A Denotational Semantics for Stateflow *

Grégoire Hamon
Chalmers Institute of Technology
Göteborg, Sweden
hamon@cs.chalmers.se

ABSTRACT

We present a denotational semantics for Stateflow, the graphical Statecharts-like language of the Matlab/Simulink toolsuite. This semantics makes use of continuations to capture even the most complex constructions of the language, such as inter-level transitions, junctions, or backtracking. An immediate application of this semantics is a formal compilation scheme for the language.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.2.6 [**Software Engineering**]: Programming Environments—*Graphical Environments*

General Terms

Design, Languages

Keywords

Stateflow,denotational semantics, continuations, compilation

1. INTRODUCTION

As embedded systems grow in complexity and criticality, designers increasingly face problems of scalability and quality. One answer to these problems has been the widespread adoption of model-based development environments. Model-based environments allow a high-level, graphical, description of the system, close to its specification. This high-level of abstraction, combined with extensive tool capabilities for simulation or validation, greatly helps to improve design quality and scalability.

One of the most widespread model-based development environment is the Matlab/Simulink suite from The Mathworks, widely used in several industries, such as aerospace, or automotive. Stateflow [11] is a component of the Matlab suite, dedicated to the design of discrete controllers. Stateflow allows hierarchical state-machines *à la* Statecharts and flowchart diagrams to be combined. It is very well integrated within Matlab. Typically, the controller under design is modeled in Stateflow, and its environment modeled in Simulink, another component of the suite. The whole system can be simulated using all the capabilities of Matlab in that regard. This integration of the suite makes it probably the most versatile design tool of its kind.

However, Stateflow lacks any formal definition. The semantics of a program is given by the result of its simulation within the Mathworks tools. This absence of formal definition is a big obstacle to static analysis, verification, or automatic test-cases generation of Stateflow designs. These are becoming crucial as designs increase in complexity and criticality. To prevent possible runtime errors, or unexpected behaviors, users have been using guidelines[5, 2] to define "safe" subset of the language. These informal guidelines are often both too restrictive to the user, and too permissive to ensure safety.

In [8], we proposed an operational semantics for Stateflow, based on the premise that Stateflow, while allowing allowing a parallel description of the system, has a purely sequential behavior. We have used this semantics to define static analysis, an interpretor, and a compiler to the SAL language. SAL[4] is the input language of various model-checkers and formal tools, we are using it to automatically generate test-suites from Stateflow diagrams[7]. However, doing these experiments on industrial-size models shows the following limitations:

- the operational semantics is defined for a subset of Stateflow, in which transitions, in particular, have been restricted. These restriction were imposed to keep the semantics simple. Some examples are not supported by the semantics.

- while the operational semantics formalizes an interpreter for the language, it does not formalize a compilation process. Extensions and maintenance of the compiler is difficult and an easy source for errors.

The first limitation can eventually be solved by considering a more complex semantics. However, for the formalization to be usable, it should be kept reasonably simple. The second limitation is somewhat deeper. The operational approach taken, which has great advantages for understanding the behavior of a program, does not formalize a compilation process.

*This work has been partially financed by the Swedish Foundation for Strategic Research.

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

In order to address both those issues, we now take a denotational approach. A denotational semantics is a good basis for a compiler, the problem is in finding a denotation for the complex features of Stateflow, specifically its transition mechanism. Following our previous idea, and considering Stateflow as an imperative language, transitions are constructions that can move the control point in a non-local manner to any other part of the program. In other words, they are a special form of jumps. Jumps, and other non-local constructions have been given denotational semantics by means of *continuations* – those are key to our semantics.

Continuations[16] have been introduced by Strachey and Wadsworth (among others, see[13]) to give a semantics to full jumps (*i.e.* gotos). They have since been used to formalize various non-local control constructions like exceptions, backtracking, or coroutines (see[15]). The idea is surprisingly simple and elegant. It came from the realization that, to formalize non-local constructions, you need the ability to manipulate "the rest of the program"[19]. A jump simply ignores this "rest of the program" and replaces it. Other constructions will alter it in various ways. To gain this ability, the "rest of the program" is passed as an extra-argument – the continuation – to the semantic functions.

In this article, we use continuations to give a denotational semantics of Stateflow. This semantics answers both of our concerns. By using continuations, we can describe the transition mechanism completely, with all its complex features, such as backtracking, or inter-level transitions, and thus consider the full language. Moreover, this denotational approach gives a natural compilation scheme for the language.

We first give a short introduction to the language in section 2. Then, in section 3 we focus on the transition mechanism, detail its behavior, and show how continuations can be used to formalize it. In section 4, we formalize the language and present its complete semantics. The compilation scheme obtained from the semantics is presented in section 5. Finally, we look at related and future works in section 6.

2. STATEFLOW

Stateflow combines two well-known notations: hierarchical state machines, similar to Statecharts, and flowcharts, in a unique formalism. While having a notation close to those of Statecharts and flowcharts, Stateflow has its own, peculiar semantics. Describing the whole language is beyond the scope of this paper (see [11]), we present a simple example that includes both kinds of notation and sketch its execution.

2.1 A stopwatch

Figure 1 presents the Stateflow specification of a stopwatch with lap time measurement. This stopwatch maintains an internal counter represented by the three variables `min`, `sec`, and `cent`). It updates a display, represented as the 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 this one is stopped. This behavior is 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 grouped by pairs inside two main states: `Run` and `Stop`, active respectively when the counter is running 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

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 can be 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 any internal actions (internal transitions, then `during` actions).
3. Execute any internal state that is active.

Transitions can be guarded by events or conditions, 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. It 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, `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.

In this model, the counter is implemented using a flowchart. 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.

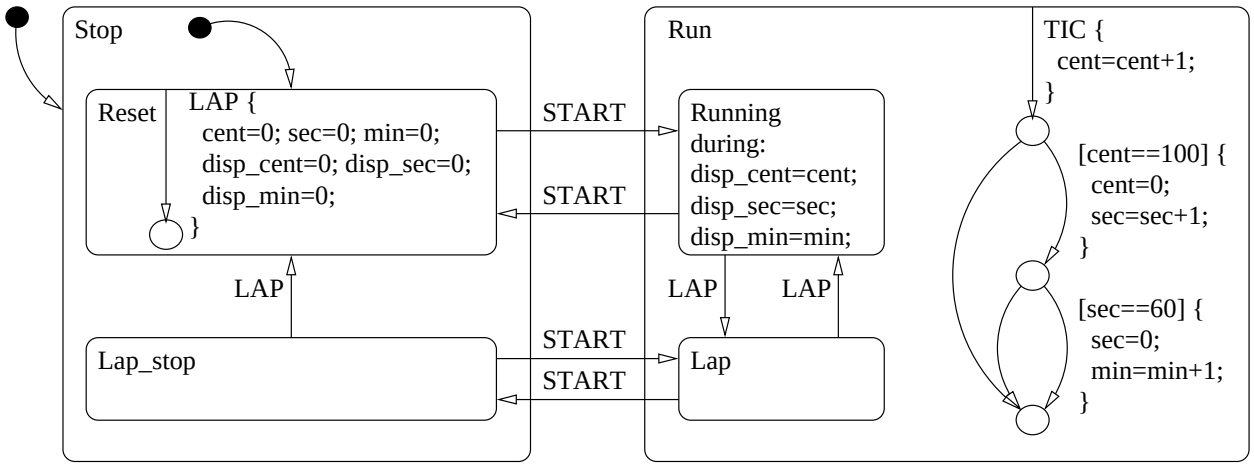

Figure 1: A Stopwatch in Stateflow

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. It already 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. STATEFLOW'S TRANSITIONS

We now focus on Stateflow's transition mechanism. The transition mechanism is specific to Stateflow, and is in a great part responsible for both the expressive power of the language and its complexity. We first present it informally, and then show how continuations can be used to give it a denotational semantics.

3.1 Presentation

As we've seen with the stopwatch example, the transition mechanism is used to express both transitions between states and flowchart diagrams. Complex transitions can be expressed by composing transitions through junctions. The mechanism support advanced features such as inter level transitions, or backtracking. Figure 2 presents the general form of a transition in Stateflow. A transition goes from a source `s` to a destination `d`. Source and destination are either a state or a junction. It is guarded by an event `e` and a condition `c`. If (and only if) the event is received, and the condition is true, the transition succeeds. Finally, the transition carries two actions, `a1`, and `a2`. `a1` is executed as soon as the transition succeeds). `a2` is left pending, and will only be executed if the transition leads to a successful path, that is if executing the destination `d` will ultimately reach a state.

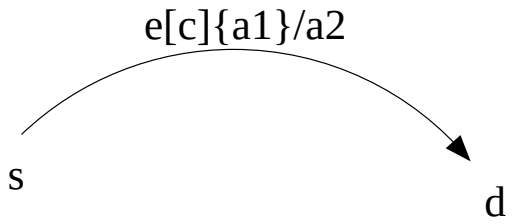

Figure 2: A simple transition

Figure 3 shows transitions combined through junctions. Supposing the event `e` is active, when the first transition is evaluated, if `X` is greater than 2, `Z` is immediately incremented. The action `X=0` is left pending, and evaluation continues with the next transition. If the condition `Y>Z` is true, it succeeds and `Y=0` is left pending. Executing the pending actions will be done when reaching a state, in sequence (here, `X=0`, then `Y=0`). However, if the condition `Y>Z` is not met, the whole branch fails, and the pending actions are discarded. As we see on this example, each component of the transition can be omitted (triggering event, condition, actions). A transition without triggering event is triggered by any event.

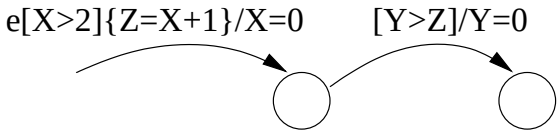

Figure 3: Example of transitions

A source, i.e. either a state or a junction, can have several outgoing transitions. In that case, they are evaluated one by one, until one of them succeeds. The order of evaluation is given by a partial order over the form of the transitions: transitions guarded by event first, then transitions guarded by a condition, then unguarded transitions. This order is made total by using a graphical argument to solve conflicts: transitions are taken clock-wisely from their position on the source, starting at 12 o'clock (this rule is known as the 12

o'clock rule). Figure 4 presents a junction, with three transitions. Transition 1 and 3 are guarded by events, and will be evaluated before transition 2 which is guarded only by a condition. Transition 1 comes first in the graphical ordering and is thus evaluated before transition 3.

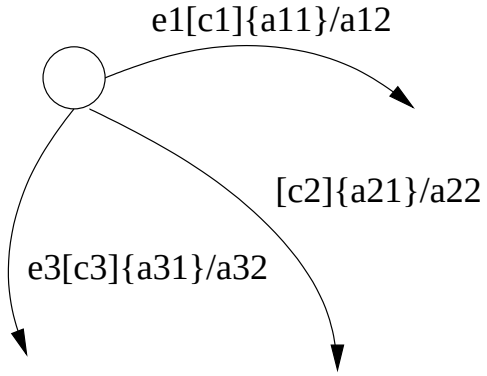

Figure 4: A junction with several transitions

3.2 Backtracking

Transitions in a list are evaluated sequentially until one succeeds. A transition succeeds only if it leads, eventually through a flowchart, to a state or a terminal junction (*i.e.* a junction without outgoing transition). This allow the definition of backtracking algorithm, which combined with instantaneous modification of the global environment can lead to unexpected behaviors. Figure 5 presents a flowchart whose evaluation can involve backtracking. Supposing `X` as the

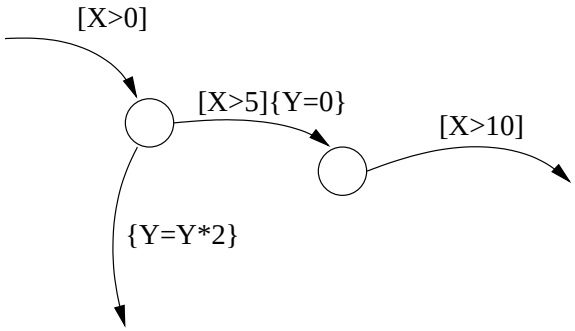

Figure 5: Backtracking

value 7 in the current environment, The first transition succeeds. At the junction, the guarded transition is evaluated, its condition, `X>5` is true, the action is executed immediately, `Y` is set to 0. The next transition fails, as $\texttt{X} \leq \texttt{10}$, the whole branch fails. Evaluation then backtracks to the previous branching point, where the unguarded transition is now taken, and its action executed. Notice here that, although the first branch failed, the action `Y=0` was executed instantaneously. Actions left pending on a failing path are discarded and never executed.

3.3 Inter-level Transitions

Stateflow supports inter-level transition (or *super-transition*) between states. Those cross different levels of state hierarchy. While being present in Statecharts, they are often considered harmful, and are often forbidden to allow for a simpler formalization. They are an important feature of Stateflow, and most programs are making use of them. The stopwatch example (figure 1), contains several of these: all the transition guarded by `START` are inter-level.

Inter-level transition have a complex behavior. Their effect is clearly non local, as they cross components boundaries. As they change the control point of hte program, they implement a form of jump. However, they do so with an important twist: when moving the control from one state to another, all the states crossed on the way have to be closed, or opened. For example, in the stopwatch, when taking the transition from the state `Reset` to `Running`, the state `Stop` has to be closed, and the state `Run` opened. As closing/opening states can be associated to actions, those have to be executed in the right order before making the jump. As an added complexity, jumping to the new location can fail, if we jump to a junction which does not lead to a state: we might have to jump back. This last features was not captured by our previous semantics[8], restricting the language. As we will see, continuation allow us to give a denotation to this behavior.

3.4 A Semantics for Transitions

The behavior associated to a transition can be described informally as: *if the event is present, and the condition true, then evaluate the first action, keep the second one pending and continue by evaluating the destination, else continue.* If we try to formalize this definition, we need to explain the precise meaning of *keep the second one pending* and *else continue*. How should we keep pending actions, and what does continuing mean? Those questions are very close to the questions arising when trying to give a semantics to, for example, jumps in imperative languages. The question of jumps was answered by introducing continuations[16]. Continuations allow one to manipulate the *rest of the program*, modify it, or ignore it altogether. We propose to use continuations to give a semantics to the transition mechanism. A transition can have two followups, or *rest of the program*, depending whether it is successful or not. A semantic function for transitions thus takes two additional arguments: the continuation corresponding to those two cases.

If we have a success continuation, and a failure continuation, the informal description of the behavior of a transition can now be reformulated as, *if the event is present, and the condition true, then evaluate the first action, add the second one to the success continuation and continue by evaluating the destination, else evaluate the failure continuation.* This new description is much more precise, it details that leaving the second action pending actually means adding it to the success continuation, and continuing means evaluating the failure continuation.

The interest of being able to explicitly manipulate continuations appears when evaluating lists of transition. Evaluating a list of transition consists in evaluating them in order until one succeeds. However, as we have seen, when combined with junctions, a transition can initially succeed, then lead to only failing transitions, in which case backtracking is needed. Thanks to our failure continuations, the tasks

of keeping track of the backtracking points, and evaluating transitions in the correct environment is fairly easy. We simply formulate the evaluation of a list of transition as: *evaluate the first transition, with a failure continuation that is the evaluation of the rest of the list.*

Those descriptions actually leads to the formal definition of the semantic functions for transitions and lists of transition. Before giving these however, we need to formaly define the Stateflow language, and introduce notations for environments and continuations.

4. DENOTATIONAL SEMANTICS

We now define a formal language for Stateflow, and give this language a denotational semantics. This section gives all the definitions for the notations used previously.

4.1 The Language

Figure 6 presents a formalization of Stateflow. A program (*i.e.* a chart) is a list of source components, that is, state and junction definitions. A state definition associates a name to a tuple composed of a triplet of actions, two lists of outer and inner transitions and an internal composition. A composition is either exclusive (*Or*), and given by a list of default transitions and a list of state names, or a parallel composition (*And*) given by a list of state names. A transition is defined by a triggering event, a condition, two actions, and a destination. A destination is either a path or a junction. A path is a list of state names. Finally, a junction definition associates a junction name to a list of transitions.

$$
\begin{array}{rrcl}
\textit{Program} & P & ::= & (s, [src_0, ..., src_n]) \\
\textit{SrcComp} & src & ::= & p : sd \mid j : T \\
\textit{StateDef} & sd & ::= & ((a_e, a_d, a_x), T_o, T_i, C) \\
\textit{Comp} & C & ::= & \textit{Or}\ (T, [s_0, ...s_n]) \mid \textit{And}\ ([s_0, ...s_n]) \\
\textit{Trans} & t & ::= & (e, c, (a_c, a_t), d) \\
\textit{Dest} & d & ::= & p \mid j \\
\textit{TransLst} & T & ::= & \emptyset | t.T \\
\textit{Path} & p & ::= & \emptyset | s.p
\end{array}
$$

Figure 6: The Stateflow Language

Actions a, events e, and conditions c are left abstract. State names s and junction names j are unique names. This language is close to the one used in [8]. In a similar way, lists of transitions and lists of states are supposed to be well-ordered, as we have seen previously, order of execution in Stateflow rely on graphical arguments which can not be used in this textual representation.

4.2 Environments and Continuations

A Stateflow program runs in an environment where variables are bound to values. We also want to keep track of active states in the programs, to do so, we add a boolean variable for each state. This variable indicates whether a state is active or not:

$$
\begin{array}{rrcl}
\textit{Env} & \rho & ::= & \{\ x_0 : v_0, ..., x_n : v_n, \\
 & & & \quad p_0 : b_0, ..., p_k : b_k\ \}
\end{array}
$$

We use the notation $\rho(x)$ to denote the value bound to variable x in the environment ρ. We also introduce two function to change the boolean variable associated to a state: `open` ρ p returns environment ρ where state p is bound to *true*, conversely `close` ρ p returns ρ with state p bound to *false*.

As is done in semantics for jumps[17], we build an environment for continuations, associating continuations to *labels*, here to state names. Continuation environments are defined as follows:

$$
\begin{array}{l}
\textit{Kenv}\ \theta ::= \\
\quad \{\ p_0 : (\mathcal{S}[\![p_0 : sd_0]\!]^e\ \theta, \mathcal{S}[\![p_0 : sd_0]\!]^d\ \theta, \mathcal{S}[\![p_0 : sd_0]\!]^x\ \theta), \\
\quad\ ... \\
\quad\ p_n : (\mathcal{S}[\![p_n : sd_n]\!]^e\ \theta, \mathcal{S}[\![p_n : sd_n]\!]^d\ \theta, \mathcal{S}[\![p_n : sd_n]\!]^x\ \theta), \\
\quad\ j_0 : \mathcal{T}[\![T_0]\!]\ \theta, ..., j_k : \mathcal{T}[\![T_k]\!]\ \theta\ \}
\end{array}
$$

A continuation environment θ is a list of bindings from path to a triplet of functions, and from junction names to a function. These function are the semantic functions associated to source element. For states, the first function enters the state, the second one executes it, and the last one exits it; we detail their definitions in the next section. We use the notation $\theta^e(p)$ (resp. $\theta^d(p)$, $\theta^x(p)$) to denote the entering (resp. executing, exiting) function associated to path p in the environment θ. Similarly, $\theta(j)$ denotes the function bound to junction j in θ.

When evaluating transitions, we need to pass them possible continuations, to execute depending on the outcome of evaluating the transition. We thus have two kinds of continuations: *success* continuations and *fail* continuations. A success continuation is a function of type:

$$k+ : \textit{Env} \rightarrow \textit{path} \rightarrow \textit{Env}$$

It takes as arguments the destination state of the transition. A fail continuation simply has type:

$$k\text{-} : \textit{Env} \rightarrow \textit{Env}$$

4.3 Semantic Rules

The rules are written in a a meta-language with lambda-abstraction, application, conditional and local let-binding. The rules for composition also use a folding operator over list. We give here the rules for Stateflow itself, as the action language as been left abstract, we suppose given semantic functions for evaluating actions ($\mathcal{A}[\![.]\!]$) and boolean conditions ($\mathcal{B}[\![.]\!]$). Those are of type:

$$
\begin{array}{rcl}
\mathcal{A}[\![]\!] & : & \textit{action} \rightarrow \textit{KEnv} \rightarrow \textit{Env} \rightarrow \textit{Env} \\
\mathcal{B}[\![]\!] & : & \textit{condition} \rightarrow \textit{Env} \rightarrow \textit{Bool}
\end{array}
$$

4.3.1 Transitions

This section formalize the informal descriptions given previously in 3.4. The evaluation of a transition takes place in an environment, a continuation environment, with two possible continuations, and is triggered by an event. It produces a new environment. The type of the semantics function is thus the following:

$$\tau[\![.]\!] : \texttt{trans} \rightarrow \texttt{env} \rightarrow \texttt{kenv} \rightarrow \texttt{k+} \rightarrow \texttt{k-} \rightarrow \texttt{event} \rightarrow \texttt{env}$$

And its definition follows the description seen before: : *"if the event is present, and the condition true, then evaluate the first action, add the second one to the success continuation and continue by evaluating the destination, else evaluate the*

failure continuation".

$$\begin{array}{l}\tau[\![(e_t, c, (a_c, a_t), d)]\!]\ \theta\ \rho\ success\ fail\ e = \\ \quad \texttt{if}\ (e_t = e) \wedge (\mathcal{B}[\![c]\!]\ \rho)\ \texttt{then} \\ \quad\quad \texttt{let}\ success' = \\ \quad\quad\quad \lambda\rho_s.\lambda p.\texttt{if}\ \ p = []\ \texttt{then}\ \ success\ \rho_s\ p \\ \quad\quad\quad\quad\quad\quad\quad\quad \texttt{else}\ \ success\ (\mathcal{A}[\![a_t]\!]\ \theta\ \rho_s)\ p\ \texttt{in} \\ \quad\quad \mathcal{D}[\![d]\!]\ \theta\ (\mathcal{A}[\![a_c]\!]\ \theta\ \rho) success'\ fail\ e \\ \quad \texttt{else} \\ \quad\quad fail\ \rho\end{array}$$

The additional test in the extended success continuation is used to detect if we have reached a state, in which case we evaluate the pending action, or a terminal junction, in which case we simply succeed.

The rule for lists of transition is the rule that, by carefully building the continuations given to the transitions is adding support for backtracking. As for a transition, a list of transitions can either succeed or fail, the semantic function thus takes two continuations. Given a list of transition, we first evaluate its head. This evaluation is done with the current success continuation, the failure continuation on the other hand needs to be build: it consists in the evaluation of the rest of the list. A singleton is evaluated with the current failure continuation as argument, and an empty list succeeds. This is used to differentiate between failure of all the transition, and success upon reaching a terminal junction.

$$\begin{array}{l}\mathcal{T}[\![\emptyset]\!]\ \theta\ \rho\ success\ fail\ e = success\ \rho\ [] \\ \mathcal{T}[\![t.\emptyset]\!]\ \theta\ \rho\ success\ fail\ e = \tau[\![t]\!]\ \theta\ \rho\ success\ fail\ e \\ \mathcal{T}[\![t.t'.T]\!]\ \theta\ \rho\ success\ fail\ e = \\ \quad \texttt{let}\ fail' = \lambda\ \rho_f.\mathcal{T}[\![t'.T]\!]\ \theta\ \rho_f\ success\ fail\ e\ \texttt{in} \\ \quad \tau[\![t]\!]\ \theta\ \rho\ success\ fail'\ e\end{array}$$

Those two rules implement the whole transition mechanism of Stateflow. While not *simple*, continuation based semantics are often a bit puzzling, they are surprisingly short given the complexity of this mechanism. This is a very interesting result, first, it shows that Stateflow, while having a complex behavior, is far from badly conceived, in particular we don't need to encode a vast number of special cases. Second, those rules can be directly, and easily implemented.

4.3.2 States

As seen when defining the continuation environments, states have three semantic functions. Those function describe, respectively, how to enter ($\mathcal{S}[\![]\!]^e$), execute ($\mathcal{S}[\![]\!]^d$), and leave ($\int[\![]\!]^x$) a state. The rules take as argument a state declaration given by a path and a state definition. Entering the state can be done *from top*, i.e. because its parent state activates it, or *from bottom*, if the activation of one of its children activates it. In the first case, we do not know which child to open (case []), in the second case, we have an explicit path leading to it.

$$\begin{array}{l}\mathcal{S}[\![p : ((a_e, a_d, a_x), T_o, T_i, C)]\!]^e\ \theta\ \rho\ []\ e = \\ \quad \texttt{open}\ \ p \circ \mathcal{C}[\![C]\!]^e\ \theta\ (\mathcal{A}[\![a_e]\!]\ \theta\ \rho)\ e \\ \mathcal{S}[\![p : ((a_e, a_d, a_x), T_o, T_i, C)]\!]^e\ \theta\ \rho\ s.p_d\ e = \\ \quad \texttt{open}\ \ p \circ \theta^e(p.s)(\mathcal{A}[\![a_e]\!]\ \theta\ \rho)\ p_d\ e\end{array}$$

Exiting a state is straightforward:

$$\begin{array}{l}\mathcal{S}[\![p : ((a_e, a_d, a_x), T_o, T_i, C)]\!]^x\ \theta\ \rho\ e = \\ \quad \texttt{close}\ \ p \circ \mathcal{A}[\![a_x]\!]\ \theta \circ \mathcal{C}[\![C]\!]^x\ \theta\ \rho\ e\end{array}$$

The most complex rule is the execution one. It needs to build the continuations required to evaluate the outer and inner transitions. The definition uses the intermediate function `open_path` (defined below) to correctly close, and open states as needed when a transition fires. The *success* continuations calls `open_path` with the state as source. The *fail* continuation, called if no outer transition succeeds, execute the inner transitions with a failing continuation executing the composition and the internal actions. Thus this definition can be read as: *try the outer transitions, if this fails, try the inner ones, if this fails, execute the internal action and the composition.*

$$\begin{array}{l}\mathcal{S}[\![p : ((a_e, a_d, a_x), T_o, T_i, C)]\!]^d\ \theta\ \rho\ e = \\ \quad \texttt{let}\ fail = \lambda\rho_f. \\ \quad\quad \texttt{let}\ fail_i = \lambda\rho_{fi}.\mathcal{C}[\![C]\!]^d\ \theta\ (\mathcal{A}[\![a_d]\!]\ \theta\ \rho_{fi})\ e\ \texttt{in} \\ \quad\quad \texttt{let}\ \ success_i = \\ \quad\quad\quad \lambda\rho_{si}.\lambda p_d.\texttt{open\_path}\ \theta\ \rho_{si}\ []\ p\ p_d\ fail_i\ e\ \texttt{in} \\ \quad\quad \mathcal{T}[\![T_i]\!]\ \theta\ \rho_f\ success_i\ fail_i\ e\ \texttt{in} \\ \quad \texttt{let}\ success = \lambda\rho_s.\lambda p_d.\texttt{open\_path}\ \theta\ \rho_s\ []\ p\ p_d\ fail\ e\ \texttt{in} \\ \quad \mathcal{T}[\![T_o]\!]\ \theta\ \rho\ success\ fail\ e\end{array}$$

Let us now define the `open_path` function. This is the function implementing the inter-level mechanism, if the destination of the transition is a state (*i.e.* a non-empty path)it first finds the higher state in the hierarchy which need to be closed by computing the common prefix of the source and destination paths. It then closes it, that will, by definition close its whole sub-tree. Finally, the destination state is entered. If the destination is a terminal junction, we simply call the continuation.

$$\begin{array}{l}\texttt{open\_path}\ \theta\ \rho\ p\ p_s\ []\ term\ e = term\ \ \rho \\ \\ \texttt{open\_path}\ \theta\ \rho\ p\ x.p_s\ y.p_d\ term\ e = \\ \quad \texttt{if}\ x = y\ \texttt{then} \\ \quad\quad \texttt{open\_path}\ \theta\ \rho\ p.x\ p_s\ p_d\ e \\ \quad \texttt{else} \\ \quad\quad \texttt{let}\ \rho_x = \theta^x(p.x)\ \rho\ e\ \texttt{in} \\ \quad\quad \theta^e(p.y)\ \rho_x\ p_d\ e\end{array}$$

4.3.3 Destinations

If the destination is a state, we just execute the success continuation. If it is a junction, we fetch the corresponding continuation from the environment and execute it.

$$\begin{array}{lcl}\mathcal{D}[\![p]\!]\ \theta\ \rho\ success\ fail\ e & = & success\ \rho\ p \\ \mathcal{D}[\![j]\!]\ \theta\ \rho\ success\ fail\ e & = & \theta(j)\ \rho\ success\ fail\ e\end{array}$$

4.3.4 Compositions

Compositions are just iterating through their components. The default transitions of an *Or* composition have not to fail, their failure continuation raises an error ($\perp$) – it is clearly desirable to statically ensure that such a case will not happen, this can be done by syntactic restrictions, or by formal tools. We have implemented such a tool, making use of automatic theorem proving to check the absence of possible failure. The success continuation open the reached state.

$$\begin{array}{l}\mathcal{C}[\![Or\ (T, [])]\!]^e\ \theta\ \rho\ e = \rho \\ \mathcal{C}[\![Or\ (T, S)]\!]^e\ \theta\ \rho\ e = \mathcal{T}[\![T]\!]\ \theta\ \rho\ (\lambda\rho_s.\lambda p.\theta^e(p)\ \rho_s\ []\ e)\ \perp\ e \\ \\ \mathcal{C}[\![Or\ (T, [])]\!]^d\ \theta\ \rho\ e = \rho \\ \mathcal{C}[\![Or\ (T, p.S)]\!]^d\ \theta\ \rho\ e = \\ \quad \texttt{if}\ \rho(p)\ \texttt{then}\ \theta^d(p)\ \rho\ e\ \texttt{else}\ \mathcal{C}[\![Or\ (T, S)]\!]^d\ \theta\ \rho\ e \\ \\ \mathcal{C}[\![Or\ (T, S)]\!]^x\ \theta\ \rho\ e = \\ \quad \texttt{fold}\ (\lambda p.\lambda\rho.\texttt{if}\ \rho(p)\ \texttt{then}\ \theta^x(p)\ \rho\ e\ \texttt{else}\ \rho)\ S\ \rho\end{array}$$

And composition are straightforward.

$$\mathcal{C}[\![And\ (S)]\!]^e\ \theta\ \rho\ e = \texttt{fold}\ (\lambda p.\lambda\rho.\theta^e(p)\ \rho\ []\ e)\ S\ \rho$$

$$\mathcal{C}[\![And\ (S)]\!]^d\ \theta\ \rho = \texttt{fold}\ (\lambda p.\lambda\rho.\theta^d(p)\ \rho\ e)\ S\ \rho$$
$$\mathcal{C}[\![And\ (S)]\!]^x\ \theta\ \rho = \texttt{fold}\ (\lambda p.\lambda\rho.\theta^x(p)\ \rho\ e)\ S\ \rho$$

4.3.5 Main program

The semantics function of the main program builds the continuation environment from the code. It initializes the main state if required, *i.e*, if we are at the first instant of the execution, otherwise it executes it.

$$\mathcal{P}[\![(s, Srcs)]\!]\ \rho = \texttt{if}\ \ \rho(s)\ \texttt{then}\ \theta^d(s)\ \texttt{else}\ \theta^e(s)$$
$$\text{where } \theta = \{k \mid k = SrcCont\ \theta\ Srcs\}$$

4.4 Local Events

Stateflow event mechanism supports instantaneous sending and receiving of local events, to parts of the design, or the whole design at once. This allows the definition of recursive behaviors, which naturally can lead to non-termination.

In [8], we had to adapt our semantics in order to support local events, here, the semantics directly supports the full local event mechanism. However, it can lead to infinite recursive calls, and thus requires static analysis, or syntactic restrictions as we did in our previous work, to ensure safety.

Sending an event is done, in the action language, through an action `send`. `send` (p, e) sends the event e to the state p. The semantic function associated to this is the following:

$$\mathcal{A}[\![\texttt{send}\ (p, e)]\!]\ \theta\ \rho = \texttt{if}\ \ \rho(p)\ \texttt{then}\ \theta^d(p)\ \rho\ e\ \texttt{else}\ \ \rho$$

If the destination state is active, it is processed with the event as argument, otherwise, we simply return an unmodified environment.

4.5 Correction of the Semantics

The semantics of Stateflow being informally defined as the result of the simulation when a program is run using the Mathworks tools, establishing correction is not possible. We have validated our semantics by systematic comparison of traces produced by the Mathworks tools, our previous interpreter build from an operational semantics, and interpretation of this semantics. Building an interpreter from the semantics is trivial: the rules can be entered nearly as-is as ML code for example, and evaluated.

An attractive possibility would be to prove equivalence of our two semantics. However, this is difficult as they consider different formalization of the language and different subsets of it. Although, as this one consider a strictly larger subset we could try to establish a partial correction lemma.

5. COMPILING STATEFLOW

One of our motivations in developing this denotational semantics is to formalize a compilation scheme for Stateflow. Indeed, it is very easy to derive a compilation process from the semantics we presented: given a program, partial evaluation of the semantic rules produces a functional program, taking an environment and an event as inputs, and returning a modified environment. The partial evaluation process consists in inlining the continuations, and reducing redexes.

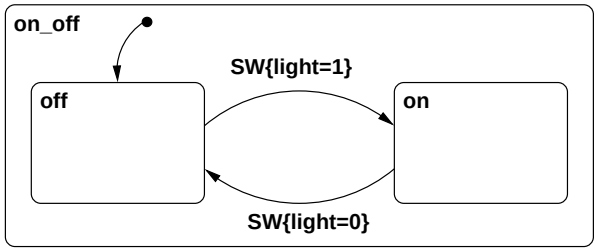

Figure 7: A light-switch

5.1 Example

Consider the example of figure 7, implementing a simple light-switch. It reacts to a single event `SW` and toggles between two states `on` and `off`. When the `on` state is active, the `light` variable is true, when the state `off` is active, the variable is false. The semantics of this program has the following form, given by the rule for the main program:

$$\mathcal{P}[\![\texttt{on\_off}, srcs]\!]\ \rho\ e =$$
$$\quad \texttt{if}\ \rho(\texttt{on\_off})\ \texttt{then}\ \theta^d(\texttt{on\_off})\ \texttt{else}\ \theta^e(\texttt{on\_off})$$

where $srcs$ is the list of source components in the charts, and θ the continuation environment build from those. The value associated to executing the chart,$\theta^d(\texttt{on\_off})$, is by definition the semantic function for executing a state, S_d, applied to this chart. Inlining this function, we get the following:

$$\begin{array}{l}\mathcal{P}[\![\texttt{on\_off}, srcs]\!]\ \rho\ e = \\ \quad \texttt{if}\ \rho(\texttt{on\_off})\ \texttt{then} \\ \quad\quad \texttt{fold}\ (\lambda p.\lambda\rho.\texttt{if}\ \ \rho(p)\ \texttt{then}\ \theta^d(p)\ \rho\ e)\ \{\texttt{off}, \texttt{on}\}\ \rho \\ \quad \texttt{else} \\ \quad\quad \theta^e(\texttt{on\_off})\end{array}$$

Unfolding this new definition, and pursuing evaluations and reductions as much as possible raises the following code (we use the notation $\rho[x \leftarrow v]$ to denote the environment ρ where the variable x is associated to the value v):

$$\begin{array}{l}\mathcal{P}[\![\texttt{on\_off}, srcs]\!]\ \rho\ e = \\ \quad \texttt{if}\ \rho(\texttt{on\_off})\ \texttt{then} \\ \quad\quad \texttt{if}\ \rho(\texttt{off})\ \texttt{then} \\ \quad\quad\quad \texttt{if}\ \ e = \texttt{SW}\ \texttt{then} \\ \quad\quad\quad\quad \texttt{let}\ \rho = \rho[\texttt{light} \leftarrow 1]\ \texttt{in} \\ \quad\quad\quad\quad \texttt{let}\ \rho = \rho[\texttt{off} \leftarrow \mathit{false}]\ \texttt{in} \\ \quad\quad\quad\quad \texttt{let}\ \rho = \rho[\texttt{on} \leftarrow \mathit{true}]\ \texttt{in} \\ \quad\quad\quad\quad \rho \\ \quad\quad\quad \texttt{else} \\ \quad\quad\quad\quad \rho \\ \quad\quad \texttt{else if}\ \rho(\texttt{on})\ \texttt{then} \\ \quad\quad\quad \texttt{if}\ \ e = \texttt{SW}\ \texttt{then} \\ \quad\quad\quad\quad \texttt{let}\ \rho = \rho[\texttt{light} \leftarrow 0]\ \texttt{in} \\ \quad\quad\quad\quad \texttt{let}\ \rho = \rho[\texttt{on} \leftarrow \mathit{false}]\ \texttt{in} \\ \quad\quad\quad\quad \texttt{let}\ \rho = \rho[\texttt{off} \leftarrow \mathit{true}]\ \texttt{in} \\ \quad\quad\quad\quad \rho \\ \quad\quad\quad \texttt{else} \\ \quad\quad\quad\quad \rho \\ \quad\quad \texttt{else} \\ \quad\quad\quad \texttt{let}\ \rho = \rho[\texttt{off} \leftarrow \mathit{true}]\ \texttt{in} \\ \quad\quad\quad \texttt{let}\ \rho = \rho[\texttt{on\_off} \leftarrow \mathit{true}]\ \texttt{in} \\ \quad\quad\quad \rho\end{array}$$

This code can be translated to either a functional or an im-

perative language by simple pretty-printing. In a functional language, the environment can be represented using named records, in an imperative one it can be modified in place. The reductions we have done are inlining of semantic functions and continuations, and beta-reduction. It is easy to see that the final code does not contain any abstraction left: all lambdas are applied, and disappear during translation.

5.2 Limitations and Extensions

This simple compilation scheme by interpretation of the semantics rules, can of course be explosive, or even loop if the Stateflow program itself contains loops. Different solutions can be proposed to overcome this problem, depending on the target language, and the end purpose. To avoid code-size explosion, a simple solution is to limit the amount of inlining performed. In practice, we haven't encountered such problems even when compiling industrial-size examples. Loops can be detected, and compiled as loops in the target language, our C backend does this. For model-checking purpose, one possibility is to expand the loops a given finite number of times, and verify we cannot exaust those expansions, our SAL backend does that.

5.3 Implementation

As was our goal when starting this work, we have implemented a new Stateflow compiler. This implementation is based on the denotational semantics, and following the compilation scheme presented here. It contains special support for loops. The code of this compiler is about 20% shorter than our previous implementation, while supporting a richer language. At the core of the implementation is the semantics, which is implemented directly. As an example, here is the code implementing the rule for evaluating a list of transition, the code is written in OCaml[10]:

```
and trans_list tl kenv env success fail e =
  match tl with
  | [] -> success env []
  | [x] -> trans x kenv env success fail e
  | t::tl' ->
      let fail' =
        fun env_f ->
          trans_list tl' kenv env_f success fail e in
      trans t kenv env success fail' e
```

This direct translation of the semantics makes the code much easier to maintain. The new compiler is also more efficient and scales well, we have been able to experiment with examples of industrial size (over 50K lines), and obtained satisfying code (10K lines).

The compiler can produce SAL[4], C, and OCaml code. As is presented here, the compiler first produces a program in an intermediate form, from which all these can be obtained by pretty-printing. Each of those pretty-printer is around 150 lines of code.

The quality of the code is very satisfactory. Is it short, very readable and reasonnably efficient. The C code compares well with other Stateflow to C compilers (see 6). When generating SAL code, the models obtained are directly exploitable, we use them for automatic test-cases generation[7] without modification, the compiler automatically instruments the code with the required *trap-conditions*.

6. RELATED AND FUTURE WORKS

We have presented a complete formalization of Stateflow, and its denotational semantics. The semantics is continuation-based, this allows us to capture complex features of the language, in particular describe its transition mechanism in its entirety. This semantics also gives us a simple and automatic compilation scheme.

6.1 Related Works

This work is naturally related to other works on formalization of Stateflow, [18], [8], and [14]. In [18], Tiwari *et al.* present a translation of both Simulink and Stateflow to pushdown automata. This approach shows rapid explosion of the number of variables needed to encode the Stateflow chart, and originally drove us to try a different approach. In [8], we presented an SOS semantics for Stateflow. This semantics, being low-level, allowed us to understand precisely the language. In particular understand that, in essence, Stateflow is an imperative language, and semantics tools developed for imperative language are adapted to its description. This semantics is also a good basis for developing syntax directed static analysis. However, as said before, it shows limits when trying to formalize the full language, and does not formalize a compilation scheme for Stateflow. In [14], Caspi *et al.* propose simple static analysis that can be used to restrict Stateflow to a "safe" subset. They propose an unformal translation of this subset to Lustre[6]. They consider a restricted transition mechanism, not allowing for example inter-level transitions. Our goal in this work was different, as we wanted to consider as much as possible the whole language, and we where interested in defining a formal semantics. Statecharts supports an inter-level transition mechanism. Huizing[9] proposed the use of continuations to describe those. However, Statecharts does not support other features of Stateflow like composition of transitions through junctions or backtracking. In general, the two language are really different, and their semantics have very little relations.

This work is also related to works on compilation of modelling languages, in particular works done by Anton *et al*[1] within the Forges project at Kestrel Institute. Their approach is based on a Stateflow interpreter written in the Oscar language. It seems that they only consider a subset of the language [12]. An interesting point is that the code they produced has received good reviews from industrial users, the C code produced by our compiler is very close to theirs on available examples.

6.2 Future Works

The semantics presented here constitutes a strong basis for reasoning about Stateflow and its compilation. Future works are three-folds. First, in the domain of the compilation of modelling languages. Industrial users of Stateflow, or other similar tools often design the model, then hand-write the corresponding code. Going to generating code would clearly be an important step. Formally defined and easily adaptable code generators that can produce good code might help in that regard. The other area where we are planning on using this work is in the use of formal methods to help in the design of systems. The area of formal methods has produced tools that can provide amazing results, these can be used not only to certify a model, but also to help in its design, by providing automatic, early, and precise diagnosis. This requires a strong understanding of the

modelling language. Our work on automatic test-cases generation[7] lies in this area. Finally, a really interesting, and necessary work is to better understand Simulink, and the combination Stateflow/Simulink, to propose a formalization of it. This would open the way to integration-level analysis, considering a controller in its environment, and not in isolation. We are considering an approach based on both [3], for the discrete part of SIMULINK, and [18], for the continuous part. Understanding precisely the separation between the discrete and the continuous parts of a design is critical to efficient analysis.

7. ACKNOWLEDGEMENTS

The author would like to thank John Rushby, for his many helpful comments on this work.

8. REFERENCES

[1] John Anton, Paulo da Costa, and Lindsay Errington. Formal synthesis of generators for embedded systems. Technical report, Kestrel Technology, May 2005.

[2] Daniel Buck and Andreas Rau. On modelling guidelines: Flowchart patterns for Stateflow. *Softwaretechnik-Trends*, 21(2), August 2001.

[3] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, and Stavros Tripakis. Translating discrete-time Simulink to Lustre. In *ACM Conference on Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 84–99. Springer, October 2003.

[4] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *Computer-Aided Verification, CAV 2004*, volume 3114 of *LNCS*, pages 496–500, Boston, MA, July 2004. Springer.

[5] 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`.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[7] Grégoire Hamon, Leonardo de Moura, and John Rushby. Generating efficient test sets with a model checker. In *International Conference on Software Engineering and Formal Methods (SEFM)*, pages 261–270. IEEE Computer Society Press, 2004.

[8] Grégoire Hamon and John Rushby. An operational semantics for Stateflow. In *Fundamental Approaches to Software Engineering (FASE)'04*, volume 2984 of *LNCS*, pages 229–243, Barcelona, Spain, 2004. Springer.

[9] C. Huizing. *Semantics of Reactive Systems: Comparison and Full Abstraction*. PhD thesis, Eindhoven Technical University, 1991.

[10] Xavier Leroy. The Objective Caml system release 3.08 Documentation and user's manual. Technical report, Institut National de Recherche en Informatique et Automatique (INRIA), 2003.

[11] The Mathworks. *Stateflow and Stateflow Coder, User's Guide*, release 13sp1 edition, September 2003.

[12] Bill Milam. Mobies, midterm report. Technical report, Ford Research, 2002. Available at `http://vehicle.me.berkeley.edu/mobies/evaluations/tools/mobies_evaluation_codegen.pdf`.

[13] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

[14] Norman Scaife, Christos Sofronis, Paul Caspi, Stavros Tripakis, and Florence Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In *ACM international conference on Embedded software (EMSOFT)*, Pisa, Italy, September 2004.

[15] David A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[16] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Computing Laboratory, Oxford University, England, 1974. Reproduced in [17].

[17] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1-2):135–152, April 2000.

[18] A. Tiwari, N. Shankar, and J. Rushby. Invisible formal methods for embedded control systems. *Proceedings of the IEEE*, 91(1):29–39, January 2003.

[19] Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1-2):131–133, April 2000.