

An Operational Semantics for Stateflow[★]

Grégoire Hamon and John Rushby

¹ Chalmers Institute of Technology, Göteborg, Sweden

² Computer Science Laboratory, SRI International, Menlo Park CA, USA

Received: date / Revised version: date

Abstract. We present a formal operational semantics for Stateflow, the graphical Statecharts-like language of the Matlab/Simulink tool suite that is widely used in model-based development of embedded systems. Stateflow has many tricky features but our operational treatment yields a surprisingly simple semantics for the subset that is generally recommended for industrial applications. We have validated our semantics by developing an interpreter that allows us to compare its behavior against the Matlab simulator. We have used the semantics as a foundation for developing prototype tools for formal analysis of Stateflow designs.

1 Introduction

The design process for embedded systems has changed dramatically over the last few years. Increasingly, designers use *model-based development environments*; these allow the system, including its software, the plant that it will control, and the environment in which it will operate, to be represented in graphical form at a high level of abstraction. Model-based development environments provide extensive tools for validation through simulation, and code generators that can compile an executable controller from its graphical representation. One of the most widely used environments of this kind is the Matlab suite from Mathworks which, with more than 500,000 licensees, is widespread throughout aerospace, automotive, and several other industries, and ubiquitous in engineering education.

Stateflow is a component of the Simulink graphical language used in Matlab. It combines hierarchical

state-machine diagrams of the kind introduced by Statecharts [10] with traditional flowchart diagrams. Stateflow is generally used to specify the discrete controller (i.e., the software) in the model of a hybrid system where the continuous dynamics (i.e., the behavior of the plant and environment) are specified using other capabilities of Simulink. As part of the Matlab tool suite, Stateflow inherits all its simulation and code generation capabilities.

The evolution of industrial practice toward model-based development has been driven by the growing number of embedded systems, and their increasing complexity. Alongside these developments has been an increase in the criticality of embedded systems, with regard to both human safety (e.g., fly-by-wire control systems) and the cost of faults (e.g., systems deployed in huge quantities in automobiles and domestic appliances). This increasing criticality creates a need for improved methods of analysis and verification and provides an opportunity for formal methods. Formal methods can provide tools to check properties of a design and they can also apply a computational procedure, such as generation of test cases, systematically and automatically, to all parts of a design. However, notations like Stateflow were not built with formal methods in mind, and do not appear to be well suited to formalization.

1.1 Understanding Stateflow

Stateflow is a complex language (its User's Guide [12] is 896 pages long) with numerous, complicated, and often overlapping features lacking any formal definition. Its documentation [12, Chapter 4] describes the semantics in informal operational terms supported by many examples, but the actual definition of the language is the "simulation semantics" given by its behavior when simulated in the Matlab environment. Proposing formal

[★] This material is based on work performed at SRI and supported by the National Science Foundation under Grant No. CCR-0086096 through the University of Illinois and by NASA Langley Research Center under Contract NAS1-00079.

tools for Stateflow requires first giving it a formal definition.

This complexity of the language can be seen as an obstacle to formalization. On the other hand, it makes the need for tools to help programmers clearly visible, and users of the language are asking for them. For example, a Stateflow program can fail with a runtime exception for any of several reasons, and it is desirable to be able to avoid such failures, or at least be able to detect when a program may be vulnerable to them. One popular way to do this is to rely on programming guidelines [3, 5] that restrict the language to a safe subset. But these guidelines have no more formal basis than the language itself and are based on experience. Precisely identifying the reasons for runtime errors or, dually, giving a formal definition for a safe subset, would allow development of static analysis tools that could guarantee their absence.

1.2 A framework for formal tools

In this work, we propose a formalization of Stateflow that can be used as a starting point for the definition of formal tools. Thus, we choose not to idealize the language but to follow strictly the simulation semantics given by the Mathworks documentation and tools, even in its shortcomings. The main result lies in understanding that although Stateflow is superficially similar to other statechart notations, it is in truth a sequential imperative language. As such, the problems arising when formalizing the language are different in nature than those for other statechart languages, and different solutions are required. We use a formal operational semantics as it precisely captures the order of execution of the different components of a Stateflow chart and is able to express the more complicated features of the language.

We have used this formalization in the development of several tools for Stateflow; it provides a detailed understanding of the language, and readily supports the construction of static analyzers and translation to formal tools such as model checkers.

1.3 Overview of the paper

We first introduce Stateflow through an example. Section 3 then develops a formalization for a subset of the language (the subset is comparable to that generally recommended for industrial applications) and gives it an operational semantics. We describe how this semantics has been used to develop prototype static analysis, model checking, and test generation tools for Stateflow. In Section 4, we outline a recently developed denotational semantics for Stateflow that complements the operational semantics described here. Finally, in Section 5 we compare our approach with related work and suggest directions for further research.

2 Introduction to Stateflow

The Stateflow language provides hierarchical state machines, similar to those of Statecharts (although these two languages give different semantics to the state machines). It includes complicated features like interlevel transitions, complex transitions through junctions (these appear as small circles in Stateflow diagrams such as Figure 1) and event broadcasting. Stateflow also provides flowcharts, which are specified using internal transitions leading to terminal junctions. Describing the whole language is beyond the scope of this paper, so we present here a simple example program that includes both kinds of notation and sketch its execution.

2.1 A stopwatch in Stateflow

Figure 1 presents the Stateflow specification for a stopwatch with lap time measurement. This stopwatch contains a counter represented by three variables (`min`, `sec`, `cent`) and a display, also represented as three variables (`disp_min`, `disp_sec`, `disp_cent`).

The stopwatch is controlled by two command buttons, `START` and `LAP`. The `START` button switches the time counter on and off; the `LAP` button freezes the display to show the lap time when the counter is running and resets the counter when the counter is stopped. This behavior can be modeled as four exclusive states:

- **Reset**: the counter is stopped. A `LAP` event resets the counter and the display, while `START` causes a transition to the **Running** state.
- **Lap_Stop**: the counter is stopped. A `LAP` event causes a transition to the **Reset** state, while `START` causes a transition to the **Lap** state.
- **Running**: the counter is running, and the display updated. A `START` event causes a transition to the **Stop** state, while `LAP` causes a transition to the **Lap** state.
- **Lap**: the counter is running, but the display is not updated, thus showing the last value it received. Receiving a `START` event causes a transition to **Lap_Stop**, while `LAP` causes a transition to **Running**.

These four states are here grouped by pairs inside two main states: **Run** and **Stop**, active respectively when the counter is counting or stopped. The counter itself is specified within the **Run** state as a flowchart, incrementing its value every time a clock `TIC` is received (every 1/100s).

2.2 Executing the stopwatch chart

A Stateflow chart always has one active state. Executing the chart means executing the active state (which may then make another state active) each time an event occurs in the environment. Events here are either an action on one of the buttons (`START` or `LAP`) or a clock tick (`TIC`). Executing the active state is done in three steps:

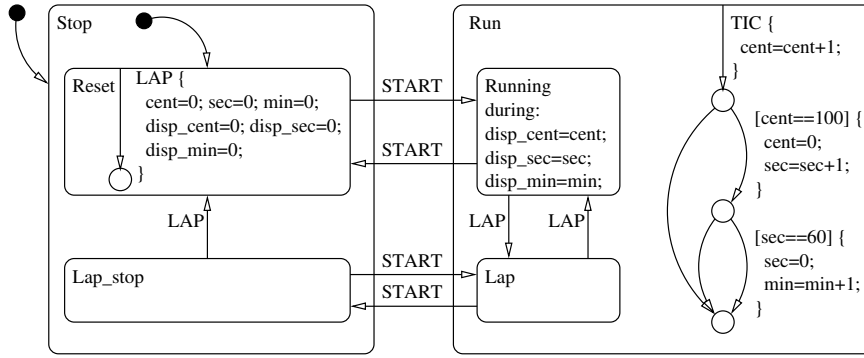


Fig. 1. A simple stopwatch in Stateflow

1. See if a transition leaving the state can be taken, else goto step 2.
2. Execute internal actions (internal transitions, then **during** actions).
3. Execute any internal state that is active.

Transitions can be guarded by events or conditions or both, and they can trigger actions. The internal transition in state **Reset** for example is guarded by the **LAP** event and triggers a series of actions reinitializing the counter and the display. Supposing that the **Run** state is active, with the **Running** substate active, receiving the **START** event will trigger the following sequence of reactions:

- there is no transition leaving the state (the transitions guarded by **START** belong to its substates),
- the flowchart is executed, but is guarded by **TIC**, thus does nothing,
- the active substate is executed, it has a transition which can be fired, leading to **Reset**, itself a substate of **Stop**; **Running** and then **Run** are exited, and **Stop** and then **Reset** are entered.

This step is completed, and execution will continue from the newly active state next time an event is received from the environment.

The model contains a flowchart that implements the counter. Flowcharts are described using transitions between junctions. Unlike states, a junction is exited instantaneously when entered, and the flowchart executes until a terminal junction (one without outgoing transitions) is reached, or all paths have failed. Backtracking can occur if a wrong path is tried. In our example, the flowchart is guarded by the **TIC** event. If activated by this event, the **cent** variable is incremented and the first junction reached. Two transitions leave it, and the guarded one is always executed first. If **cent** is equal to 100, the guarded transition is taken, **cent** is reset to 0 and **sec** incremented, the second junction is reached, and execution continues. If **cent** is not equal to 100, the guarded transition fails, the unguarded one is tried and, being unguarded, succeeds, leading to the third junction, which is terminal, so execution ends.

This short example does not present all Stateflow features, but it introduces hierarchical states, inter-level transitions, and mixed design with statecharts and flowcharts. Our informal description of the execution of this example is actually close to the presentation of the language's semantics in its documentation.

3 Formalizing Stateflow

Studying the language, we came to realize that, although superficially similar to other statechart notations, Stateflow greatly differs from them. In particular, all possibilities of non-determinism are avoided by relying on strict ordering rules, and the scheduling between concurrent components is always statically known. Thus, we decided to consider Stateflow as an imperative language, and to use a structural operational semantics (SOS) [15], which is well-adapted to the description of such languages. This semantics is efficient in dealing with the complexity of Stateflow, which lies in the intricacy of its constructions, not in concurrency or non-determinism.

3.1 A Stateflow subset

We now introduce a linear language that encodes a strict subset of Stateflow. This language eliminates some difficulties of the graphical notation, by making the order between components explicit. We then give this language a formal semantics.

The language is presented in Figure 2. Its basic components (reading from the bottom) are states s , junctions j , events e , actions a , and conditions c . We also define active states s_a (which may be nothing or a state), transition events e_t (nothing or an event), and paths (lists of states).

Transitions t are guarded by a transition event and a condition, can execute two actions and go to a destination d (either a path or a junction). The first action is executed as soon as the transition is valid; the second one is executed only if taking the transition leads somewhere.

composition	$C = \text{Or}(s_a, p, T, SD) \mid \text{And}(b, SD)$
state definition	$sd = ((a, a, a), C, T_i, T_o, J)$
state definition list	$SD = \{s_0 : sd_0; \dots; s_n : sd_n\}$
junction definition list	$J = \{j_0 : T_0; \dots; j_n : T_n\}$
transition	$t = (e_t, c, a, a, d)$
transition list	$T = \emptyset_T \mid t.T$
state s	active state $s_a = \emptyset_s \mid s$
junction j	
path $p = \emptyset_p \mid s.p$	destination $d = p \mid j$
event e	transition event $e_t = \emptyset_e \mid e$
action a	condition c

Fig. 2. The language

Transitions are grouped into lists T . Junction definition lists J associate lists of transitions with junctions. State definition lists SD associate state definitions sd with states. A state definition is a triplet of actions, executed respectively upon entering, during, and exiting the state, an internal composition, a list of inner transitions, a list of outgoing transitions, and a junction definition list. State definitions will be written (A, C, nT_i, T_o, J) , with A representing a triplet of actions: the **entering**, **during**, and **exit** actions, which will be noted respectively as $A.e$, $A.d$, and $A.x$.

Finally, a composition C is a composition of states, and is either an **And** or an **Or** composition. An **And** composition is defined by a Boolean (*true* if the composition is active) and a state definition list. An **Or** composition is an active state, a path, a set of default transitions, and a state definition list.

We note the following properties of this language.

- Actions a and conditions c are expressions of the *action language*, which is distinct from Stateflow itself; we keep this distinction here. The action language is a very simple imperative language. For the same reason we do not have variables here, they are part of the action language, not of Stateflow itself.
- Transition lists T and state definition lists SD are ordered, and their order is significant. When using the graphical representation of a program, the order is determined by the position of the components on the chart: states are ordered top to bottom and then left to right. Transitions are ordered following the *12 o'clock rule*: they are first ordered using a partial ordering on the form of their guards (transitions guarded by an event are evaluated before transitions guarded only by a condition, and unguarded transitions come last), and when this ordering fails, they are ordered by following their source clockwise starting from a 12 o'clock position.

To illustrate the language, the **Stop** state from the stopwatch is shown in Figure 3. The \diamond symbol represents both an empty action and an empty condition. The name j corresponds to the terminal junction found in state **Reset** (junctions being anonymous in Stateflow, they are given unique ids during the translation).

We see here that **Stop** is a state, containing an **Or** composition made from states **Reset** and **Lap_Stop**. **Reset** contains an internal transition guarded by **LAP** and a transition guarded by **START** going to state **Run.Running**; **Lap_Stop** contains two transitions, one guarded by **LAP** going to state **Stop.Reset**, the other guarded by **START** going to state **Run.Lap**.

3.2 Operational semantics

Executing a Stateflow program consists, on each (discrete) step, in processing an input event through the program. This processing can modify the value of variables in the environment, raise output events, and change the program itself as it may change the active states if transitions occur.

We propose here an SOS semantics for the language. This semantics precisely expresses the sequence of actions involved in processing an event through a chart. It is made of rules with the following general form:

$$e, D \vdash P \xrightarrow{D'} P', tv$$

Processing an event e in an environment D (a binding of variables to values) through a program component P produces a new environment D' , a new program P' , and a transition value tv . P here can denote any syntactic class of the language. Transition values tv are used for communication between different parts of the chart. The rules for some of the particular syntactic classes given below extend and slightly differ from this general form.

```

Stop: ((◊, ◊, ◊),
  Or(◊s, Stop, (◊e, ◊, ◊, ◊, Stop.Reset).◊T,
    { Reset: ((◊, ◊, ◊), Or(◊s, Stop.Reset, ◊t, { }),
      (LAP, ◊, (cent ← 0; sec ← 0; min ← 0; disp_cent ← 0; disp_sec ← 0; disp_min ← 0), ◊, j).◊T,
      (START, ◊, ◊, ◊, Run.Running). ◊T, {j : ◊T }));
    Lap_stop: ((◊, ◊, ◊), Or(◊s, Stop.Lap_stop, ◊T, { }), ◊T,
      (LAP, ◊, ◊, ◊, Stop.Reset).
      (START, ◊, ◊, ◊, Run.Lap). ◊T, { }
    )
  ),
  ◊T, ◊T, { })

```

Fig. 3. The Stop state from the stopwatch in the linear language

Definition 1 (Environment D). Environments D contain bindings from variables of the action language to values and the list of output events that are raised in the current instant.

$$D ::= [x_0 : v_0; \dots; x_n : v_n; e_0; \dots e_k]$$

Definition 2 (Transition value tv). Transition values indicate if a transition has fired or not. If no transition has fired, two distinct values, **End** and **No**, are necessary to distinguish a failing transition from the final transition of a flowchart. If a transition has fired, we keep track of its destination and of an eventual pending action.

$$tv ::= \text{Fire}(d, a) \mid \text{End} \mid \text{No}$$

We abstract from the details of the action language and assume that we have semantic rules for actions and conditions such that

$$e, D \vdash a \leftrightarrow D' \quad e, D \vdash c \rightarrow b$$

Evaluating an action when processing event e in environment D produces a new environment D' ; evaluating a condition when processing e in D produces a Boolean value b .

We now present the semantic rules for the different syntactic classes. For compactness, we detail only the rules for transitions, transition lists and parallel compositions; the remaining rules are available in the appendix.

Transitions (Figure 4) – A transition (e_0, c, a_c, a_t, d) fires to destination d if e_0 corresponds to the processed event e or is empty, and if the condition c is *true*. In this case, the action a_c is immediately executed and a_t is left pending in the returned value (rule *t-FIRE*). If e_0 is different from the processed event and is not empty (rule *t-NO₁*) or if the condition is *false* (rule *t-NO₂*), the transition fails and returns **No**.

Transition lists (Figure 5) – Lists of transitions, together with junctions, are used to model both flowcharts and complex transitions between states. The important point here is that a list of transitions is processed sequentially and the first transition that can fire is taken, as shown by rules *T-No* and *T-FIRE*.

If a transition fires to a junction, the list of transitions associated with this junction needs to be processed: evaluation continues instantaneously when reaching a transition. This goes on until we fire to a path (rule *T-FIRE-J-F*), we reach a terminal junction (rule *T-END*) or we fail, in which case we have to back-track and try the next transition in our first list (rule *T-FIRE-J-N*).

State definitions (Figure 8 in the Appendix)

– The rules exhibit the order of execution. Different rules are necessary for entering, executing, and exiting a state. When executing a state, outgoing transitions are tested first; if they fail, the *during* code is executed, then the internal transitions and then the internal composition. If the composition fires, the transition actions are executed followed by the exit code. If the outgoing transitions fire, the transition actions are executed, the internal composition exited, and the exit code executed.

Or compositions (Figures 9 and 10 in the Appendix) – Rules for Or compositions take care of the control changes between states and handle interlevel transitions. The currently active state is executed. If this state fires, either it fires to one of its siblings, in which case this sibling is entered and becomes the active state, or it fires outside of the composition.

And compositions (Figure 6) – Executing an And composition consists in sequentially entering, executing, or exiting all its parallel substates, each state being executed in the environment returned by the execution of its predecessor. It is important to notice here that the parallel construction is in fact completely sequential, and the order of execution statically known: none of the problems associated with concurrency appears here.

3.3 Supporting local events

We now extend our treatment to include one of the trickiest features of Stateflow, the local events mechanism, which the preceding semantics does not consider. This mechanism allows actions to send an event to a state;

$$\begin{array}{c}
\text{\textit{t-FIRE}} \\
\frac{(e = e_0) \vee (e_0 = \emptyset_e) \quad e, D \vdash c \rightarrow \text{true} \quad e, D \vdash a_c \hookrightarrow D'}{e, D \vdash (e_0, c, a_c, a_t, d) \xrightarrow{D'} \text{Fire}(d, a_t)} \\
\\
\text{\textit{t-NO}_1} \\
\frac{(e \neq e_0) \wedge (e_0 \neq \emptyset_e)}{e, D \vdash (e_0, c, a_c, a_t, d) \xrightarrow{D} \text{No}} \\
\\
\text{\textit{t-NO}_2} \\
\frac{(e = e_0) \vee (e_0 = \emptyset_e) \quad e, D \vdash c \rightarrow \text{false}}{e, D \vdash (e_0, c, a_c, a_t, d) \xrightarrow{D} \text{No}}
\end{array}$$

Fig. 4. Rules for transitions t

$$\begin{array}{c}
\text{\textit{\emptyset}_t} \\
\frac{\emptyset_t}{e, D, J \vdash \emptyset_t \xrightarrow{D} \text{End}} \\
\\
\text{\textit{T-FIRE}} \\
\frac{e, D \vdash t \xrightarrow{D'} \text{Fire}(p, \text{act})}{e, D, J \vdash t.T \xrightarrow{D'} \text{Fire}(p, \text{act})} \\
\\
\text{\textit{T-NO-LAST}} \\
\frac{e, D \vdash t \xrightarrow{D_1} \text{No}}{e, D, J \vdash t.\emptyset_T \xrightarrow{D_1} \text{No}} \\
\\
\text{\textit{T-No}} \\
\frac{T \neq \emptyset_T \quad e, D \vdash t \xrightarrow{D_1} \text{No} \quad e, D_1, J \vdash T \xrightarrow{D_2} \text{tv}}{e, D, J \vdash t.T \xrightarrow{D_2} \text{tv}} \\
\\
\text{\textit{T-FIRE-J-F}} \\
\frac{e, D \vdash t \xrightarrow{D_1} \text{Fire}(j, a_1) \quad e, D_1, J[j : T_j] \vdash T_j \xrightarrow{D_2} \text{Fire}(p, a_2)}{e, D, J[j : T_j] \vdash t.T \xrightarrow{D_2} \text{Fire}(p, a_1; a_2)} \\
\\
\text{\textit{T-END}} \\
\frac{e, D \vdash t \xrightarrow{D_1} \text{Fire}(j, a_1) \quad e, D_1, J[j : T_j] \vdash T_j \xrightarrow{D_2} \text{End}}{e, D, J[j : T_j] \vdash t.T \xrightarrow{D_2} \text{End}} \\
\\
\text{\textit{T-FIRE-J-N}} \\
\frac{e, D \vdash t \xrightarrow{D_1} \text{Fire}(j, a_1) \quad e, D_1, J[j : T_j] \vdash T_j \xrightarrow{D_2} \text{No} \quad e, D_2, J[j : T_j] \vdash T \xrightarrow{D_3} \text{tv}}{e, D, J[j : T_j] \vdash t.T \xrightarrow{D_3} \text{tv}}
\end{array}$$

Fig. 5. Rules for transition list T

$$\begin{array}{c}
\text{\textit{AND}} \\
\frac{(tv = \text{No}) \vee (tv = \text{End}) \quad \forall i \in [0..n], \quad e, D_i, J \vdash sd_i \xrightarrow{D_{i+1}} sd'_i, \text{No}}{e, D_0, J, tv \vdash \text{And}(\{s_0 : sd_0; \dots; s_n : sd_n\}) \xrightarrow{D_{n+1}} \text{And}(\{s_0 : sd'_0; \dots; s_n : sd'_n\}), \text{No}} \\
\\
\text{\textit{AND-INIT}} \\
\frac{p_j = p \quad \forall i \neq j, p_i = \emptyset_p \quad \forall i \in [0..n], \quad e, D_i, p_i \vdash sd_i \uparrow^{D_{i+1}} sd'_i}{e, D_0, s_j.p \vdash \text{And}(\{s_0 : sd_0; \dots; s_n : sd_n\}) \uparrow^{D_{n+1}} \text{And}(\{s_0 : sd'_0; \dots; s_n : sd'_n\})} \\
\\
\text{\textit{AND-EXIT}} \\
\frac{\forall i \in [0..n], \quad e, D_i \vdash sd_{n-i} \downarrow^{D_{i+1}} sd'_{n-i}}{e, D_0 \vdash \text{And}(\{s_0 : sd_0; \dots; s_n : sd_n\}) \downarrow^{D_{n+1}} \text{And}(\{s_0 : sd'_0; \dots; s_n : sd'_n\})}
\end{array}$$

Fig. 6. Rule for AND compositions

when this occurs, the current processing is interrupted while the sent event is processed through the receiving state. The receiving state acts here as a function, the action of sending it an event being the function call. However, this mechanism also introduces some complicated cases and fully supporting it in the general case appears difficult. We exhibit a restricted form of this mechanism that is both expressive and supports a simple semantics.

We first try to extend our semantics with a simple interpretation of local events. The action $\mathbf{send}(e, s)$ sends event e to a named state s (broadcasting an event to the whole chart consists in sending an event to the **main** state). Its behavior can be expressed by the following rule:

$$\text{SEND} \quad \frac{e', D[s : sd], \emptyset_J \vdash sd \xrightarrow{D'} sd', tv}{e, D[s : sd] \vdash \mathbf{send}(e', s) \hookrightarrow D'[s : sd']}$$

Sending e' to state s results in processing it through the definition of s . We have extended environments by the definitions of states:

Definition 3 (Extended Environment D).

$$D ::= [x_0 : v_0; \dots; x_n : v_n; e_0; \dots e_k; SD]$$

where SD is a list of state definitions. The notation $D[s : sd]$ denotes the environment D in which s is associated with sd .

However, this rule alone does not fully handle event sending; deeper modifications of the semantics are needed. Processing the local event changes the definition of the destination state (in the rule, the definition of \mathbf{s} is \mathbf{sd}' after processing the event). The destination state can be an ancestor of the current state, which might have been modified. It is necessary, whenever an action is performed, to read the (eventually new) definition of the current state and continue the execution at the corresponding control point in this new definition. If the active state has been modified by the call, the return point may even not be active anymore, which leads in Stateflow to a runtime error.

Investigating this mechanism to understand its behavior and its expressive power, we distinguished two different usages:

- Describing recursive behaviors. Recursion occurs if the caller sends an event to itself or one of its ancestors. In practice, those recursions are very difficult to control (the event sending action might get executed by the recursive call) and to understand. Providing tools to check that the recursion will stop is difficult (see [18]). Moreover, these recursions easily lead to runtime errors and their use is discouraged in industrial applications.
- Explicit scheduling of parallel states. Parallel states are normally ordered statically given their position on the chart. Local events can be used to make some

explicit, or dynamic, scheduling of parallel states, guarding the states by local events and having a caller that executes them in the expected order. This particular use is much simpler to understand.

Our proposal is to limit the use of local events to the definition of sequencing behaviors. This can be obtained by imposing the following restrictions:

- Local events can be sent only to parallel states.
- Transitions out of parallel states are forbidden (this is already imposed by Stateflow, see Section 3.4 for more details).
- Loops in the broadcasting of events are forbidden (i.e., if state A broadcasts an event to state B , B cannot in turn broadcast an event to A).

Given these restrictions¹ sending an event can really be seen as a function call. Forbidding transitions out of parallel states ensures that context modifications are kept local to the destination. Forcing sending to parallel states and forbidding loops ensures that no infinite calls will occur. The rule for sending an event is the rule presented before. In addition to this, we need to change only the rule for parallel execution to that shown in Figure 7. This rule is similar to the original one, with the addition of the state definitions in the environment, where they are updated during execution.

This definition of local events in our opinion captures the most interesting of their uses in Stateflow, supports a simple semantics, and does not introduce new runtime error or infinite loop possibilities. The Ford guidelines for Stateflow [5] make use of local events in this exact same way.

3.4 Additional and unsupported features

Our subset supports nearly the whole language with the restrictions on local events presented above. The only interesting feature still missing is the history junction mechanism that keeps track of the configuration a state was in before it was last exited, and re-enters it in this configuration. Our semantics easily extends to support this mechanism; we omitted it here for the sake of simplicity. The necessary modifications are to add a history component (a Boolean) in the state definitions to determine whether they carry such junctions, and to add rules to handle this component when entering and leaving states and compositions.

Two restrictions are also imposed on transitions: transitions out of a parallel compositions and interlevel transitions going to a junction are forbidden. Transitions directly out of a parallel state are already forbidden in Stateflow, but can be simulated by taking an interlevel

¹ To keep equivalence with Stateflow, we further impose that local events can be sent only to already-visited states; this is due to initialization problems in Stateflow itself.

$$\text{AND} \frac{(tv = \text{No}) \vee (tv = \text{End}) \quad \forall i \in [0..n], \quad e, D_i, J \vdash sd_i \xrightarrow{D_{i+1}} sd'_i, \text{No}}{e, D_0, J, tv \vdash \text{And}(\{s_0 : sd_0; \dots; s_n : sd_n\}) \xrightarrow{D_{n+1}} \text{And}(\{s_0 : sd'_0; \dots; s_n : sd'_n\}), \text{No}}$$

Fig. 7. Modified rule for AND compositions with local events

transition from a substate of a parallel state. The behavior of such transitions is quite unpredictable, and introduces possible runtime errors (e.g., two states fire simultaneously out of the composition to different destinations). Forbidding interlevel transitions to a junction allows the semantics to be local. When taking an interlevel transition to a state, pending actions can be executed and the state closed before entering the destination. If the transition goes to a junction, we cannot be sure that it is leading somewhere, and cannot close the state before opening the destination.

3.5 Equivalence with the simulation semantics

Our language is intended to be a strict subset of Stateflow, so that tools developed for it will apply to programs designed using Mathworks' tools—as long as the programs are within our subset (which is checked by a tool, see Section 3.6).

For this to succeed, the semantics we are using and the simulation semantics of Stateflow must be equivalent on our subset. Our semantics was conceived with this goal in mind, precisely following Stateflow documentation but, because the simulation semantics is not formal, it is not possible to prove this equivalence. However, our SOS semantics is directly executable and can easily be used to define a Stateflow interpreter whose outputs can be compared to those from the Matlab simulator. We have done this and systematically examined many examples; for all these examples, the traces obtained by the two tools were the same.

3.6 Applications

Our goal was to propose a formalization of Stateflow that provides a good foundation for construction of formal tools for the language. The presented semantics appears to meet this goal very well: while sometimes large, the rules of the semantics are simple and syntax directed, which makes them well adapted to automatic processing.

One kind of tool in which we are interested is static analysis for detecting flaws in programs and to enforce or enhance programming guidelines such as [5]. The proposed semantics, by giving a low-level view of a program's execution makes it possible to detect and understand causes of runtime errors (through lack of an applicable rule in the semantics). We have developed such a

tool that checks for possible runtime errors and that also detects non fatal flaws, such as possible backtracking or reliance on the 12 o'clock rule. Having a syntax-directed semantics allows a precise diagnosis to be given to the user. This tool also verifies that a program lies within the subset considered by our semantics.

We are also interested in model checking Stateflow, which can be used to check properties of designs and to automate test case generation. Our operational semantics provides a basis for efficiently compiling Stateflow to an imperative language or to the input language of a model checker. We have developed a translator to the SAL language used by SRI's model checkers [14]; the translation produces efficient code similar in size to the Stateflow model. The SAL translation can be used to verify properties or to detect errors through model checking: for example, the Stopwatch of Figure 1 contains a bug which the model-checker easily finds (updating the display is only done when staying at least one instant in the **Running** state; if several **LAP** and **START** events occur between two **TICs**, the display can show an erroneous value). This translation is also able to insert the "trap variables" used in automated test generation [8]; the manual for the `sal-atg` automated test generator [9] describes experiments in test generation for a gearbox shift-selector specified in Stateflow.

4 A Denotational Semantics for Stateflow

In recent work [7], we propose another approach to Stateflow semantics, which is denotational. Our motivation for developing this new semantics were twofold: first, to remove the restrictions on the language imposed by the semantics presented here, in particular the ones on interlevel transitions leading to junctions; second, to formalize a compilation process for Stateflow. The denotational semantics addresses both these questions.

The key to the operational semantics presented here was to recognize that Stateflow, although looking like a statechart notation, is in fact an imperative language, thanks to (albeit complex) total ordering rules, and the absence of nondeterminism. This recognition also provides the key to a denotational semantics. If Stateflow is an imperative language, its transitions are governed by a mechanism that can modify the control point of the program, possibly in a non-local way. The denotational

semantics thus considers the whole Stateflow transition mechanism as a complex and highly specialized form of non-local control structure. Denotational semantics for many forms of such non-local constructions have been given by means of continuations [17], and we use continuations to describe the complete transition system for Stateflow. The denotational treatment is able to give a concise semantics to interlevel transitions leading to junctions that were excluded from the operational semantics in order to keep its rules tractable.

Although more recent than the semantics presented here, the denotational semantics does not replace the present work, but complements it in our overall goal, which is to understand Stateflow and to design formal tools for it. Our main purpose in designing the denotational semantics was to formalize the compilation process that we use to translate Stateflow to SAL and to imperative programs. This complements the syntax-directed static analysis tools for Stateflow that are based on the operational semantics presented here. Between these two approaches, we now have a very strong basis for developing formal tools for Stateflow.

5 Conclusion and related work

We have presented an operational semantics for the Stateflow language. Our semantics covers virtually the whole language, excluding only those features that are generally discouraged in industrial applications [5]. A formal semantics is the necessary basis for building formal tools for the language. The operational approach chosen here leads to a surprisingly simple semantics and thus constitutes a good starting point for such developments.

5.1 Related work

Little work has directly addressed the semantics of Stateflow. A natural idea when considering Stateflow is to evaluate work on formalization of Statecharts [10]. However, the two languages have very different semantics (and Stateflow also includes flowcharts), so denotational approaches proposed for Statecharts semantics do not easily or usefully adapt to Stateflow.

A popular approach to Statecharts semantics is to translate the language into a simpler formalism for which a semantics is already known. This approach was followed by Mikk et al. for Statemate [13] by translation to hierarchical automata. Their semantics was adapted to UML-Statecharts by Gnesi, Latella and Massink [6]. A similar semantics was proposed for Stateflow by Tiwari, Shankar and Rushby [18] by translation to push-down automata. However, encoding the complex Stateflow language constructions requires introduction of a vast number of control variables that make using the translation by formal tools difficult.

Lüttgen, von der Beeck, and Cleaveland [11] have proposed an SOS semantics for a subset of Statecharts. They wanted to define a compositional semantics and do not consider interlevel transitions. We can notice the same effect here: although our semantics is not compositional (the language contains absolute reference to states), it can be made compositional by forbidding interlevel transitions. They also need to consider execution on a micro and on a macro level, which is not necessary here due to the completely deterministic nature of Stateflow.

The appeal of the proposed SOS semantics for Stateflow, and what makes it work, is that it exhibits the sequential behavior of Stateflow: the language does not have true concurrency nor any kind of nondeterminism. Seeing Stateflow as an imperative language, the choice for an operational approach is natural, and has the advantage of scaling well to a rich language allowing a large subset to be considered.

A similar approach is implicit in the work of Banphawattharak, Krogh, and Butts [2], who describe a translator from Stateflow into the input language of the SMV model checker. Although they do not construct an explicit semantics, the considerations that guide their translation are very close to ours and reflect a similar focus on the sequential nature of Stateflow execution and the importance of accurately representing its sequencing rules. Similar approaches can also be found in the works of Caspi et al. [16], which points to the present work for a formal definition of the language, and in the work of Anton, da Costa, and Errington [1]. Finally, we have already mentioned our own work presenting a denotational semantics for the language [7].

In future work, we plan to investigate the formalization of the whole Simulink/Stateflow environment. One possible direction is to combine this work with existing work on Simulink [4].

References

1. Anton, J., da Costa, P., Errington, L.: Formal synthesis of generators for embedded systems. Tech. rep., Kestrel Technology (2005)
2. Banphawattharak, C., Krogh, B.H., Butts, K.: Symbolic verification of executable control specifications. In: Proceedings of the Tenth IEEE International Symposium on Computer Aided Control System Design, pp. 581–586. Kohala Coast—Island of Hawai'i, HI (1999)
3. Buck, D., Rau, A.: On modelling guidelines: Flowchart patterns for Stateflow. *Softwaretechnik-Trends* **21**(2) (2001)
4. Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S.: Translating discrete-time Simulink to Lustre. In: Third International ACM Conference on Embedded Software, *Lecture Notes in Computer Science*, vol. 2855, pp. 84–99. Springer-Verlag (2003)
5. Ford: Structured analysis and design using Matlab/Simulink/Stateflow - modeling style guidelines.

- Tech. rep., Ford Motor Company (1999). URL <http://vehicle.me.berkeley.edu/mobies/papers/stylev242.pdf>. Available at <http://vehicle.me.berkeley.edu/mobies/papers/stylev242.pdf>
6. Gnesi, S., Latella, D., Massink, M.: Modular semantics for a UML Statechart diagrams kernel and its extension to Multicharts and branching time model checking. *The Journal of Logic and Algebraic Programming* **51**(1), 43–75 (2002)
 7. Hamon, G.: A denotational semantics for Stateflow. In: EMSOFT 2005: Proceedings of the Fifth ACM Workshop on Embedded Software, pp. 164–172. Association for Computing Machinery, Jersey City, NJ (2005)
 8. Hamon, G., de Moura, L., Rushby, J.: Generating efficient test sets with a model checker. In: 2nd International Conference on Software Engineering and Formal Methods, pp. 261–270. IEEE Computer Society, Beijing, China (2004)
 9. Hamon, G., de Moura, L., Rushby, J.: Automated test generation with SAL. Technical note, Computer Science Laboratory, SRI International, Menlo Park, CA (2005). Available at <http://www.csl.sri.com/users/rushby/abstracts/sal-atg>
 10. Harel, D.: Statecharts: A visual formalism for complex systems. *Science of Computer Programming* **8**(3), 231–274 (1987)
 11. Lüttgen, G., von der Beeck, M., Cleaveland, R.: A compositional approach to Statecharts semantics. In: D. Rosenblum (ed.) Eighth International ACM Symposium on Foundations of Software Engineering, pp. 120–129. San Diego, California (2000)
 12. The Mathworks: Stateflow and Stateflow Coder, User’s Guide, release 13sp1 edn. (2003). URL http://www.mathworks.com/access/helpdesk/help/pdf_doc/statefl%ow/sf_ug.pdf
 13. Mikk, E., Lakhnech, Y., Petersohn, C., Siegel, M.: On formal semantics of Statecharts as supported by Statemate. In: 2nd BCS-FACS Northern Formal Methods Workshop. BCS-EWIC (1997)
 14. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: R. Alur, D. Peled (eds.) *Computer-Aided Verification, CAV ’2004, Lecture Notes in Computer Science*, vol. 3114, pp. 496–500. Springer-Verlag, Boston, MA (2004)
 15. Plotkin, G.: A structural approach to operational semantics. Tech. Rep. DAIMI-FN-19, Aarhus University (1981)
 16. Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maranchi, F.: Defining and translating a “safe” subset of Simulink/Stateflow into Lustre. In: EMSOFT 2004: Proceedings of the Fourth ACM Workshop on Embedded Software, pp. 259–268. Pisa, Italy (2004)
 17. Strachey, C., Wadsworth, C.P.: Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation* **13**, 135–152 (2000). Originally published 1974
 18. Tiwari, A., Shankar, N., Rushby, J.: Invisible formal methods for embedded control systems. *Proceedings of the IEEE* **91**(1), 29–39 (2003)

A The complete semantics

We give here the complete operational semantics, with rules for all syntactic classes of the language:

- transitions and transition lists in Figures 4 and 5.
- state definitions in Figure 8.
- OR compositions in Figures 9 and 10.
- AND compositions in Figure 6, as updated by Figure 7.

SD-NO	
$e, D_0, J_0 \vdash T_o \xrightarrow{D_1} tv$	$(tv = \mathbf{No}) \vee (tv = \mathbf{End}) \quad e, D_1 \vdash A.d \hookrightarrow D_2 \quad e, D_2, J \vdash T_i \xrightarrow{D_3} tv \quad e, D_3, J, tv \vdash C \xrightarrow{D_4} C', \mathbf{No}$
$e, D_0, J_0 \vdash (A, C, T_o, T_i, J) \xrightarrow{D_4} (A, C', T_o, T_i, J), \mathbf{No}$	
SD-INT-FIRE	
$e, D_1 \vdash A.d \hookrightarrow D_2$	$e, D_0, J_0 \vdash T_o \xrightarrow{D_1} tv \quad (tv = \mathbf{No}) \vee (tv = \mathbf{End})$
$e, D_2 \vdash T_i \xrightarrow{D_3} tv$	$e, D_3, J, tv \vdash C \xrightarrow{D_4} C', \mathbf{Fire}(d, a) \quad e, D_4 \vdash a \hookrightarrow D_5 \quad e, D_5 \vdash A.x \hookrightarrow D_6$
$e, D_0, J_0 \vdash (A, C, T_o, T_i, J) \xrightarrow{D_5} (A, C', T_o, T_i, J), \mathbf{Fire}(d, \diamond)$	
SD-FIRE	
$e, D_0, J_0 \vdash T_o \xrightarrow{D_1} \mathbf{Fire}(d, a)$	$e, D_1 \vdash a \hookrightarrow D_2 \quad e, D_2 \vdash C \Downarrow C' \quad e, D_3 \vdash A.x \hookrightarrow D_4$
$e, D_0, J_0 \vdash (A, C, T_o, T_i, J) \xrightarrow{D_4} (A, C', T_o, T_i, J), \mathbf{Fire}(d, \diamond)$	
SD-INIT	SD-EXIT
$e, D_0 \vdash A.e \hookrightarrow D_1$	$e, D_0 \vdash C \Downarrow C' \quad e, D_1 \vdash A.x \hookrightarrow D_2$
$e, D_1, p \vdash C \Uparrow C'$	$e, D_1 \vdash A.x \hookrightarrow D_2$
$e, D_0, p \vdash (A, C, T_o, T_i, J) \Uparrow (A, C', T_o, T_i, J)$	$e, D_0 \vdash (A, C, T_o, T_i, J) \Downarrow (A, C', T_o, T_i, J)$

Fig. 8. Rules for state definitions

OR-EXT-FIRE	
$e, D_0 \vdash a \hookrightarrow D_1$	$e, D_1 \vdash sd_0 \Downarrow sd'_0 \quad e, D_2, p' \vdash sd_1 \Uparrow sd'_1$
$e, D_0, J, \mathbf{Fire}(p.s1.p', a) \vdash \mathbf{Or}(s_0, p, T, SD[s_0 : sd_0; s_1 : sd_1]) \xrightarrow{D_3} \mathbf{Or}(s_1, p, T, SD[s_0 : sd'_0; s_1 : sd'_1]), \mathbf{No}$	
OR-EXT-FIRE-OUT	
$\neg \mathbf{prefix}(p', p)$	$e, D_0 \vdash a \hookrightarrow D_1 \quad e, D_1 \vdash sd \Downarrow sd'$
$e, D_0, J, \mathbf{Fire}(p', a) \vdash \mathbf{Or}(s, p, T, SD[s : sd]) \xrightarrow{D_2} \mathbf{Or}(\emptyset_s, p, T, SD[s : sd']), \mathbf{Fire}(p', \diamond)$	
OR-NO	
$(tv = \mathbf{No}) \vee (tv = \mathbf{End}) \quad e, D_0, J \vdash sd_0 \xrightarrow{D_1} sd'_0, \mathbf{No}$	
$e, D_0, J, tv \vdash \mathbf{Or}(s_0, p, T, SD[s_0 : sd_0]) \xrightarrow{D_1} \mathbf{Or}(s, p, T, SD[s_0 : sd'_0]), \mathbf{No}$	
OR-INT-FIRE	
$(tv = \mathbf{No}) \vee (tv = \mathbf{End})$	$e, D_0, J \vdash sd_0 \xrightarrow{D_1} sd'_0, \mathbf{Fire}(p', act) \quad p' = p.s1.p'' \quad e, D_1, p'' \vdash sd_1 \Uparrow sd'_1$
$e, D_0, J, tv \vdash \mathbf{Or}(s_0, p, T, SD[s_0 : sd_0; s_1 : sd_1]) \xrightarrow{D_2} \mathbf{Or}(s_1, p, T, SD[s_0 : sd'_0; s_1 : sd'_1]), \mathbf{No}$	
OR-FIRE	
$(tv = \mathbf{No}) \vee (tv = \mathbf{End})$	$e, D_0, J \vdash sd \xrightarrow{D_1} sd', \mathbf{Fire}(p', a) \quad \neg \mathbf{prefix}(p', p)$
$e, D_0, J, tv \vdash \mathbf{Or}(s, p, T, SD[s : sd]) \xrightarrow{D_1} \mathbf{Or}(\emptyset_s, p, T, SD[s : sd']), \mathbf{Fire}(p', a)$	

Fig. 9. Rules for Or-compositions

<p style="text-align: center;">OR-INIT-NO-STATE</p> $e, D, \emptyset_p \vdash \mathbf{Or}(\emptyset_s, p, \emptyset_T, []) \xrightarrow{D} \mathbf{Or}(\emptyset_s, p, \emptyset_T, [])$	<p style="text-align: center;">OR-INIT-\emptyset_p</p> $\frac{e, D_0, J \vdash T \xrightarrow{D_1} \mathbf{Fire}(s.p, a) \quad e, D_1, p \vdash sd \xrightarrow{D_2} sd' \quad e, D_2 \vdash a \hookrightarrow D_3}{e, D_0, \emptyset_p \vdash \mathbf{Or}(\emptyset_s, p_0, T, SD[s : sd]) \xrightarrow{D_3} \mathbf{Or}(s, p_0, T, SD[s : sd'])}$
<p style="text-align: center;">OR-INIT</p> $\frac{e, D_0, p \vdash sd \xrightarrow{D_1} sd'}{e, D_0, s.p \vdash \mathbf{Or}(\emptyset_s, p_0, T, SD[s : sd]) \xrightarrow{D_1} \mathbf{Or}(s, p_0, T, SD[s : sd'])}$	<p style="text-align: center;">OR-EXIT</p> $\frac{e, D_0 \vdash sd \xrightarrow{D_1} sd'}{e, D_0 \vdash \mathbf{Or}(s, p, T, SD[s : sd]) \xrightarrow{D_1} \mathbf{Or}(\emptyset_s, p, T, SD[s : sd'])}$

Fig. 10. Rules for entering and exiting Or-compositions