Recursive Modeling of Stateflow as Input/Output-Extended Automaton

Meng Li and Ratnesh Kumar
Dept. of Electrical and Computer Engineering, Iowa State University, Ames, IA

Abstract—**Stateflow, a graphical interface tool for Matlab, is a common choice for design of event-driven software and systems. In order for their offline analysis (testing/verification) or online analysis (monitoring), the Stateflow model must be converted to a form that is amenable to formal analysis. In this paper, we present a systematic method, which translates Stateflow into a formal model, called Input/Output Extended Finite Automata (I/O-EFA). The translation method treats each state of the Stateflow model as an atomic module, and applies composition/refinement rules for each feature (such as state-hierarchy, local events) recursively to obtain the entire model. The size of the translated model is linear in the size of the Stateflow chart. Our translation method is sound and complete in the sense that it preserves the discrete behaviors as observed at the sample times. Further, the translation method has been implemented in a Matlab tool, which outputs the translated I/O-EFA model that can itself be simulated in Matlab.**

Note to Practitioners: The paper presents a methodology and tool that enables a formal reasoning about the correctness of a reactive/control software written in Simulink/Stateflow through its translation into an automaton model. The discrete modeling formalism of an automaton extended with input/ouput/state variables suffices, since Simulink/Stateflow is a discrete-time system and the translated model preserves all the discrete-time behaviors. This work complements our earlier work that translated only the time-driven blocks; the event-driven components (Stateflow blocks) are also translated here.

I. 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. Owing to the correctness, safety, security, etc. requirements of such systems, methods to analyze the system designs are needed. Simulink/Stateflow however has originally been designed for the simulation of the designs, and does not provide a model that makes it amenable for formal analysis such as verification and validation.

Semantic translation of embedded control models (such as Simulink/Stateflow) for formal analysis is thus widely investigated. [2] converted a subclass of Simulink/Stateflow diagrams into a semantically equivalent hybrid automaton. [3] derived hybrid automata models from Simulink/Stateflow models to improve simulation coverage. [4] [5] [6] translated discrete-time Simulink diagrams into a synchronous language, called Lustre. [7] presented a translation scheme from Simulink/Stateflow to a model of concurrent processes communicating with FIFO queues or registers, called a SPI model. [8] defined a translation from a visual specification formalism called StateCharts [9] to hierarchical automata. [10] [11] described a discrete translation of hybrid automata for reachability analysis. [12] introduced a translation of a hybrid data-flow formalism to hybrid automata.

In our previous work [13], we introduced a recursive modeling method to translate Simulink diagram 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. This paper completes the modeling approach by extending it to allow the translation of Stateflow charts that are event-driven blocks.

Stateflow, which has been adopted from StateCharts [9], allows hierarchical modeling of discrete behaviors consisting of parallel and exclusive decompositions, with tedious semantics, which makes it challenging to capture and translate into formal models. Several authors have attempted different forms of Stateflow translation. Scaife et al. [14] translated a subset of Simulink/Stateflow into Lustre and verified the model using a model checking tool called Lesar. Gadkari et al. [15] proposed a translation from Simulink/Stateflow to Symbolic Analysis Laboratory (SAL). [16] defined a mapping from Stateflow to pushdown automata. [17] presented a translation framework from Simulink/Stateflow to a specification language called Circus. [18] described a translation of Stateflow diagrams to a formal modeling language called CSP#. However, the prior works have not provided a recursive method for the translation from Stateflow to an automaton, preserving the discrete behaviors (behaviors observed at discrete time steps when the inputs are sampled and the outputs are computed).

In this work, we continue to use I/O-EFA as the target model for translation so as to retain consistency with our previous work that translated 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

The research was supported in part by the National Science Foundation under the grants NSF-ECS-0601570, NSFECCS-0801763, NSF-CCF-0811541, and NSF-ECCS-0926029; email:mengl, rkumar@iastate.edu

Stateflow and Simulink models is introduced to combine them into a single complete model.

We have also implemented our algorithm 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. The contributions of this paper include:

- 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.
- We have developed an automated translation tool 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).

The present paper is an extension of our conference version [19]. The extension includes details of original work that were omitted in conference version, complete set of examples, plus a formal proof of correctness.

II. 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*.

III. 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 1. 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. Each node of the tree can be modeled as an atomic I/O-EFA model, which we describe below in this section.

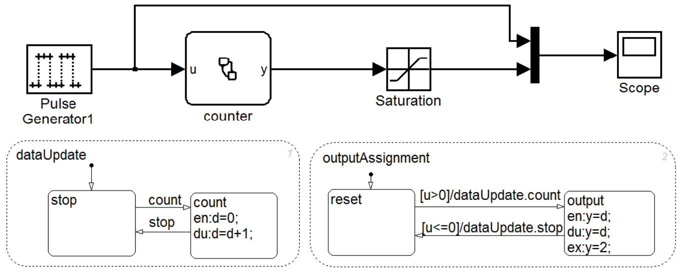


Fig. 1. Simulink diagram of a Counter system (top) and the Stateflow chart of the counter (below)

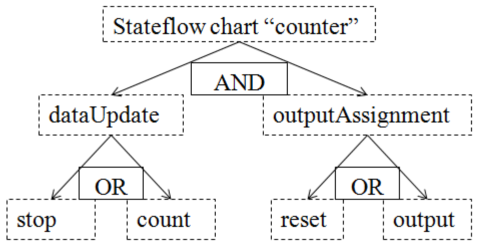
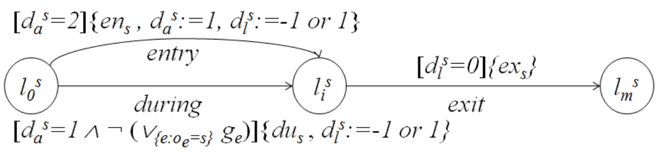


Fig. 2. Hierarchical Structure of Conuter's Stateflow chart

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 3.



$[d_a^s=2]\{en_s, d_a^s:=1, d_l^s:=-1 \text{ or } 1\}$
entry
l_0^s
l_i^s
l_m^s
$[d_l^s=0]\{ex_s\}$
during
exit
$[d_a^s=1 \wedge \neg(\vee_{\{e:o_e=s\}} g_e)]\{du_s, d_l^s:=-1 \text{ or } 1\}$

Fig. 3. Atomic Model 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
- $E^s = \{l_0^s, l_i^s, [d_a^s = 2], \{en_s, d_a^s := 1, d_l^s := -1$ (if s has a substate) or 1 (otherwise)$\}\}$
  $\bigcup\{l_0^s, l_i^s, [d_a^s = 1 \wedge \neg(\bigvee_{\{e:o_e=s\}} g_e)], \{du_s, d_l^s := -1$ (if s 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.

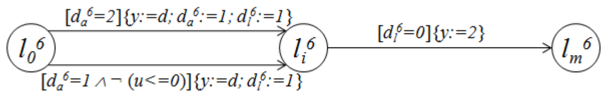
The above atomic model captures a Stateflow state's behavior as follows:

• 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 1: Consider the counter system of Figure 1. Figure 4 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.



$[d_a^6=2]\{y:=d; d_a^6:=1; d_l^6:=1\}$
l_0^6
l_i^6
$[d_l^6=0]\{y:=2\}$
l_m^6
$[d_a^6=1 \wedge \neg(u<=0)]\{y:=d; d_l^6:=1\}$

Fig. 4. Atomic Model for State outputAssignment.output in the Counter

IV. 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 3. 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

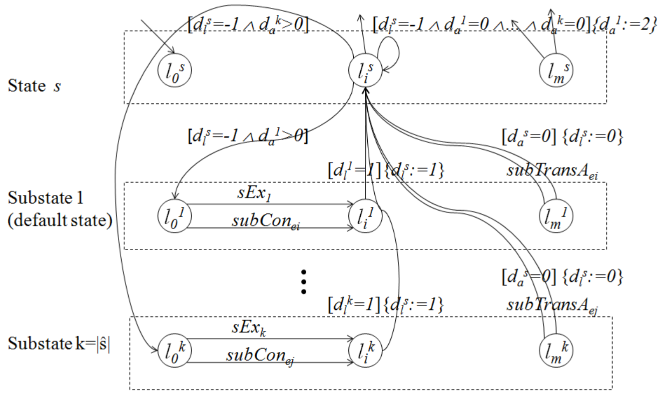
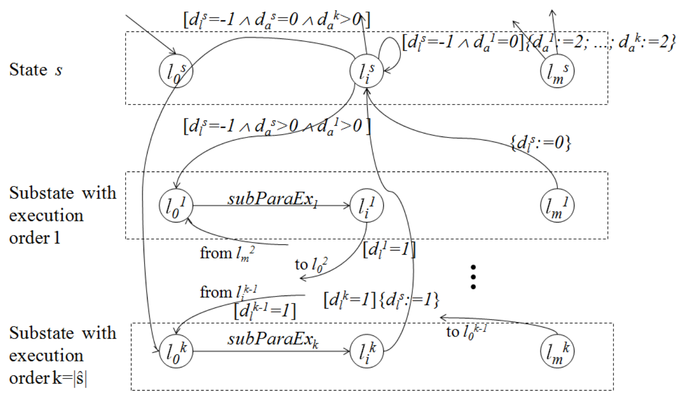


Fig. 5. OR-Complex state modeling. $\forall r \in \widehat{s} - \{s\}$: $sEx_r \equiv [d_a^s = 0]\{d_a^r := 0; d_l^r := -1or0\}$; $\forall e : o_e \in \widehat{s} - \{s\}$: $subCon_e \equiv [d_a^{o_e} = 1 \wedge g_e \wedge d_a^s > 0]\{ca_e; d_a^{o_e} := 0; d^e := 1; d_l^{o_e} := -1or0\}$, and $subTransA_e \equiv [d_a^s > 0 \wedge d^e = 1]\{ta_e; d_a^{t_e} := 2; d^e := 0; d_l^s := -1\}$



State s
Substate with execution order 1
Substate with execution order k=|ŝ|
[d_l^s=-1 ∧ d_a^s=0 ∧ d_a^k>0]
[d_l^s=-1 ∧ d_a^1=0]{d_a^1:=2; ... ; d_a^k:=2}
[d_l^s=-1 ∧ d_a^s>0 ∧ d_a^1>0]
{d_l^s:=0}
subParaEx_1
from l_m^2
to l_0^2
[d_l^1=1]
from l_m^{k-1}
[d_l^{k-1}=1]
[d_l^k=1]{d_l^s:=1}
to l_0^{k-1}
subParaEx_k

Fig. 6. AND-Complex state modeling. $\forall r \in \widehat{s} - \{s\}$: $subParaEx_r \equiv [d_a^r = 1 \wedge d_a^s = 0]\{d_a^r := 0; d_l^r := -1or0\}$

defining the corresponding composition rules. We first introduce a few notation to make the presentation clearer.

Definition 2: 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 $\widehat{s}$.

A. Modeling state with OR-substates

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 5:
  $\bigcup_{r \in \widehat{s}-\{s\}} \{l_i^s, l_0^r, [d_l^s = -1 \wedge d_a^r > 0], -\}$
  $\bigcup \{l_i^s, l_i^s, [d_l^s = -1 \wedge (\bigwedge_{r \in \widehat{s}-\{s\}} (d_a^r = 0)], \{d_a^{s^*} := 2\}\}$
  $\bigcup_{r \in \widehat{s}-\{s\}} \{l_0^r, l_i^r, [d_a^s = 0], \{d_a^r := 0; d_l^r := -1$ (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.

B. Modeling state with AND-substates

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 6:
  $\{l_i^s, l_0^1, [d_l^s = -1 \wedge d_a^s > 0 \wedge d_a^1 > 0], -\}$
  $\bigcup \{l_i^s, l_i^s, [d_l^s = -1 \wedge d_a^1 = 0], \{\forall r \in \widehat{s} - \{s\} : d_a^r := 2\}\}$
  $\bigcup \{l_i^k, l_i^s, [d_l^k = 1], \{d_l^s := 1\}\}$
  $\bigcup_{r \in \widehat{s}-\{s,k\}} \{(l_i^r, l_0^{r+1}, [d_l^r = 1], -\}$
  $\bigcup_{r \in \widehat{s}-\{s\}} \{l_0^r, l_i^r, [d_a^r = 1 \wedge d_a^s = 0], \{d_a^r := 0; d_l^r := 1$ (if s has a substate) or 0 (otherwise)$\}\}$
  $\bigcup \{l_i^s, l_i^k, [d_l^s = -1 \wedge d_a^s = 0 \wedge d_a^k > 0], -\}$
  $\bigcup_{r \in \widehat{s}-\{s,1\}} \{l_m^r, l_0^{r-1}, -, -\}$
  $\bigcup \{l_m^1, l_i^s, -, \{d_l^s := 0\}\}$.

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 2: Consider the counter system of Figure 1. 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 7 and Figure 8 respectively. Next a model for the top-level AND-complex state (the Stateflow chart) is obtained by composing the models of Figures 7 and 8 using the AND-Connecting Rule (Algorithm 3); the result is shown in Figure 9.

V. 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

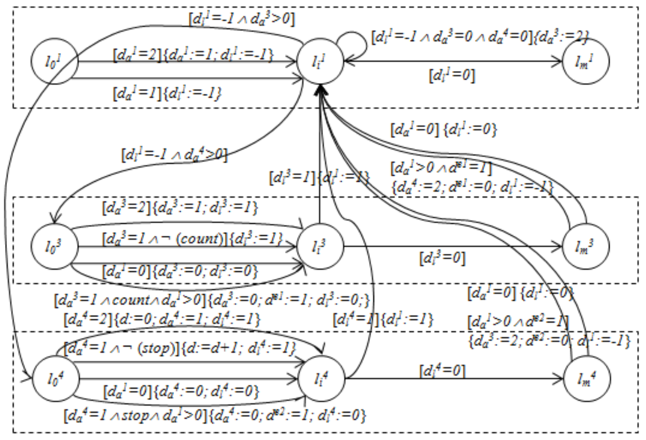


Fig. 7. Modeling of OR-Complex state dataUpdate within Statechart of Counter

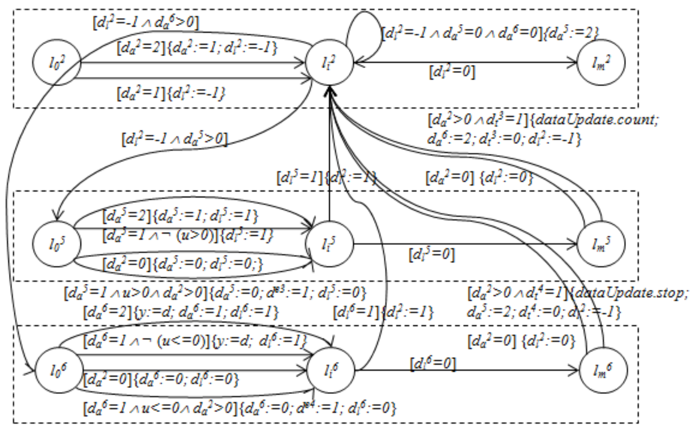


Fig. 8. Modeling of OR-Complex state outputAssignment within Statechart of Counter

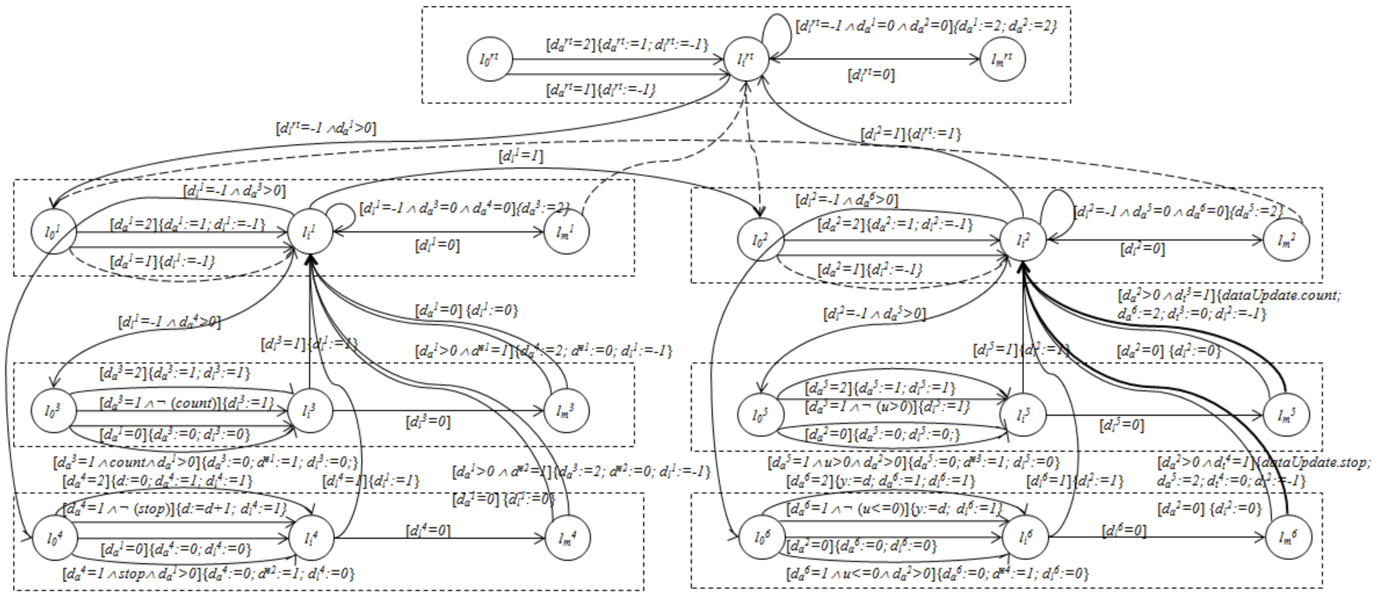


Fig. 9. Modeling state hierarchy within Statechart of Counter; certain places are missing spacings in Fig 5 and 6; the following edges, whose guards are never satisfied, are drawn as dotted lines and their labels are omitted for simplicity: $\{l_i^{rt}, l_0^2, [d_l^{rt} = -1 \wedge d_a^{rt} = 0 \wedge d_a^2 > 0], -\}$, $\{l_0^2, l_i^2, [d_a^2 = 1 \wedge d_a^{rt} = 0], \{d_a^2 := 0; d_l^2 := -1\}\}$, $\{l_m^2, l_0^1, -, -\}$, $\{l_0^1, l_i^1, [d_a^1 = 1 \wedge d_a^{rt} = 0], \{d_a^1 := 0; d_l^1 := -1\}\}$, and $\{l_m^1, l_i^{rt}, -, \{d_l^{rt} := 0\}\}$

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 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 [15], 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 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 10 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 $E - E^{ev}$)..

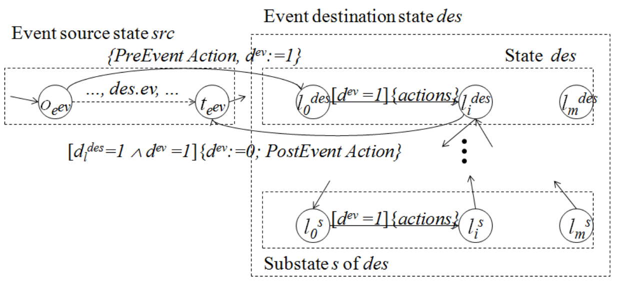


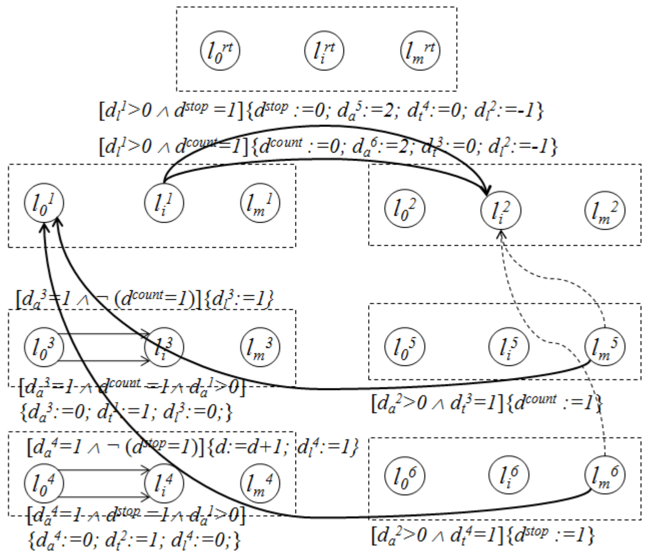
Fig. 10. Modifying the model capturing state hierarchy to also model local events

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 3: Consider the counter system of Figure 1. 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 9 labeled by the events "count" and "stop" respectively, and shown in bold in Figure 9, are broken down into two parts with corresponding parts re-routed as shown in Figure 11, where the re-routed edges are shown in bold and the original edges are drawn in dotted lines.

VI. 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 [13]. Accordingly, the model of a Stateflow chart obtained from applying Algorithms 1-4 is adapted by adding another layer as shown in Figure 12. As is the case with the model of a time-driven Simulink block (see [13] for the details), the final model of a Stateflow



l_0^{rt}
l_i^{rt}
l_m^{rt}
$[d_l^1>0 \wedge d^{stop}=1]\{d^{stop}:=0; d_a^5:=2; d_l^4:=0; d_l^2:=-1\}$
$[d_l^1>0 \wedge d^{count}=1]\{d^{count}:=0; d_a^6:=2; d_l^3:=0; d_l^2:=-1\}$
l_0^1
l_i^1
l_m^1
l_0^2
l_i^2
l_m^2
$[d_a^3=1 \wedge \neg(d^{count}=1)]\{d_l^3:=1\}$
l_0^3
l_i^3
l_m^3
l_0^5
l_i^5
l_m^5
$[d_a^3=1 \wedge d^{count}=1 \wedge d_a^1>0]$
$\{d_a^3:=0; d_l^1:=1; d_l^3:=0;\}$
$[d_a^2>0 \wedge d_l^3=1]\{d^{count}:=1\}$
$[d_a^4=1 \wedge \neg(d^{stop}=1)]\{d:=d+1; d_l^4:=1\}$
l_0^4
l_i^4
l_m^4
l_0^6
l_i^6
l_m^6
$[d_a^4=1 \wedge d^{stop}=1 \wedge d_a^1>0]$
$\{d_a^4:=0; d_l^2:=1; d_l^4:=0;\}$
$[d_a^2>0 \wedge d_l^4=1]\{d^{stop}:=1\}$

Fig. 11. Modification of Fig 9 to capture the local events within Statechart of Counter

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 12.

- 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 12 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.

Example 4: Consider the counter system of Figure 1. 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.

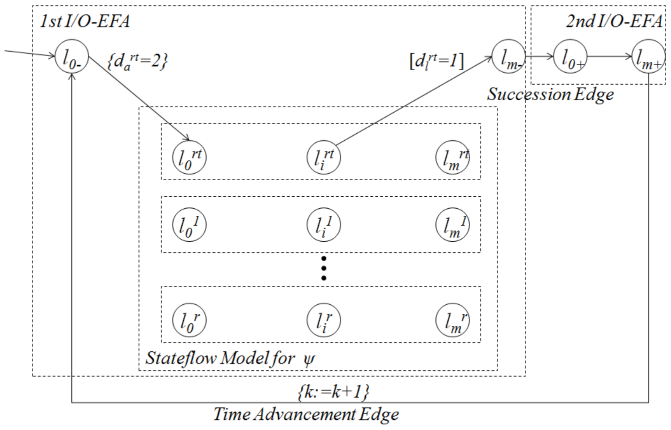


Fig. 12. Finalized model of Stateflow chart

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 9.

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 11.

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 12. 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 [13]. The result is shown in Figure 13.

VII. 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 computation path in the I/O-EFA model of a Stateflow chart.

Definition 3: 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 $\{\pi = e_0^{\pi}...e_{|\pi|-1}^{\pi} \in E^* | o_{e_0^{\pi}} \in L_0, t_{e_{|\pi|-1}^{\pi}} \in L_m, \forall i \in [1, |\pi|-2] : o_{e_i^{\pi}} = t_{e_{i-1}^{\pi}}\}$. We say that a cc-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 3),

- 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 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 5),

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

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 6),

- entering substates is represented by the c-path: $l_0^s \to l_i^s \to l_0^1 \to l_i^1 \to l_0^2 \to ... \to l_i^{k-1} \to l_0^k \to l_i^k \to l_i^s$ (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: $l_0^s \to l_i^s \to l_0^k \to l_i^k \to l_m^k \to l_0^{k-1} \to ... \to l_m^2 \to l_0^1 \to l_i^1 \to l_m^1 \to l_i^s \to l_m^s$ (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. ■

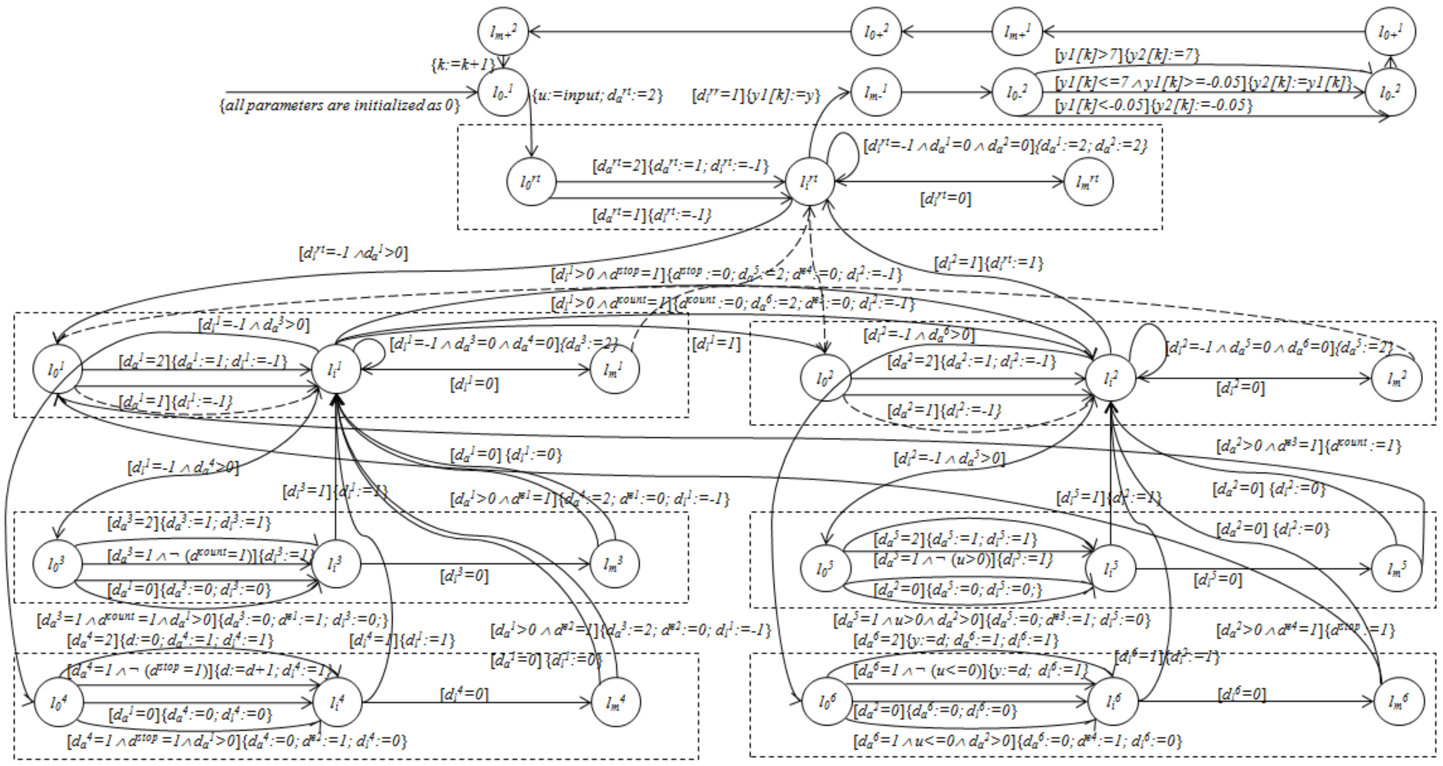


Fig. 13. Finalized complete model of the Statechart Counter for counter system of Figure 1. The following edges, whose guards are never satisfied, are drawn as dotted lines and their labels are omitted for simplicity: $\{l_i^{rt}, l_0^2, [d_l^{rt} = -1 \wedge d_a^{rt} = 0 \wedge d_a^2 > 0], -\}$, $\{l_0^2, l_i^2, [d_a^2 = 1 \wedge d_a^{rt} = 0], \{d_a^2 := 0; d_l^2 := -1\}\}$, $\{l_m^2, l_0^1, -, -\}$, $\{l_0^1, l_i^1, [d_a^1 = 1 \wedge d_a^{rt} = 0], \{d_a^1 := 0; d_l^1 := -1\}\}$, and $\{l_m^1, l_i^{rt}, -, \{d_l^{rt} := 0\}\}$

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 10),

- execution of each event ev in its source state transition $o_{e^{ev}} \rightarrow d_{e^{ev}}$ is modeled by the c-path: $o_{e^{ev}} \rightarrow l_0^{des} \rightarrow l_i^{des} \rightarrow \ldots \rightarrow l_i^{des} \rightarrow t_{e^{ev}}$, 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 12), 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 [13].) ■

VIII. Implementation and Validation

The Stateflow modeling approach described above, together with the Simulink modeling method for time-driven blocks of our previous work [13], 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 5: The simulation result comparison between the I/O-EFA model of the counter (see Figure 13) and the original counter system (see Figure 1) is shown in Figure 14. 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.

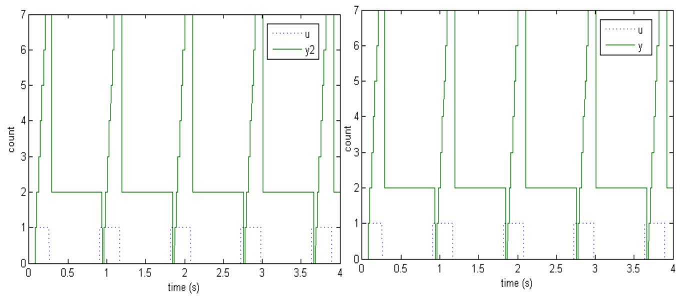


Fig. 14. Simulation to compare the execution of Statechart Counter (left) and its I/O-EFA model of Fig 13 (right)

Example 6: This example is of a servo velocity control system (shown in Figure 15) consisting of a controller, a fault monitor (both written in Stateflow, shown respectively in Figure 16 and Figure 17), and a motor (written in Simulink, shown in Figure 18). There 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 19) is identical to the discrete behaviors of the original Simulink/Stateflow model.

IX. Conclusion

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. Generally, for test generation, certain set of the computational paths within one time step are extracted according to the coverage criterion, and checked for their feasibility. The input sequences that can achieve the feasible paths are the test cases. We have discussed these in [20].

References

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

[2] A. Agrawal, G. Simon, and G. Karsai, "Semantic translation of simulink/stateflow models to hybrid automata using graph transformations," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 109, pp. 43–56, December 2004.

[3] R. Alur, A. Kanade, S. Ramesh, and K. C. Shashidhar, "Symbolic analysis for improving simulation coverage of simulink/stateflow models," *In EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pp. 89–98, 2008.

[4] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, "Translating discrete-time simulink to lustre," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 4, no. 4, pp. 779–818, 2005.

[5] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From simulink to scade/lustre to tta: a layered approach for distributed embedded applications," *In LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, vol. 38, no. 7, pp. 153–162, June 2003.

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

[7] M. Jersak, D. Ziegenbein, F. Wolf, K. Richter, R. Ernst, F. Cieslok, J. Teich, K. Strehl, and L. Thiele, "Embedded system design using the spi workbench," *In Proc. of the 3rd International Forum on Design Languages*, 2000.

[8] E. Mikk, Y. Lakhnechi, and M. Siegel, "Hierarchical automata as model for statecharts," *Advances in Computing Science - ASIAN'97*, vol. 1345, pp. 181–196, 1997.

[9] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231 – 274, 1987.

[10] S. Jiang, "Reachability analysis of linear hybrid automata by using counterexample fragment based abstraction refinement," *American Control Conference, 2007. ACC '07*, pp. 4172 –4177, July 2007.

[11] R. Kumar, C. Zhou, and S. Jiang, "Safety and transition-structure preserving abstraction of hybrid systems with inputs/outputs," *2008 Workshop on Discrete Event Systems*, pp. 206–211, May 2008.

[12] P. Schrammel and B. Jeannet, "From hybrid data-flow languages to hybrid automata: a complete translation," *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, pp. 167–176, 2012.

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

[14] 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.

[15] 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.

[16] A. Tiwari, "Formal semantics and analysis methods for simulink stateflow models," *Technical report, SRI International*, 2002.

[17] A. Miyazawa and A. Cavalcanti, "Refinement-oriented models of stateflow charts," *Science of Computer Programming*, vol. 77, pp. 1151–1177, September 2012.

[18] C. Chen, J. Sun, Y. Liu, J. Dong, and M. Zheng, "Formal modeling and validation of stateflow diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 14, pp. 653–671, 2012.

[19] 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.

[20] ——, "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.

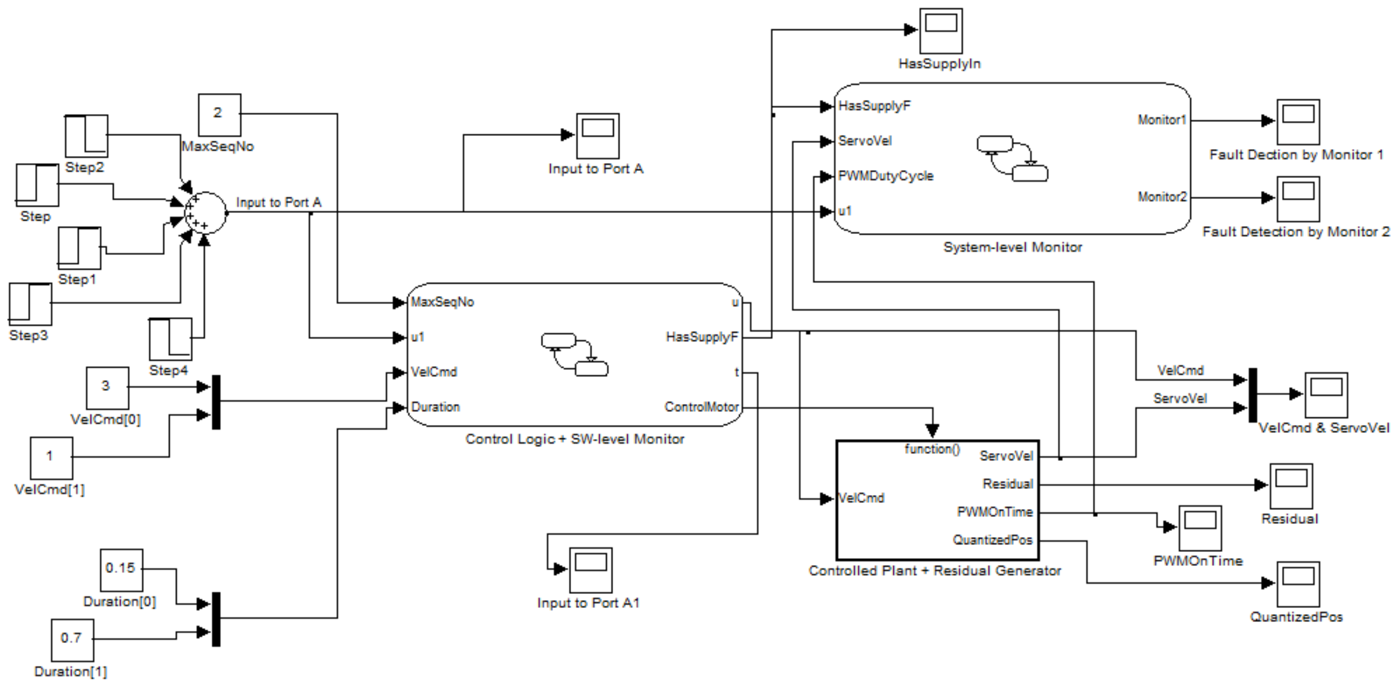


Fig. 15. High level Simulink/Stateflow model of servo velocity control

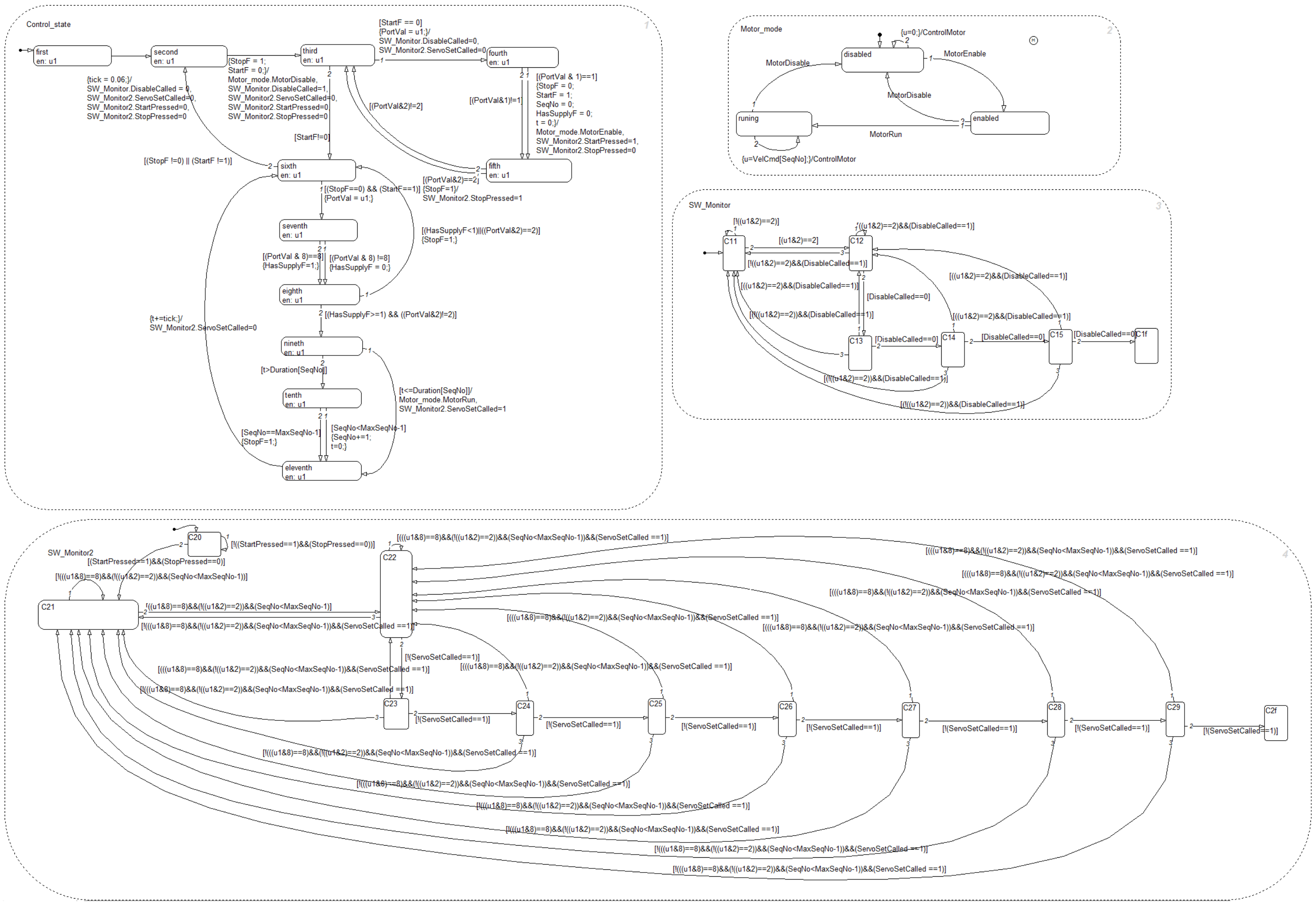


Fig. 16. Controller Stateflow chart (named “Counter Logic + SW-level Monitor”) of servo velocity control

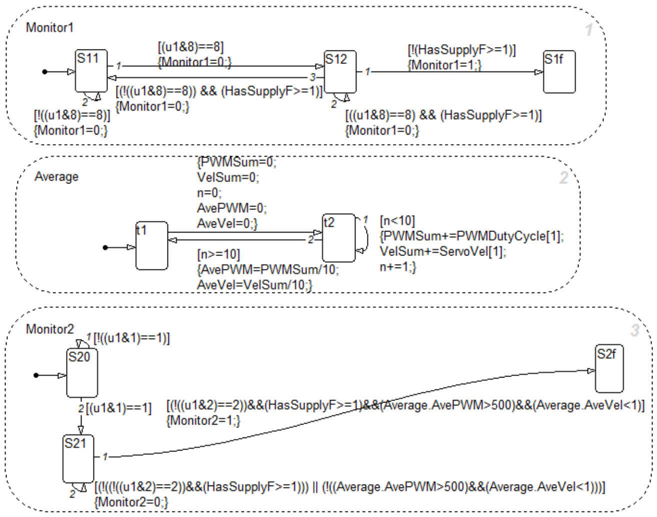


Fig. 17. Fault monitor Stateflow chart (named “System-level Monitor”) of servo velocity control

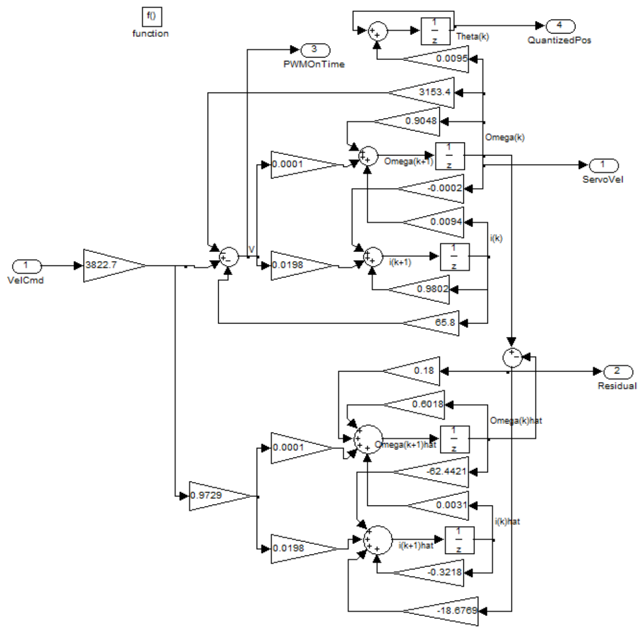


Fig. 18. Motor Subsystem (named “Controlled Plant + Residual Generator”) of servo velocity control

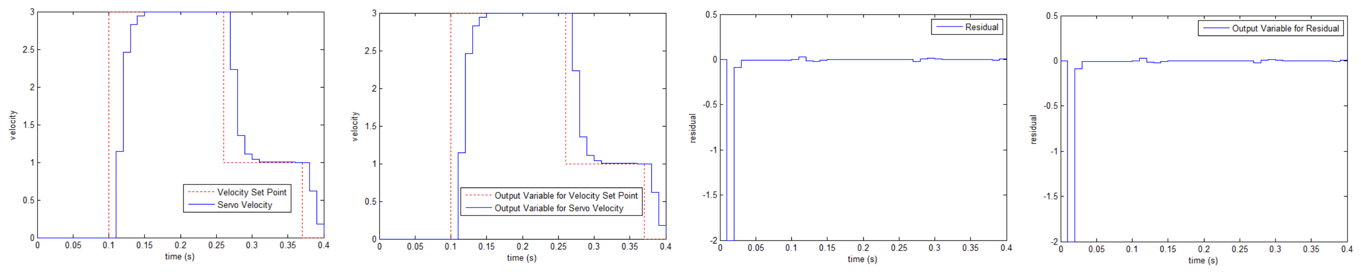


Fig. 19. Simulation results for the velocity set point and actual servo velocity and residue; first (resp., second) figure is for the set point and actual servo velocity of the original Simulink/Stateflow model of servo system (resp., translated I/O-EFA model); third (resp., fourth) figure is for the residue of the original Simulink/Stateflow model of servo system (resp., translated I/O-EFA model)