition e is active or not (0/1).

A formal description of this atomic model is given in the following algorithm.

Algorithm 1 A Stateflow state s can be represented as an I/O-EFA $(L^s, D^s, -, -, -, -, \{l_0^s, l_i^s\}, D_0^s, \{l_i^s, l_m^s\}, E^s)$, where

- $L^s = \{l_0^s, l_i^s, l_m^s\}$,
- D^s is the set of data variables consisting of $\{d_a^s, d_l^s\} \cup \{d^e | o_e = s\}$,
- D_0^s is the set of initial data values, and

- $E^s = \{l_0^s, l_i^s, [d_a^s = 2], \{en_s, d_a^s := 1, d_l^s := -1 \text{ (if } s \text{ has a substate) or 1 (otherwise)}\}\}$

  $\bigcup\{l_0^s, l_i^s, [d_a^s = 1 \wedge \neg(\bigvee_{\{e:o_e=s\}} g_e)], \{du_s, d_l^s := -1 \text{ (if } s \text{ has a substate) or 1 (otherwise)}\}\}$

  $\bigcup\{l_i^s, l_m^s, [d_l^s = 0], \{ex_s\}\}$, where

  - en_s is the entry actions of s,
  - du_s is the during actions of s,
  - ex_s is the exit actions of s, and
  - g_e is the guard of the transition e.

The above atomic model captures a Stateflow state's behavior as follows:

• When the Stateflow state s is newly activated and the location is transitioned to l_0^s, d_s^a is set to 2 (as described later). Accordingly, initially the transition labeled "entry" is enabled, and the state entry action en_s is executed, and also d_a^s is set to 1 to notate that the state has already been activated; d_l^s is set to -1 (if s has a substate and so the execution flow should be downward into the hierarchy to a substate) or 1 (if s has no substate and so the execution flow can only be upward to a superstate) to indicate that the state has finished executing in this time step, and execution flow should go to another state;

• Once the state has been activated, d_a^s equals 1 and upon arrival at l_0^s if none of the outgoing transitions is enabled, the transition labeled "during" is executed causing the state during action du_s to be executed; d_a^s remains unchanged since the state is still in the execution phase; d_l^s is set to -1 or 1 as described above in the previous bullet;

• When leaving the state, d_l^s is set to 0 (as discussed later), so upon arrival to l_i^s the transition labeled "exit" is executed, causing the execution of the state exit action ex_s.

Example 2 Consider the counter system of Figure 2.3. Figure 2.6 shows an example of the atomic model for the bottom-level state "outputAssignment.output" in the Stateflow chart of the counter. The entry action, during action, and exit action are represented by three edges in the I/O-EFA model following Algorithm 1.

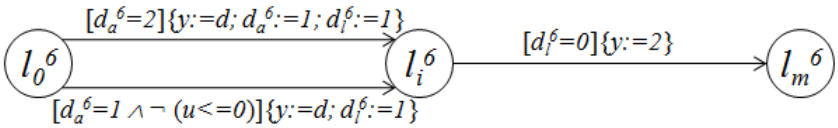

Figure 2.6 Atomic Model for State outputAssignment.output in the Counter

2.3.2 Modeling State Hierarchy

Stateflow provides for hierarchical modeling of discrete behaviors by allowing a state to possess substates which can be organized into a tree structure. The root node of the tree is the Stateflow chart, the internal nodes are the substates of the Stateflow chart, and the leaves are the bottom-level states with no substates of their own. As described in the previous section, each state, which is a node of the tree, is modeled as an atomic model of the type shown in Figure 2.5. The next step in the modeling is to connect these atomic models according to the type (AND vs. OR) of the children nodes.

In case of AND substates, all substates must be active simultaneously and must execute according to their execution order at each time step, whereas in case of OR substates, at most one of the substates can be active at each time step, and one of the substates is deemed default substate which gets activated at the first time step its superstate becomes active. For the execution order of a state with substates, two rules must be followed: 1) The substates can be executed only when their superstate is activated, and 2) A state finishes execution only when all its substates have been evaluated for execution.

After the execution of a transition labeled “entry” or “during” of a state, all its outgoing transitions are evaluated for enablement (if no outgoing transition is enabled, another execution of “during” action is performed). The enabled transition with the highest priority is selected for execution, and the particular transition is activated. Also the exit phase of the state is initiated. Exit phase generally has the following execution sequence: The condition action of the activated transition, the exit actions of the leaving state, the transition action of the activated transition and the entry action of the entering state. Furthermore, if there are multiple exit actions to be executed (i.e. the leaving state has substates), then those are

ordered according to the following rule: The leaving state, along with all its substates, exits by starting from the last-entered state's exit action, and progressing in reverse order to the first-entered state's exit action.

With the above knowledge of the semantics of the AND/OR hierarchy, we can now model the hierarchical behaviors by defining the corresponding composition rules. We first introduce a few notation to make the presentation clearer.

Definition 3 A complex state $\widehat{s}$ is the state system consisting of the state s and all its immediate substates. $\widehat{s}$ is said to be an AND- (resp., OR-) complex state if it possesses AND (resp., OR) substates. We define $|\widehat{s}|$ to indicate the number of substates in the complex state $\widehat{s}$.

Following the state transition semantics, the modeling rule for a state with OR-substates can be defined by the following algorithm. For an OR-complex state $\widehat{s}$ we use s^* to denote its default state.

Algorithm 2 An OR-complex state $\widehat{s}$ can be represented as an I/O-EFA $(L^{\widehat{s}}, D^{\widehat{s}}, -, -, -, -, \{l_0^s, l_i^s\}, D_0^{\widehat{s}}, \{l_i^s, l_m^s\}, E^{\widehat{s}})$, where

- $L^{\widehat{s}} = \bigcup_{s\in\widehat{s}} L^s$,
- $D^{\widehat{s}} = \prod_{s\in\widehat{s}} D^s$,
- $D_0^{\widehat{s}} = \prod_{s\in\widehat{s}} D_0^s$,
- $E^{\widehat{s}} = E \bigcup_{s\in\widehat{s}} E^s$, where E is all newly introduced edges as shown in Figure 2.7:

  $\bigcup_{r\in\widehat{s}-\{s\}}\{l_i^s, l_0^r, [d_l^s = -1 \wedge d_a^r > 0], -\}$

  $\bigcup\{l_i^s, l_i^s, [d_l^s = -1 \wedge (\bigwedge_{r\in\widehat{s}-\{s\}}(d_a^r = 0)], \{d_a^{s^*} := 2\}\}$

  $\bigcup_{r\in\widehat{s}-\{s\}}\{l_0^r, l_i^r, [d_a^s = 0], \{d_a^r := 0; d_l^r := -1$ (if s has a substate) or 0 (otherwise)$\}\}$

  $\bigcup_{r\in\widehat{s}-\{s\},\{e:o_e=r\}}\{(l_0^r, l_i^r, [d_a^r = 1 \wedge g_e \wedge d_a^s > 0], \{ca_e; d_a^r := 0; d^e := 1; d_l^r := -1$ (if s has a substate) or 0 (otherwise)$\}\}$

  $\bigcup_{r\in\widehat{s}-\{s\}}\{l_i^r, l_i^s, [d_l^r = 1], \{d_l^s := 1\}\}$

  $\bigcup_{r\in\widehat{s}-\{s\}}\{l_m^r, l_i^s, [d_a^s = 0], \{d_l^s := 0\}\}$

  $\bigcup_{r\in\widehat{s}-\{s\},\{e:o_e=r\}}\{l_m^r, l_i^s, [d_a^s > 0 \wedge d^e = 1], \{ta_e; d_a^{t_e} := 2; d^e := 0; d_l^s := -1\}\}$, where

- g_e is guard condition of the transition e,
- ca_e is condition action of the transition e,
- ta_e is transition action of the transition e, and
- o_e (resp., t_e) is the origin (resp., terminal) state of edge e.

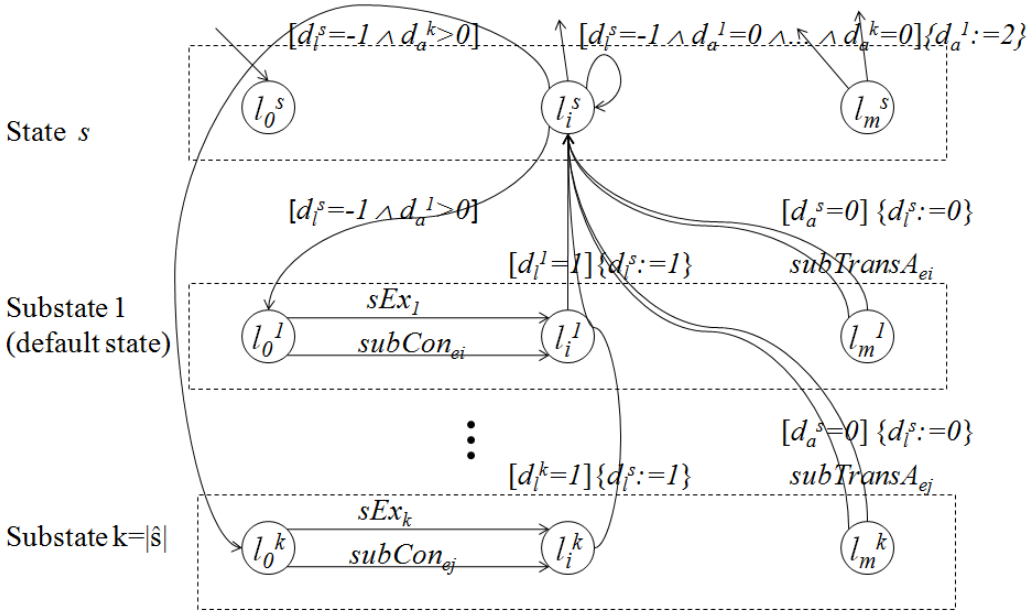

Figure 2.7 OR-Complex state modeling. $\forall r \in \widehat{s} - \{s\}$: $sEx_r \equiv [d_a^s = 0]\{d_a^r := 0; d_l^r := -1 or 0\}$; $\forall e : o_e \in \widehat{s} - \{s\}$: $subCon_e \equiv [d_a^{o_e} = 1 \wedge g_e \wedge d_a^s > 0]\{ca_e; d_a^{o_e} := 0; d^e := 1; d_l^{o_e} := -1 or 0\}$, and $subTransA_e \equiv [d_a^s > 0 \wedge d^e = 1]\{ta_e; d_a^{t_e} := 2; d^e := 0; d_l^s := -1\}$

For an AND-complex state $\widehat{s}$, its substates, although simultaneously active, are executed in a certain order. With a slight abuse of notation we use r to denote the substate whose execution order is r among all the substates of $\widehat{s}$. Also for simplicity of notation let $k = |\widehat{s}|$. The modeling rule for a state with AND-substates is defined as follows.

Algorithm 3 An AND-complex state $\widehat{s}$ can be represented as an I/O-EFA $(L^{\widehat{s}}, D^{\widehat{s}}, -, -, -, -, \{l_0^s, l_i^s\}, D_0^{\widehat{s}}, \{l_i^s, l_m^s\}, E^{\widehat{s}})$, where

- $L^{\widehat{s}} = \bigcup_{s \in \widehat{s}} L^s$,
- $D^{\widehat{s}} = \prod_{s \in \widehat{s}} D^s$,

- $D_0^{\widehat{s}} = \prod_{s \in \widehat{s}} D_0^s$,
- $E^{\widehat{s}} = E \bigcup_{s \in \widehat{s}} E^s$, where E is all newly introduced edges as shown in Figure 2.8:

  $\{l_i^s, l_0^1, [d_l^s = -1 \wedge d_a^s > 0 \wedge d_a^1 > 0], -\}$

  $\bigcup \{l_i^s, l_i^s, [d_l^s = -1 \wedge d_a^1 = 0], \{\forall r \in \widehat{s} - \{s\} : d_a^r := 2\}\}$

  $\bigcup \{l_i^k, l_i^s, [d_l^k = 1], \{d_l^s := 1\}\}$

  $\bigcup_{r \in \widehat{s} - \{s,k\}} \{(l_i^r, l_0^{r+1}, [d_l^r = 1], -\}$

  $\bigcup_{r \in \widehat{s} - \{s\}} \{l_0^r, l_i^r, [d_a^r = 1 \wedge d_a^s = 0], \{d_a^r := 0; d_l^r := 1$ (if s has a substate) or 0 (otherwise)$\}\}$

  $\bigcup \{l_i^s, l_i^k, [d_l^s = -1 \wedge d_a^s = 0 \wedge d_a^k > 0], -\}$

  $\bigcup_{r \in \widehat{s} - \{s,1\}} \{l_m^r, l_0^{r-1}, -, -\}$

  $\bigcup \{l_m^1, l_i^s, -, \{d_l^s := 0\})\}$.

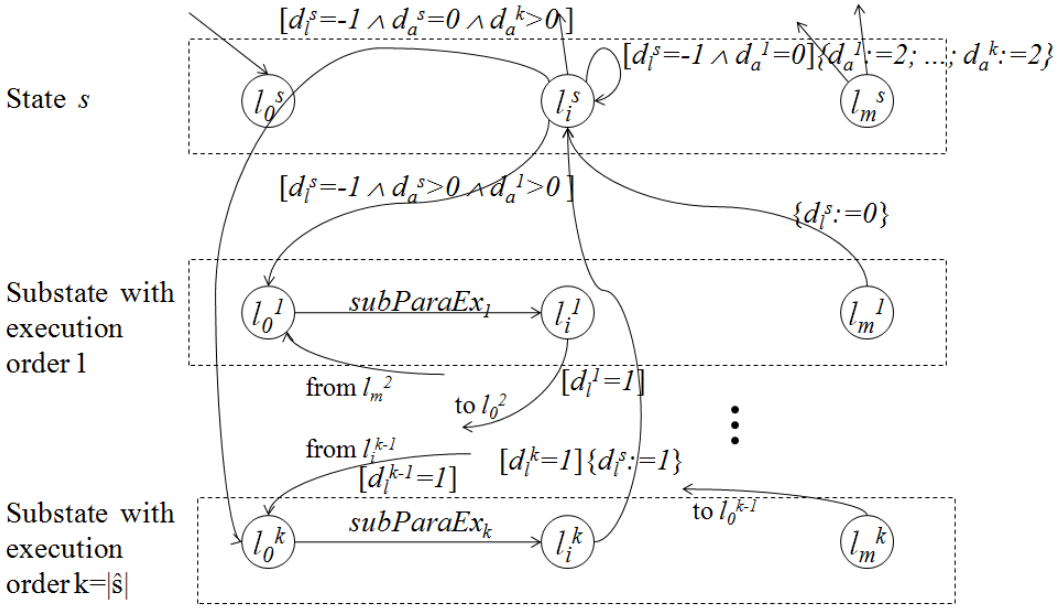

Figure 2.8 AND-Complex state modeling. $\forall r \in \widehat{s} - \{s\}$: $subParaEx_r \equiv [d_a^r = 1 \wedge d_a^s = 0]\{d_a^r := 0; d_l^r := -1 or 0\}$

With the above two composition rules, an overall model of a Stateflow chart, capturing only the state hierarchy feature, can be obtained by applying the rules recursively, over the tree structure of the state hierarchy, in a bottom-up fashion.

Example 3 Consider the counter system of Figure 2.3. We start from the two bottom level OR-substates "dataUpdate.stop" and "dataUpdate.count", and compose them using the OR-state connecting rule (Algorithm 2) to obtain the OR-complex state "dataUpdate" and "outputAssignment". The results are shown in Figure 2.9 and Figure 2.10 respectively. Next a model for the top-level AND-complex state (the Stateflow chart) is obtained by composing the models of Figures 2.9 and 2.10 using the AND-Connecting Rule (Algorithm 3); the result is shown in Figure 2.11.

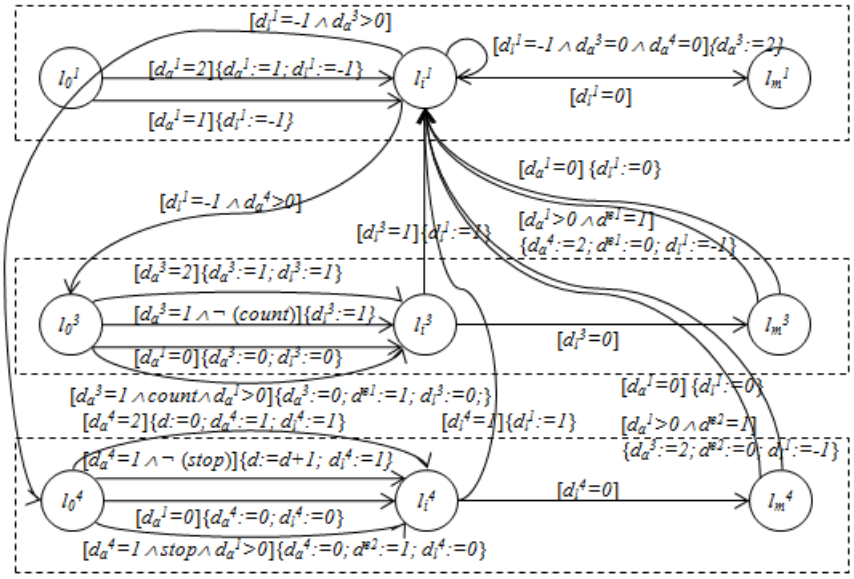

Figure 2.9 Modeling of OR-Complex state dataUpdate within Statechart of Counter

2.3.3 Model Refinement for Other Features

Besides the state hierarchy, Stateflow provides many additional features, such as events, historical node and interlevel transitions. We capture these features into our model by refining the I/O-EFA model obtained by recursively applying Algorithms 1-3. We illustrate the model refinement by modeling one of the important features of Stateflow, namely a local event which is a commonly used event type.

A local event is triggered at a certain source state as part of one of the actions, where along with the event name, the destination states for the event broadcast are also specified. When

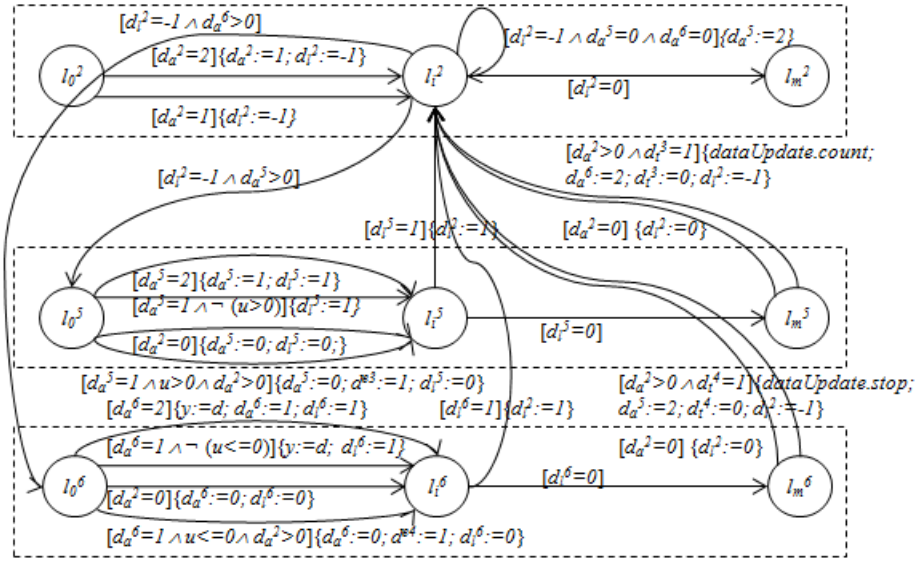

Figure 2.10 Modeling of OR-Complex state outputAssignment within Statechart of Counter

an event is triggered, it is immediately broadcast to its destination state for evaluation. At this point, the destination state, including all of its substates, is executed by treating the event condition to be true in all of the guard conditions where it appears. Then the execution flow returns to the breakpoint where the event was triggered and resumes the execution.

The Stateflow event semantics permits an infinite chaining of events since each event can cause an action in its destination state that triggers a new or the same event. Such recursive behavior cannot be captured in the I/O-EFA modeling framework. However, practical systems avoid infinite chaining of events by way of satisfying the following requirements [3], which we assume to hold:

- Local events can be sent only to parallel states,
- Transitions out of parallel states are forbidden,
- Loops in broadcasting of events are forbidden, and
- Local events can be sent only to already-visited states.

For local event ev that is triggered in some source state src of a Stateflow chart, let $e^{ev} \in E$ be an edge that broadcasts the event ev. The model refinement step for modeling the local

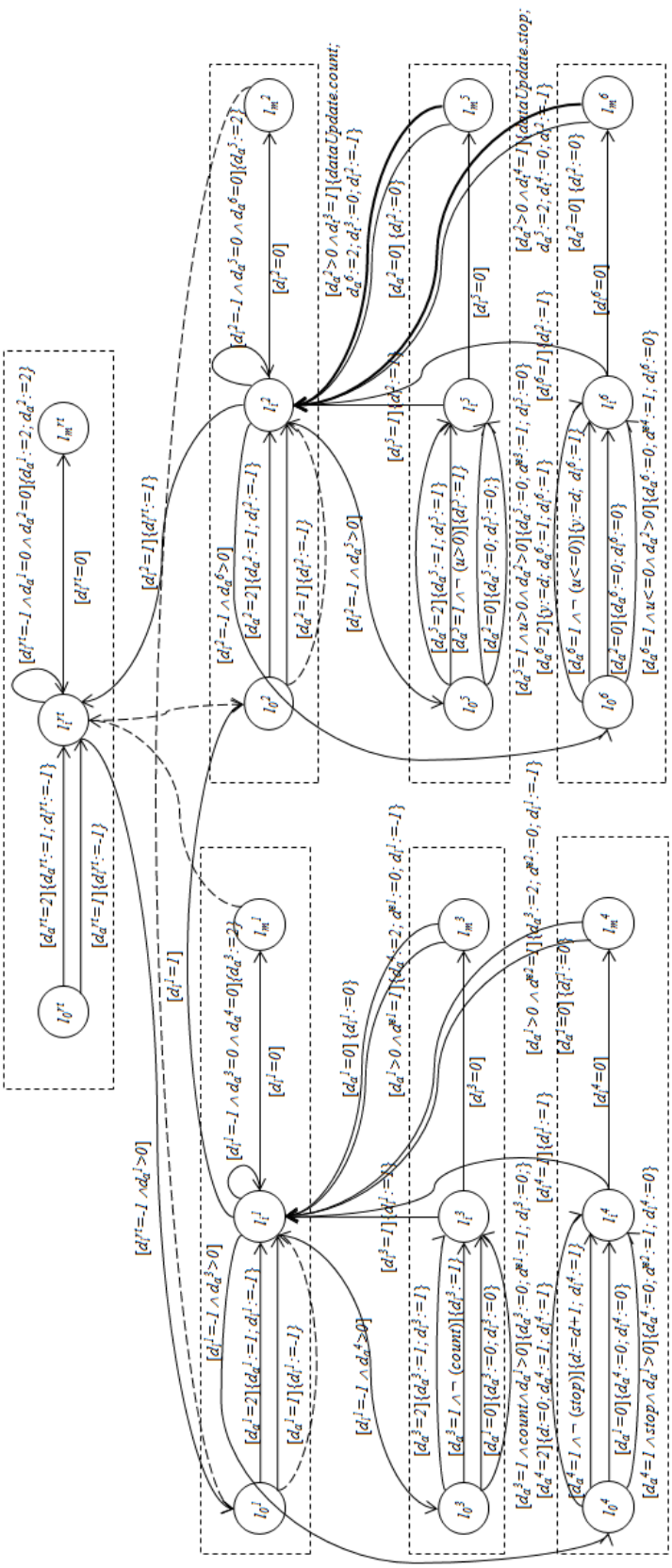

Figure 2.11 Modeling state hierarchy within Statechart of Counter; certain places are missing spacings in Fig 2.7 and 2.8; the following edges, whose guards are never satisfied, are drawn as dotted lines and their labels are omitted for simplicity: $\{l_i^{rt}, l_0^2, [d_l^{rt} = -1 \wedge d_a^{rt} = 0 \wedge d_a^2 > 0], -\}$, $\{l_0^2, l_i^2, [d_a^2 = 1 \wedge d_a^{rt} = 0], \{d_a^2 := 0; d_l^2 := -1\}\}$, $\{l_0^2, l_i^2, [d_a^2 = 1 \wedge d_a^{rt} = 0], \{d_a^2 := 0; d_l^2 := -1\}\}$, $\{l_m^1, l_i^{rt}, -, \{d_l^{rt} := 0\}\}$

event behavior requires replacing the event-triggering edge e^{ev} with a pair of edges between the event source state src and the event destination state des, one in each direction (see Figure 2.12 for illustration). Also letting E^{ev} denote the set of edges in the destination state's I/O-EFA model where the event ev is received, then for each edge $e \in E^{ev}$, its event label $\sigma_e (= ev)$ is replaced by the guard condition $[d^{ev} = 1]$, where the binary variable d^{ev} captures whether or not the event ev has been triggered.

Algorithm 4 Given an I/O-EFA model $(L, D, -, -, -, -, L_0, D_0, L_m, E)$ obtained from recursive application of Algorithms 1-3, an edge $e^{ev} \in E$ that broadcasts an event ev to the destination state des, and a set of edges E^{ev} in the destination state that receive the event (i.e., $\forall e \in E^{ev} : \sigma_e = ev$), the refined I/O-EFA model is given by $(L, D, -, -, -, -, L_0, D_0, L_m, E')$, where

- $E' = [E|_{\{\sigma_e \to [d^{ev}=1] | e \in E^{ev}\}} - \{e^{ev}\}]$

  $\bigcup \{o_{e^{ev}}, l_0^{des}, -, \{PreEventAction, d^{ev} := 1\}, -\}$

  $\bigcup \{l_i^{des}, t_{e^{ev}}, [d_l^{des} = 1 \wedge d^{ev} = 1], \{d^{ev} := 0; PostEventAction\}\}$,

  where $PreEventAction$ (resp., $PostEventAction$) denotes all the guard conditions and actions appearing on the event-triggering edge prior to (resp., after) the event-broadcast label, and $E|_{\{\sigma_e \to [d^{ev}=1] | e \in E_o\}}$ is the set of edges obtained by replacing the event label $\sigma_e (= ev)$ of each edge $e \in E^{ev}$ by the guard condition $[d^{ev} = 1]$ (no relabeling is done for the remaining edges in $E - E^{ev}$)..

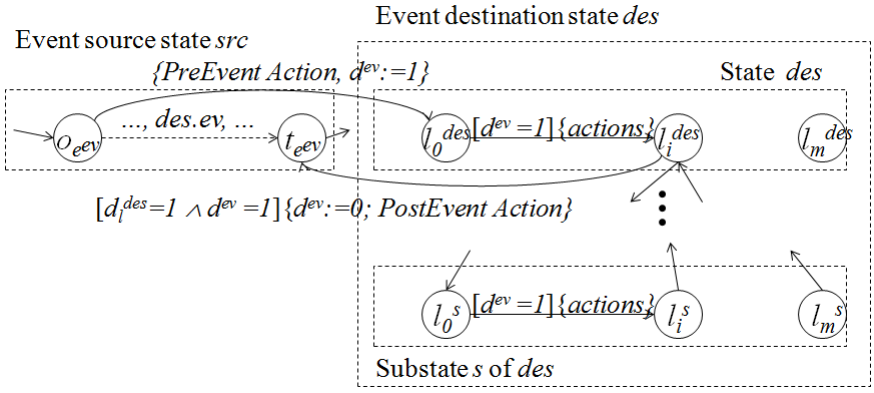

Figure 2.12 Modifying the model capturing state hierarchy to also model local events

Recursive application of Algorithm 4 with respect to each event-triggering edge is required to complete the model-refinement step for modeling the local events. Additional features such

as historical node and interlevel transitions can also be modeled by similar refinement steps, conforming to their respective Stateflow semantics. Due to space limitations, those are not included.

Example 4 Consider the counter system of Figure 2.3. There are two local events "count" and "stop" in the "outputAssignment" state with the destination "dataUpdate" state. Following Algorithm 4, the two edges of Figure 2.11 labeled by the events "count" and "stop" respectively, and shown in bold in Figure 2.11, are broken down into two parts with corresponding parts re-routed as shown in Figure 2.13, where the re-routed edges are shown in bold and the original edges are drawn in dotted lines.

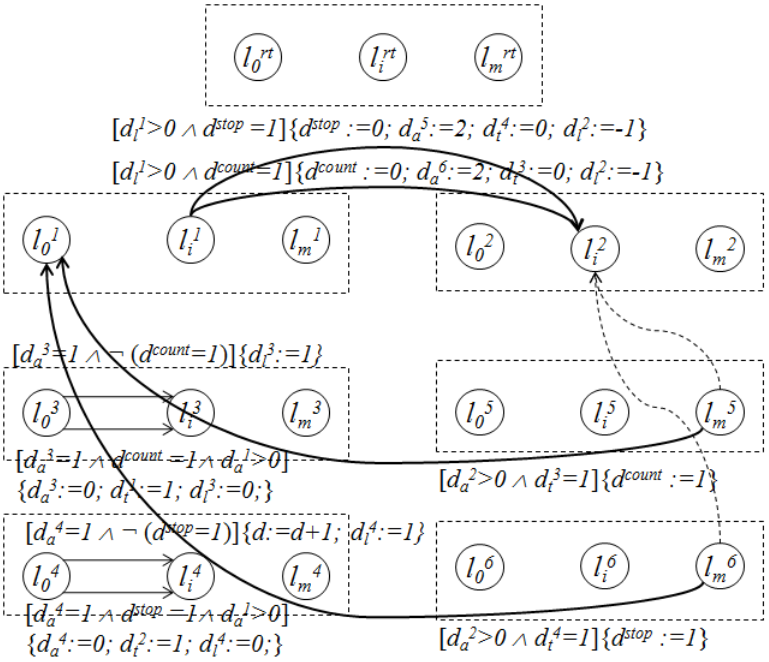

Figure 2.13 Modification of Fig 2.11 to capture the local events within Statechart of Counter

2.3.4 Final Touches: Finalizing the Model

At the top level, a Stateflow chart is also a Simulink block (the distinction being that it is event-driven as opposed to time-driven). So to be consistent, at the very top level, the model of a Stateflow chart ought to resemble the model of a time-driven Simulink block as introduced

in [12]. Accordingly, the model of a Stateflow chart obtained from applying Algorithms 1-4 is adapted by adding another layer as shown in Figure 2.14. As is the case with the model of a time-driven Simulink block (see [12] for the details), the final model of a Stateflow chart is composed of two I/O-EFA parts that are linked by a "succession edge" (connects the final location of 1st I/O-EFA to initial location of 2nd I/O-EFA) and a "time-advancement edge" (connects the final location of 2nd I/O-EFA to initial location of 1st I/O-EFA and increments a time-step counter k). The final model of a Stateflow chart is obtained as follows.

Algorithm 5 Given an I/O-EFA model $(L, D, -, -, -, -, L_0, D_0, -, E)$ obtained from recursive application of Algorithms 1-4 and model refinement concerning other features, the final I/O-EFA model P^{ϕ} of a Stateflow chart ϕ is composed of two I/O-EFAs connected through succession and time-advancement edges as in Figure 2.14.

- The 1st I/O-EFA model is given by $(L_-, D_-, U_-, Y_-, -, -, \{l_{0-}\}, D_{0-}, \{l_{m-}\}, E_-)$, where
  - $L_- = L \cup \{l_{0-}, l_{m-}\}$,
  - $D_- = D$,
  - $D_{0-} = D_0$, and
  - $E_- = \{l_{0-}, l_0^{rt}, -, \{d_a^{rt} := 2, \}\}$
    $\bigcup \{l_i^{rt}, l_{m-}, [d_l^{rt} = 1]\}$.
- The 2nd I/O-EFA model is given by $(L_+, -, U_+, Y_+, -, -, \{l_{0+}\}, -, \{l_{m+}\}, E_+)$, where
  - $L_+ = \{l_{0+}, l_{m+}\}$, and
  - $E_+ = \{l_{0+}, l_{m+}, -, -\}$.

Figure 2.14 depicts the final model of a Stateflow chart, ready to be integrated with models of other components. The 1st I/O-EFA model goes down the state hierarchy to perform the Stateflow computations. When the Stateflow computations of the current time step are completed, the first I/O-EFA model returns to its final location in the top layer. The 2nd I/O-EFA model is vacuous and is only included to retain consistency with the Simulink model.

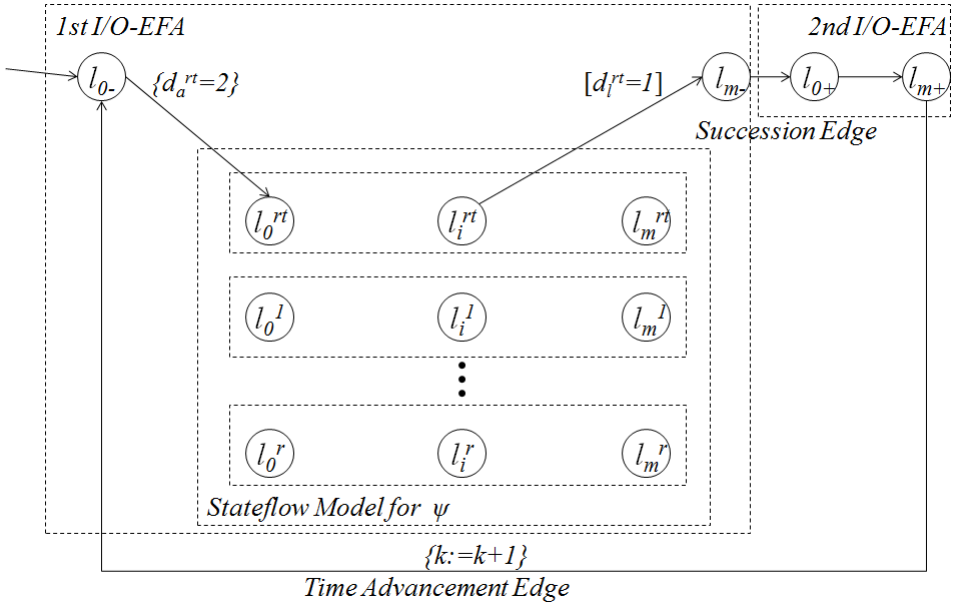

Figure 2.14 Finalized model of Stateflow chart

Example 5 Consider the counter system of Figure 2.3. It can be translated into an I/O-EFA model as follows:

1. Modeling states: We first construct atomic model for each of the seven states (including the Stateflow root) of Figure 2.4.

2. Modeling state hierarchy: We apply OR Complex State Composition Rule (Algorithm 2) on the models obtained in step 1 of the two bottom-level OR complex states, and AND Complex State Composition Rule (Algorithm 3) on models obtained in step 1 of the top-level AND complex state. The result is as shown in Figure 2.11.

3. Modeling local events: Each edge in the model obtained in step 2 containing the event "count" or "stop" is replaced with a pair of edges to connect the source state "outputAssignment" and the destination state "dataUpdate", in either direction. At the same time the evaluation of "count" (resp., "stop") is modified to $d^{count} = 1$ (resp., $d^{stop} = 1$). The result is as shown in Figure 2.13.

4. Obtaining final model: The model obtained in step 3 is augmented by applying Algorithm 5 to obtain a final model (this step introduces four extra locations, and a few extra edges as shown in Figure 2.14. Finally, the I/O-EFA model for Stateflow chart (of counter) is combined with the I/O-EFA model of Simulink block (of saturation) using the connecting rule

introduced in [12]. The result is shown in Figure 2.15.

2.3.5 Correctness of Stateflow Modeling

In order to show that the I/O-EFA model preserves the Stateflow discrete behaviors, we introduce the concept of a feasible c-path (c-path is defined in Definition 2) in the I/O-EFA model of a Stateflow chart. We say that a c-path is *feasible* if it can be enabled under some input(s) and initial data value(s).

Given an I/O-EFA model, its behavior is defined by its set of feasible c-paths. We show that an I/O-EFA model at a sampling time correctly models the discrete behaviors of the corresponding Stateflow chart at the same sampling time. In the correctness proof below, we only provide a sketch of the proof-steps; the details are intentionally omitted, as those are notationally cumbersome and do not add any extra insight.

Lemma 1 Given a Stateflow state s, its discrete behavior is correctly modeled by its I/O-EFA model, that is there is one-to-one mapping between each of s's discrete behaviors and the feasible c-paths in its I/O-EFA model.

Proof: The possible discrete behaviors of a Stateflow state consists of "entering phase", "during phase", and "exit phase". From Algorithm 1 (also refer to Figure 2.5),

- entering phase is represented by the c-path: $\{l_0^s, l_i^s, [d_a^s = 2], \{en_s, d_a^s := 1, d_l^s := -1 \text{ or } 1\}\}$, and
- during phase is represented by the c-path: $\{l_0^s, l_i^s, [d_a^s = 1 \wedge \neg(\bigvee_{\{e:o_e=s\}} g_e)], \{du_s, d_l^s := -1 \text{ or } 1\}\}$, and
- exit phase is represented by the c-path: $\{l_i^s, l_m^s, [d_l^s = 0], \{ex_s\}\}$.

Each of the three discrete behaviors of a Stateflow state map one-to-one to the above c-paths. ■

For simplicity in the following proofs the c-paths (sequences of edges) are represented as sequences of locations. If there is unique edge between the consecutive locations, the location

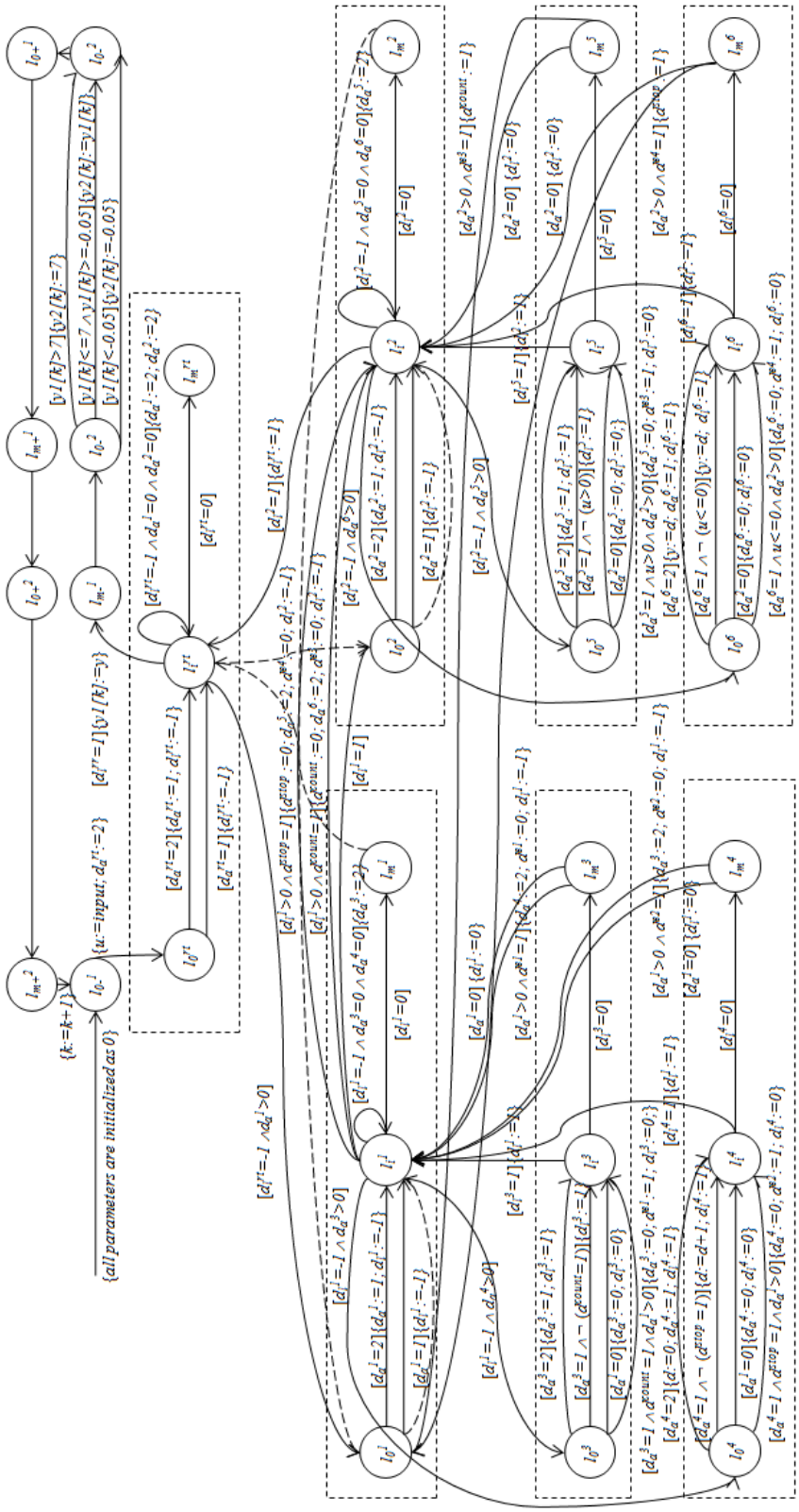

Figure 2.15 Finalized complete model of the Statechart Counter for counter system of Figure 2.3. The following edges, whose guards are never satisfied, are drawn as dotted lines and their labels are omitted for simplicity: $\{l_i^t, l_0^2, [d_l^t = -1 \wedge d_a^t > 0], -\}$, $\{l_0^2, l_i^2, [d_a^2 = 1 \wedge d_a^{rt} = 0], \{d_a^2 := 0; d_l^2 := -1\}\}$, $\{\{l_m^2, l_0^1, -, -\}, \{l_0^1, l_i^1, [d_a^1 = 1 \wedge d_a^{rt} = 0], \{d_a^1 := 0; d_l^1 := -1\}\}$, and $\{l_m^1, l_i^{rt}, -, \{d_l^{rt} := 0\}\}$

pair represents the unique edge. If there are multiple edges between the consecutive locations, the location pair represents the edge which is feasible.

Lemma 2 Given an complex state $\widehat{s}$, its discrete behaviors are correctly modeled by its I/O-EFA model.

Proof: First we prove the state transition behavior of a complex state $\widehat{s}$ is correctly modeled by its I/O-EFA model, that is each discrete behavior of $\widehat{s}$ is mapped one-to-one to a feasible c-path in its I/O-EFA model.

(i) If the complex state is an OR-complex state, from Algorithm 2 (also refer to Figure 2.7),

- entering default substate 1 is represented by the c-path: $l_0^s \rightarrow l_i^s \rightarrow l_i^s \rightarrow l_0^1 \rightarrow l_i^1 \rightarrow l_i^s$, and
- entering active substate s_i is represented by the c-path: $l_0^s \rightarrow l_i^s \rightarrow l_0^{s_i} \rightarrow l_i^{s_i} \rightarrow l_i^s$, and
- leaving substate s_i is represented by the c-path: $l_0^s \rightarrow l_i^s \rightarrow l_0^{s_i} \rightarrow l_i^{s_i} \rightarrow l_m^{s_i} \rightarrow l_i^s$, and
- switching from substate s_i to substate s_j is represented by the c-path: $l_0^s \rightarrow l_i^s \rightarrow l_0^{s_i} \rightarrow l_i^{s_i} \rightarrow l_m^{s_i} \rightarrow l_i^s \rightarrow l_0^{s_j} \rightarrow l_i^{s_j} \rightarrow l_i^s$, and
- leaving OR-complex state $\widehat{s}$ with active substate s_i is represented by the c-path: $l_0^s \rightarrow l_i^s \rightarrow l_0^{s_i} \rightarrow l_i^{s_i} \rightarrow l_m^{s_i} \rightarrow l_i^s \rightarrow l_m^s$.

Also note all the above c-paths possess the correct sequence of guards and updates (details omitted).

(ii) If the complex state is an AND-complex state, from Algorithm 3 (also refer to Figure 2.8),

- entering substates is represented by the c-path: $l_0^s \rightarrow l_i^s \rightarrow l_0^1 \rightarrow l_i^1 \rightarrow l_0^2 \rightarrow ... \rightarrow l_i^{k-1} \rightarrow l_0^k \rightarrow l_i^k \rightarrow l_i^s$ (edges of substates with execution order between 1 and k are omitted), and
- leaving AND-complex state $\widehat{s}$ is represented by the c-path: $l_0^s \rightarrow l_i^s \rightarrow l_0^k \rightarrow l_i^k \rightarrow l_m^k \rightarrow l_0^{k-1} \rightarrow ... \rightarrow l_m^2 \rightarrow l_0^1 \rightarrow l_i^1 \rightarrow l_m^1 \rightarrow l_i^s \rightarrow l_m^s$ (edges of substates with execution order between 1 and k are omitted).

Also note all the above c-paths possess the correct sequence of guards and updates (details omitted). Since the discrete behaviors of a complex state consist of transitions to substates, as correctly modeled above, together with the evolution within each substate, which by Lemma 1 has been proven to be correct, the discrete behaviors of a complex state $\widehat{s}$ is correctly modeled by its I/O-EFA model. ■

Next we prove the correctness of Algorithm 4 that is used to refine the model that captures also the local events.

Lemma 3 The local events within a complex state $\widehat{s}$ are corrected modeled by its modified I/O-EFA model obtained by applying Algorithm 4.

Proof: We first prove that the discrete behavior of each local event ev is correctly modeled by its modified I/O-EFA model. From Algorithm 4 (also refer to Figure 2.12),

- execution of each event ev in its source state transition $o_{e^{ev}} \rightarrow d_{e^{ev}}$ is modeled by the c-path: $o_{e^{ev}} \rightarrow l_0^{des} \rightarrow l_i^{des} \rightarrow ... \rightarrow l_i^{des} \rightarrow t_{e^{ev}}$, where the first transition of the c-path passes the control to the destination state and also sets the binary data variable d^{ev} to 1, that serves as a guard of the subsequent transitions of the c-path that occur, respectively, at the destination state and its substates, whereas the last transition of the c-path returns control back to the source state.

Thus the local event behavior is correctly modeled by the modified I/O-EFA model. Since the discrete behaviors of a complex state with local events consist of the event behavior and the complex state behaviors without any local events, where the latter are correctly modeled by the I/O-EFA model prior to modification of Algorithm 4 (as proved in Lemma 2), we can conclude that the discrete behaviors of a complex state with local events is correctly modeled by the modified I/O-EFA model of Algorithm 4. ■

Using the above three lemmas, we can prove the main theorem that proves the correctness of the final I/O-EFA model.

Theorem 1 Given a Stateflow chart block ϕ, the discrete behaviors of ϕ is correctly modeled by its I/O-EFA model P^{ϕ} obtained from Algorithm 5.

Proof: From Algorithm 5 (also refer to Figure 2.14), the c-paths of the I/O-EFA model P^{ϕ} of a Stateflow chart block are equivalent to the c-paths of the I/O-EFA model obtained by applying Algorithm 1-4, as no additional paths are added, while the behaviors of the existing paths is preserved. Further, the discrete behaviors of Stateflow chart block ϕ is equivalent to the discrete behaviors of its root state $\widehat{s}$. According to Lemma 3, the discrete behaviors of a complex state $\widehat{s}$ is correctly modeled by the I/O-EFA model obtained by applying Algorithm 1-4. Thus, the I/O-EFA model P^{ϕ} also correctly models the discrete behaviors of Stateflow chart block ϕ. (Note that the difference between the outputs of Algorithm 4 versus 5 is only structural; the two models are behaviorally equivalent. Algorithm 5 is needed to make the final model conform the standard models that were proposed for I/O-EFA based modeling of time-driven blocks of Simulink in [12].) ■

2.4 Implementation and Validation

The Stateflow modeling approach described above, together with the Simulink modeling method for time-driven blocks of our previous work [12], have been written in the Matlab script, and implemented in an automated translation tool SS2EFA. Upon specifying a source Simulink/Stateflow model together with the input and output ports, the tool can be executed to output the corresponding I/O-EFA model in form of a "flat" Stateflow chart, which can itself be simulated in Matlab. Above we proved the correctness of the translation, and below we also validate this through several simulations to ensure that the result of simulating the I/O-EFA is the same as that of simulating the source Simulink/Stateflow model.

Example 6 The simulation result comparison between the I/O-EFA model of the counter (see Figure 2.15) and the original counter system (see Figure 2.3) is shown in Figure 2.16. The simulation (using Intel Core 2 Duo P8400 2.27GHz, 2GB RAM) time is 4 seconds with sampling period of 0.03 seconds, and the results are consistent with the behaviors of the counter.

Example 7 This example is of a servo velocity control system (shown in Figure 2.17) consisting of a controller, a fault monitor (both written in Stateflow, shown respectively in Figure 2.18 and Figure 2.19), and a motor (written in Simulink, shown in Figure 2.20). There

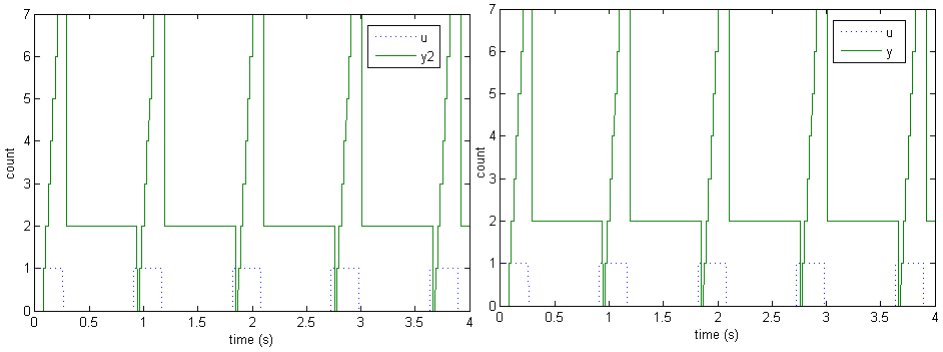

Figure 2.16 Simulation to compare the execution of Statechart Counter (left) and its I/O-EFA model of Fig 2.15 (right)

are 45 number of atomic blocks with 48 number of Stateflow states in the overall model. The Simulink/Stateflow diagram of the servo velocity control system is translated by our translation tool. The translated I/O-EFA model is a flat Stateflow diagram consisting of 382 number of states and 646 number of transitions. The CPU time (using Intel Core 2 Duo P8400 2.27GHz, 2GB RAM) for the translation is 45.1 seconds. The simulation result of the translated model (shown in Figure 2.21) is identical to the discrete behaviors of the original Simulink/Stateflow model.

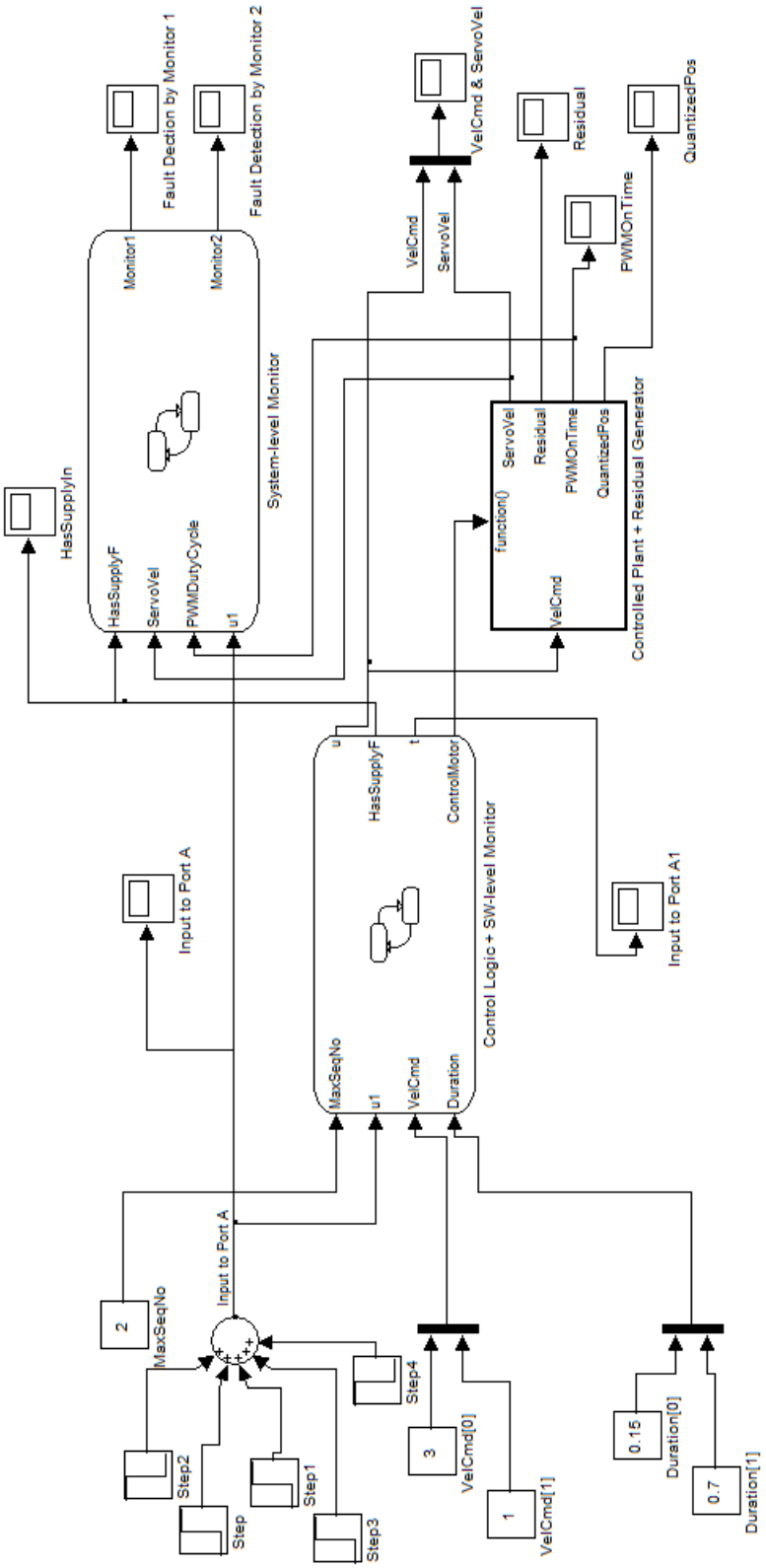

Figure 2.17 High level Simulink/Stateflow model of servo velocity control

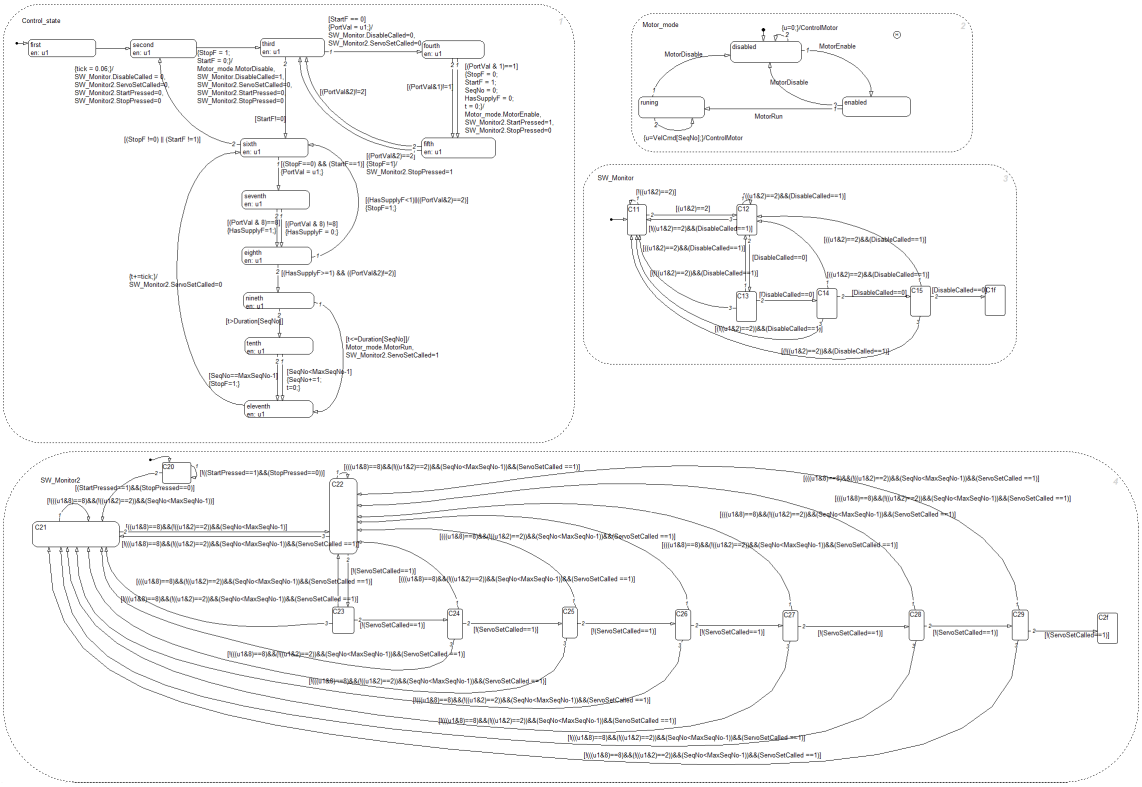

Figure 2.18 Controller Stateflow chart (named "Counter Logic + SW-level Monitor") of servo velocity control

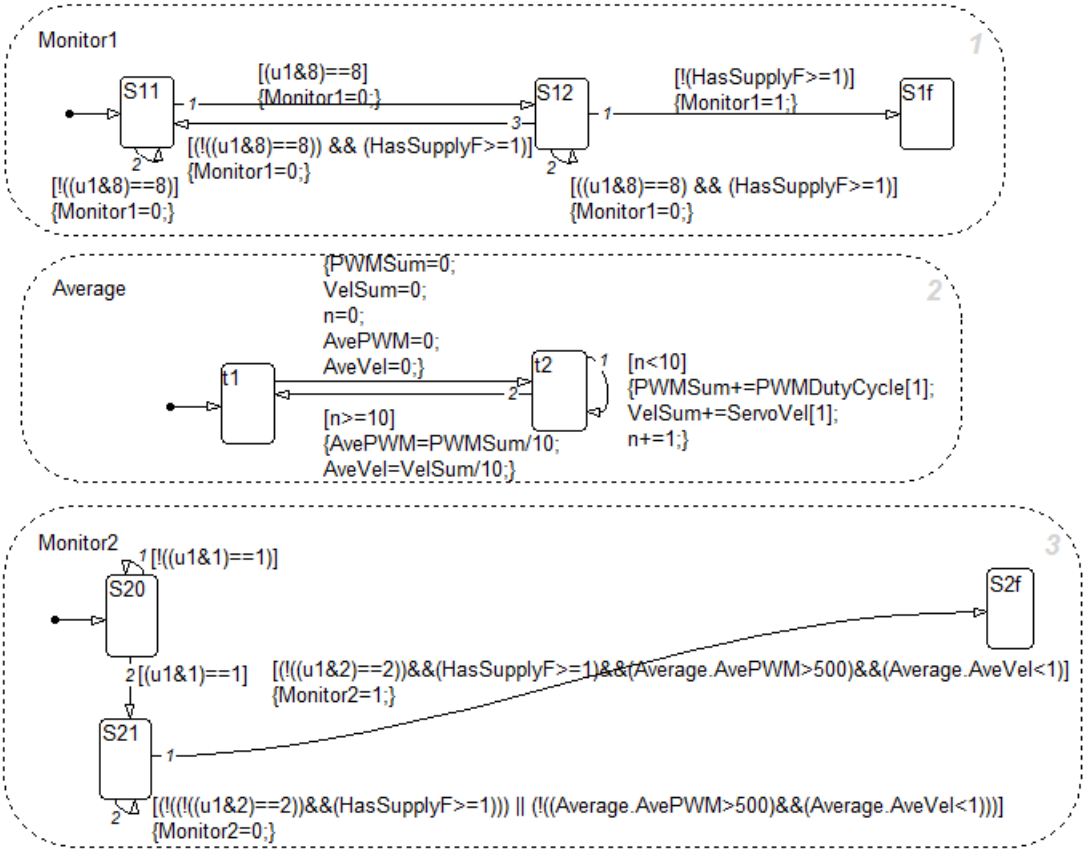

Figure 2.19 Fault monitor Stateflow chart (named "System-level Monitor") of servo velocity control

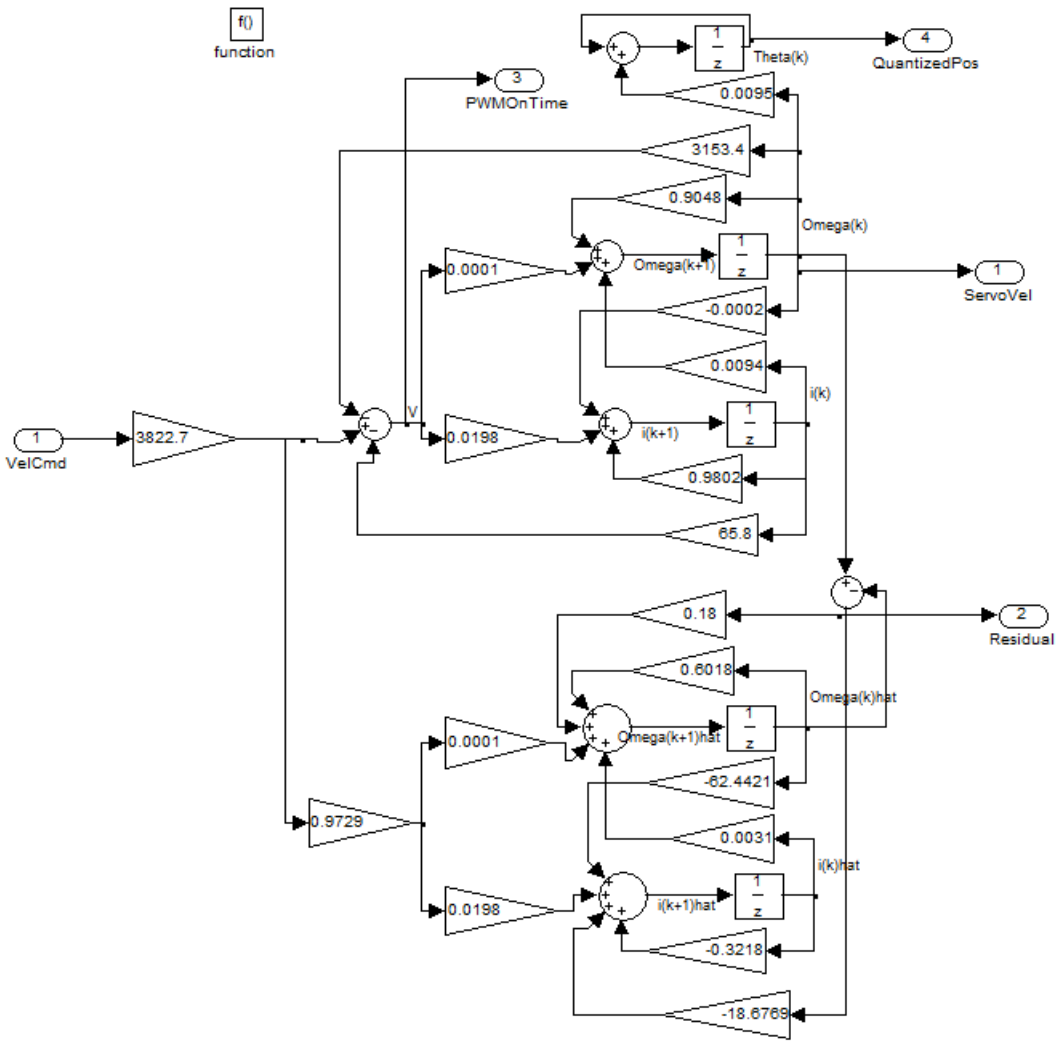

Figure 2.20 Motor Subsystem (named "Controlled Plant + Residual Generator") of servo velocity control

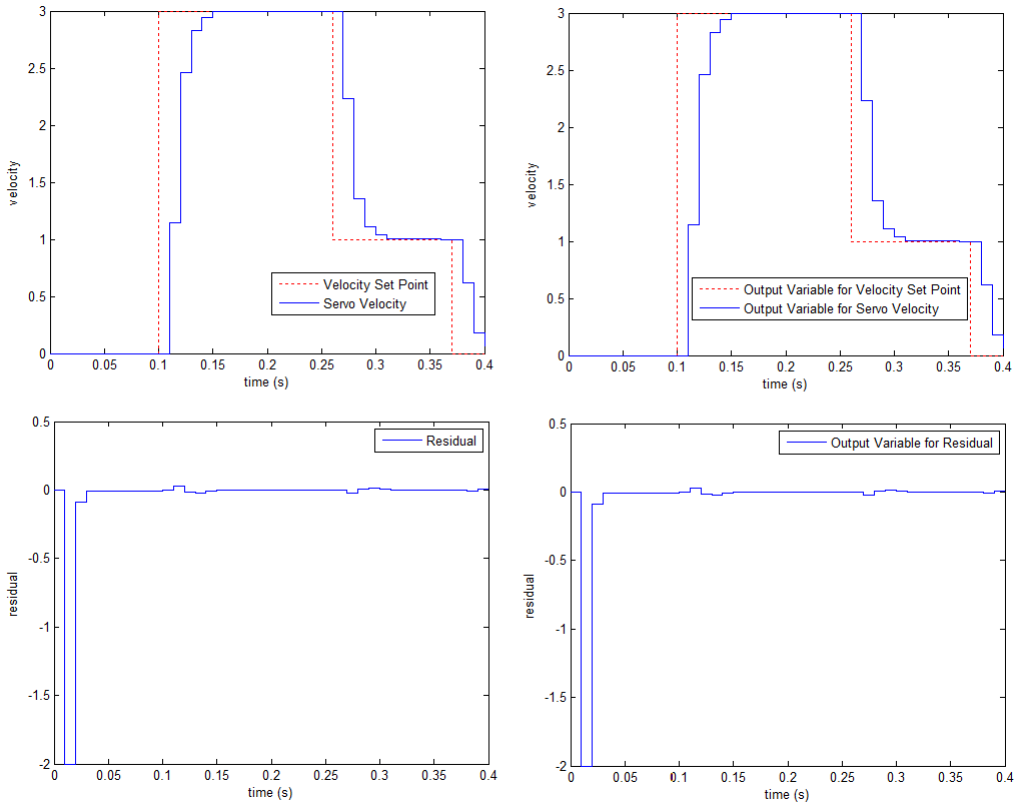

Figure 2.21 Simulation results for the velocity set point and actual servo velocity and residue; up-left (resp., up-right) figure is for the set point and actual servo velocity of the original Simulink/Stateflow model of servo system (resp., translated I/O-EFA model); down-left (resp., down-right) figure is for the residue of the original Simulink/Stateflow model of servo system (resp., translated I/O-EFA model)

CHAPTER 3 Test Generation of Simulink/Stateflow

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 Simulink/Stateflow diagram satisfies the design requirements, and whether the generated code (for example ANSI C) preserves the functional behaviors of the Simulink/Stateflow diagram. Testing is an essential step of validation, and while formal verification can catch many errors early in the design, it is unable to catch the errors introduced by compilation or asynchrony of the underlying execution platform.

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

- Errors in the Simulink/Stateflow diagram block selection and connection.
- Errors in the Simulink/Stateflow diagram block parameter settings.
- 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 testing approach to reveal these errors is to create a set of test cases from Simulink/Stateflow, and then validate or execute them against the requirements or the generated code to see if the test passes or fails. Any failed test cases can be used to find the errors introduced in the Simulink/Stateflow design or code generation process.

In Chapter 2, we introduced a recursive method to translate a Simulink/Stateflow diagram to I/O-EFA, which captures each computation cycle of Simulink/Stateflow in form of an automaton extended with data-variables to capture internal states and also the input and output variables. This chapter discusses the method to generate test cases for the Simulink/Stateflow diagram based on the corresponding I/O-EFA derived using the approach in Chapter 2. 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.

Test generation based on the requirements is also discussed by translating the requirements to an equivalent automaton. Test cases are obtained as acyclic executions accepted by the automaton and are applied to test the requirements.

3.1 Model-based Test Generation Approach

Our I/O-EFA model-based 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.

Chapter 2 formalizes and automates this mapping from computation sequences of Simulink/Stateflow diagram to the c-paths (defined in Definition 2) in the translated I/O-EFA model. 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 $\pi_5 = e_2e_8e_9e_{12}e_{13}e_{19}e_{20}e_{21}$ over the edges e_2 and e_9, where $y5(k) := 2$ on edge e_2, whereas $y5(k) > 7$ 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 $y5(k) < -0.5$ 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 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 Chapter 2. 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.

3.2 Algorithms for Model-based Test Generation

The proposed model-based 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.

3.2.1 Implementation using Model-Checking

Model-checking is a method to check automatically whether a model of a system meets a given specification. NuSMV [17] 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 4 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 discrete time counter $k < steplimit$ in the I/O-EFA model. In the NuSMV file, when the system evolves exceeding the defined value of "steplimit", the variable "location" is given 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: $value_{new} = round(value_{old} \cdot 10^{precision})$, where $value_{old}$ is the value in the I/O-EFA model, and $value_{new}$ is the value in NuSMV file.

• Each variable d_j is converted to an integer with upper limit $d_j^{max} \cdot 10^{precision}$ and lower limit $d_j^{min} \cdot 10^{precision}$, so that data space is finite. d_j^{max} and d_j^{min} are determined by the requirements on the system.

Definition 4 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

- $S = L \times D^f$ is the set of its states, where D^f is the finite abstraction of D,
- $E^f := \{((l_1, d_1^f), \sigma, u^f, \delta, y^f, (l_2, d_2^f)) \mid \exists d_1 \in d_1^f, u \in u^f, y \in y^f, d_2 \in d_2^f : (l_1, d_1) \xrightarrow{\sigma,u,\delta,y} (l_2, d_2)\}$ is its set of transitions,
- $S_0 = L_0 \times D_0^f$ is the set of its initial states, where D_0^f is the finite abstraction of D_0,
- U^f is the finite abstraction of U,
- Y^f 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_e:$D \times U \rightarrow D$ are expressed by the "next()" 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 6 A c-path $\pi = e_0^\pi ... e_{|\pi|-1}^\pi$ 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 ϕ is the CTL formula $EF(e_0^\pi \wedge EX(e_1^\pi \wedge \cdots EXe_{|\pi|-1}^\pi) \cdots)$, meaning path π can eventually be activated in the finite abstraction P^f. An input sequence that makes π 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 π. The final test suite is the set of input sequences obtained from a subset of reachable paths Π 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-output sequences activating those paths as the test suite. Report the unreachable paths for the analysis of model soundness.

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.

3.2.2 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:

$$\begin{aligned} &minimize_x && f(x) \\ &subject\ to && g_i(x) \leq 0, i = 1, \ldots, m \\ & && h_i(x) = 0, i = 1, \ldots, p \end{aligned}$$

where

- $f(x) : R^n \rightarrow R$ is the objective function to be minimized over the variable x,
- $g_i(x) \leq 0$ are called inequality constraints, and
- $h_i(x) = 0$ are called equality constraints.

Finding whether a c-path π of an I/O-EFA is reachable can be converted to a constraint solving problem, which is an optimization problem without regard to an objective value as follows:

$$\begin{aligned} &minimize_{(d,u)} && 1 \\ &subject\ to && G_\pi(d, u) \end{aligned}$$

where, $G_\pi(d, u, y)$ is called the path-predicate of the path π. It is a set of conditions over (d, u) activating the path π. The above constraint solving problem has solution if the path

predicate $G_\pi(d,u)$ is satisfiable (does not equate to *False*). The path predicate $G_\pi(d,u)$ along with its data $f_\pi(d,u)$ and output $h_\pi(d,u)$ can be obtained as follows.

Algorithm 7 For a path $\pi = e_0^\pi ... e_{|\pi|-1}^\pi$, its path-predicate $G_\pi(d,u)$ can be computed recursively backward, and data $f_\pi(d,u)$ and output $h_\pi(d,u)$ can be computed recursively forward as:

Base step:

$j = |\pi| - 1, k = (|\pi| - 1) - j;$

$G_\pi^j(d,u) := G_{e_{|\pi|-1}^\pi}(d,u);$

$f_\pi^k(d,u) := f_{e_0^\pi}(d,u);$

$h_\pi^k(d,u) := h_{e_0^\pi}(d,u).$

Recursion step:

$G_\pi^{j-1}(d,u) := G_{e_{j-1}^\pi}(d,u) \wedge G_\pi^j(f_{e_{j-1}^\pi}(d,u), \{u, h_{e_{j-1}^\pi}(d,u)\});$

$f_\pi^{k+1}(d,u) := f_{e_{k+1}^\pi}(f_\pi^k(d,u), \{u, h_\pi^k(d,u)\});$

$h_\pi^{k+1}(d,u) := h_{e_{k+1}^\pi}(f_\pi^k(d,u), \{u, h_\pi^k(d,u)\}).$

Termination step:

If $j \neq 0$, then decrement j and return to recursion step; else stop, and set:

$G_\pi(d,u) := G_\pi^0(d,u);$

$f_\pi(d,u) := f_\pi^{|\pi|-1}(d,u);$

$h_\pi(d,u) := h_\pi^{|\pi|-1}(d,u).$

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

Constraint solving problem is constructed to check if $G_\pi(d,u) \neq False$, in which case, the path π is feasible. The feasible paths obtained in Algorithm 7 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 Π 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 Π is as follows.

Algorithm 8 Given a path-sequence $\Pi = \pi_0 \dots \pi$ starting at the initial condition $I(d)$ ending with the path π, the feasibility/reachability of Π can be checked recursively backward, while its data update and output assignment can be computed recursively forward as:

Base step:

$j = (|\Pi| - 1), k = (|\Pi| - 1) - j;$

$G^j_\Pi(d, u_j) := G_\pi(d, u_j);$

$f^k_\Pi(d, u_k) := f_{\pi_0}(d, u_k);$

$h^k_\Pi(d, u_k) := h_{\pi_0}(d, u_k).$

Recursion step:

$G^{j-1}_\Pi(d, \{u_{j-1}, \dots, u_{|\Pi|-1}\}) := G_{\pi_{j-1}}(d, u_{j-1}) \wedge G^j_\Pi(f_{\pi_{j-1}}(d, u_{j-1}), \{\{u_j, \dots, u_{|\Pi|-1}\},$
$h_{\pi_{j-1}}(d, u_{j-1})\});$

$f^{k+1}_\Pi(d, \{u_0, \dots, u_{k+1}\}) := f_{\pi_{k+1}}(f^k_\Pi(d, \{u_0, \dots, u_k\}), \{u_{k+1}, h^k_\Pi(d, \{u_0, \dots, u_k\})\});$

$h^{k+1}_\Pi(d, \{u_0, \dots, u_{k+1}\}) := h_{\pi_{k+1}}(f^k_\Pi(d, \{u_0, \dots, u_k\}), \{u_{k+1}, h^k_\Pi(d, \{u_0, \dots, u_k\})\}).$

Termination condition:

If $j \neq 0$, then decrement j and return to recursion step; else stop and set:

$G_\Pi(d, \{u_0, \dots, u_{|\Pi|-1}) = G^0_\Pi(d, \{u_0, \dots, u_{|\Pi|-1});$

$f_\Pi(d, \{u_0, \dots, u_{|\Pi|-1}) = f^{|\Pi|-1}_\Pi(d, \{u_0, \dots, u_{|\Pi|-1});$

$h_\Pi(d, \{u_0, \dots, u_{|\Pi|-1}) = h^{|\Pi|-1}_\Pi(d, \{u_0, \dots, u_{|\Pi|-1});$

and declare Π as reachable iff $G^0_\Pi(d, \{u_0, \dots, u_{|\Pi|-1}\}) \wedge I(d) \neq False.$

Given a feasible path π, if none of the path-sequence with $|\Pi| \leq steplimit$ ending with π is reachable, π is unreachable within the *steplimit*. Otherwise, π is reachable. Note *steplimit* is the test case length requirement of the system.

Given a reachable path-sequence $\Pi = \pi_0 ... \pi$ ending with path π, a test input-output sequence $t_\pi = (u_0, y_0) ... (u_{|\Pi|-1}, y_{|\Pi|-1})$ activating the reachable path π is obtained by selecting $u_0, \dots, u_{|\Pi|-1}$ such that $G^0_\Pi(d, \{u_0, \dots, u_{|\Pi|-1}\})$ holds, and for $j = 0, \dots, |\Pi| - 1$, $y_j = h^j_\Pi(d, \{u_0, \dots, u_j\})$. This input-output sequence t_π is the test case for path π.

The constraint solving based test suite is derived with an open source optimization tool CVX [18], 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 7 to obtain the feasible paths;
- For each feasible path π, apply Algorithm 8 on each path-sequence Π that ends in π and $|\Pi| \leq$ *steplimit* to determine the reachability of π;
- Report the unreachable paths identified in the previous two steps for the analysis of model soundness.

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.

3.3 Software Implementation of Model-based Test Generation Algorithms

Both of the model-based 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 8 Model Checker based Test Generator: Consider the I/O-EFA model (see Figure 2.2) of the counter system (see Figure 2.1). By specifying *steplimit* $= 10$, *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 3.1.

Table 3.1 Reachable Paths and Test Cases from Implementation with Model-Checking

Path	Test Case ($(u, y2)$ at each sample time)
$e_0e_3e_4e_5e_6e_7e_8e_{10}e_{12}e_{14}e_{15}$ $e_{16}e_{17}e_{18}e_{19}e_{20}e_{21}$	$(1,0)$
$e_1e_3e_4e_5e_6e_7e_8e_{10}e_{12}e_{14}e_{15}$ $e_{16}e_{17}e_{18}e_{19}e_{20}e_{21}$	$(1,0),(1,1)$
$e_1e_3e_4e_5e_6e_7e_8e_9e_{12}e_{14}e_{15}$ $e_{16}e_{17}e_{18}e_{19}e_{20}e_{21}$	$(1,0),(1,1),(1,2),(1,3),$ $(1,4),(1,5),(1,6),(1,7),(1,8)$
$e_2e_8e_{10}e_{12}e_{13}e_{20}e_{21}$	$(0,2)$

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 9 Constraint Solving based Test Generator: Consider the same I/O-EFA model (see Figure 2.2) of the counter system (see Figure 2.1). By specifying $steplimit = 10$, $u \in \{0, 1\}$, and path-covered criterion (all paths be covered), the constraint solving based test generator provides five feasible paths as shown in Table 3.2 and four of them are reachable (π_3 is identified as unreachable). The test cases are generated as shown in Table 3.3.

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.

3.4 Requirements-based Test Generation

Design requirements are properties that a Simulink/Stateflow model is expected to satisfy. They need to be tested against the Simulink/Stateflow model for satisfiability. The goal of Requirements-based Test Generation for Simulink/Stateflow is to generate test cases to be able to ensure testing of the specified requirements.

Table 3.2 Feasible Paths from Implementation with Constraint Solving

Path No.	Path Predicate	Path Data	Path Outputs
π_0	$u(k) > 0 \wedge$ $d' = 0$	$d(k) := 0,$ $d' := 1,$ $d(k+1) := 1,$ $k := k+1$	$y2(k) := 0,$ $y3(k) := 1,$ $y5(k) := 0,$ $y4(k) := 1$
π_1	$u(k) > 0 \wedge$ $d' = 1 \wedge$ $-0.5 \leq d(k)$ ≤ 7	$d(k+1) :=$ $d(k)+1,$ $k := k+1$	$y2(k) := d(k),$ $y3(k) := 1,$ $y5(k) := d(k),$ $y4(k) := d(k)+1$
π_2	$u(k) > 0 \wedge$ $d' = 1 \wedge$ $d(k) > 7$	$d(k+1) :=$ $d(k)+1,$ $k := k+1$	$y2(k) := 7,$ $y3(k) := 1,$ $y5(k) := d(k),$ $y4(k) := d(k)+1$
π_3	$u(k) > 0 \wedge$ $d' = 1 \wedge$ $d(k) < -0.5$	$d(k+1) :=$ $d(k)+1,$ $k := k+1$	$y2(k) := -0.5,$ $y3(k) := 1,$ $y5(k) := d(k),$ $y4(k) := d(k)+1$
π_4	$u(k) \leq 0$	$d(k+1) := d(k),$ $d' := 0,$ $k := k+1$	$y2(k) := 2,$ $y5(k) := 2$

Table 3.3 Test Cases from Implementation with Constraint Solving

Path Number	Test Case ($(u, y2)$ at each sample time)
π_0	$(1,0)$
π_1	$(1,0),(1,1)$
π_2	$(1,0),(1,1),(1,2),(1,3),$ $(1,4),(1,5),(1,6),(1,7),(1,8)$
π_4	$(0,2)$

Each model requirement, being a property of an input/output computation sequence, can be expressed as an Linear Temporal Logic (LTL) formula, with the propositions as the predicates over the input/output variables of the I/O-EFA model of the Simulink/Stateflow diagram. For example, the counter in Figure 2.1 has the requirement that "output can never exceed 7". The corresponding LTL formula is $\phi = [\neg F(y2(k) > 7)]$.

[19] [20] [21] [22] discuss algorithms to compute a Büchi automaton accepting all infinite sequences satisfying a given LTL formula. A Büchi automaton is defined as follows.

Definition 5 A Büchi automaton is a 5-tuple $R = (Q, \Gamma, \Xi, Q_0, Q_m)$, where

- Q is a finite set of states,
- Γ is a finite set of symbols,
- $\Xi \subseteq Q \times \Gamma \times Q$ is the set of state transitions,
- $Q_0 \in Q$ is the set of initial states, and
- $Q_m \subseteq Q$ is the set of marked states.

The states in R correspond to subformulas of ϕ and so $|Q|$ is of the exponential order in $|\phi|$; the edge-labels Γ are Boolean formulas over the predicates in ϕ; since each edge in R processes a new input-output pair, it implicitly advances the time counter by 1. R accepts exactly those infinite runs (state sequences) that are initialized at Q_0, follow the transition relation, and visit Q_m infinitely often. Thus it can be assumed without loss of generality that all marked states are in some strongly connected component. Since we only generate finite length test cases, we can only test those LTL properties that are properties of finite length runs/computations. This is precisely the *safety fragment* of LTL [23]. For the safety fragment of LTL, its Büchi model R is deterministic and it accepts finite traces from initial locations $q_0 \in Q_0$ to final locations $q_m \in Q_m$ (within a strongly connected component). A test case that activates the requirement is generated from an acyclic path in R from $q_0 \in Q_0$ to $q_m \in Q_m$ that *imposes constraints* on the outputs. (A test case that does not constraint an output vacuously passes). The Requirements-based Test Generation is defined in the algorithm below.

Algorithm 9 Given a LTL requirement ϕ, the test cases activating ϕ are generated in the following steps.

1. Compute the Büchi automaton $R = (Q, \Gamma, \Xi, Q_0, Q_m)$ accepting all input/output sequences that satisfy ϕ.

2. Find all acyclic paths from $q_0 \in Q_0$ to $q_m \in Q_m$ (q_m is within a strongly connected component), where there exists at least one proposition along the edges of the path that constraints an output.

3. For each acyclic path $tr = G_0 \dots G_{|tr|-1}$, where G_i is the guard predicate on the $(i-1)$th edge of the path, the test case is a finite sequence $t_{tr} = (u_0, y_0) \dots (u_{|tr|-1}, y_{|tr|-1})$, where for $i = 0, \dots, |tr| - 1$, (u_i, y_i) satisfies the guard G_i.

Example 10 Consider the counter in Figure 2.1 with requirement that "output is zero whenever input is non-positive for a first time, and then on output can be arbitrary". The LTL formula for this requirement is $[u \leq 0 \Rightarrow y2 = 0]U[L(u \leq 0)]$, where U denotes *until* and L denotes *in last step*. The Büchi automaton computed from the LTL formula is as shown in Figure 3.1. There is one acyclic path $tr = [[u \leq 0] \wedge [y2 = 0]]$ that has a constraint on the output. The test case obtained from tr is $t_{tr} = \{(u_0, y_0) = (0, 0)\}$.

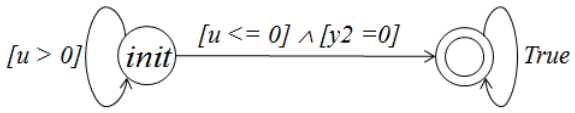

Figure 3.1 Büchi automaton computed from LTL formula $[u \leq 0 \Rightarrow y2 = 0]U[L(u \leq 0)]$

Remark 1 Note the requirements-based test generation is a much simpler exercise than the model-based test generation since each acyclic accepted path of the requirements model can be activated on its own, without having to execute any preparatory prefix path-sequence. This is because the path-predicates in the requirements-model do not involve any data-variables, only the input-output variables, and so no prior preparation is needed to set the data-variables to the right values. In contrast, the model-based test generation requires the execution of an earlier prefix path-sequence so as to "prepare" the data-variables to the right values. Accordingly,

the requirements-based test generation is implemented simply as a special case of model-based test generation.

CHAPTER 4 Reduction of Test Generation to Reachability and its Novel Resolution

Test generation approach in Chapter 3 involves a bounded length search and finds the test cases with lengths within the bound. It starts backwards from a target computation path (c-path) of the I/O-EFA model to be tested and finds a sequence of prefix computation-paths to reach the initial condition. Note the execution of a prefix is required to set the data values for the target c-path to become executable. Determining whether or not such an enabling prefix exists is undecidable in general. Our technique in Chapter 3 checks all the path-sequences ending with the target c-path within certain time step limit to see if the c-path can be reached from the initial condition. As expected due to the undecidability of the problem, the approach is not guaranteed to find test cases for all reachable c-paths.

Test cases with length longer than the bound used by the approach of [15] may exist for Simulink/Stateflow diagrams possessing loops since it may take several iterations along the loop before a data value suitable to enable a target computation-path becomes available. In the case that we are able to "collapse" the computation of those several iterations into a single step by analytically solving the computations performed, it will become possible to explore arbitrary length iterations. Building on this idea, this chapter improves the test generation approach to reduce the test generation time and remove constraints on the maximum test case length for those Simulink/Stateflow models for which the computation results of the multiple iterations of each computation path can be analytically computed (this for example is the case for linear update functions). We first apply the techniques in our previous papers [24] [12] [13] [15] to convert the Simulink/Stateflow diagram into I/O-EFA model and enumerate all feasible c-paths whose enabling predicates are non-False (note feasibility is only a necessary condition

for reachability since the enabling predication, while non-False, may not be reachable from the initial condition). To be able to check reachability of a target computation-path, we next create a computation-succession automaton which is an instance of an discrete-time hybrid automaton and preserves the computations of the I/O-EFA. Test generation is then done by performing reachability analysis on this hybrid automaton.

Reachability analysis for hybrid automata is widely studied subject, and in general undecidable. See for example some recent surveys [25–27]. Our contribution lies in the utilization of the analytical solutions of the dynamics, whenever feasible, in arriving at a novel reachability resolution technique.

Our approach iteratively refines the hybrid automaton to obtain an equivalent refined hybrid automaton. The idea is to split each location into a number of locations and associate a stronger invariant condition with the locations and stronger guard conditions with the incoming/outgoing edges so that the reachability in the hybrid automaton reduces to the reachability in the underlying graph, ignoring all the dynamics. We show that the termination of the iterative refinement is guaranteed when the I/O-EFA model possesses a *finite late-bisimilar quotient*. Also while the iterative refinement may not terminate in general, we show through examples that the approach is more effective compared to the test generation approaches in Chapter 3 in the sense that it is able to provide a larger test coverage due to the use of the analytical solutions that allows unbounded length computations. Also the approach turns out to be more efficient needing less time for test generation since the results of unbounded length computations are analytically derived offline. The hybrid automaton-based approach is also applied on the defect-detection and requirements-satisfaction to more efficiently and effectively detect defects and unsatisfied requirements.

4.1 Introduction to I/O-HA

In Chapter 2, we modeled a Simulink/Stateflow diagram as an Input/Output Extended Finite Automaton (I/O-EFA) model preserving its discrete behaviors. In this chapter 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 6 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,
  - $f_l : D \times U \to D$ is data-update function, and
  - $h_l : D \times U \to Y$ 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,
- Σ is the set of symbolic-inputs,
- Δ 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,
  - $\sigma_e \in \Sigma \cup \{\varepsilon\}$ is symbolic-input,
  - $\delta_e \in \Delta \cup \{\varepsilon\}$ is symbolic-output,
  - $G_e \subseteq D \times U$ is enabling guard (a predicate),

- $f_e : D \times U \to D$ is data-update function, and
- $h_e : D \times U \to Y$ 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 σ_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 δ_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 $True$, location update and assignment functions as the identity maps.

4.2 Computation-Succession Hybrid Automaton

In Chapter 3, 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.

For each c-path sequence $\omega = \pi_0 \dots \pi_{|\omega|-1|}$, it is possible to compute its enabling guard $G_\omega \subseteq D \times U$ recursively backwards, and its data-update function $f_\omega : D \times U \to D$ and its output-assignment function $h_\omega : D \times U \to Y$ recursively forward using Algorithm 7-8. Then a c-path π is immediately executable (equivalently, feasible) if and only if its enabling guard $G_\pi(d, u)$ is satisfiable, and π is eventually executable (equivalently, reachable) if there is a path-sequence ω ending in π (i.e., $\pi_{|\omega|-1} = \pi$), such that $G_\omega(d, \{u_0 \dots u_{|\omega|-1}\})$ is satisfiable.

Example 11 Consider the Simulink diagram of a bounded counter shown in Figure 4.1, which is a modified version of the counter in Figure 2.3 with upper bound of the saturation block

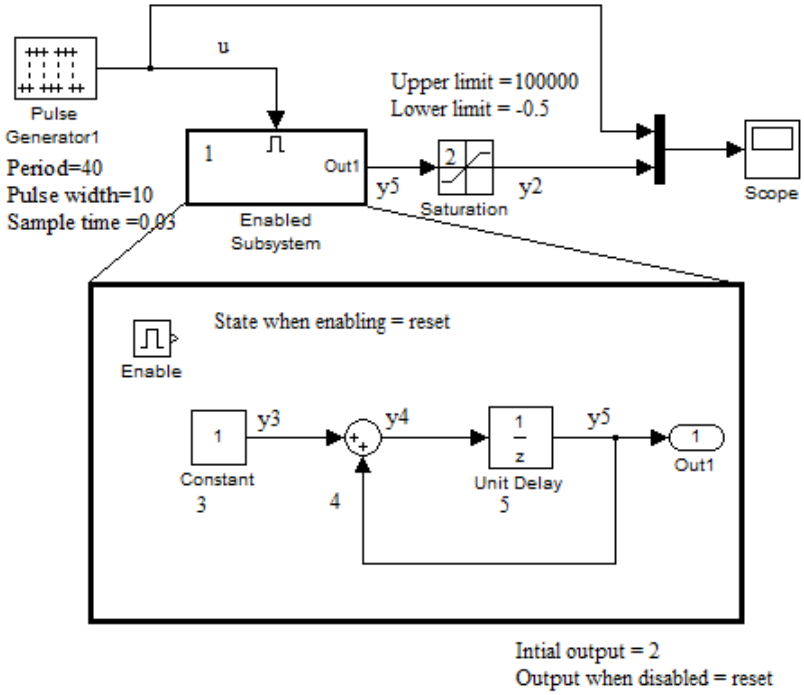

Figure 4.1 Simulink Diagram of a Counter System

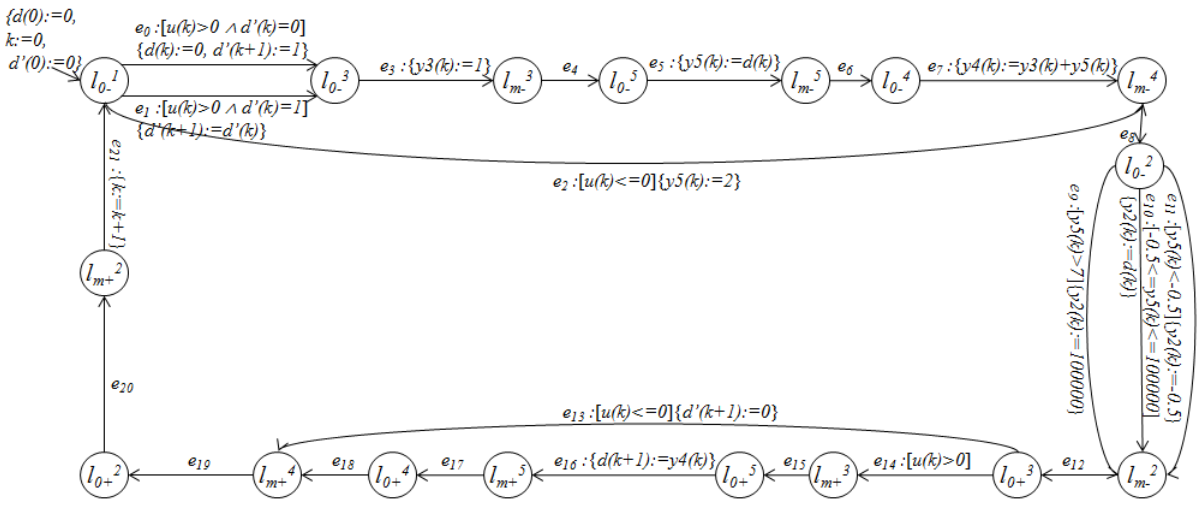

Figure 4.2 I/O-EFA model of the Counter System in Figure 4.1

increased to 100000. It consists 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 $y5$ resets to its initial value when the control input u is not positive. The saturation block limits the value of $y5$ in the range between -0.5 and 100000. The translated I/O-EFA P using the method of Chapter 2 is shown in Figure 4.2. As can be seen the translated I/O-EFA has 18 different 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 c-paths are feasible, as analyzed by Algorithm 7, and only 4 of 5 are reachable. These are listed in Table 4.1.

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 10 Given a set of feasible c-paths Π^P of a I/O-EFA model $P = (L, D, U, Y, \Sigma, \Delta, L_0, D_0, L_m, E)$, its Computation-Succession Hybrid Automaton (CS-HA) is obtained as, $P^\Pi = (L^\Pi, D, U, Y, \Sigma, \Delta, L_0^\Pi, D_0, L^\Pi, E^\Pi)$, 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 Π^P. (P^Π has one location l^π for each feasible c-path π of P, and l^π's invariant/data-update/output-assignment are the same as the guard/data-update/output-assignment of π.)

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

- $E^\Pi = \bigcup_{\pi, \pi' \in \Pi^P, \pi \neq \pi'} \{(l^\pi, l^{\pi'}, -, -, G_{\pi'}(d,u), -, -)\}$ 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^Π 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

Table 4.1 Path Analysis of I/O-EFA model of Figure 4.2

Path π	Path Guard G_π	Path Data f_π	Path Outputs h_π
$\pi_0 =$ $e_0e_3e_4e_5$ $e_6e_7e_8e_{10}e_{12}$ $e_{14}e_{15}e_{16}e_{17}$ $e_{18}e_{19}e_{20}e_{21}$	$u(k) > 0 \wedge$ $d'(k) = 0$	$d(k) := 0,$ $d'(k+1) := 1,$ $d(k+1) := 1,$ $k := k+1$	$y2(k) := 0,$ $y3(k) := 1,$ $y5(k) := 0,$ $y4(k) := 1$
$\pi_1 =$ $e_1e_3e_4e_5e_6$ $e_7e_8e_{10}e_{12}$ $e_{14}e_{15}e_{16}e_{17}$ $e_{18}e_{19}e_{20}e_{21}$	$u(k) > 0 \wedge$ $d'(k) = 1 \wedge$ $-0.5 \leq d(k)$ ≤ 100000	$d(k+1) :=$ $d(k)+1,$ $d'(k+1) := 1,$ $k := k+1$	$y2(k) := d(k),$ $y3(k) := 1,$ $y5(k) := d(k),$ $y4(k) := d(k)+1$
$\pi_2 =$ $e_1e_3e_4e_5e_6$ $e_7e_8e_9e_{12}$ $e_{14}e_{15}e_{16}e_{17}$ $e_{18}e_{19}e_{20}e_{21}$	$u(k) > 0 \wedge$ $d'(k) = 1 \wedge$ $d(k) > 100000$	$d(k+1) := d(k)+1,$ $d'(k+1) := 1,$ $k := k+1$	$y2(k) := 100000$ $y3(k) := 1,$ $y5(k) := d(k),$ $y4(k) := d(k)+1$
$\pi_3 =$ $e_1e_3e_4e_5e_6$ $e_7e_8e_{11}e_{12}$ $e_{14}e_{15}e_{16}e_{17}$ $e_{18}e_{19}e_{20}e_{21}$	$u(k) > 0 \wedge$ $d'(k) = 1 \wedge$ $d(k) < -0.5$	$d(k+1) := d(k)+1,$ $d'(k+1) := 1,$ $k := k+1$	$y2(k) := -0.5,$ $y3(k) := 1,$ $y5(k) := d(k),$ $y4(k) := d(k)+1$
$\pi_4 =$ $e_2e_8e_{10}e_{12}$ $e_{13}e_{20}e_{21}$	$u(k) \leq 0$	$d(k+1) := d(k),$ $d'(k+1) := 0,$ $k := k+1$	$y2(k) := 2,$ $y5(k) := 2$

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^Π is also deterministic. Finally note that the CS-HA P^Π 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 2 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^π is reachable in the corresponding CS-HA P^Π.

Example 12 Given the feasible paths in Table 4.1, the corresponding CS-HA is shown in Figure 4.3.

4.3 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 11 Given a CS-HA P^Π, for each $l \in L^\Pi$, do the following:

Base step:

$j = 0, i = (N-1) - j;$

$G_l^i(d, u_{k+i}) := G_l(d, u_{k+i});$

$f_l^j(d, u_{k+j}) := f_l(d, u_{k+j});$

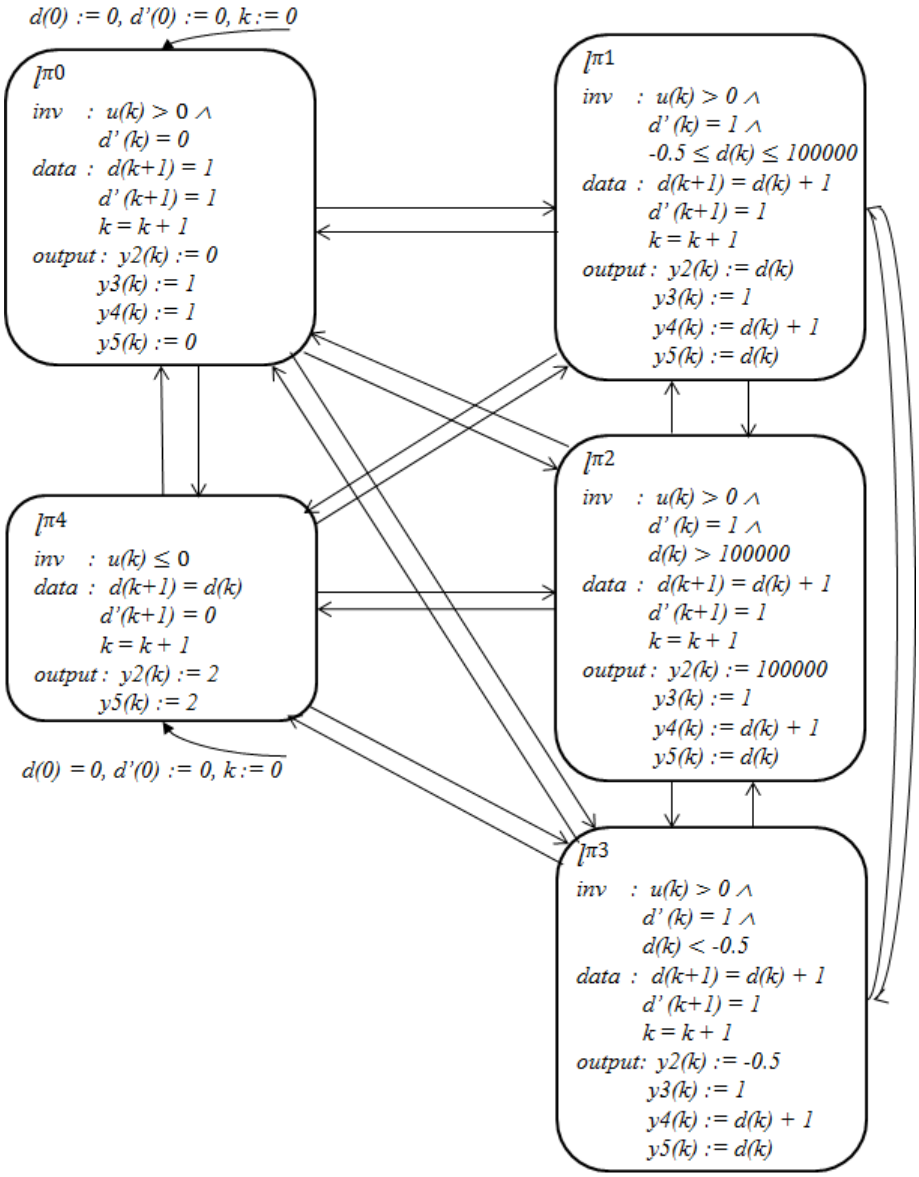

Figure 4.3 CS-HA of the I/O-EFA model in Figure 4.2. Transition guards, which are the same as the invariants of the destination location, are omitted.

$h_l^j(d, u_{k+j}) := h_l(d, u_{k+j})$. (Let π_l be the feasible c-path of P represented by a location l of P^{Π}. Then the base step computes the guard for the last, i.e., Nth execution of π_l, together with the data-update and output-assignment of the first execution of π_l.)

Recursion step:

$G_l^{i-1}(d, \{u_{k+i-1}, \ldots, u_{k+N-1}\}) := G_l(d, u_{k+i-1}) \wedge G_l^i(f_l(d, u_{k+i-1}), \{\{u_{k+i}, \ldots, u_{k+N-1}\},$ $h_l(d, u_{k+i-1})\})$;

$f_l^{j+1}(d, \{u_k, \ldots, u_{k+j+1}\}) := f_l(f_l^j(d, \{u_k, \ldots, u_{k+j}\}),$ $\{u_{k+j+1}, h_l^j(d, \{u_k, \ldots, u_{k+j}\})\})$;

$h_l^{j+1}(d, \{u_k, \ldots, u_{k+j+1}\}) := h_l(f_l^j(d, \{u_0, \ldots, u_{k+j}\}),$ $\{u_{k+j+1}, h_l^j(d, \{u_k, \ldots, u_{k+j}\})\})$. (Recursion step computes the guard of the last $N - (i - 1)$ executions of π_l, together with the data-update and output-assignment of the first $j + 1$ executions of π_l. Note the former calculation uses the guard for last $N - i$ executions of π_l (a backward recursion), whereas the latter calculation uses the data-update/output-assignment of the first j executions of π_l (a forward recursion).)

Termination step:

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

$$G_{ll'}(d) = \bigvee_{N \geq 1} \Big[\exists \{u_k, \ldots, u_{k+N}\} : G_l^0(d, \{u_k, \ldots, u_{k+N-1}\}) \wedge G_{l'}(f_l^{N-1}(d, \{u_k, \ldots, u_{k+N-1}\}), u_{k+N})\Big].$$

(Upon termination, the precondition to transit from l to successor l' by evolving at l for one or more steps is computed. Note that $G_{ll'}(d)$ is solvable whenever G_l^0 and f_l^{N-1} 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 12 Given a CS-HA $P^{\Pi} = (L^{\Pi}, D, U, Y, \Sigma, \Delta, L_0^{\Pi}, D_0, L^{\Pi}, E)$, the refinement algorithm iteratively computes for each iteration n, a refined hybrid automaton $P^n = (L^n, D, U, Y, \Sigma, \Delta, L_0^n, D_0, L^n, E^n)$, where $L_0^n := \{l \in L^n \mid G_l \wedge D_0 \neq False\}$, and $E^n := \{(l, l', -, -, G_{l'}, -, -) \mid l, l' \in L^n, l \neq l', G_{ll'}(d) \neq False\}$, as follows (note for each $n \geq 0$, only L^n needs to be iteratively computed since definitions of L_0^n and E_0^n are derived from that of L^n):

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^{Π}.)

Recursion step: $L^{n+1} = \bigcup_{l \in L^n, G(d) \in \mathcal{G}_l} \{(G_l(d,u) \wedge G(d), f_l(d,u), h_l(d,u))\}$, where for each $l \in L^n$, $\mathcal{G}_l := \bigcup_{L' \subseteq succ(l)} \{\bigwedge_{l' \in L'} G_{ll'}(d) \bigwedge_{l' \in succ(l) - L'} \neg G_{ll'}(d)\}$ is the partition induced by $\{G_{ll'}(d) \mid l' \in succ(l)\}$. (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 $\overline{P^{\Pi}} := P^n$; 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 13 Consider the CS-HA shown in Figure 4.4. The CS-HA is refined according to Algorithm 12. Firstly, since l^{π_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^{π_1} is split into three locations $l_0^{\pi_1}$, $l_1^{\pi_1}$, and $l_2^{\pi_1}$ as in Figure 4.5. This requires the application of Algorithm 11 to find the guards $G_{l^{\pi_1} l^{\pi_2}}(d(k)) = [d(k) = 2]$, $G_{l^{\pi_1} l^{\pi_3}}(d(k)) = [d(k) = 3]$, $G_{l^{\pi_1} l^{\pi_4}}(d(k)) = [2 < d(k) < 3 \vee 3 < d(k) \leq 4]$, and then performing the refinement as in Algorithm 12 that splits l^{π_1} into $l_0^{\pi_1}, l_1^{\pi_1}, l_2^{\pi_1}$ with the invariants $G_{l^{\pi_1} l^{\pi_2}}, G_{l^{\pi_1} l^{\pi_3}}, G_{l^{\pi_1} l^{\pi_4}}$ respectively, and with the same data-update and output-assignment functions as l^{π_1}. Next, since l^{π_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^{π_0} is split into three locations $l_0^{\pi_0}$, $l_1^{\pi_0}$, and $l_2^{\pi_0}$ as in Figure 4.6. Again this requires applying Algorithm 11 to find the

guards $G_{l^{\pi_0}l_0^{\pi_1}}(d(k)) = [d(k) = 0]$, $G_{l^{\pi_0}l_1^{\pi_1}}(d(k)) = [d(k) = 1]$, $G_{l^{\pi_0}l_2^{\pi_1}}(d(k)) = [0 < d(k) < 1]$, and then performing the refinement as in Algorithm 12 that splits l^{π_0} into $l_0^{\pi_0}, l_1^{\pi_0}, l_2^{\pi_0}$ with the invariants $G_{l^{\pi_0}l_0^{\pi_1}}, G_{l^{\pi_0}l_1^{\pi_1}}, G_{l^{\pi_0}l_2^{\pi_1}}$ respectively, and with the same data-update and output-assignment functions as l^{π_0}. Note only the node $l_0^{\pi_0}$ remains an initial node since the invariant condition for $l_1^{\pi_0}$ and $l_2^{\pi_0}$ are $[d(k) = 1]$ and $[0 < d(k) < 1]$, which are disjoint from the initial condition $[d(k) = 0]$. Also only the node $l_1^{\pi_0}$ remains reachable from l^{π_2}, whose outgoing edge guard $[d(k) = 1]$ has nonempty overlap with only the invariant of $l_1^{\pi_0}$. 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 12 to terminate and yielding the refined CS-HA of Figure 4.6.

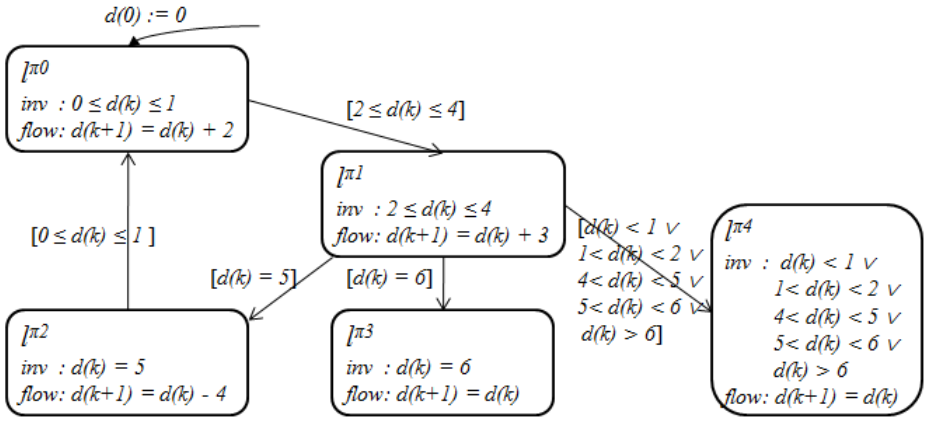

Figure 4.4 CS-HA with a cycle-location possessing more than one successor

The following theorem establishes that the refinement step indeed resolves the reachability.

Theorem 3 When Algorithm 12 terminates in finite steps with $L^{n+1} = L^n$, then the refined PS-HA $\overline{P^{\Pi}}$ from Algorithm 12 has the property that, if there exists a path from the initial locations to a target location, then the target location is reachable.

Proof: Since location invariants satisfy the preconditions to reach their successors, each location can eventually transit to its successors by selecting a sequence of input. If there exists a path from an initial location to the target location, initial location can transit along the path to any locations on the path and eventually to the target location. Target location is reachable. ■

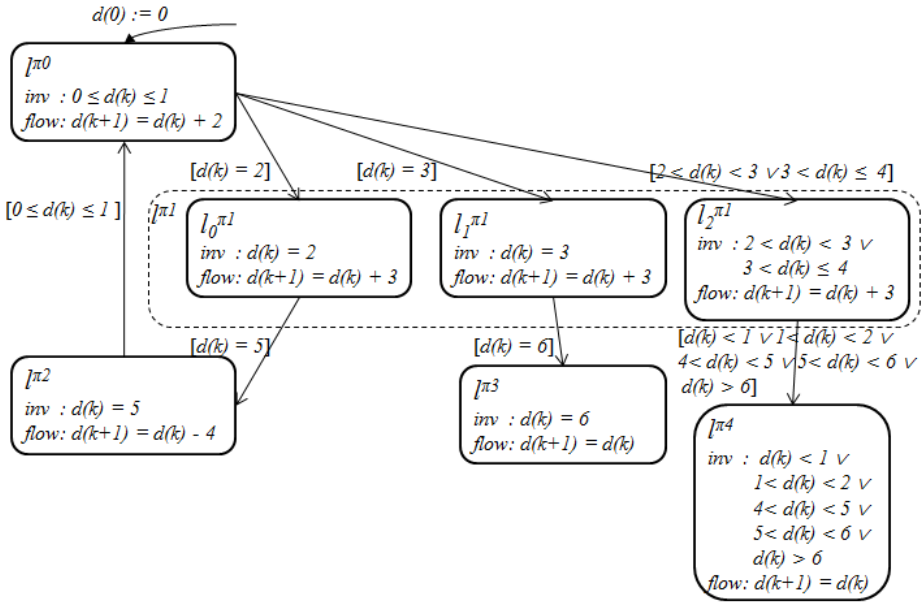

Figure 4.5 Refined model of the CS-HA in Figure 4.4 with l^{π_1} split. Dotted line encloses the locations after the split.

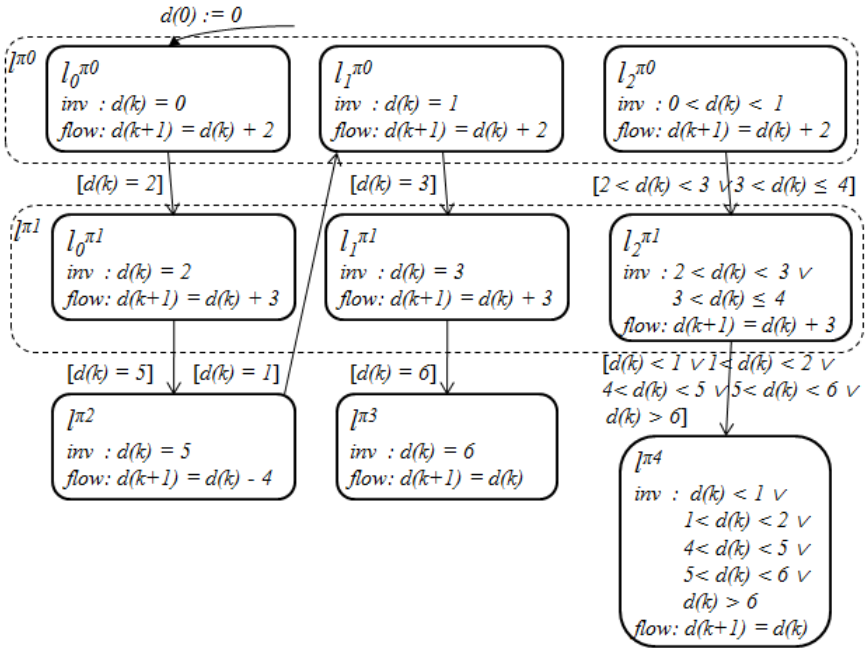

Figure 4.6 Refined model of the CS-HA in Figure 4.5 with l^{π_0} split. Dotted line encloses the locations after the split.

The following theorem provides a condition for the termination of Algorithm 12. It employs the notion of late-bisimilarity and late-bismulation quotient, which can be found in [28]; a brief overview is also presented in the appendix for completeness.

Theorem 4 Algorithm 12 terminates if and only if the CS-HA of the Simulink/Stateflow model preserves a finite late-bisimilar quotient [28].

Proof: Necessity is obvious since Algorithm 12, when it terminates with a finite n, it actually finds a finite late-bisimilar quotient of the CS-HA. For sufficiency, suppose the CS-HA possesses a finite late-bisimilar quotient, then it must be finer than the one introduced by the location-invariants of P^{Π} [28, Proposition 2]. Suppose for contradiction that Algorithm 12 does not terminate, then the partition of the data space will not terminate and eventually the partition will be finer than the partition of the coarsest finite late-bisimilar quotient, which means the refined CS-HA at that point would be a finite late-bisimilar quotient, causing Algorithm 12 to terminate, and arriving at a contradiction to the hypothesis. ■

Example 14 Consider the CS-HA of the counter in Figure 4.3. By applying Algorithm 12 on the CS-HA, the refined CS-HA is obtained in Figure 4.7. The refinement terminates in 1 iteration; the details are omitted for brevity. It turns out that the refinement step does not introduce any new splits, but out of the total 20 edges (see Figure 4.3), only 8 edges survive; the others have False guard conditions. From the connectivity information of the refined CS-HA model of Figure 4.7, it is evident that 4 out of the 5 feasible c-paths are reachable. (π_3 is the only unreachable c-path.)

Remark 2 Example 14 shows a drastic improvement compared to the approach of [15], since to reach the c-path π_2, a prefix of length 100000 must be executed first. (The guard condition for π_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 π_1 is executed.) Finding such a path using the search employed in [15] 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.

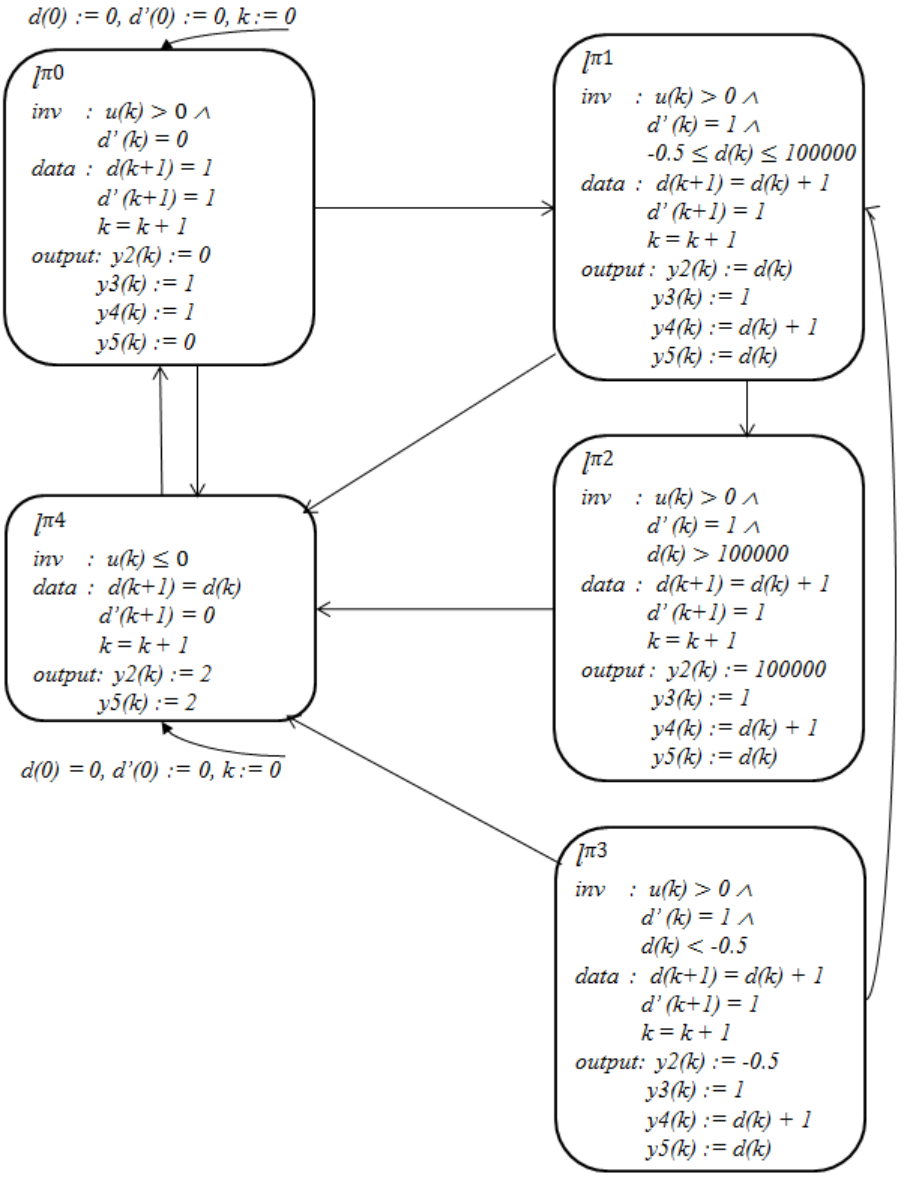

Figure 4.7 Refined model of the CS-HA in Figure 4.3

Remark 3 Note that the reachability resolution approach can also be applied to general hybrid automata. If a hybrid automaton satisfies the property in Theorem 4, Algorithm 12 terminates within finite steps, yielding a refined hybrid automaton that is a finite late-bisimilar quotient, and for which the reachability is decidable.

Note that Algorithm 12 applies to a CS-HA with the following property, that we designate as Θ for future reference:

Θ: Each edge guard is equivalent to the invariant of its destination location and there are no data-update or output-assignment functions on the edge.

4.4 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 13 A c-path $\pi \in \Pi^P$ is reachable if there exists a location $l \in L^n = L^{n+1}$ in the refined CS-HA, with $G_l \Rightarrow G_\pi$, and a path $\omega = l_0 \dots l_{|\omega|-1}$ starting at an initial location $l_0 \in L_0^n$ and ending with the location $l_{|\omega|-1} = l$. For the path ω, its test case can be computed iteratively forward as:

Base step:

$j = k = 0$, and solve for $d_j, N_j, \{u_k, \dots, u_{k+N_j}\}$ such that the following holds:

$D_0 \wedge G^0_{l_j}(d_j, \{u_k, \dots, u_{k+N_j-1}\}) \wedge G_{l_{j+1}}(f^{N_j-1}_{l_j}(d_j, \{u_k, \dots, u_{k+N_j-1}\}), u_{k+N_j})$. ((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_{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 $d_{j+1} := f^{N_j-1}_{l_j}(d_j, \{u_k, \dots, u_{k+N_j-1}\})$, $\{y_{k+i} := h^i_{l_j}(d_j, \{u_k, \dots, u_{k+i}) \mid 0 \le i \le N_j - 1\}, k := k + N_j, j := j + 1$, and solve for N_j and $\{u_{k+1}, \dots, u_{k+N_j}\}$ such that the following holds:

$G^0_{l_j}(d_j, \{u_k, \ldots, u_{k+N_j-1}\}) \wedge G_{l_{j+1}}(f^{N_j-1}_{l_j}(d_j, \{u_k, \ldots, u_{k+N_j-1}\}), u_{k+N_j})$, and return to recursion step. (Similar to the base step, the recursion finds jth sequence of N_j inputs so that the jth c-path l_j can be executed N_j number of times, so that the resulting data d_{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 ω has been executed a certain number of times in the same order as appearing in ω.)

Remark 4 In order to compute a test case for a reachable c-path, Algorithm 13 requires an analytical solution of all $\{f^j_l, h^j_l : l \in \omega\}$, 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 14 1. Obtain the I/O-EFA model P of a given Simulink/Stateflow diagram according to Chapter 2.

2. Apply Algorithm 7 to enumerate the feasible c-paths Π^P of P.
3. Apply Algorithm 10 on Π^P to obtain the CS-HA P^Π of I/O-EFA P.
4. Apply Algorithm 12 to refine P^Π and obtain the refined CS-HA $\overline{P^\Pi}$.
5. Apply Algorithm 13 on $\overline{P^\Pi}$ to identify reachable c-paths of P, and to generate their test cases.

Example 15 Consider the refined CS-HA in Figure 4.7 of the counter in Figure 2.3. By applying Algorithm 13 on the refined CS-HA, the test cases to reach the reachable c-paths are obtained in Table 4.2. 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 Chapter 3. This illustrates the effectiveness of the new approach proposed here in terms of providing a

Table 4.2 Test cases generated from the refined CS-HA in Figure 4.7

c-Path	Path in CS-HA	Test Case
π_0	l^{π_0}	$u = \{1\}$
π_1	l^{π_0}, l^{π_1}	$u = \{1, 1\}$
π_2	$l^{\pi_0}, l^{\pi_1}, l^{\pi_2}$	$u = \{1, \ldots, 1\}$ $\|u\| = 100001$
π_3	Not Reachable	N/A
π_4	l^{π_4}	$u = \{0\}$

better test coverage, and also its efficiency in terms of the time needed for automated test generation.

4.5 Applications of CS-HA in Defect-Detection/Requirements-Satisfaction

Simulink/Stateflow models may possess defects, such as overflow conditions (e.g., divided-by-zero), design ambiguity/conflicts, un-testable condition, etc., which need to be avoided during the design process. Defect-detection is a step before the test generation to detect whether any block-specific or user-defined defects exist in the Simulink/Stateflow model. Failure of detecting the defects may result in severe accidents in case of safety critical applications. Requirements, on the other hand, are properties defined by users that the Simulink/Stateflow model must satisfy. Requirements-satisfaction is a step before the test generation to verify the satisfaction of the critical requirements. Both defect-detection and requirements-satisfaction can be performed more efficiently using the compact modeling formalism of CS-HA of Simulink/Stateflow diagram introduced above, together with the reachability resolution method also proposed above.

Each model defect/requirement that we analyze is assumed to be a property of an input/output computation, and so can be expressed as a predicate over the input/output variables of the I/O-EFA model of the Simulink/Stateflow diagram. For example, the counter in Figure 4.1 has the requirement that "output can never exceed 100000 or fall below 0". The corresponding predicate is $\phi(u, y) = [\neg(y2(k) > 100000 \vee y2(k) < 0)]$.

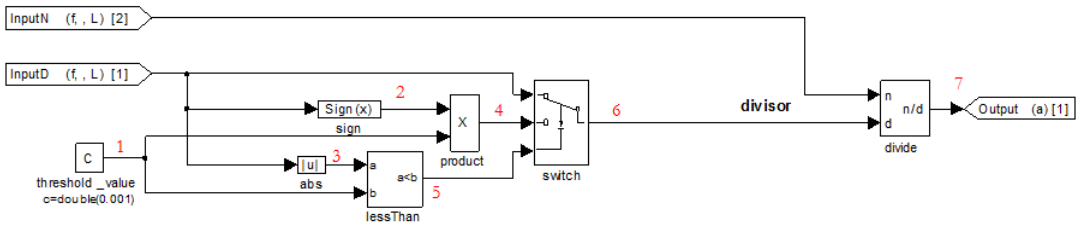

Figure 4.8 Simulink model for a division operation

Note the negation of a defect serves as a requirement, and with this observation we can have a common approach for defect-detection/requirements-satisfaction, where each defect specification is first negated to turn into a requirements specification. Next a refined CS-HA of the Simulink model is obtained, in which a fault-location f is introduced whose reachability corresponds to defect-witness or requirement-violation. The refined CS-HA is obtained as in the algorithm below. Each location as well as each edge is partitioned into two cases.

Algorithm 15 Given a requirement ϕ and a CS-HA $P = (L, D, U, Y, \Sigma, \Delta, L_0, D_0, L, E)$, the ϕ-refined CS-HA is $P^\phi = (L^\phi, D, U, Y, \Sigma, \Delta, L_0^\phi, D_0, L^\phi, E^\phi)$, where

- $L^\phi = L^t \cup \{f\}$ is a set of locations obtained from refining guards of locations L in ϕ and adding a fault-location f, where
  - $L^t = \bigcup_{l \in L} \{G_l \wedge \phi(u, h_l(d,u)), f_l, h_l\}$ is the set of locations, with invariants also satisfying ϕ, and
  - $f = \{-, -, -\}$ is the fault-location with no dynamics since once the fault-location is reached, the specification is violated and there is no need for further evolution.
- $L_0^\phi = \bigcup_{l \in L_0} \{G_l \wedge \phi(u, h_l(d,u)), f_l, h_l\}$ is the set of initial locations, with invariants also satisfying ϕ,
- $E^\phi = E^t \cup E^f$ is the set of edges, where
  - $E^t = \bigcup_{e \in E} \{o_e, t_e, -, -, G_{t_e} \wedge \phi(u, h_{t_e}(d,u)), -, -\}$ is the set of edges with guards also satisfying the requirement as evaluated at the destination locations and performing transitions among locations in L^t;

– $E^f = \bigcup_{l \in L^t} \{l, f, -, -, (G_l \wedge \neg\phi(u, h_l(d,u))) \bigvee_{e \in E, o_e = l} (G_{t_e} \wedge \neg\phi(u, h_{t_e}(d,u))), -, -\}$ is the set of edges to the fault-location f which are taken when either the location invariants in L^t or the edge guards in E^t violate the requirement ϕ.

The purpose of the refinement algorithm above is to convert the defect-detection and requirements-violation problems into reachability problems. The refinement merely partitions the location invariants and edge guards into cases that satisfy the requirement versus the ones that don't. As a result, (1) the refinement continues to satisfy the property Θ needed for the application of Algorithm 12 for reachability resolution, and (2) the behaviors executed by the refinement P^ϕ are identical to those of P, with the exception that the behaviors violating the specification are simply terminated at the fault-location.

Example 16 Consider the Simulink model for performing a division operation in Figure 4.8. The input to the denominator (InputD) is compared with a threshold 0.001 to avoid divided-by-zero defect. If the absolute value of the denominator input is less than the threshold 0.001, then the denominator remains at 0.001 or -0.001 (negativity depends on the negativity of the denominator input); else, the denominator equals to the input. Defect-detection aims to detect if there is divided-by-zero defect, i.e., if zero is a possible value of the denominator. By performing the steps 1-3 of Algorithm 14, the CS-HA of the division Simulink model, satisfying the property Θ is obtained as in Figure 4.9. The predicate for the divided-by-zero defect is $[y6(k) = 0]$. The requirement is to avoid the defect and thus has the predicate $\phi = [y6(k) \neq 0]$. By applying Algorithm 15 on the CS-HA in Figure 4.9 with requirement ϕ, the refinement is obtained with five locations, which also satisfies the property Θ. The result of refinement performed by the reachability resolution Algorithm 12 also possesses five locations, and is shown in Figure 4.10. As can be seen in this figure, the fault-location f is not reachable. Therefore, the requirement is satisfied by the division Simulink model (i.e. the divided-by-zero defect is not present in the model).

Example 17 Consider the counter in Figure 4.1 with requirement that "whenever input is non-positive output is zero, otherwise output can be arbitrary". The predicate for this re-

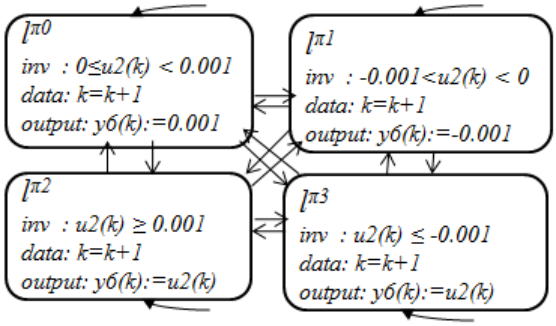

Figure 4.9 CS-HA of the division model in Figure 4.8. Only the output-assignment function that assigns the second input to the denominator is shown.

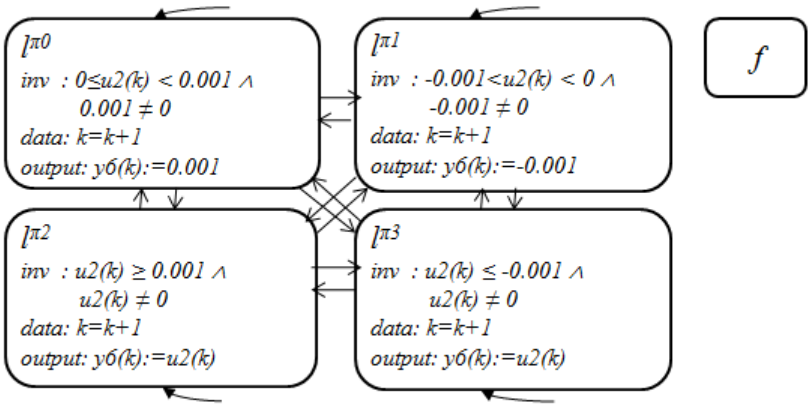

Figure 4.10 Refinement of the CS-HA in Figure 4.9 against the requirement $\phi = [y6(k) \neq 0]$. Only the output-assignment function that assigns the second input to the denominator is shown.

quirement is $\phi = [(u(k) \leq 0 \wedge y2(k) = 0) \vee u(k) > 0]$. By performing the refinement, using Algorithm 15, of the CS-HA in Figure 4.7 against the requirement ϕ, the result is obtained with six locations. The result of applying Algorithm 12 possesses five locations and is shown in Figure 4.11. Invariant of l^{π_4} is False and the location is not shown, since evolvement in l^{π_4} violates the requirement ϕ ($y2(k)$ should be zero when $u(k) \leq 0$, but $y2(k)$ is assigned with 2 in l^{π_4}). As can be seen in Figure 4.11, the fault-location f is reachable. Therefore, the requirement is not satisfied by the counter Simulink model.

Remark 5 Application of CS-HA for defect-detection and requirements-satisfaction *shares the same advantage* as its application to the test generation problem, namely: The existence of the defects and the satisfiability of the requirements are reduced to the reachability problem of the fault-location in the CS-HA, for which the reachability resolution method for CS-HA introduced in Section IV offers better efficiency (faster analysis time) and effectiveness (no pre-defined search depth is needed).

4.6 Case Study: a Thermal Control

In this section, the Simulink/Stateflow test generation approach described above is validated with a realistic application of a thermal control of a house from the Simulink demo [1], as shown in Figure 4.12. This system models the outdoor environment, the thermal characteristics of the house, and the house heating system. "Set Point" is a constant block. It specifies the temperature that must be maintained indoors, and equals 70 degrees Fahrenheit by default. Temperatures are given in Fahrenheit, but then are converted to Celsius to perform the calculations. "Thermostat" is a subsystem that contains a Relay block. The thermostat allows fluctuations of 5 degrees Fahrenheit above or below the desired room temperature. When air temperature drops below 65 degrees Fahrenheit, the thermostat turns on the heater. The thermostat signal turns the heater on or off. When the heater is on, it blows hot air at temperature THeater (50 degrees Celsius = 122 degrees Fahrenheit by default) at a constant flow rate of Mdot (1kg/sec = 3600kg/hr by default). "House" is a subsystem that calculates room temperature variations. It takes into consideration the heat flow from the heater and heat

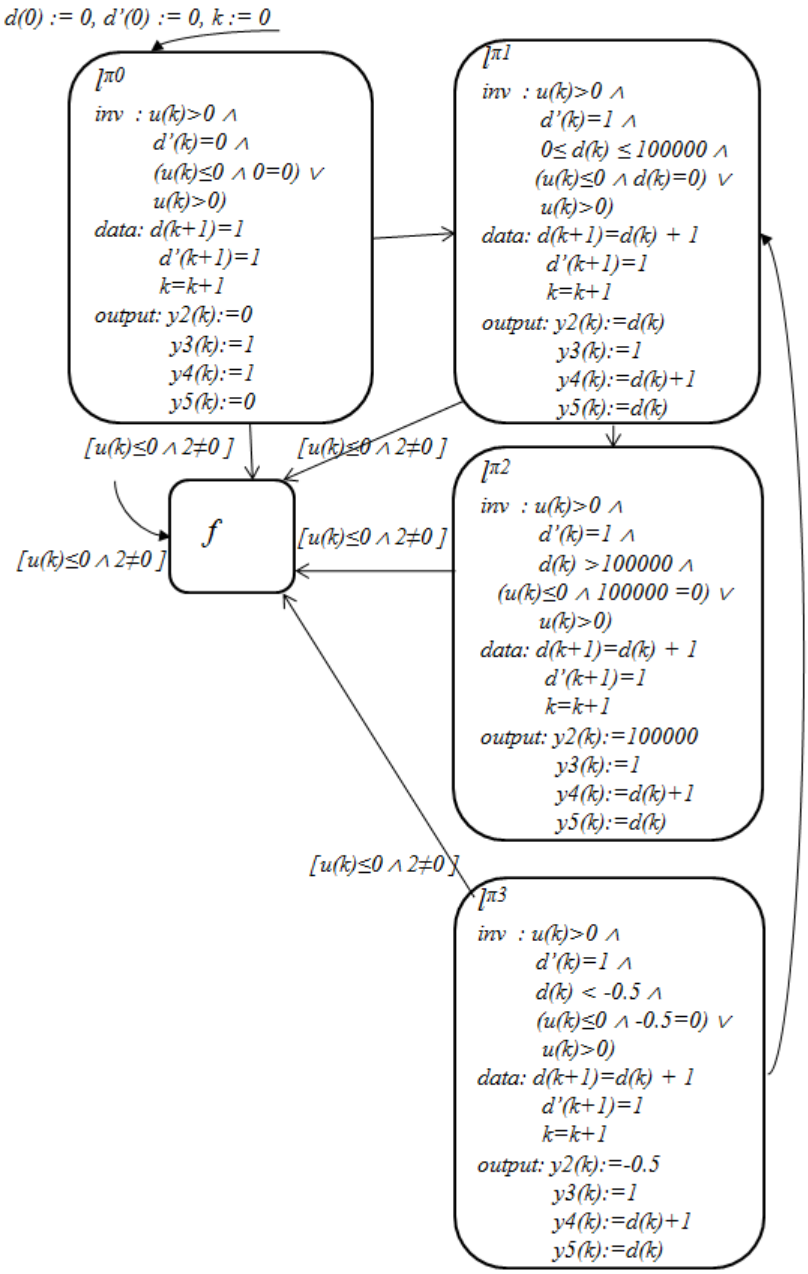

Figure 4.11 Refinement of the CS-HA in Figure 4.7 against the requirement $\phi = [(u(k) \leq 0 \wedge y2(k) = 0) \vee u(k) > 0]$. The guards of the edges with destination locations other than the fault-location are the same as the invariants in their destination locations and therefore are omitted.

losses to the environment. "Cost Calculator" is a Gain block. "Cost Calculator" integrates the heat flow over time and multiplies it by the energy cost. We model the environment as a heat sink with infinite heat capacity and a constant temperature Tout. The sample time of the model is $T = 0.001$.

The I/O-EFA model of the thermal control of a house can be obtained as in Figure 4.13. There are 16 possible c-paths, out which four feasible as determined by applying Algorithm 7. The corresponding CS-HA is obtained as in Figure 4.14 by executing Algorithm 10. In Figure 4.14, l^{π_0} represents the computation that when the house temperature is lower than 5 degrees below the desired room temperature, the heater is on to increase the temperature of the house; l^{π_1} represents the computation that when the house temperature is within ± 5 degrees of the desired room temperature with the heater previously on, the heater remains on to increase the temperature of the house; l^{π_2} represents the computation that when the house temperature is greater than 5 degrees above the desired room temperature, the heater is off so that house temperature decreases due to the heat losses to the environment; l^{π_3} represents the computation that when the house temperature is within ± 5 degrees of the desired room temperature with the heater previously off, the heater remains off so that the house temperature decreases due to the heat losses to the environment.

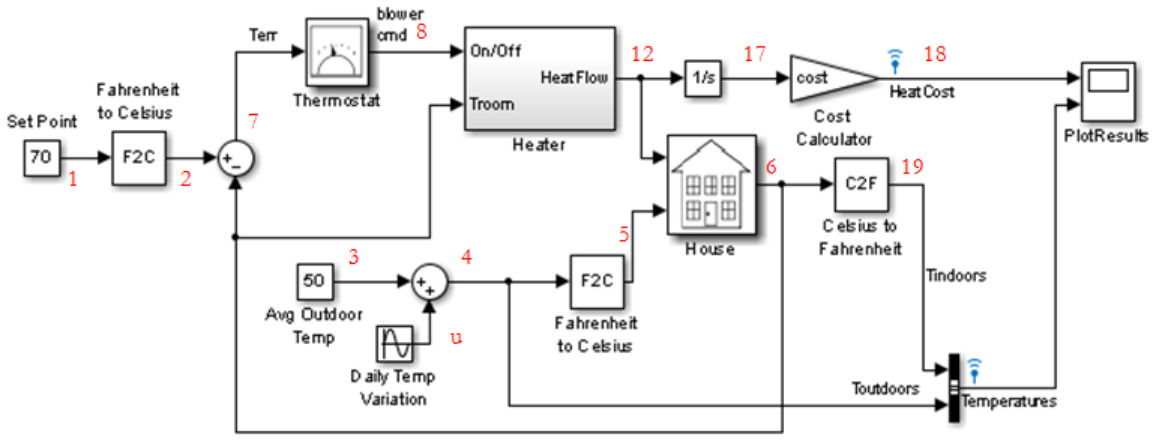

Figure 4.12 Simulink model for a thermal model of a house

The refined CS-HA of the thermal model of a house can be obtained as in Figure 4.15 by applying Algorithm 12 for one iteration on the CS-HA in Figure 4.14. The edge guards that

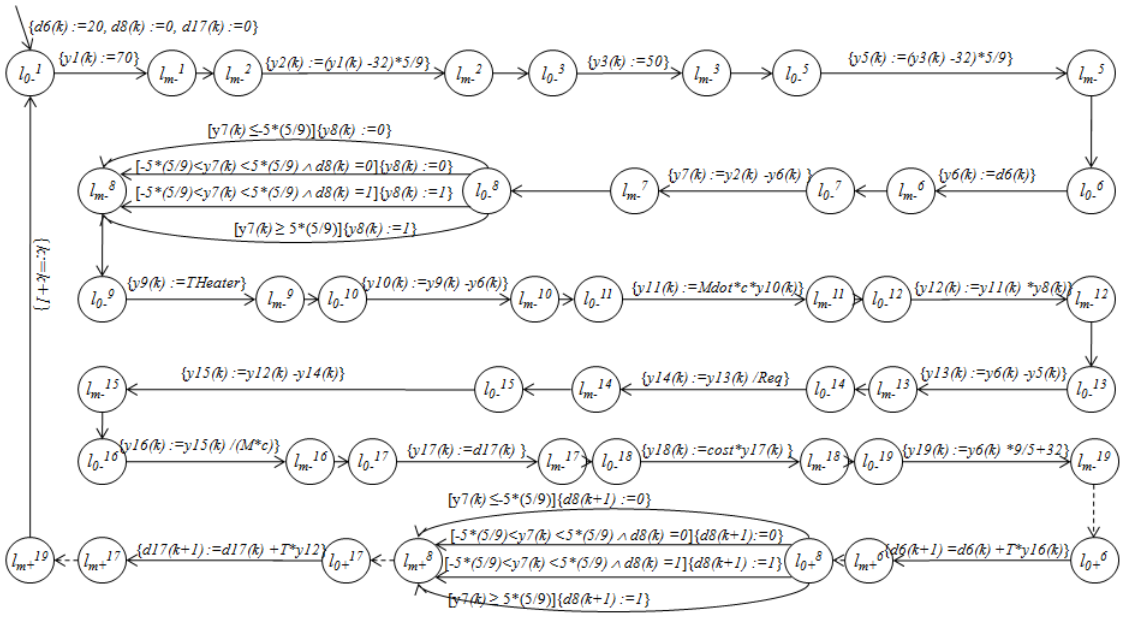

Figure 4.13 I/O-EFA model for the thermal model of a house in Figure 4.12, where $THeater = 50$, $Req = 4.26976e-07$, $Mdot = 3600$, $c = 1005.4$, $M = 1778.37$, and $T = 0.001$. I/O-EFA modules with empty edge labels are omitted and represented as dotted arrows.

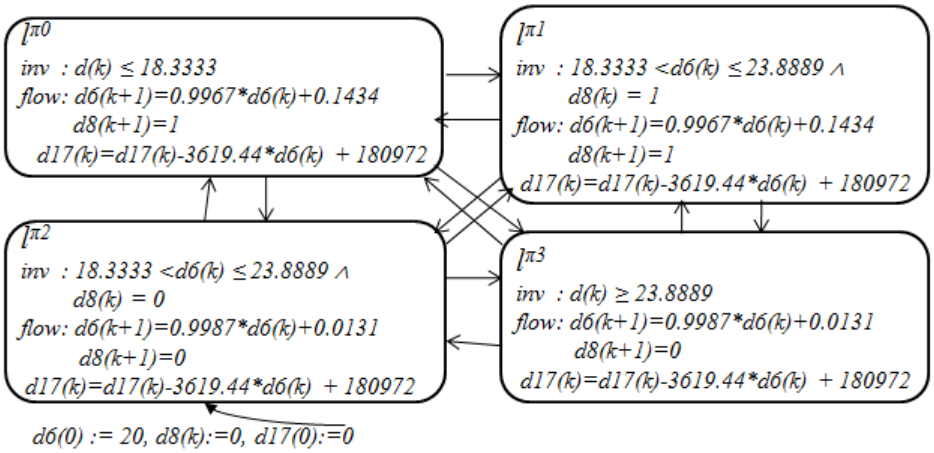

Figure 4.14 CS-HA model for the thermal model of a house in Figure 4.12. Output-assignment functions of the locations are omitted.

Table 4.3 Test Cases of the Thermal Model of a House

c-Path Number	Test Case (number of time steps to run the model)
π_0	$k = 142$
π_1	$k = 143$
π_2	$k = 0$
π_3	$k = 220$

become false after the initial iteration are omitted from the figure. In Figure 4.15, there exists a path from the initial location to each location in the CS-HA, therefore, all four locations are reachable. By applying Algorithm 13, a set of test cases can be obtained by analytically solving the difference equations in each location (totally four computations). The result of the test generation is shown in Table 4.3, where we can see that by running the model for 220 time steps all four c-paths of the model can be executed.

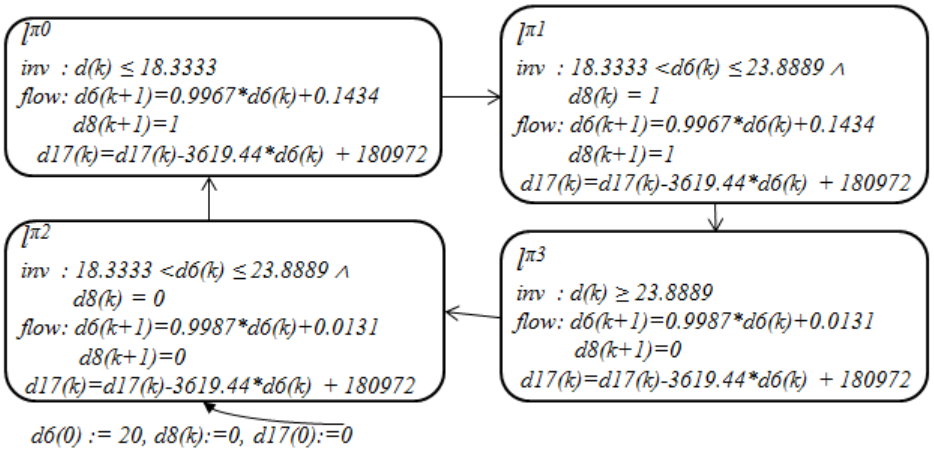

Figure 4.15 Refined CS-HA model for the thermal model of a house in Figure 4.12 from the CS-HA in Figure 4.14. Output-assignment functions of the locations are omitted.

The case study illustrates the benefit of the proposed test generation approach that, instead of exhaustively searching for 4^{220} iterations to obtain the test cases using a search-based test generation approach, the proposed approach resolves the reachability of each c-path by applying one iteration of Algorithm 12 and finds the test cases in four computations. Moreover, no pre-

defined maximum test case length is required for the proposed approach. In contrast, most of the existing tools require a user-defined maximum test case length, and also do not exploit the fact that certain discrete-time flows may be analytically solvable, so multiple time-steps could all be explored at once. Selecting a maximum test case length is adhoc, and an inappropriate selection of maximum test case length may either reduce the test coverage (selected length is smaller than required length) or increase the time for the tools to generate test cases (selected length is larger than required length).

CHAPTER 5 Test Validation and Error Localization

In this chapter, we complete the testing process by introducing the validation methods to validate the model-based tests against the requirements and the requirements-based tests against the model for "fail/pass" obtained from Chapter 3, 4. Further, we develop an error localization approach that uses the failed versus passed tests to locate the errors within the Simulink/Stateflow blocks.

5.1 Test Validation

First unit test for each block is conducted to verify that the output of output-assignment function from I/O-EFA and the output from the generated code of the block is close to each other (within specified tolerance). First, test generation approach in Chapter 3, 4 is performed on the I/O-EFA of the block to generate test cases for the block. The generated test cases are executed on I/O-EFA and the compiled generated code to measure the difference of the outputs. If the outputs are within specified tolerance, the unit test passes; otherwise, test fails.

A test from M-test (resp. R-test) passes/fails if the requirements (resp. the model) accepts/rejects it. We first present the algorithm to validate the cases generated from M-test against the requirements.

Algorithm 16 Given a LTL requirement ϕ of a Simulink/Stateflow model and a M-test t for the Simulink/Stateflow model, t is validated in the following steps.

1. Compute the Büchi automaton $R = (Q, \Gamma, \Xi, Q_0, Q_m)$ accepting ϕ;

2. Simulate R with the test t to check if t is accepted by R, i.e. if starting from $q_0 \in Q_0$, the simulation reaches $q_m \in Q_m$ (q_m is within a strongly connected component);

3. If t is accepted by R, t passes; else, t fails.

Next we present an algorithm to validate a R-test against the model. For this, as a first step, an augmented I/O-EFA is obtained from the I/O-EFA model of the Simulink/Stateflow model and a R-test, as defined in the algorithm below.

Algorithm 17 Given an I/O-EFA $P = (L, D, U, Y, \Sigma, \Delta, L_0, D_0, L_m, E)$ and a test case $t = (u_0, y_0) \dots (u_{|t|-1}, y_{|t|-1})$, its augmented I/O-EFA P^t is a tuple $P^t = (L^t, D, U, Y, \Sigma, \Delta, L_0, D_0, L_m, E^t)$, where

- $L^t = L \cup \{l_f\}$, where the added l_f denotes a faulty location,
- E^t is obtained by replacing each edge $e = (o_e, t_e, \sigma_e, \delta_e, G_e, f_e, h_e) \in E$ by a pair of edges $e^t = (o_e, t_e, \sigma_e, \delta_e, G_e \wedge (y = h_e), f_e, h_e)$, and $e^f = (o_e, l_f, \sigma_e, \delta_e, G_e \wedge (y \neq h_e), f_e, h_e)$.

Note that the augmented I/O-EFA rejects a R-test t if and only if the simulation of the augmented I/O-EFA P^t with the test t leads to the faulty location l_f. Thus the R-tests are validated as in the algorithm below.

Algorithm 18 Given an I/O-EFA model P of a Simulink/Stateflow model and a R-test t, t is validated in the following steps.

1. Apply Algorithm 17 on P and t to obtain the augmented I/O-EFA P^t.
2. Simulate P^t with the test t for $|t| - 1$ time steps to check if P^t reaches l_f.
3. If P^t reaches l_f, t fails; else, t passes.

5.2 Localizing Errors

Recall that a M-test t is a path-sequence in the I/O-EFA model and we let π^t denote its edge sequence. On the other hand, a R-test t is executed in the augmented model P^t as in Algorithm 18 and it traces a sequence of edges which we again denote as π^t. The failing of a test t signifies the manifestation of a fault that resides in a *subsequence* π^f of the edge-sequence π^t executed by t. For notational convenience we denote a subsequence π^f of the edge-sequence π^t executed by t as $\pi^f \preceq \pi^t$. A subsequence $\pi^f \preceq \pi^t$ serves as plausible root cause for the fault witnessed by the test t, called a *fault-seed* as introduced in [29], if for *any* edge-sequence that

contains π^f as a subsequence, a failure is inevitable. [29] showed that the problem of checking whether a subsequence $\pi^f \preceq \pi^t$ of a failed test t is a fault-seed if and only if the following CTL formula is satisfied in $P^f \| R$: $EFm \wedge AG(m \rightarrow AFf)$. Here m is the length of π^f; P^f is a refinement of P that can execute all edge-sequences of P that contain π^f as a subsequence; and R is an automaton accepting all runs of a requirement. The definition of P^f can be found in [29]; the definition of the composition $P^f \| R$ is given below; and the CTL formula itself has the following meaning: "Exists a run that fully executes the subsequence π^f" (as captured by EFm) and "For all runs always if π^f is a subsequence, then for all subsequent runs eventually fault occurs" (as captured by $AG(m \rightarrow AFf)$). The composition of an I/O-EFA P^f and a Büchi automaton R is obtained so as to synchronize each edge of R, which as stated earlier implicitly advances the time counter, with the time-advancement edge e^{time} of P^f. So the time-advancement edge $e^{time} = \{l_m, l_0, -, \{k := k+1\}\}$ of P^f (l_m and l_0 are final and initial locations of P^f and k is time counter), is paired with each edge in edge set Ξ of R (as captured by E^{time} below), and each non time-advancement edge $E - \{e^{time}\}$ of P^f is executed asynchronously in which the state of R does not change (as captured by $E - \{e^{time}\} \times Q$ below). Finally to track the violation of the LTL requirement modeled by R, additional edges are added that steer the composition to a newly added faulty state f (as captured by E^f).

Definition 7 Given a Büchi Automata $R = (Q, \Gamma, \Xi, Q_0, Q_m)$ of a LTL requirement ψ and a I/O-EFA model $P^f = (L, D, U, Y, \Sigma, \Delta, \{l_0\}, D_0, \{l_m\}, E)$, their composition is an I/O-EFA, $P \| R = ((L \times Q) \cup \{f\}, D, U, Y, \Sigma, \Delta, \{l_0\} \times Q_0, D_0, \{l_m\} \times Q_m, E^{\|})$, where

- $E^{\|} = E^{time} \bigcup ((E - \{e^{time}\}) \times Q) \bigcup E^f$ is the set of edges, where
  - $E^{time} = \bigcup_{(q_i, \gamma, q_j) \in \Xi} \{(l_m, q_i), (l_0, q_j), \gamma, \{k := k+1\}\}$ is the set of time-advancement of $P^f \| R$ in which both P^f and R transition synchronously;
  - $(E - e^{time}) \times Q$ is the set of non time-advancement edges of $P^f \| R$ in which only P^f transitions asynchronously;
  - $E^f = \bigcup_{q \in Q} \{(l_m, q), f, [\neg \bigvee_{(q, \gamma, q') \in \Xi} \gamma], -\}$
    is the set of transition edges to the faulty location f.

It can be seen that the above definition creates one copy of the I/O-EFA P^f for each state $q \in Q$ of R in the composition $P^f \| R$. The results computed by fully executing a copy of P^f are used to decide a next edge guard satisfied in R and advancing R along that edge, and also simultaneously passing the control to the next copy of P^f. Note whenever the execution of P^f violates the requirement R, it reaches the faulty location f.

The search of a fault-seed can be enhanced by noting that a fault-seed cannot be a subsequence of a passed test, and so the set of failed tests can be compared with the set of passed tests to narrow down the search for a fault-seed. The overall algorithm for the error localization is as follows.

Algorithm 19 Given a set of failed test T^f, a set of passed test T^p, the I/O-EFA P, and the requirement model R, the fault-seeds are identified as follows.

1. Map each failed/passed test t to its executed sequence of edges π^t.

2. Identify the set of candidate subsequences $\{\pi^f \preceq \pi^t, t \in T^f - T^p\}$ that are subsequences of a failed test but not a passed test.

3. Arrange the candidate subsequences in the order of increasing length, and for each candidate π^f, picked in the order from shortest to longest, do the following:

- obtain the refined P^f as defined in [29, Algorithm 1, Step 5] that can execute all edge-sequences of P that contain π^f as a subsequence;

- form the composition $P^f \| R$ using Definition 7;

- model-check $P^f \| R$ against the CTL formula $EFm \wedge AG(m \rightarrow AFf)$;

- return π^f as fault-seed if and only if the CTL formula is satisfied.

5.3 Mapping Faulty Edges Back to Simulink/Stateflow Diagram

In order to debug the Simulink/Stateflow design, each fault-seed (sequence of edges) is mapped back to the corresponding Simulink/Stateflow blocks to help locate the error. Each edge of a fault-seed is unambiguously mapped to a computation by a block of the underlying Simulink/Stateflow diagram.

Consider the I/O-EFA model of the counter in Figure 2.2. The edge e_2 of the I/O-EFA model corresponds to the computation "reset the counter to the initial output". The computation occurs in the "Enabled Subsystem" block and the related block parameters are "enabling type" and "Initial output". If a fault-seed is $\pi^f = e_2$, the root-cause of the error occurs in the "enabling type" or "Initial output" parameters of the "Enabled Subsystem" block.

Example 18 Suppose the counter in Figure 2.1 erroneously sets the initial output as 2, and suppose the requirement is given by, $\phi = [u \leq 0 \Rightarrow y2 = 0]U[L(u \leq 0)]$, whose acceptor Büchi automaton R is shown in Figure 3.1. By validating the M-tests listed in Table 3.3 against the requirements model R according to Algorithm 16, we can note that the test case $t = \{(0, 2)\}$ fails. The edge sequence that is executed by this test is $\pi^t = e_2e_8e_{10}e_{12}e_{13}e_{20}e_{21}$. By applying Algorithm 19 on the failed test, the fault seed is identified as $\pi^f = e_2 \preceq \pi^t$, which maps to the Simulink/Stateflow block "Enabled Subsystem". Since the edge e_2 references two parameters of this block, "Initial Output" and "Enabling Type", the fault must reside in one of the two parameters. A manual inspection can then pin-point the fault to the erroneous "Initial Condition".

CHAPTER 6 Conclusion and Future Work

6.1 Summary

In this dissertation, we studied and proposed a model-based approach to automated test generation and error localization for Simulink/Stateflow, which overcomes many limitations of the existing approaches and tools. Below summarizes the dissertation.

We presented a translation approach from Stateflow chart to Input/Output Extended Finite Automata (I/O-EFA). A Stateflow state, which is the most basic component of Stateflow chart, is modeled as an atomic model. The composition rules for AND/OR hierarchy are defined to connect the atomic state models. An overall I/O-EFA model is obtained by recursively applying the two composition rules in a bottom-up fashion over the tree structure of the state hierarchy. Rules for further refining the model to incorporate other Stateflow features such as events, historical information, interlevel transitions, etc. have been developed. Finally, the Stateflow model is adapted to resemble a Simulink model, since at the highest level a Stateflow chart is a block in the Simulink library. The size of the translated model is *linear* in the size of the Stateflow chart. Both the Stateflow and Simulink translation approaches have been implemented in an automated translation tool SS2EFA. The translated I/O-EFA models are validated to preserve the discrete behaviors of the original Simulink/Stateflow models. The translated I/O-EFA models can be used for further formal analysis such as verification and test generation.

Further, we presented a model and requirements based test generation approach for Simulink/Sateflow. 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 their feasibility. 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. Requirements-based test generation was also discussed. This was done by modeling each LTL requirement as a Büchi automaton, and selecting all acyclic paths between the initial and final states.

We then presented an improved test generation approach for Simulink/Stateflow extending and enhancing the test generation approach presented in Chapter 3. 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 is decidable for the class of Simulink/Stateflow diagrams possessing a finite late-bisimilar quotient.

Finally, for validation purposes, we presented the approach where model-based tests are validated against the requirements, whereas the requirements-based tests are validated against the model. In both cases, the failed versus passed tests are compared and analyzed to determine a fault-seed, or a plausible root cause. This was further mapped to the Simulink/Stateflow

diagram to identify the plausible faulty blocks and their erroneous parameters.

6.2 Directions for Further Research

This dissertation has laid a foundation for automated verification/validation of Simulink/Stateflow based on extended finite automata, and opens up several avenues for future work in this area.

1. Robustness of Simulink/Stateflow regarding platform inaccuracies: The target platforms where the generated code from Simulink/Stateflow will be deployed on have limited computation power and input-output signals may be perturbed by noise, therefore, the implemented system may perform different behaviors than that should be performed in the Simulink/Stateflow. Simulink/Stateflow model along with the target platform and associated environment can be verified to ensure the implementation preserves the control and data flow of the Simulink/Stateflow model, such that possible failures can be avoided at the early stage of the design process.

2. System-level verification/testing: A cyber-physical system consists three main components: computation component, communication component and physical component. Verification&Validation method should not only consider the verification/testing within each component, but also include the interaction among these components. However, components are modeled separately and isolated from each other when verification/testing is performed. Future directions can attempt to translate the model of each component into a uniform model or propose a compositional approach, where an interface will be designed which can let verification tool of each component to communicate with each other.

3. Concurrency with Simulink/Stateflow: Simulink/Stateflow and the generated code are executed sequentially. When multiple pieces of generated code are executed concurrently on the platform, the concurrent system needs to be verified and tested to ensure its absence of concurrency issues, such as deadlock an starvation. However, since computa-

tions in a concurrent system can interact with each other while they are executing, the number of possible execution paths in the system can be extremely large, which may lead to state explosion problem. Future work can study the problem of concurrency verification/testing with Simulink/Stateflow and the approach to reduce the search space of the verification/testing techniques.

APPENDIX A Finite Bisimulation Quotient

Note that a discrete-time hybrid automaton can be modeled as an 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. So instead of defining the properties of a discrete-time hybrid automaton it suffices to define the properties of an I/O-EFA, as below.

Simulation (resp., late-simulation) and bisimulation (resp., late-bisimulation) relations are defined as follows. For an I/O-EFA P, the notation $(l,d)\xrightarrow{\sigma,\delta,u,y}(l',d')$ implies the existence of $e \in E$ such that $o_e = l, t_e = l', \sigma_e = \sigma, \delta_e = \delta$, $G_e(d,u)$ holds, and $d' = f_e(d,u)$ and $y = h_e(d,u)$.

Definition 8 Given an I/O-EFA P, a *simulation* relation over its states is a binary relation $\Phi \subseteq (L \times D) \times (L \times D)$ such that $((l_1,d_1),(l_2,d_2)) \in \Phi$ implies $\forall e_1, \forall u, \exists e_2 : \sigma_{e_2} = \sigma_{e_1} := \sigma$, and $[(l_1,d_1)\xrightarrow{\sigma,\delta,u,y}(l_1',d_1'), l_1\xrightarrow{e_1}l_1'] \Rightarrow \exists[(l_2,d_2)\xrightarrow{\sigma,\delta,u,y}(l_2',d_2'), l_2\xrightarrow{e_2}l_2']$ s.t. $((l_1',d_1'),(l_2',d_2')) \in \Phi$.

Similarly, a *late-simulation* relation over states of P is a binary relation $\Phi \subseteq (L \times D) \times (L \times D)$ such that $((l_1,d_1),(l_2,d_2)) \in \Phi$ implies $\forall e_1, \exists e_2 : \sigma_{e_2} = \sigma_{e_1} := \sigma$, and $\forall u$, $[(l_1,d_1)\xrightarrow{\sigma,\delta,u,y}(l_1',d_1'), l_1\xrightarrow{e_1}l_1'] \Rightarrow \exists[(l_2,d_2)\xrightarrow{\sigma,\delta,u,y}(l_2',d_2'), l_2\xrightarrow{e_2}l_2']$ s.t. $((l_1',d_1'),(l_2',d_2')) \in \Phi$.

A symmetric simulation (resp., late-simulation) relation is called bisimulation (resp., late-bisimulation) relation. Two systems P_1 and P_2 are said to be bisimilar (resp., late-bisimilar) if there exists a bisimulation (resp., late-bisimulation) relation $\Phi \subseteq (L_1 \times D_1) \times (L_2 \times D_2)$ such that for each $(l_{10},d_{10}) \in L_{10} \times D_{10}$ there exists $(l_{20},d_{20}) \in L_{20} \times D_{20}$ such that $((l_{10},d_{10}),(l_{20},d_{20})) \in \Phi$.

Given a partition of the set of the data and the inputs, one can obtain a quotient system of an I/O-EFA as follows.

Definition 9 Given an I/O-EFA $P = (L, D, U, Y, \Sigma, \Delta, L_0, D_0, L_m, E)$ and a partition $\mathcal{G}$ of $D \times U$ that refines the partition induced by the set of guards of P, the *quotient* of P with respect to the partition $\mathcal{G}$ is $P^{\mathcal{G}} = (L^{\mathcal{G}}, D, U, Y, \Sigma, \Delta, L_0^{\mathcal{G}}, D_0, L^{\mathcal{G}}, E^{\mathcal{G}})$, where

- $L^{\mathcal{G}} = L \times \mathcal{G}$,
- $L_0^{\mathcal{G}} = L_0 \times \mathcal{G}$, and
- $((l, G), (l', G'), \sigma, \delta, G', f, h) \in E^{\mathcal{G}} \Leftrightarrow [\exists (l, l', \sigma, \delta, \overline{G}, f, h) \in E : (G'(d, u) \Rightarrow \overline{G}(d, u))]$.

In other words, each location in P is split into a number of locations, one per partition-cell, and each edge in P is split into a number of edges, one per pair of partition-cells, with the edge-guard in $P^{\mathcal{G}}$ being the same as the successor location's partition-cell (which by definition is stronger than the edge-guard in P).

$P^{\mathcal{G}}$ is called a bisimilar (resp., late-bisimilar) quotient of P if $P^{\mathcal{G}}$ is bisimilar (resp., late-bisimilar) to P.

Bibliography

[1] Simulink/Stateflow, "http://www.mathworks.com/products/simulink/."

[2] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi, "Defining and translating a "safe" subset of simulink/stateflow into lustre," *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pp. 259–268, 2004.

[3] A. Gadkari, S. Mohalik, K. Shashidhar, A. Yeolekar, J. Suresh, and S. Ramesh, "Automatic generation of test-cases using model checking for sl/sf models," *4th International Workshop on Model Driven Engineering, Verification and Validation*, 2007.

[4] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rmmer, and G. Weissenbacher, "Mutation-based test case generation for simulink models," in *Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, F. Boer, M. Bonsangue, S. Hallerstede, and M. Leuschel, Eds. Springer Berlin Heidelberg, 2010, vol. 6286, pp. 208–227.

[5] N. He, P. Rümmer, and D. Kroening, "Test-case generation for embedded simulink via formal concept analysis," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11, New York, NY, USA, 2011, pp. 224–229.

[6] J. Oh, M. Harman, and S. Yoo, "Transition coverage testing for simulink/stateflow models using messy genetic algorithms," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, ser. GECCO '11, New York, NY, USA, 2011, pp. 1851–1858.

[7] P. Peranandam, S. Raviram, M. Satpathy, A. Yeolekar, A. Gadkari, and S. Ramesh, "An integrated test generation tool for enhanced coverage of simulink/stateflow models," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, march 2012, pp. 308 –311.

[8] "Simulink design verifier," Available at: http://www.mathworks.com/products/sldesignverifier/.

[9] D. Bhatt, G. Madl, and K. Schloegel, "Towards scalable veri

cation of commercial avionics software," *Proceedings of the AIAA Infotech@Aerospace Conference*, April 2010.

[10] "Reactis," Available at: http://www.reactive-systems.com/.

[11] "T-vec," Available at: http://www.t-vec.com/.

[12] C. Zhou and R. Kumar, "Semantic translation of simulink diagrams to input/output extended finite automata," *Discrete Event Dynamic Systems*, pp. 1–25, 2010.

[13] M. Li and R. Kumar, "Stateflow to extended finite automata translation," *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pp. 1–6, July 2011.

[14] ——, "Recursive modeling of stateflow as input/output-extended automaton," *IEEE Transactions on Automation Science and Engineering*, 2013.

[15] ——, "Model-based automatic test generation for simulink/stateflow using extended finite automaton," *In proceedings of the eighth IEEE International Conference on Automation Science and Engineering (CASE 2012)*, August 2012.

[16] ——, "Reduction of automated test generation for simulink/stateflow to reachability and its novel resolution," *In proceedings of the eighth IEEE International Conference on Automation Science and Engineering (CASE 2013)*, August 2013.

[17] "Nusmv," Available at: http://nusmv.fbk.eu/.

[18] "Cvx," Available at: http://cvxr.com/cvx/.

[19] M. Y. Vardi and P. Wolper, "Reasoning about infinite computations," *Information and Computation*, vol. 115, pp. 1–37, 1994.

[20] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli, "A decision algorithm for full propositional temporal logic," *Computer Aided Verification*, vol. 697, pp. 97–109, 1993.

[21] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification (PSTV95)*, June 1995.

[22] P. Gastin and D. Oddoux, "Fast ltl to büchi automata translation," *Computer Aided Verification*, vol. 2102, pp. 53–65, 2001.

[23] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*. Elsevier, 1995, pp. 995–1072.

[24] C. Zhou and R. Kumar, "Modeling simulink diagrams using input/output extended finite automata," *Computer Software and Applications Conference, Annual International*, vol. 2, pp. 462–467, 2009.

[25] R. Alur, "Formal verification of hybrid systems," in *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, Oct. 2011, pp. 273–278.

[26] J.-F. Raskin, "Reachability problems for hybrid automata," in *Reachability Problems*, ser. Lecture Notes in Computer Science, G. Delzanno and I. Potapov, Eds. Springer Berlin Heidelberg, 2011, vol. 6945, pp. 28–30.

[27] H. Guéguen, M.-A. Lefebvre, J. Zaytoon, and O. Nasri, "Safety verification and reachability analysis for hybrid systems," *Annual Reviews in Control*, vol. 33, no. 1, pp. 25 – 36, 2009.

[28] C. Zhou and R. Kumar, "Finite bisimulation of reactive untimed infinite state systems modeled as automata with variables," vol. 10, no. 1, Jan. 2013, pp. 160–170.

[29] C. Zhou, R. Kumar, and S. Jiang, "A formal analysis approach of runtime data-log for embedded software-fault localization," *2011 American Control Conference*, June 2011.