Presented at *Fundamental Approaches to Software Engineering* (FASE)
Barcelona, Spain, March 2004. Springer Verlag LNCS 2984, pp. 229–243.

An Operational Semantics for Stateflow*

Grégoire Hamon and John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 - USA
{hamon,rushby}@csl.sri.com

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 allows hierarchical state-machine diagrams *à la* Statecharts to be combined with flowchart diagrams in a very flexible way. 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.

* This material is based on work 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.

The evolution to 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 this 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 [1] is 896 pages long) with numerous, complicated, and often overlapping features lacking any formal definition. Its documentation [1, Chapter 4] describes the semantics in informal operational terms, supported by numerous examples, but the actual definition of the language is the "simulation semantics" given by its behavior when simulated in the Matlab environment. Proposing formal 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 [2, 3] that restrict the language to a safe kernel. These guidelines have no more formal basis than the language itself and are based on experience. Precisely identifying the reasons for runtime errors 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 statecharts 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. This operational approach satisfies our

goal and is able to express the more complicated features of the language where alternative approaches (e.g., denotational) might get lost.

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 develops a formalization of a subset of the language and gives it an operational semantics. Finally, in Section 4 we compare our approach with related work and sketch why the approach proposed here seems to be a good basis for developing formal tools for the language.

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 (which are portrayed as small circles), 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

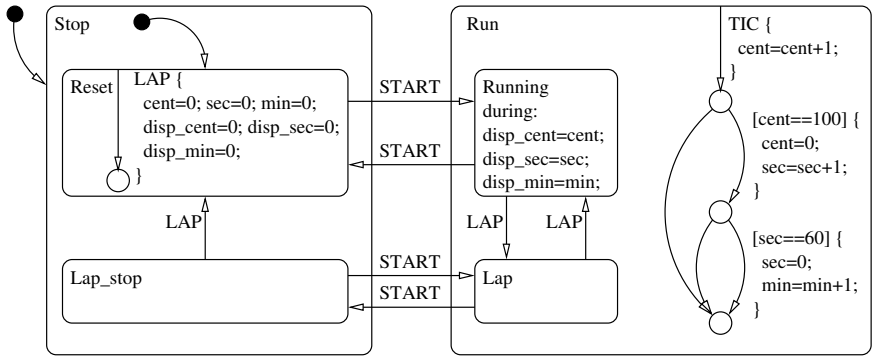

Fig. 1. A simple stopwatch in Stateflow

Figure 1 presents the Stateflow specification of 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 fixes 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. Receiving `LAP` resets the counter and the display, receiving `START` changes the control to the `Running` mode.
- `Lap_Stop`: the counter is stopped. Receiving `LAP` changes to the `Reset` mode and receiving `START` to the `Lap` mode.
- `Running`: the counter is running, and the display updated. Receiving `START` changes to the `Stop` mode, pressing `LAP` changes to the `Lap` mode.
- `Lap`: the counter is running, but the display is not updated, thus showing the last value it received. Receiving `START` changes to `Lap_Stop`, receiving `LAP` changes 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 consists in executing the active state 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:

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 would 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 substate of `Stop`; `Running` then `Run` are exited, and `Stop` 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 (a junction 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 under this event, the `cent` variable is incremented and the first junction reached. Two transitions leave it, the guarded one is always executed first. If `cent` is equal to 100, the guarded transition is taken, `cent` initialized 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, interlevel transitions, and mixed design with 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) [4], 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 is a strict subset of Stateflow. This language eliminates some difficulties of the graphical notation, by making the order between components explicit (we describe translation from the graphical form below). We then give this language a formal semantics.

The language – The language is presented in Figure 2. Its basic components are states s, junctions j, events e, actions a, and conditions c. We also define active states s_a (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.

Transitions are grouped into lists T. Junction definition lists J associate lists of transitions to junctions. State definition lists SD associate state definitions sd to states. A state definition is a triplet of actions, executed respectively upon entering, executing and exiting the state, an internal composition, a list of inner transitions, a list of outgoing transitions, and a junction definition list. 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.

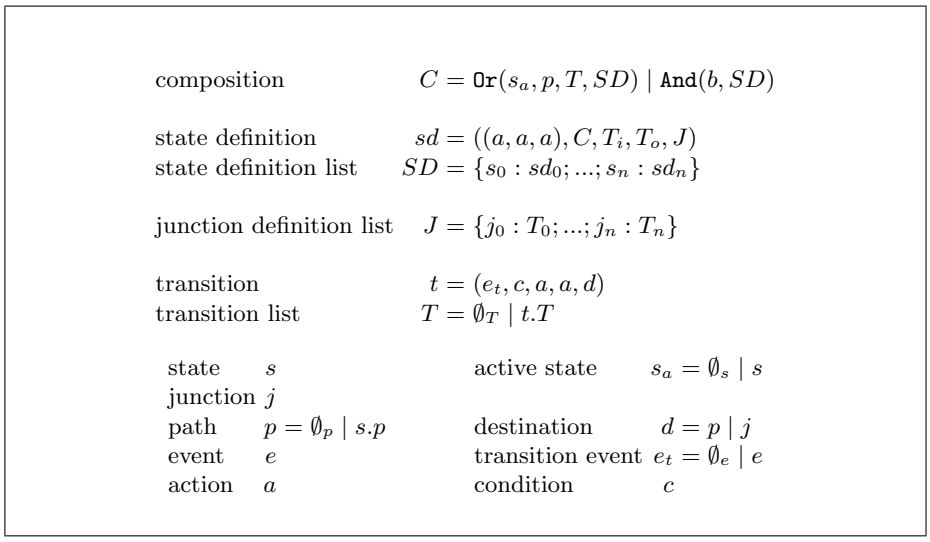

Fig. 2. The language

Notes on the 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 list 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.
- In the following, 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 will be noted respectively as $A.e$, $A.d$, and $A.x$.

An example – the `Stop` state from the stopwatch:

```
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, { })
```

The ◇ 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 through a program component P produces a new environment D', a new program P', and a transition value tv. P can here 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.

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.

Definition 1 (Environment D).

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

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.

Definition 2 (Transition value tv).

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

As for the definition of the language, we do not detail the semantics of actions and conditions here but consider that we have semantics rules such that

$$e, D \vdash a \hookrightarrow D' \qquad\qquad e, D \vdash c \rightarrow b$$

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 brevity we only detail rules for transitions, transition lists and parallel compositions, the full rules are available in an appendix.

Transitions (Figure 3) – 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-No$_1$) or if the condition is false (rule t-No$_2$), the transition fails and returns `No`.

Transition lists (Figure 4) – 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 backtrack and try the next transition in our first list (rule T-Fire-j-N).

$$\textit{t-Fire}\quad \frac{(e = e_0) \vee (e_0 = \emptyset_e) \qquad e, D \vdash c \rightarrow \mathit{true} \qquad e, D \vdash a_c \hookrightarrow D'}{e, D \vdash (e_0, c, a_c, a_t, d) \xrightarrow{D'} \texttt{Fire}(d, a_t)}$$

$$\textit{t-No}_1\quad \frac{(e \neq e_0) \wedge (e_0 \neq \emptyset_e)}{e, D \vdash (e_0, c, a_c, a_t, d) \xrightarrow{D} \texttt{No}} \qquad \textit{t-No}_2\quad \frac{(e = e_0) \vee (e_0 = \emptyset_e) \qquad e, D \vdash c \rightarrow \mathit{false}}{e, D \vdash (e_0, c, a_c, a_t, d) \xrightarrow{D} \texttt{No}}$$

Fig. 3. Rules for transitions t

$$\emptyset_t\quad \frac{}{e, D, J \vdash \emptyset_t \xrightarrow{D} \texttt{End}} \qquad \text{T-Fire}\quad \frac{e, D \vdash t \xrightarrow{D'} \texttt{Fire}(p, act)}{e, D, J \vdash t.T \xrightarrow{D'} \texttt{Fire}(p, act)} \qquad \text{T-No-Last}\quad \frac{e, D \vdash t \xrightarrow{D_1} \texttt{No}}{e, D, J \vdash t.\emptyset_T \xrightarrow{D_1} \texttt{No}}$$

$$\text{T-No}\quad \frac{T \neq \emptyset_T \qquad e, D \vdash t \xrightarrow{D_1} \texttt{No} \qquad e, D_1, J \vdash T \xrightarrow{D_2} \texttt{tv}}{e, D, J \vdash t.T \xrightarrow{D_2} \texttt{tv}}$$

$$\text{T-Fire-J-F}\quad \frac{e, D \vdash t \xrightarrow{D_1} \texttt{Fire}(j, a_1) \qquad e, D_1, J[j : T_j] \vdash T_j \xrightarrow{D_2} \texttt{Fire}(p, a_2)}{e, D, J[j : T_j] \vdash t.T \xrightarrow{D_2} \texttt{Fire}(p, a1; a2)}$$

$$\text{T-End}\quad \frac{e, D \vdash t \xrightarrow{D_1} \texttt{Fire}(j, a_1) \qquad e, D_1, J[j : T_j] \vdash T_j \xrightarrow{D_2} \texttt{End}}{e, D, J[j : T_j] \vdash t.T \xrightarrow{D_2} \texttt{End}}$$

$$\text{T-Fire-J-N}\quad \frac{e, D \vdash t \xrightarrow{D_1} \texttt{Fire}(j, a_1) \qquad e, D_1, J[j : T_j] \vdash T_j \xrightarrow{D_2} \texttt{No} \qquad e, D_2, J[j : T_j] \vdash T \xrightarrow{D_3} \texttt{tv}}{e, D, J[j : T_j] \vdash t.T \xrightarrow{D_3} \texttt{tv}}$$

Fig. 4. Rules for transition list T

State definitions – The rules exhibit their 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 – 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 5) – Executing an And composition consists in sequentially entering/executing/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.

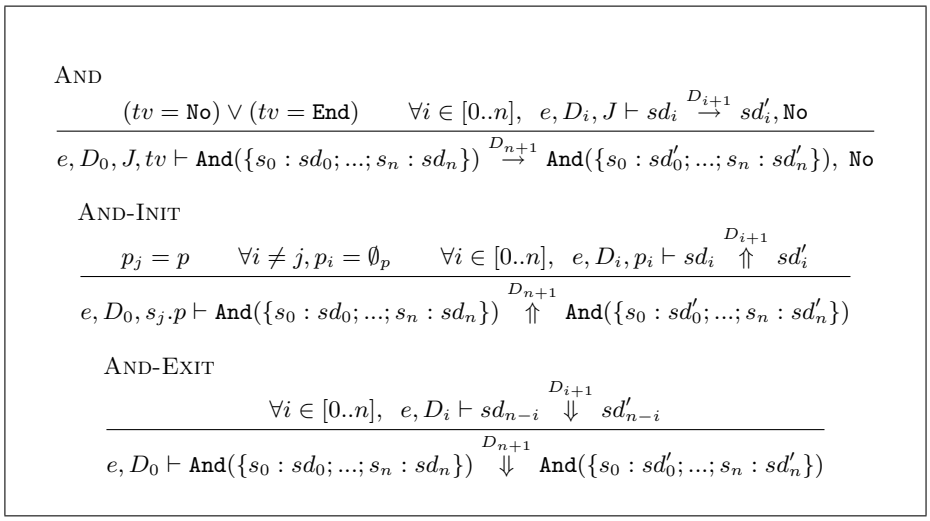

Fig. 5. Rule for AND compositions

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; 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 $\texttt{send}(e, s)$ 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:

$$\textsc{Send}\quad \frac{e', D[s : sd], \emptyset_J \vdash sd \xrightarrow{D'} sd', tv}{e, D[s : sd] \vdash \texttt{send}(e', s) \hookrightarrow D'[s : sd']}$$

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 (Environment D).

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

where SD is a list of state definitions. The notation $D[s : sd]$ denotes the environment D in which s is associated to 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 `s` is `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 [5]). 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 proposition 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 those limitations[1] 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:

AND

$$\frac{\begin{array}{c} D_0 = D[s_0 : sd_0; ...; s_n : sd_n] \\ \forall i \in [0..n],\ \ e, D_i, \{\} \vdash D_i(s_i) \xrightarrow{D'_i} sd'_i, \texttt{No} \quad D_{i+1} = D'_i[s_i : sd'_i] \end{array}}{\begin{array}{r} e, D \vdash \texttt{And}(\{s_0 : sd_0; ...; s_n : sd_n\}) \xrightarrow{D_{n+1} \setminus \{s_0,...,s_n\}} \\ \texttt{And}(\{s_0 : D_{n+1}(s_0); ...; s_n : D_{n+1}(s_n)\}),\ \texttt{No} \end{array}}$$

The 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 guideline for Stateflow [2] makes 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 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.

[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.

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 4.2).

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 interpretor 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.

4 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 [2]. 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.

4.1 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 [6]. 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 [7] by translation to hierarchical automata. Their semantics was adapted to UML-Statecharts by Gnesi, Latella and Massink [8]. A similar semantics was proposed for Stateflow by Tiwari, Shankar and Rushby [5] 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 [9] 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 big subset to be considered.

A similar approach is implicit in the work of Banphawatthanarak, Krogh, and Butts [10], 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.

4.2 A good basis for formal tools

Our goal was to propose a formalization of Stateflow that would constitute 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 [2]. The proposed semantics, by giving a low-level view of a program's execution makes it possible to understand causes of runtime errors (through missing rules in the semantics). We have developed such a tool that checks for possible runtime errors and 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, which can be used to check properties of programs and to automate test case generation [11, 12]. 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; the translation produces efficient code similar in size to the Stateflow model. The SAL translation can be used to check properties of a design: our example (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 `TIC`s, the display can show an erroneous value). We are currently using this translation to do automatic test-case generation for Stateflow.

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 [5, 13].

References

1. The Mathworks: Stateflow and Stateflow Coder, User's Guide. Release 13sp1 edn. (2003)
2. Ford: Structured analysis and design using Matlab/Simulink/Stateflow - modeling style guidelines. Technical report, Ford Motor Company (1999) Available at http://vehicle.me.berkeley.edu/mobies/papers/stylev242.pdf.
3. Buck, D., Rau, A.: On modelling guidelines: Flowchart patterns for Stateflow. Softwaretechnik-Trends **21** (2001)
4. Plotkin, G.: A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University (1981)
5. Tiwari, A., Shankar, N., Rushby, J.: Invisible formal methods for embedded control systems. Proceedings of the IEEE **91** (2003) 29–39
6. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8** (1987) 231–274
7. 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)
8. 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** (2002) 43–75
9. Lüttgen, G., von der Beeck, M., Cleaveland, R.: A compositional approach to Statecharts semantics. In Rosenblum, D., ed.: Eighth International ACM Symposium on Foundations of Software Engineering, San Diego, California (2000) 120–129
10. Banphawatthanarak, 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, Kohala Coast—Island of Hawai'i, HI (1999) 581–586
11. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In Nierstrasz, O., Lemoine, M., eds.: ESEC/FSE '99: Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering. Volume 1687 of Lecture Notes in Computer Science., Toulouse, France, Springer-Verlag (1999) 146–162
12. Rayadurgam, S., Heimdahl, M.P.E.: Coverage based test-case generation using model checkers. In: 8th Annual IEEE Conference and Workshop on the Engineering of Computer Based System (ECBS '01). (2001)
13. Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S.: Translating discrete-time Simulink to Lustre. In: Third International ACM Conference on Embedded Software. Volume 2855 of Lecture Notes in Computer Science., Springer-Verlag (2003) 84–99

A The complete semantics

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

- transitions and transition lists in figure 6.
- state definitions in figure 7.
- or compositions in figures 8 and 9.
- and compositions in figure 10.

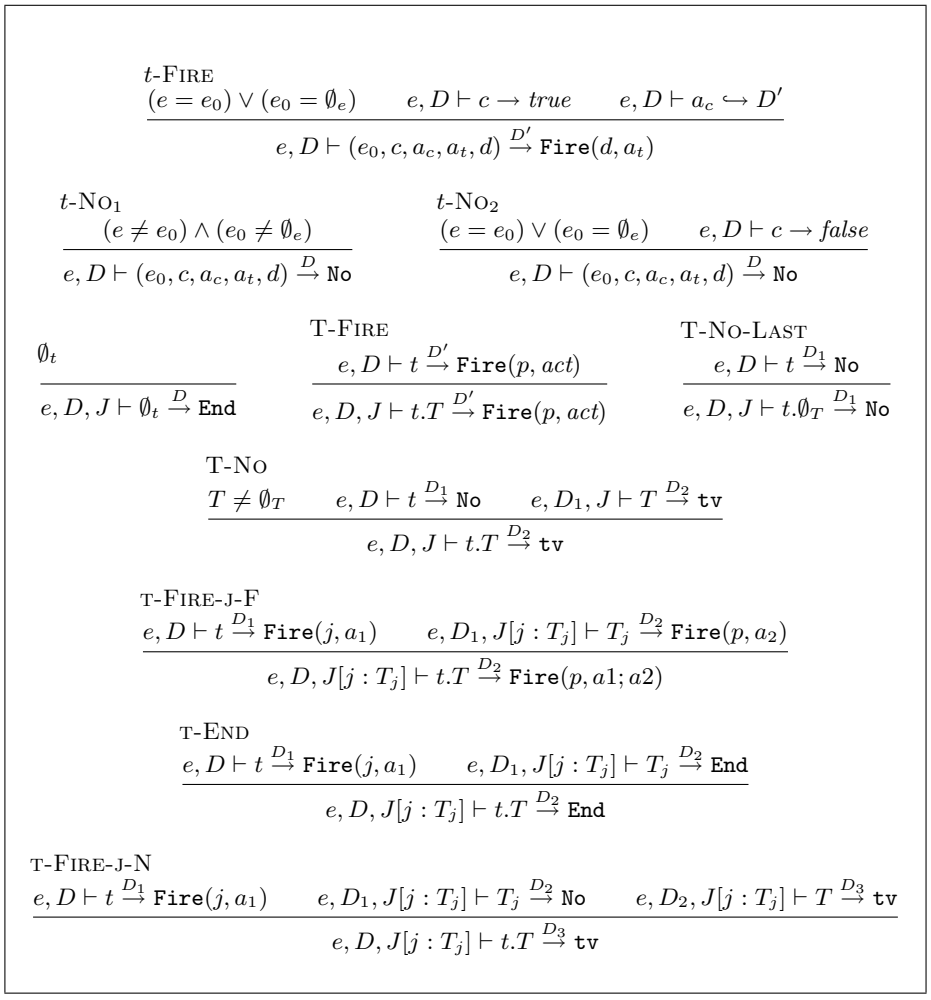

Fig. 6. Rules for transitions and lists of transitions

SD-NO

$$\frac{e, D_0, J_0 \vdash T_o \overset{D_1}{\to} tv \qquad (tv = \texttt{No}) \vee (tv = \texttt{End}) \qquad e, D_1 \vdash A.d \hookrightarrow D_2 \qquad e, D_2, J \vdash T_i \overset{D_3}{\to} tv \qquad e, D_3, J, tv \vdash C \overset{D_4}{\to} C', \texttt{No}}{e, D_0, J_0 \vdash (A, C, T_o, T_i, J) \overset{D_4}{\to} (A, C', T_o, T_i, J),\ \texttt{No}}$$

SD-INT-FIRE

$$\frac{e, D_0, J_0 \vdash T_o \overset{D_1}{\to} tv \qquad (tv = \texttt{No}) \vee (tv = \texttt{End}) \qquad e, D_1 \vdash A.d \hookrightarrow D_2 \qquad e, D_2 \vdash T_i \overset{D_3}{\to} tv \qquad e, D_3, J, tv \vdash C \overset{D_4}{\to} C', \texttt{Fire}(d, a) \qquad e, D_4 \vdash a \hookrightarrow D_5 \qquad e, D_5 \vdash A.x \hookrightarrow D_6}{e, D_0, J_0 \vdash (A, C, T_o, T_i, J) \overset{D_5}{\to} (A, C', T_o, T_i, J),\ \texttt{Fire}(d, \diamond)}$$

SD-FIRE

$$\frac{e, D_0, J_0 \vdash T_o \overset{D_1}{\to} \texttt{Fire}(d, a) \qquad e, D_1 \vdash a \hookrightarrow D_2 \qquad e, D_2 \vdash C \overset{D_3}{\Downarrow} C' \qquad e, D_3 \vdash A.x \hookrightarrow D_4}{e, D_0, J_0 \vdash (A, C, T_o, T_i, J) \overset{D_4}{\to} (A, C', T_o, T_i, J),\ \texttt{Fire}(d, \diamond)}$$

SD-INIT

$$\frac{e, D_0 \vdash A.e \hookrightarrow D_1 \qquad e, D_1, p \vdash C \overset{D_2}{\Uparrow} C'}{e, D_0, p \vdash (A, C, T_o, T_i, J) \overset{D_2}{\Uparrow} (A, C', T_o, T_i, J)}$$

SD-EXIT

$$\frac{e, D_0 \vdash C \overset{D_1}{\Downarrow} C' \qquad e, D_1 \vdash A.x \hookrightarrow D_2}{e, D_0 \vdash (A, C, T_o, T_i, J) \overset{D_2}{\Downarrow} (A, C', T_o, T_i, J)}$$

Fig. 7. Rules for state definitions

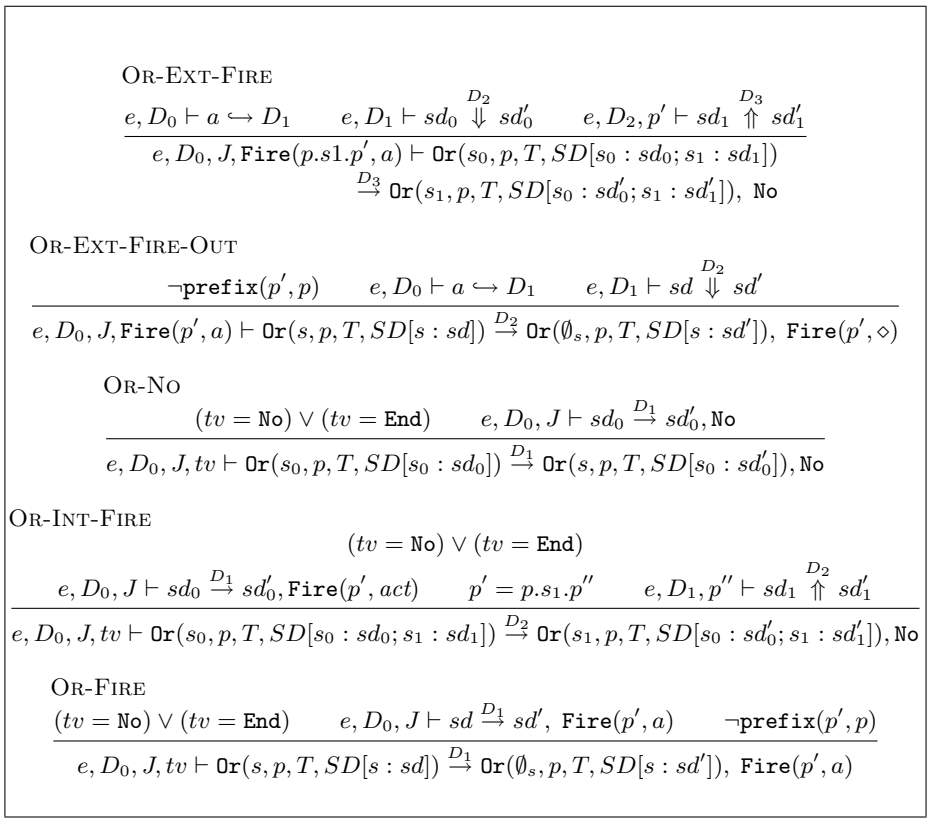

Fig. 8. Rules for Or-compositions

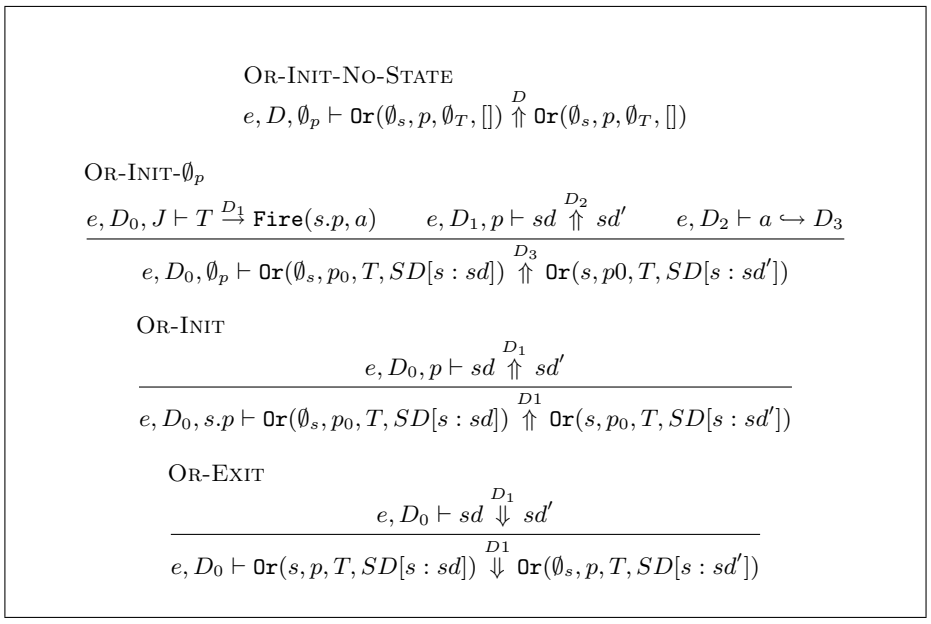

Fig. 9. Rules for entering and exiting Or-compositions

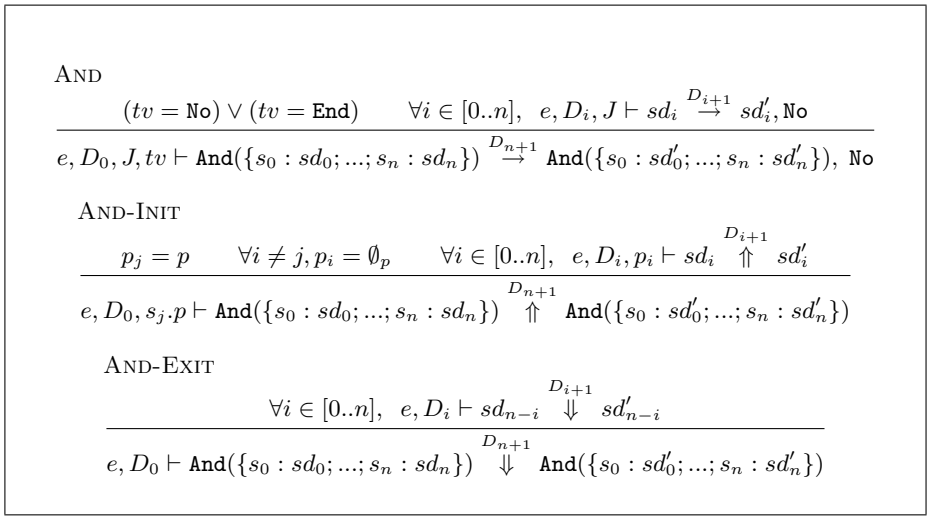

Fig. 10. Rule for AND compositions