A Parametric Structural Operational Semantics for Stateflow, UML Statecharts, and Rhapsody

Mike Whalen
Department of Computer Science and Engineering
University of Minnesota

Abstract

In this report, we define a parametric structural operational semantics that can be used to define the behavior of three Statecharts variants: Stateflow from the Mathworks, Inc., UML Statecharts from the Object Management Group, and Rhapsody from Rational/IBM Corporation. We believe that these dialects are the most commonly used variants of Statecharts in industrial applications, and are increasingly used to construct safety-critical applications. We believe that our semantics for each dialect is more complete than prior research and matches the informal documentation of each notation more closely than other approaches. In the formalization process, we have discovered deep similarities between the semantics, and we are able to create a parametric operational semantics that factors out the variabilities between the dialects in a modular way.

1. Introduction

Following the publication of Harel’s seminal paper describing Statecharts [Harel87:statecharts], dozens, if not hundreds, of notational and semantic variants have been created by research groups and different commercial companies. Over time, most of the variants have been abandoned, leaving a handful of widely used dialects of Statecharts. Some, such as SyncCharts from Esterel Technologies [Andre03:synccharts] were designed from a formal semantics, but most dialects were not originally designed with an eye towards formalization. Unfortunately, this means that many dialects exhibit complex behaviors that can be difficult to formalize (or understand!) for large charts, or simply charts whose features interact in ways not expected by language designers.

In this report, we formalize a significant subset of the three most popular Statecharts dialects: Stateflow from the Mathworks, Inc. [Stateflow], UML Statecharts from the Object Management Group [OMG:UML], and Rhapsody from Rational/IBM Corporation [Harel01:rhapsody]. The Statecharts variants in question can have complex behaviors and few formalizations have attempted to cover any single dialect completely. In our approach, we believe that we provide a more complete formalization of each dialect than has been previously attempted. Furthermore, we have discovered deep similarities between the semantics, and are able to provide most of the semantics in common “core” rules. The semantic variations between the dialects can be captured in only a handful of parametric rules.

We are motivated to create the semantics by a desire to analyze models in each of the dialects, and further to check models for different kinds of equivalence across dialects. Complex projects, such as the Constellation project in NASA, have used all three dialects in different software modules within a system architecture. The semantics provide a mechanism for checking whether models will behave equivalently (or at least preserve important safety properties) if they are interpreted in another dialect. Additionally, the UML Statecharts and Rhapsody dialects are underspecified: different tools may meet the evaluation constraints defined on the notation and yet yield different behaviors. We model this underspecification as non-determinism, and our semantics allow checking whether a model will behave as intended under all possible deterministic interpretations of the model.

More specifically, within a single dialect, one would like to check whether the model satisfies
- **Well-formedness constraints** to ensure that the model does not ‘go wrong’ structurally during its execution,
- **Determinism constraints** to ensure that the behavior of the machine is deterministic for all input event sequences
- **Temporal properties**, which check the model against declarative requirements

Between dialects, we can also check for various kinds of equivalences, including:

- **State equivalence**, which demonstrates that the state machines quiesce to the same state configuration given the same input trace
- **Output equivalence**, which demonstrates that the state machines produce the same output trace given the same input trace
- **Property equivalence**, which checks that the higher-level requirements are maintained between dialects.

The semantics in this report is derived primarily from a denotational semantics for Stateflow written by Gregoire Hamon [Hamon05:stateflow] and secondarily from an Abstract State Machine formulation of UML Statecharts by Egon Börger [Borger00:UML]. Hamon’s elegant work provides not only a concise formal description for the language but also a means to transparently create an efficient compiler for the language. Börger’s work presents a nice formulation of UML Statecharts and defines syntactic equivalences between some elements of UML Statecharts that we also utilize that significantly simplify the formal presentation.

We have rewritten and generalized Hamon’s rules as a structural operational semantics in order to straightforwardly describe the variations in behavior (and non-determinism) that are possible in UML Statecharts and Rhapsody. While we do not claim that our semantics are “correct” (as there is no clear definition of correctness without a formal semantics), we believe that our semantics match the informal descriptions of behavior for each of the variants. We use [Stateflow] as the reference for Stateflow, [Harel01:rhapsody] as the reference for Rhapsody, and [OMG:UML] as the reference for UML Statecharts. Both Hamon [Hamon05:stateflow] and Borger [Borger00:UML] demonstrably diverge from the informal descriptions of semantics provided by the informal references for Stateflow and UML Statecharts, respectively, and we attempt to fix these subtle flaws in our presentation. Where there is ambiguity or incompleteness in the informal description (for example, the 29 problems with the informal description of UML Statecharts presented in [Fecher05:UML]), we describe our semantic choice and the consequences of that choice.

We present a semantics for most of Stateflow, Rhapsody, and UML Statecharts. The constructs that we ignore in the current semantics are:

- history junctions (All notations)
- entry and exit pseudostates (UML and Rhapsody)
- deferred, timing, and change events (UML)
- termination, creation, and destruction of objects (UML)
- do actions (UML)
- internal transitions (UML)

There are no significant difficulties in formalizing history junctions, internal transitions, entry and exit pseudostates, and deferred actions, and we expect to add these to the semantics in this report in the near future. The other activities are problematic because they either involve timing or actions upon the “external world” which cannot be easily formalized.

Additionally, we treat certain features of each notation as **syntactic sugar** and define them in terms of an immediate source-to-source transformation rather than define them directly within the operational semantics of the language. These features are:

- Counter events (Stateflow)
Finally, we describe only a handful of the expressions within the action and condition languages used by the notations formally. To formalize the entire action and condition language for each notation would be relatively straightforward, but would also be very verbose.

The operational semantics presented here are designed with translation in mind and can be straightforwardly converted to a translation tool to generate code from any of the three variants into a functional or imperative language. The generated code can be left non-deterministic by appealing to a choice construct, such as a random number generator, in the target language (good for analysis with code-level model checkers such as JPF). It can also be made more efficient by choosing a particular deterministic execution strategy for the code (this is the strategy pursued by the UML and Rhapsody code generation tools). In later sections of this report, we define alternate rules for more efficient compilation and also pre- and post-processing steps for further optimization.

Section 2 presents an informal description of the semantics for each of the dialects and provides examples of models that exhibit semantic discrepancies depending on the dialect used for interpretation. Section 3 presents preliminaries, including a description of structural operational semantics and presentation style for the rules. Section 4 describes the abstract syntax for the dialects and the formal semantics for each dialect. Section 5 presents related work. Section 6 describes future work and concludes.

2. Statecharts Notation

Statecharts is a graphical notation invented by David Harel [Harel87:statecharts, Harel96] that defines a generalization of finite state machines. Statecharts consist of finite state machines that can be composed hierarchically and in parallel. Finite state machine transitions are generalized to allow transition across machine borders. An example of a Statechart (in the Stateflow dialect) is shown in Figure 1.
Figure 1: An Example Stateflow Statechart

The solid lines define hierarchical state machines, and dashed lines define state machines that execute in parallel. For example, the PowerOn state contains three state machines that execute in parallel: FAN1, FAN2 and SpeedValue. Inside the FAN1 state is a sequential state machine consisting of the states On and Off. Transitions can be triggered either by events, such as the transition between PowerOn and PowerOff, triggered by the event SWITCH. Transitions can also be triggered by conditions, such as the transition between On and Off in FAN1, which is taken when the condition [temp < 120] is true. Statecharts can contain non-graphical variables, such as temp that can be used to describe or record numeric quantities. Additionally, transitions and states can perform actions that generate new events or assign variables. In Figure 1, there is an entry action in the PowerOff state that is performed when the state is entered: airflow = 0, and a during action in the SpeedValue state that is performed when the state is occupied (called a during action): du: airflow = in(FAN1.On) + in(FAN2.On). In the next few sections, we describe each of the common elements of Statecharts in more detail.

2.1 Pseudostates

In addition to states, statecharts have various mechanisms for simplifying complex transitions. The most important is pseudostates, which allow several simplifications of transitions. Pseudostates describe transient locations within a state machine; when entered, they must immediately be exited by a transition segment. Each dialect supports a different set of pseudostates; in the formalization, we define include the complete set of pseudostate types between the dialects, but for the moment we cover the common pseudostates: junctions, history connectors, and initial states.

---

1 Figure from The Mathworks Stateflow User Guide [Stateflow]
2.1.1 Junction Pseudostates

All of the notations have a notion of a junction pseudostate that allows complex transitions to be created from transition segments. An example containing a junction pseudostate is shown in Figure 2, where the small circle in Figure 2 denotes a junction. A junction has the effect of conjoining the guards and events from the segments joined by a junction. For example, in Figure 2, we could create an equivalent diagram with no junctions by turning each complete path into its own transition and conjoining the guards as shown in Figure 3.

![Figure 2: Statechart containing junction pseudostates](image)

![Figure 3: Equivalent Statechart containing no junction pseudostates](image)

2.1.2 History Pseudostates

History pseudostates are used to save and restore the state of a state machine. An example of a history pseudostate in the Stateflow notation is shown in Figure 4.
In Figure 4, the first time that state S1 is entered, the initial transition is used to enter state S2. When S1 is exited (due to the transition to S4), the currently active state is recorded by the history pseudostate. Suppose that in some execution, when S1 is exited, the child state S3 was active. In this case, when S1 is re-entered, S3 (rather than the default state S2) is also re-entered.

There are slight semantic differences in the implementation of history pseudostates within the three notations. For example, in UML Statecharts, the history behavior is only used if there exists a transition directly to the history pseudostate rather than to the parent state containing the pseudostate. UML Statecharts also allow deep history pseudostates, which restore the entire subgraph of an ancestor state, while Rhapsody and Stateflow only allow shallow history pseudostates that restore the immediate children of the state containing the pseudostate.

### 2.1.1 Initial Pseudostates

Initial states serve to define the ‘default’ transition into a child state for each parent state. When the parent state is entered, they define which child state will be initially active. They are denoted by a transition from a small filled oval, as seen in Figure 4. In this figure, there are transitions into S1 and S2 from initial pseudostates.

### 2.2 Transitions

Transitions in UML Statecharts and Rhapsody connect locations within a statemachine. In Statecharts, transitions have labels of the form: `Trigger [Guard] / Actions`. The `Trigger` describes the triggering event, the `Guard` describes an additional condition (Boolean expression) that must evaluate to True for the transition to fire, and `Actions` describe actions activities that are performed when the transition fires. `Actions` are either assignments to variables or generation of additional events. None of the label components are required.

Transitions in Statecharts may cross state boundaries, as shown in Figure 5, and may involve traversing through several pseudostates. We define a transition segment to be any arc between two locations within a chart (either states or pseudostates), and a transition or compound transition a sequence of transition segments that leads from a source state (or history junction) to a destination state through a sequence of zero or more intermediate pseudostates\(^2\). [OMG:UML].

---

\(^2\) In UML Statecharts and Rhapsody, a complete transition can originate in a set of states and terminate in a set of states in case join and fork pseudostates, respectively, occur along the transition path.
The Stateflow notation splits actions into condition actions and transitions actions that are performed at different times within the evaluation of the transition. The format of transitions for Stateflow becomes: Trigger \{Condition\} \{ConditionActions\} / TransitionActions. These differences between these two types of actions will be described in Section 2.4, which describes the evaluation of Stateflow.

2.4 Informal Sketch of Evaluation

The semantics of each Statecharts dialect center around the evaluation of an event. Given an event (which may be from the external environment or internally generated), the system finds the set of enabled transitions. A transition is enabled if its triggering event matches the current event and its guarding condition evaluates to true. A subset of these transitions then fires, causing the system to change state and potentially generate new events that further evolve the state machine.

The complex behavioral elements that are mostly common between the different dialects are:
1. How to exit and enter compound states (that is, states with child states) when transitions ‘fire’.
2. How to evaluate inter-level transitions, that is, transitions that cross state boundaries
3. How to evaluate compound transitions, that is, transitions containing multiple segments that are joined by pseudostates

In our view, the most complex component of the semantics for Statecharts involves evaluating complex transitions containing multiple segments. Fortunately, this portion of the semantics is largely common between the different semantics, and the core semantics is primarily concerned with this aspect.

The most significant differences between the dialects in this report have to do with
1. how and when internal events, that is, events generated by actions of the state machine, are consumed
2. how and when transitions without an explicit triggering event (called completion transitions in UML Statecharts and Rhapsody) are evaluated.
3. In the case that multiple conflicting transitions are simultaneously enabled, which subset are chosen to fire
4. Whether the evaluation of the set of enabled transitions and the update of system state for a particular event is performed once or incrementally as a sequence of sub-steps.

---

3 Figure from The Stateflow User’s Guide [Stateflow]
These aspects are dealt with in the parametric portion of the rule sets, and are instantiated separately for each dialect. Fortunately, these aspects are relatively straightforward to formalize, leading to relatively few variant rules between the different dialects.

### 2.8 Discrepancies between Semantics

A good example of the discrepancies between the dialects is shown in Figure 6. This example is taken from an excellent paper describing the differences between UML Statecharts, Rhapsody Statecharts, and “Classical” statecharts (the semantics of Harel’s STATEMATE tool) [Crane05:StatechartsDialects].

![Figure 6: Simple model with different semantics in each dialect](image)

The question is: what happens when the model is in state A and event e occurs? The answer is: it depends on the dialect. In Stateflow, outer transitions have priority over inner transitions, so the system moves to state D. In Rhapsody and UML Statecharts, inner transitions have priority, so the transition from state A to the composite state on the right is fired. In Rhapsody, evaluation is in terms of microsteps, where each complete transition becomes a microstep. In this case, x is assigned 1 prior to the evaluation of the state machine on the right, so the system transitions to state C. In UML Statecharts, the system evaluates to state B, as all transition guards are evaluated before a transition actually fires. There are many additional simple models that can be constructed to show several semantic variations between notations.

In the following subsections, we describe the various differences between the notations in terms of their syntactic elements.

#### 2.8.1 States

States are largely the same between dialects. Each dialect allows parallel (AND) and hierarchical (OR) states. The differences that exist have to do with the notion of actions that occur when the state is entered, exited, or occupied and neither exited nor entered (during actions). All notations support entry and exit actions, which are performed when the state is entered / exited, respectively. For example, the model snippet below sets the variable ‘enabled’ to true when state S3 is entered and to ‘false’ when it is exited.
Stateflow also allows *during* actions, which are performed on any step in which a state is occupied and neither entered or exited. UML Statecharts also allow *activities*, which are tasks in the host language that are started at the time the state is entered and aborted when the state is exited.

Finally, Rhapsody allows *static reactions* which are conditionally-activated actions that occur when in a particular state. These reactions use the same notation as transition labels but are bound to a state rather than a transition; they do not cause a state change.

### 2.8.1.1 Evaluation Order for Parallel States

Statecharts allow parallel state machines via the use of parallel (AND) states. For example, in Figure 1, there are three parallel state machines: FAN1, FAN2, and SpeedValue that are children of the FanOn state. In UML Statecharts and Rhapsody, the order of evaluation of the parallel machines is unspecified. In Stateflow, an explicit ordering is provided by an explicit priority ordering (the grey numbers in the upper right corners of the parallel state machines in Figure 1).

### 2.8.2 Pseudostates and Transitions

We describe discrepancies between pseudostates and transition segments simultaneously, as they together describe compound transitions in each of the notations. There are significant differences between the notations here and these, along with internal event semantics, form the significant variation points in the semantics of the dialects.

The first point of variation, demonstrated by the example in Figure 6, is the precedence ordering of transitions. In UML Statecharts and Rhapsody, transitions from the innermost active state have priority over transitions from enclosing states; in Stateflow, the opposite is true.

### 2.8.2.1 Stateflow: Junction Loops and Condition Actions

Stateflow allows for more complex multi-segment transitions than either UML Statecharts or Rhapsody. In the other notations, static choice junctions can be thought of as purely syntactic sugar; it is always possible to rewrite transitions that use static choice junctions into "simple" transitions. Not so in Stateflow; in fact, it is possible to use junctions to write looping structures, as is demonstrated by Figure 7.
This example also uses condition actions, shown in curly braces {}, which are unique to Stateflow. All dialects allow transition actions, which describe the actions to be performed at the instant when a compound transition fires: that is, after the source state has been exited and prior to the destination state being entered. Stateflow also allows condition actions, which occur immediately after a transition segment fires. In the example above, the output variable is assigned each time the condition \([i < 10]\) in the segment labeled ‘1’ is satisfied. These actions are performed even if the compound transition fails; that is, if no path leads to a destination state. This behavior is also demonstrated by the example in Figure 7. While condition actions can be useful for ‘flowchart’ style specifications that allow modeling of imperative flow graphs in Stateflow, they are also a major source of unintended side effects in Stateflow charts.

### 2.8.2.2 UML Statecharts and Rhapsody: Fork and Join Pseudostates

In UML and Rhapsody, compound transitions are defined in terms of sets-of-states instead of states. The reason for this involves fork and join pseudostates, which allow explicit entry and exit of parallel states. Both of these pseudostates always involve boundary crossing transitions. For fork pseudostates, the multiple outgoing transitions from the fork must each end in states that are in orthogonal regions (that is, they are state in parallel state machines). For join pseudostates, the multiple incoming transitions must all be from states in orthogonal regions.

[Include snippet containing fork/join HERE]

### 2.8.2.3 Semantics of State Updates

A major discrepancy between the dialects involves when actions are performed during an evaluation step. Condition actions, which occur only in Stateflow, are performed immediately after a transition segment’s firing condition is satisfied. They are therefore interleaved with the conditional evaluation of guards along a single compound transition. In the example snippet below, the condition action assignment \(\{\text{data} = 1\}\) causes the transition segment to state \(S2\) to be satisfied.
Transition actions occur in all dialects and are performed after a compound transition is completed. However, there are still discrepancies in state updates related to transition actions. UML Statecharts considers a step to be first an evaluation of the set of simultaneously enabled transitions followed by the execution of the all transition actions associated with the set of transitions, whereas Rhapsody and Stateflow have a microstep semantics, in which each compound transition is evaluated separately.

### 2.8.2.4 Dynamic Choice Junctions

To make things more confusing, UML Statecharts allows a dynamic choice junction which breaks the monolithic evaluation of transitions described in the previous subsection. When a dynamic choice junction is encountered, the system state is updated and the remainder of the conditions along the compound transition are evaluated using the state updated by any actions prior to the dynamic choice junction. The problem, however, is that the UML specification is silent as to whether these changes are visible to the evaluation of other parallel state machines. This leads to a wide number of possible interpretations of the semantics of the dynamic choice junction.

### 2.8.2.5 Determinism of Simultaneously Enabled Transitions

Determinism for chart evaluation is enforced in Stateflow by the addition of explicit priorities to each transition and parallel state machine. The order of evaluation of transitions is explicitly “outside in” with transitions exiting an enclosing state having priority over transitions exiting one of the children of the enclosing state. For states containing multiple exiting transitions, such as Figure 3, each transition is numbered on the diagram to denote its priority. Transition priorities are set by the user. Similarly, the order of evaluation of parallel state machines is denoted by the number in each parallel state. In the chart in Figure 1, the Fan1 parallel state is evaluated first, followed by the Fan2, and SpeedValue, respectively. Because of the explicit orderings provided to transitions and parallel state machines, the evaluation of a Stateflow chart is both visible and deterministic.

In both of these cases, UML Statecharts and Rhapsody do not define an explicit ordering on transitions.

### 2.8.3 Event Propagation

Perhaps the largest discrepancy between Stateflow vs. UML Statecharts and Rhapsody is the internal event propagation behavior. In UML Statecharts and Rhapsody, internally-generated and externally-generated events are not distinguished; they are queued in an unspecified order. It is therefore possible that internal events due to one external event are interleaved with the execution other external events. We say that internal events in these dialects have a queueing semantics.

Stateflow, on the other hand, has a suspension semantics for internal events. This semantics states that if an event is broadcast by an action, processing of the current event is immediately suspended and the new event is evaluated by the chart. This semantics can easily lead to unintended behaviors, because the original event resumes in what may be a significantly different configuration than the one that led to the generation of the event; in fact, it is possible that the source state of the transition is no longer occupied! In this case, the processing of the current transition aborts in what is called early return logic. An example of this problem is shown in Figure 8. In this chart, event F is generated as a condition action (before State A is exited). The processing of the transition from A to B is suspended and the chart is re-evaluated with event F. This event causes A to transition to C. When control returns to the processing of event E, the complete transition from A to B is no longer possible, so it aborts. In deference to Star Trek, we sometimes call this the wormhole semantics, as it appears to exhibit temporal anomalies.
Stateflow also allows events to be broadcast to a specific state using the `SEND` action. Sent events still exhibit the function call semantics, but they do not cause re-evaluation of the entire chart and are therefore less likely to cause the wormhole behavior.

3. **Semantics Preliminaries**

We formalize the Statecharts dialects using *Structural Operational Semantics*. An operational semantics is a set of rules specifying how the state of an actual or hypothetical computer changes while executing a program [foldoc:operational-semantics]. The overall state is typically divided into a number of components, e.g., stack, heap, registers, etc. Each rule specifies certain preconditions on the contents of some components and their new contents after the application of the rule. The range of possible operational semantics for a language is very broad; at a high level of abstraction, the rules may specify abstract operations on a mathematical store, in the same sense as denotational semantics [Allison86:semantics, Winskel93:semantics]. At a low level of abstraction, every compiler provides an operational semantics for a given language in terms of state changes (machine instructions) on a target machine. For this presentation, we will focus on abstract operational semantics that are suitable for understanding the semantics of languages.

Structural operational semantics were developed by Plotkin [Plotkin81:sos, Turi97:semantics] to provide a straightforward, syntax-directed presentation of language semantics. Structural operational semantics operate over the abstract syntax of a language. The *concrete syntax* of the language describes the strings of characters, or in the case of Statecharts, the graphical position of objects, that developers use to write models in a particular language. The abstract syntax, on the other hand, is an alternate representation of programs using labeled trees. These trees provide an unambiguous and easy to manipulate representation of programs. It is possible to translate between the abstract syntax and the concrete syntax of a language using a variety of parsing and unparsing tools.

Structural operational semantics describes a language using a set of axioms and inference rules that characterize the various semantic predicates of interest for the language [Kahn87:natural-semantics]. These semantic predicates can describe the well-formedness of the abstract syntax, which programs are correctly typed, the dynamic semantics of a program, and even translations between languages. The rules, similar to rules in natural deduction, describe the operational state transformations. The premises of the rule describe under what circumstances the transformation is applicable, and the conclusion of the rule describes the result of the transformation.

This document represents SOS rules in a “Prolog style” rather than an “inference rule style”. Usually, SOS rules are presented in an inference rule style, which looks like this:
Evaluation of Sums:
\[
\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1 \\
\langle a_0 + a_1, \sigma \rangle \rightarrow n
\]
where \( n \) is the sum of \( n_0 \) and \( n_1 \).

Evaluation of Subtractions:
\[
\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1 \\
\langle a_0 - a_1, \sigma \rangle \rightarrow n
\]
where \( n \) equals \( n_0 \) minus \( n_1 \).

Evaluation of Products:
\[
\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1 \\
\langle a_0 \times a_1, \sigma \rangle \rightarrow n
\]
where \( n \) is the product of \( n_0 \) and \( n_1 \).

The examples above are for evaluating arithmetic in a simple expression language. In this form, the result below the line can be derived if it is possible to derive each of the premises above the line. The evaluation of the rules forms a proof tree, with the ultimate result at the bottom, e.g.,

\[
\langle 1, \sigma \rangle \rightarrow 1 \\
\langle 2, \sigma \rangle \rightarrow 2 \\
\langle 1 + 2, \sigma \rangle \rightarrow 3
\]

Instead, we write the rules in a “Prolog style” where the result “below the line” is written first, and the premises are written following the symbol \(-\) and comma serves as conjunction. We also distinguish language syntax from other arguments to the rule using double brackets \([[]]\). So, the evaluation of sums rule would be written:

\[
\text{Eval}([[[a_0 + a_1]], \sigma]) \rightarrow n :-
\text{Eval}([[a_0]], \sigma) \rightarrow n_0,
\text{Eval}([[a_1]], \sigma) \rightarrow n_1,
\]

\( n \) is \( n_0 + n_1 \).

Note: the use of arrows (\( \rightarrow \)) in both syntaxes of rules is notional, to guide the reader in following the rule. It does not have any semantic meaning; the semantics establish predicates (n-tuples) through the use of inference rules. However, the intuition is that the object(s) to the right of the arrow are computed by the rule.

3.1 Use of Disjunctions
We use disjunctions (\( \lor \)) within the rules to simplify their presentation. For example:

\[
C, [[[\text{Or}(T, \text{child}_d::\text{child}_d)]]] \theta \rho \rightarrow \rho' :-
\text{is\_in\_state}(\rho, \text{child}_d) \rightarrow \text{ins},
((\text{ins} = \text{TRUE}, S, [[[\theta, \text{child}_d]]] \theta \rho \rightarrow \rho') \lor
(\text{ins} = \text{FALSE}, C, [[[\text{Or}(T, \text{child}_d)]]] \theta \rho \rightarrow \rho')).
\]

This use is equivalent to defining multiple rules, one for each branch of the disjunction. In the case of the rule above, it is equivalent to the following two rules:

\[
\text{is\_in\_state}(\rho, \text{child}_d) \rightarrow \text{ins},
(\text{ins} = \text{TRUE}, S, [[[\theta, \text{child}_d]]] \theta \rho \rightarrow \rho') \lor
(\text{ins} = \text{FALSE}, C, [[[\text{Or}(T, \text{child}_d)]]] \theta \rho \rightarrow \rho')).
\]
\[C_\land[[\text{Or} (T, \text{child}_{\text{hd}}:\text{child}_{\text{tl}})] \theta \rho \rightarrow \rho']:\]
\[\text{is\_in\_state}(\rho, \text{child}_{\text{hd}}) \rightarrow \text{ins},\]
\[\text{ins} = \text{TRUE},\]
\[S_\land[[\theta_{\text{d}}(\text{child}_{\text{tl}})]] \theta \rho \rightarrow \rho'.\]

\[C_\land[[\text{Or} (T, \text{child}_{\text{hd}}:\text{child}_{\text{tl}})] \theta \rho \rightarrow \rho']:\]
\[\text{is\_in\_state}(\rho, \text{child}_{\text{hd}}) \rightarrow \text{ins},\]
\[\text{ins} = \text{FALSE},\]
\[C_\land[[\text{Or} (T, \text{child}_{\text{hd}})] \theta \rho \rightarrow \rho'.\]

### 3.2 List Operations

Lists are used extensively within the semantics. We assume the following operators for lists:

- \(\text{cons}(\text{Hd}, \text{Tl})\) (written inline as \(\text{Hd}∷\text{Tl}\)) which adds element \(\text{Hd}\) to the head of list \(\text{Tl}\).
- \(\text{tlcons}(\text{L}, \text{Last})\) (written inline as \(\text{L}.\text{Last}\)) which adds element \(\text{Last}\) to the end of list \(\text{L}\).
- \(\text{concat}(\text{L}_1, \text{L}_2)\) (written inline as \(\text{L}_1 @ \text{L}_2\)) which concatenates \(\text{L}_1\) to \(\text{L}_2\).

The empty list is written \(\phi\). List comprehensions of the form \([a_1, a_2, \ldots, a_k]\) are shorthands for multiple conses. For example, the list comprehension \([a_1, a_2, a_3]\) is equivalent to \((a_1∷a_2∷a_3∷\phi)\).

We use a \(\text{common\_prefix}(\text{L})\) operation that takes a list of lists as an argument and returns the list that is the common prefix of each element of \(\text{L}\) (this result may be the nil list). This can be defined using a helper rule \(\text{hds\_match}\) as follows:

\[\text{hds\_match} \text{elem} \phi \rightarrow (\text{TRUE}, \phi).\]
\[\text{hds\_match} \text{elem} ((\text{elem}∷\text{tl})∷\text{rest}) \rightarrow (v, \text{tl}∷\text{rest\_tls}) ::\]
\[\text{hds\_match} \text{elem} \text{tl} \rightarrow (v, \text{rest\_tls}).\]
\[\text{hds\_match} \text{elem} ((\text{hd}∷\text{tl})∷\text{rest}) \rightarrow (\text{FALSE}, \phi) :: \text{hd} \neq \text{elem}.\]

\[\text{common\_prefix} \phi \rightarrow \phi.\]
\[\text{common\_prefix} ((\text{hd}∷\text{tl})∷\text{otherPaths}) \rightarrow (\text{hd}∷\text{rest}) ::\]
\[\text{hds\_match} \text{hd} \text{otherPaths} \rightarrow (\text{TRUE}, \text{other\_tls}),\]
\[\text{common\_prefix} (\text{tl}∷\text{other\_tls}) \rightarrow \text{rest}.\]

We also use a rule \(\text{partition}\) that takes two lists and splits the first list into the portion that is common to the second list and the portion that is different.

\[\text{partition}(\text{hd}_1∷\text{tl}_1, \text{hd}_1∷\text{tl}_2) \rightarrow (\text{hd}_1∷\text{common}, \text{rest}) :: \text{partition}(\text{tl}_1, \text{tl}_2) \rightarrow (\text{common}, \text{rest}).\]
\[\text{partition}(\text{hd}_1∷\text{tl}_1, \text{hd}_2∷\text{tl}_2) \rightarrow (\phi, \text{hd}_1∷\text{tl}_1) :: \text{hd}_1 \neq \text{hd}_2.\]
\[\text{partition}(\phi, \text{L}) \rightarrow (\phi, \phi).\]
\[\text{partition}(\text{L}, \phi) \rightarrow (\phi, \text{L}).\]

### 3.3 Mechanism of Parameterization

The parameterization of the semantics is accomplished through \textit{extension} of the rules. Each rule has a signature that describes its arguments. The core language provides the rules that are common to all three semantic dialects. The dialects provide additional rules that are applicable only to that dialect. For most of the extension points, the core language provides only the \textit{signature} of the rule and no instances. To make clear the \textit{interface rules} that only are given as signatures in the common core and are extended by the dialects, the interface rules are written in green.
4. Syntax and Common Core Semantics

In this section we define an abstract syntax for Statecharts that is essentially the union of the set of constructs used by the different dialects. The rules that encapsulate the distinctions between the notations ignore those constructs that are not relevant for that dialect. It is worth noting, however, that it would be straightforward to describe hybrid notations that mix behaviors from each of the dialects.

The syntactic structure of a Statechart is provided by the following definitions. A state machine $\theta$ is defined as a root component $\text{root}$, input declarations $\text{I}$, output declarations $\text{O}$, and local declarations $\text{L}$. Input and output declarations are be either variables or events. Local declarations can be variables, events, or constants.

$\text{Chart } \theta ::= (\text{root}, [\text{src}_0...\text{src}_n], \text{I}, \text{O}, \text{L})$

$\text{SrcComp src ::= } p : v$

$\text{Vertex v ::= State(sd) | Pseudostate(psd) | GFunction(gfd)}$

A vertex ($v$) is either
   - a state definition $sd$
   - a pseudostate definition $psd$
   - a graphical function $gfd$

A GraphicalFunction (used in Stateflow) consists of input variables $\text{I}$, output variables $\text{O}$, local variables and constants $\text{L}$, and a transition list $\text{T}$ ($\text{T}$ describes the execution of the function). A StateDef consists of the actions associated with entering, staying within (during), and exiting the state ($a_e, a_d, a_t$), local variables and constants $\text{L}$ associated with the state, a transition list $\text{T}$ of transitions whose source is the state, and a composition $\text{C}$ that describes the children of the state. A composition $\text{Comp}$ is either an Or-composition, in which case it has a default transition list and a list of child state ids, or an And-composition containing just child state ids.

$\text{GraphicalFunction gfd ::= ((I, O, L), T)}$

$\text{State sd ::= ((a_e, a_d, a_t), L, T, C)}$

$\text{Pseudostate psd ::= (pty, T)}$

$\text{PseudostateType pty ::= SF_JUNCTION | STATIC_CHOICE | DYNAMIC_CHOICE}$

$\text{Comp C ::= Or (T, [s_{0h}...s_{an}]) | And ([s_{0h}...s_{an}])}$

Note that we do not include a complete set of UML/Rhapsody pseudostate types in our definition. The types that are missing are: initial, join, fork, entryPoint, exitPoint, and terminate. We handle join and fork via a source-to-source translation, as described in Sections 4.4 and 4.5 for Rhapsody and UML Statecharts. Initial pseudostates are encoded directly via the set of initial transitions for compositions. History, EntryPoint, ExitPoint, and terminate pseudostates are currently not supported in our semantics.

A transition $\text{Trans}$ contains a triggering event $e$ (which can be NULL in the case of an eventless transition in Stateflow or a completion transition in UML Statecharts or Rhapsody), guarding condition $c$, condition and transition actions ($a_e, a_t$) and destination $d$. UML and Rhapsody do not support condition actions, so this field will always be nil for these notations.

A destination is a path to a state or junction or the special destination ExternalLoop that is used for external loop transitions. The three list types TransLst, path, and ActionLst describe lists of transition segments, identifiers, and actions, respectively. $\phi$ is the symbol for the empty list (nil).

$\text{Trans t ::= (e, c, (a_e, a_t), d)}$

$\text{Dest d ::= p | EXTERNAL_LOOP}$
4.1 Evaluation Environment

For the moment, we leave the environment \((Env, \text{usually denoted by } \rho)\) that describes the dynamic state of the system abstract. There are structural differences in the environments between the different notations. These are abstracted by a handful of accessor functions that query the environment for details about the current state of the system. The \text{is_trigger_evt} and \text{set_trigger_evt} rules check and set the trigger event, respectively. The \text{is_in_state} and \text{set_in_state} rules check and set whether or not the current state is occupied. The \text{is_machined_initialized} and \text{set_machined_initialized} rules check and set whether the state machine was initialized. Finally, the \text{set_transition_occurred} rule sets a flag in the environment that a transition has occurred in this step.

\[\text{is_trigger_evt}: \text{Chart} \rightarrow \text{Env} \rightarrow \text{Path} \rightarrow \text{Event} \rightarrow \text{Bool}\]
\[\text{set_trigger_evt}: \text{Env} \rightarrow \text{Path} \rightarrow \text{Env}.\]
\[\text{is_in_state}: \text{Env} \rightarrow \text{Path} \rightarrow \text{Bool}.\]
\[\text{set_in_state}: \text{Env} \rightarrow \text{Path} \rightarrow \text{Bool} \rightarrow \text{Env}.\]
\[\text{is_machine_initialized}: \text{Env} \rightarrow \text{Env}.\]
\[\text{set_machine_initialized}: \text{Env} \rightarrow \text{Bool} \rightarrow \text{Env}.\]
\[\text{set_transition_occurred}: \text{Env} \rightarrow \text{Env}\]

4.2 Common Semantic Rules

For evaluation of transitions, we introduce one more syntactic category, called \text{Status}, which defines the evaluation result of transitions. It is defined as follows:

\[\text{DestType } dt ::= \text{STATE} | \text{TERMINAL}\]
\[\text{Status } st ::= \text{Succeed}(dt) \mid \text{Fail}\]

The meaning of this type will be explained in the subsection on transition segments.

Actions and Conditions

\text{Conditions} describe the Boolean expressions that are used for transition guards. \text{Actions} describe updates to the environment that can be specified when a transition fires or a state is entered / active / exited.

\[B: \text{condition} \rightarrow \text{Env} \rightarrow \text{Bool}\]
\[A: \text{action list} \rightarrow \text{Chart} \rightarrow \text{Env} \rightarrow \text{Env}\]

For the moment, these are left abstract. Important cases will be defined in the dialect-specific section, but for the most part all of the expressions are common between dialects. We assume that all references to variables or constants are fully qualified: if the variable is a local variable of a particular state, then the reference contains the complete path to that variable. This qualification of references can be straightforwardly performed as a preprocessing pass.

Transition Segments

\[\tau: \text{trans} \rightarrow \text{Chart} \rightarrow \text{Env} \rightarrow \text{Path list} \rightarrow A \rightarrow (\text{Env, Status})\]

\(\tau\) describes the behavior of a transition segment. In Statecharts, \text{compound transitions} can be composed of several \text{segments} that start and end with a state and visit intermediary pseudostates. In our formalization, we evaluate the transition segments one at a time and carry along the list of transition actions (\text{transact}) that from previous segments of the compound transition under evaluation. The transition segment returns the
result \((res)\) of evaluating its destination \((d)\). This contains the updated environment and also a Status flag: ‘Succeed’ if the segment yielded a complete transition to a destination state, ‘Terminal’ if the transition led to a terminal junction (this is relevant only in Stateflow), and ‘Fail’ if the segment did not lead to a destination state or terminal junction. A transition segment *fires* if the trigger event \((et)\) is the current event and the guarding condition \(c\) evaluates to TRUE.

\[
\tau \left[\left((et, c, (a_i, a), d)\right) \circ \rho \ P \ transact \rightarrow res \ : \\
\text{is\_trigger\_evt} \ \theta \ P \ et \rightarrow \text{TRUE}, \\
B \ [[c]] \ \rho \rightarrow \text{TRUE}, \\
A \ [[a_i]] \ \theta \ \rho \rightarrow \rho ', \\
D \ [[d]] \ \theta \ \rho ' \ P \ (\text{transact} \ @ \ a) \rightarrow res
\right]
\]

\[
\tau \left[\left((et, c, (a_i, a), d)\right) \circ \rho \ P \ transact \rightarrow (\rho, \text{Failed}) \ : \\
((\text{is\_trigger\_evt} \ \theta \ P \ et \rightarrow \text{FALSE}) \lor (B[[c]] \ \rho \rightarrow \text{FALSE})
\right]
\]

The arguments to \(\tau\) are the transition syntax (in the tuple below, the tuple \((et, c, (a_i, a), d)\)), \(\theta\): the structure of the chart, \(\rho\): the current dynamic environment, \(P\): the source state and list of junctions leading to this transition segment, and \(\text{transact}\): the list of transition actions that have been stored thus far. The result of \(\tau\) is a pair that describes the current environment and whether or not the transition reached a terminal. The \(\text{is\_trigger\_evt}\) rule determines whether or not the trigger event was satisfied. This rule is a variation point due to the way different notations handle completion events.

### Transition Lists

\(T\) defines the behavior of transition segment lists. The first argument is the syntax of the transition segment list. The other arguments of the rule are \(\theta\): the structure of the chart, \(\rho\): the current dynamic environment, \(P\): the source state and list of junctions that have led to this transition segment list, and \(\text{transact}\): the list of transition actions that have been stored thus far. The result of \(\tau\) is a pair that describes the current environment and whether or not the transition reached a terminal.

\[
T \ [[\psi]] \ \theta \ \rho \ P \ transact \rightarrow (\rho, \text{Failed})
\]

\[
T \ [[t : L]] \ \theta \ \rho \ P \ transact \rightarrow res \ :- \\
\text{choose\_elem} \ [[t : L]] \rightarrow (elem, rest), \\
\tau \ [[\text{elem}]] \ \theta \ P \ transact \rightarrow res', \\
(((res' = (\rho', \text{Succeeded}(dt)), res = res') \lor \\
(res' = (\rho', \text{Failed}), T [[L]] \ \theta \ \rho ' \ P \ transact \rightarrow res)
\]

The \text{choose\_elem} rule is variability between the semantics. In Stateflow, the order of evaluation of transitions from a given vertex is fixed, and therefore \text{choose\_elem} rule simply returns the head of the list. In Rhapsody and UML Statecharts, this rule is non-deterministic and can return any element of the list.

### Destinations

\(D\) defines the behavior of destinations, which are the targets of transition segments. The first argument is the syntax of the destination: either a state or a pseudostate. The other arguments of the rule are \(\theta\): the structure of the chart, \(\rho\): the current dynamic environment, \(P\): the source state and list of junctions that have led to this transition segment list, and \(\text{transact}\): the list of transition actions that have been stored thus far. The result of \(\tau\) is a pair that describes the current environment and whether or not the transition reached a terminal.

\[
D \ [\theta \ P \ dest, c, a \ P \ \text{transact} \rightarrow \text{res} : \\
\text{choose\_elem} \ [\theta \ P \ dest, c, a \ P \ \text{transact} \rightarrow \text{res}, \\
\tau \ [\theta \ P \ dest, c, a \ P \ \text{transact} \rightarrow \text{res}']
\]

The \text{choose\_elem} rule is variability between the semantics. In Stateflow, the order of evaluation of transitions from a given vertex is fixed, and therefore \text{choose\_elem} rule simply returns the head of the list. In Rhapsody and UML Statecharts, this rule is non-deterministic and can return any element of the list.
The external loop destination handles the special case in our semantics when a transition loops back to the same state from which it started. In Stateflow and UML Statecharts, a transition from a state to itself may be either external or internal. If the transition is external, then it causes the state to be exited and re-entered, causing the entry and exit actions to be evaluated. If it is internal, then the state is not exited and re-entered, and therefore the entry and exit actions do not occur. In our non-graphical semantics, it is not possible to distinguish these two cases without creating a special case. This handled by the external loop destination. It works by (artificially) adding the parent state into the ‘scope’ of the transition that is defined by the path list argument.

We present Stateflow junctions and UML/Rhapsody StaticChoice elements in this section because they are quite similar to one another; other elements that exist only in one or two of the dialects, or behave differently between the semantics are presented later in the dialect-specific sections.

Stateflow junctions have two differences from junctions (called StaticChoice elements in our abstract syntax) in UML Statecharts and Rhapsody. First, Stateflow junctions affect the scope of the transition; writing a transition segment to a junction outside the parent state of the source and destination state will cause the parent state(s) to be exited; in UML and Rhapsody, static and dynamic choices do not affect the scope of the transition. Second, Stateflow allows terminal junctions, which are junctions with no outgoing transitions, to act as terminators for complex transitions. These prevent backtracking to complete a transition, but also allow child actions to be performed.

Note that the combination of τ rules, Transition list rules, and destination rules allow backtracking between the junctions along the path of complex transitions. This is explicitly the correct behavior for Stateflow, which supports condition actions that cause immediate state changes even when the associated transition segment does not lead to a complete transition. UML and Rhapsody do not have a notion of condition actions, so the backtracking does not affect the evaluation state.

We represent Rhapsody Junctions and Choice vertices as StaticChoice junctions. From the perspective of evaluation semantics, the differences between the two pseudostate types is irrelevant. UML Junctions are also considered StaticChoice pseudostates, to distinguish them from DynamicChoice elements.
The release rule exits a subtree. The arguments of the rule are \( \theta \): the structure of the chart, \( \rho \): the current dynamic environment, \( src \) the source state and \( cp \) the common prefix of all transition segments. The result of release is the environment after exiting the source states, or if the transition does not exit the source state, the result of executing the during operations for the state. The \( \text{partition}(a, b) \) operation splits the list \( a \) into two lists: the first list matches a prefix of \( b \), and the second is the remainder of \( a \).

release: Chart \( \rightarrow \) Env \( \rightarrow \) Path \( \rightarrow \) Path \( \rightarrow \) Env
release \( \theta \rho src cp \rightarrow \rho' \)
- \( src = cp \)
- \( \theta src = [(p : ((a, a_d, a), T, C))] \)
- \( A [(a_0)] \theta \rho src \rightarrow \rho' \)

release \( \theta \rho src cp \rightarrow \rho' \)
- \( src \neq cp \)
- \( \text{partition}(src, cp) \rightarrow (\text{same}, \text{diff}::\text{rest}) \)
- \( S_x [(\theta (\text{same}.\text{diff})]] \theta \rho \rightarrow \rho' \)

The enter rule enters a subtree. The arguments of the rule are \( \theta \): the structure of the chart, \( \rho \): the current dynamic environment, \( src \) the source state and \( cp \) the common prefix of all transition segments. The result of enter is the environment after entering the source states, or if the transition does not enter the destination state (say, it is a transition from a substate to its parent), the result of executing the enter action is the identity.

enter: Chart \( \rightarrow \) Env \( \rightarrow \) Path \( \rightarrow \) Path \( \rightarrow \) env
enter \( \theta \rho dst cp \rightarrow \rho : dst = cp \)
enter \( \theta \rho dst cp \rightarrow \rho ' : \)
- \( dst \neq cp \)
- \( \text{partition}(src, cp) \rightarrow (\text{same}, \text{diff}::\text{rest}) \)
- \( S_x [(\theta (\text{same}.\text{diff})]] \theta \rho rest \rightarrow \rho' \)

The open_path rule performs the activities associated with exiting and entering states due to a complete transition segment path. The arguments of the rule are \( \theta \): the structure of the chart, \( \rho \): the current dynamic environment, \( src \), \( P \), \( dst \), the list of transition segments starting with source state \( src \), then a list of intermediate junctions \( P \) followed by the destination \( dst \). The \( \text{common_prefix}(L) \) rule takes a list of lists as an argument and returns the list that is the common prefix of each element of \( L \) (which may be empty). This finds the “parent” state of all junctions within the transition.

open_path: Chart \( \rightarrow \) env \( \rightarrow \) path list \( \rightarrow \) AL \( \rightarrow \) env
open_path \( \theta \rho (src::P.dst)\) transact \( \rightarrow \rho'' \)
- \( \text{common_prefix}(src::P.dst) \rightarrow \rho'' \)
- \( \text{release} \theta \rho cp \rightarrow \rho'' \)
- \( A [[\text{transact}]] \theta \rho'' src \rightarrow \rho'' \)
- \( \text{enter} \theta \rho'' dst cp \rightarrow \rho'' \)

State Evaluation Rules

The \( S_e \), \( S_d \), and \( S_x \) rules define the behavior of a state upon entry (\( S_e \)), “during” (enabled and neither entered or exited) (\( S_d \)), and exit (\( S_x \)). The arguments to the rule are the definition of the state, \( \theta \): the structure of the chart, \( \rho \): the current dynamic environment, and, for the entry and exit rules, \( ep \), the path of remaining states to be entered or exited. The \( \text{add_state} \) and \( \text{remove_state} \) predicates add and remove the state \( p \) to the set of active states in environment \( \rho \), respectively. \( \text{In_state} \) tests whether the current state is active.
Composition Evaluation Rules

The composition rules define how the states relate to their children (if any). There are two types of compositions, Or and And. Or compositions allow exactly one child to be active at a time, and are used for constructing hierarchical states. All children are activated in and compositions, and so these compositions describe parallel machines. For each type of composition, there are rules for entry, during (enabled and neither entered or exited), and exit. Entry rules have additional parameters that describe an entry path through the system; these parameters are used when we have transitions that cross a parent state boundary to reach a child state.

We present all the rules for or compositions first, and and compositions subsequently.

\[ S_c: \text{StateDef} \rightarrow \text{Chart} \rightarrow \text{env} \rightarrow P \rightarrow \text{env} \]
\[ S_d: \text{StateDef} \rightarrow \text{Chart} \rightarrow \text{env} \rightarrow (\text{env}, \text{bool}) \]
\[ S_e: \text{StateDef} \rightarrow \text{Chart} \rightarrow \text{env} \rightarrow P \rightarrow \text{env} \]

\[ S_c[[\text{path} : (\{a_c, a_d, a_e\}, T, C)]] \theta \rho \text{ ep} \rightarrow \rho'' \vdash \]
\[ A [[\{a_{\rho}\} \theta (\text{set in state}(\rho, \text{path}, \text{TRUE})) \text{ path} \rightarrow \rho'], \]
\[ C_s[[C]] \theta \rho' \text{ path ep} \rightarrow \rho''. \]

\[ S_d[[\text{path} : (\{a_c, a_d, a_e\}, T, C)]] \theta \rho \rightarrow \rho'' \vdash \]
\[ A [[\{a_{\rho}\} (\text{set in state}(\rho, \text{path}, \text{FALSE})) \text{ path} \rightarrow \rho'], \]
\[ C_s[[C]] \theta \rho' (\text{path}) \rightarrow \rho''. \]

\[ \text{Composition Evaluation Rules} \]

In Stateflow, it is possible that a state can have no substates, but may contain a junction network. Due to condition actions, these may cause changes to the environment. The first rule captures that behavior. The second rule captures the ‘normal’ evaluation, in which we evaluate the initial transition to find the active substate. We require that this rule succeeds; otherwise the machine would be left in an inconsistent configuration because the parent state would be active but no child states would be active. The third rule captures the case where we have a prescribed entry path, in which case we enter the prescribed child.

\[ C_c[[\text{Or}(\text{TL}, \phi)]] \theta \rho \text{ parent } \phi \rightarrow \rho'' \vdash T [[\text{TL}]] \theta \rho \text{ [parent] } \phi \rightarrow (\rho', \text{st}). \]

\[ C_c[[\text{Or}(\text{TL}, S)]] \theta \rho \text{ parent } \phi \rightarrow \rho'' \vdash T [[\text{TL}]] \theta \rho \text{ [parent] } \rightarrow (\rho', \text{Succeeded}(\text{STATE})). \]

\[ C_c[[\text{Or}(\text{TL}, S)]] \theta \rho \text{ parent } (\text{entryPath}_{a_c}:\text{entryPath}_{a_d}) \rightarrow \rho'' \vdash S_e [[\theta (\text{parent.entryPath}_{a_d})]] \rho \text{ entryPath}_{a_d} \rightarrow \rho'. \]

For the ‘during’ actions, we evaluate the active child, if the state contains children.

\[ C_d[[\text{Or}(\text{TL}, \phi)]] \theta \rho \rightarrow \rho. \]
\[ C_d[[\text{Or}(\text{TL}, \text{child}_{a_d}:\text{child}_{a_d})]] \theta \rho \rightarrow \rho'' \vdash \text{in state}(\rho, \text{child}_{a_d}) \rightarrow \text{ins}, \]

\[ \text{composition rule} \]

\[ \text{composition rule} \]

\[ \text{composition rule} \]
For exit actions, we exit all children of the composition.

\[
\begin{align*}
(\text{ins} = \text{TRUE, } S_d[[\theta_d(\text{child}_d)]] & \land \rho \rightarrow \rho') \lor \\
(\text{ins} = \text{FALSE, } C_d[[\text{Or} (T, \text{child}_d)]] & \land \rho \rightarrow \rho').
\end{align*}
\]

For the 'exit' actions, we exit the active child.

\[
\begin{align*}
C_d[[\text{Or} (\text{TL, child}_d::\text{child}_d)]] & \land \rho \rightarrow \rho' : - \\
\text{is_in_state}(\rho, \text{child}_d) & \rightarrow \text{ins},
\end{align*}
\]

\[
(\text{ins} = \text{TRUE, } S_d[[\theta_d(\text{child}_d)]] & \land \rho \rightarrow \rho') \lor \\
(\text{ins} = \text{FALSE, } C_d[[\text{Or} (T, \text{child}_d)]] & \land \rho \rightarrow \rho').
\]

For and compositions, we have similar rules. For entry, we enter all child states. When we have a
prescribed path, we enter using the 'default' behavior for each child except the one in which we have the
prescribed path. For this child, we pass down the remainder of the prescribed path.

\[
C_d[[\text{And} (\text{phi})]] \land \rho \land \text{parent} \phi \rightarrow \rho.
\]

\[
C_d[[\text{And} (\text{child}_d::\text{child}_d)]] \land \rho \land \text{parent} \phi \rightarrow \rho' : - \\
\text{choose_elem} [[\text{child}_d::\text{child}_d]] \rightarrow (\text{elem, rest}),
\]

\[
S_e[[\theta(\text{elem})]] \land \rho \phi \rightarrow \rho';
\]

\[
C_d[[\text{And} (\text{rest})]] \land \rho \phi \rightarrow \rho'.
\]

\[
C_d[[\text{And} (\text{child}_d::\text{child}_d)]] \land \rho \land \text{parent} (\text{path}_d::\text{path}_d) \rightarrow \rho' : - \\
\text{choose_elem} [[\text{child}_d::\text{child}_d]] \rightarrow (\text{elem, rest}),
\]

\[
(\text{parent.path}_d = \text{elem},
S_e[[\theta(\text{elem})]] \land \rho \phi \rightarrow \rho';
C_d[[\text{And} (\text{rest})]] \land \rho \phi \rightarrow \rho'.
\]

\[
C_d[[\text{And} (\text{child}_d::\text{child}_d)]] \land \rho \land \text{parent} (\text{path}_d::\text{path}_d) \rightarrow \rho' : - \\
\text{choose_elem} [[\text{child}_d::\text{child}_d]] \rightarrow (\text{elem, rest}),
\]

\[
(\text{parent.path}_d \neq \text{elem},
S_e[[\theta(\text{elem})]] \land \rho \phi \rightarrow \rho';
C_d[[\text{And} (\text{rest})]] \land \rho \phi \rightarrow \rho'.
\]

For entry, we pass down the remainder of the prescribed path.

\[
C_d[[\text{And} (\text{phi})]] \land \rho \rightarrow (\rho, \text{FALSE}).
\]

\[
C_d[[\text{And} (\text{child}_d::\text{child}_d)]] \land \rho \rightarrow \rho' : - \\
S_d[[\theta(\text{child}_d)]] \land \rho \rightarrow \rho';
C_d[[\text{And} (\text{child}_d)]] \land \rho \rightarrow \rho'.
\]

For exit actions, we exit all children of the composition.

\[
C_d[[\text{And} (\text{phi})]] \land \rho \rightarrow \rho.
\]

\[
C_d[[\text{And} (\text{child}_d::\text{child}_d)]] \land \rho \rightarrow \rho' : - \\
\text{choose_elem} [[\text{child}_d::\text{child}_d]] \rightarrow (\text{elem, rest}),
\]

\[
S_e[[\theta(\text{elem})]] \land \rho \phi \rightarrow \rho';
C_d[[\text{And} (\text{rest})]] \land \rho \phi \rightarrow \rho'.
\]

**Top-Level Event Processing**
To process an event at the top level, for any of the notations, we first check whether the state machine has been initialized. If not, we initialize it by entering the root-level composition and set the initialized value. If so, we execute the ‘during’ action of the root-level composition. Note that it is possible to perform the initialization prior to evaluation of events – this may be the default in UML Statecharts and Rhapsody – but in Stateflow by default the initialization occurs concurrent with the first event received. Even in Stateflow, this behavior can be modified using the InitializeAtStartup option.

\[
\text{Evt: Chart} \rightarrow \text{Env} \rightarrow \text{E} \rightarrow (\text{Env}, \text{Status})
\]

\[
\text{Evt } \theta \rho e \rightarrow \rho' \text{:-}

\text{is_statemachineinitialized}(\rho) \rightarrow \text{FALSE},
\]

\[
C_\theta[[\text{root}(\theta)]] \theta (\text{set_evt}(\rho, e)) \phi \phi \rightarrow \rho';
\]

\[
\text{set_statemachineinitialized}(\rho', \text{TRUE}) \rightarrow \rho''.
\]

\[
\text{Evt } \theta \rho e \rightarrow \text{res} \text{:-}

\text{is_statemachineinitialized}(\rho) \rightarrow \text{TRUE},
\]

\[
C_\theta[[\text{root}(\theta)]] \theta (\text{set_trigger_evt}(\rho, e)) \rightarrow \text{res}.
\]

### 4.2 Common Parametric Rules and Definitions

Some of the instantiations of the parameterized rules are common to multiple dialects. We present the rules in this section, then use them in the instantiations below.

#### List Choice Rules

First, we have functions for choosing elements nondeterministically from lists. The following rule will return one of the elements from the list, assuming that the list has at least one element.

\[
\text{choose_elem_nondet }[[hd::tl]] \rightarrow (hd, tl).
\]

\[
\text{choose_elem_nondet }[[hd::tl]] \rightarrow (elem, hd::rest) \text{ :-}

\text{choose_elem_nondet }[[tl]] \rightarrow (elem, rest).
\]

#### ‘During’ State Rules

Second, we have rules for evaluating the ‘during’ actions within states. These rules handle the variability between Stateflow, which gives priority to outermost transitions (that is, transitions of parent states), vs. UML Statecharts and Rhapsody, which give priority to innermost transitions (that is, transitions of child states).

The ‘out-in’ rules, which give priority to outermost transitions, are defined below.

\[
S_d\text{out}_{\text{in}} [[\text{path} : ((a_e, a_d, a_c), T, C)]]_d \theta \rho \rightarrow (\rho', \text{Succeeded(State)})) \text{ :-}
\]

\[
T [[T]] \theta \rho [\text{path}] \phi \rightarrow (\rho', \text{Succeeded(State)}).
\]

\[
S_d\text{out}_{\text{in}} [[\text{path} : ((a_e, a_d, a_c), T, C)]]_d \theta \rho \rightarrow \text{res} \text{ :-}
\]

\[
T [[T]] \theta \rho [\text{path}] \text{nil} \rightarrow (\rho', st),
\]

\[
(st = \text{Succeeded(Terminal)} \lor st = \text{Failed}),
\]

\[
A [[a_d]] \theta \rho' \text{path} \rightarrow \rho'';
\]

\[
C_d [[C]] \theta \rho'' \rightarrow \text{res}.
\]
The ‘in-out’ rules are defined as follows. Note that UML and Rhapsody statecharts do not have during actions, so these actions are ignored.

\[
S_d_{\text{in-out}}[\{\text{path} : ((a_e, a_d, a_x), T, C)\}] \theta \rho \rightarrow (\rho', \text{Succeeded}(v)) :
\]
\[
C_d[[C]] \theta \rho \rightarrow (\rho', \text{Succeeded}(v)).
\]

\[
S_d_{\text{in-out}}[\{\text{path} : ((a_e, a_d, a_x), T, C)\}] \theta \rho \rightarrow \text{res} :
\]
\[
C_d[[C]] \theta \rho \rightarrow (\rho', \text{Failed}),
T [[T]] \theta \rho [\text{path}] \text{nil} \rightarrow \text{res}.
\]

**Common Dynamic Environment Types**

Although each of the notations differs slightly on what is stored in their dynamic environment, they share a common core, as follows.

\[
\begin{align*}
Evt & \::= \text{id} \mid \text{NullEvt}.
\text{Bool} & \::= \text{TRUE} \mid \text{FALSE}.
\text{Val} & \::= \text{RealVal}(v) \mid \text{IntVal}(i) \mid \text{Bool}(b) \mid \ldots
\text{InputEnv} & \::= \text{id} \rightarrow v.
\text{OutputEnv} & \::= \text{id} \rightarrow v.
\text{VariableEnv} & \::= p \rightarrow v.
\text{StateEnv} & \::= p \rightarrow b.
\text{Stack} stk = \text{InputEnv list}.
\end{align*}
\]

The common environment consists of the values of variables, the current event, and a Boolean describing whether the chart has been initialized.

\[
\text{CEnv} cenv ::= ((\text{ie}, \text{oe}, \text{ve}, \text{se}), \text{evt}, \text{init})
\]

We define a complete environment for a dialect as this common environment and a dialect-specific difference portion \(\text{diff}\):

\[
\text{Env} ::= (cenv, \text{diff})
\]

Using this, we can define versions of the environment functions for the common portion

\[
\text{set-trigger-evt}: \text{Env} \rightarrow \text{Path} \rightarrow \text{Env}.
\text{set-trigger-evt} ((v, \text{evt}, \text{init}), \text{diff}) \text{new} \rightarrow ((v, \text{new}, \text{init}), \text{diff}).
\]

\[
\text{is-in-state}: \text{Env} \rightarrow \text{Path} \rightarrow \text{Bool}.
\text{is-in-state} ((v, \text{evt}, \text{init}), \text{diff}) \text{path} \rightarrow (\text{se path}).
\]

\[
\text{c-is-in-state}: \text{CEnv} \rightarrow \text{Path} \rightarrow \text{Bool}.
\text{c-is-in-state} (v, \text{evt}, \text{init}) \text{path} \rightarrow (\text{se path}).
\]

\[
\text{set-in-state}: \text{Env} \rightarrow \text{Path} \rightarrow \text{Bool} \rightarrow \text{Env}.
\text{set-in-state} (((\text{ie}, \text{oe}, \text{ve}, \text{se}), \text{evt}, \text{init}), \text{diff}) \text{path in} \rightarrow (((\text{ie}, \text{oe}, \text{ve}, \text{se}) \oplus \{\text{path} \rightarrow \text{in}\}), \text{evt}, \text{init}), \text{diff}).
\]

\[
\text{is-statemachine-initialized}: \text{Env} \rightarrow \text{Bool}.
\text{is-statemachine-initialized} ((v, \text{evt}, \text{init}), \text{diff}) \rightarrow \text{init}.
\]

\[
\text{set-statemachine-initialized}: \text{Env} \rightarrow \text{Bool} \rightarrow \text{Env}.
\text{set-statemachine-initialized}((v, \text{evt}, \text{init}), \text{diff}) \text{new} \rightarrow ((v, \text{evt}, \text{new}), \text{diff}).
\]
additionally, we add a rule to set the inputs and extract the outputs of the environment:

\[
\text{set_inputs: } \text{Env} \rightarrow \text{InputEnv} \rightarrow \text{Env}.
\]

\[
\text{set_inputs } ((\text{ie}, \text{oe}, \text{ve}, \text{se}, \text{evt}, \text{b}), \text{diff}) \rightarrow ((\text{newie}, \text{oe}, \text{ve}, \text{se}, \text{evt}, \text{b}), \text{diff}).
\]

\[
\text{get_outputs: } \text{Env} \rightarrow \text{OutputEnv}.
\]

\[
\text{get_outputs } ((\text{ie}, \text{oe}, \text{ve}, \text{se}, \text{evt}, \text{b}), \text{diff}) \rightarrow \text{oe}.
\]

\[
\text{get_trigger_evt: } \text{Env} \rightarrow \text{Evt}.
\]

\[
\text{get_trigger_evt } (v, \text{evt}, \text{b}) \rightarrow \text{evt}.
\]

**Is_Completed_State function**

In *Rhapsody* and UML Statecharts, to check whether we can transition based on the NULL event, we first must determine whether or not the state has completed. We define an *is_completed_state* function to determine whether a state has completed. Writing this as a function is a small bit of an abuse of notation, but it is much more verbose to write as a rule set.

\[
\text{active_child } \rho \phi \rightarrow \bot
\]

\[
\text{active_child } \rho (hd::tl) \rightarrow
\]

\[
\begin{cases} 
\text{if } (c\_is\_in\_state } \rho \text{ hd } = \text{TRUE}) \text{ then } hd \\
\text{else } \text{active_child } \rho \text{ tl } 
\end{cases}
\]

\[
\text{is_final_state } \theta \text{ path } =
\]

\[
(\theta(path) = path : ((a_e, a_d, a_x), T, C, TRUE)).
\]

\[
\text{is_complete_composition}[[\text{And(\phi)}]] \theta \rho = \text{TRUE}.
\]

\[
\text{is_complete_composition}[[\text{And(hd::tl)}]] \theta \rho =
\]

\[
(\text{is_complete_state } [[\text{hd}]] \theta \rho) \land 
(\text{is_complete_composition}[[\text{And(tl)}]] \theta \rho).
\]

\[
\text{is_complete_composition}[[\text{Or(T, \phi)}]] \theta \rho = \text{TRUE}.
\]

\[
\text{is_complete_composition}[[\text{Or(T, hd::tl)}]] \theta \rho =
\]

\[
\text{let } \text{active } = (\text{active_child } \rho (hd::tl)) \text{ in}
\]

\[
\begin{cases} 
\text{if } (\text{contains_final_state } \theta (hd::tl)) \text{ then } \\
\text{is_final_state } \theta \text{ active } \\
\text{else } \text{is_completed_state } \text{active } \theta \rho. 
\end{cases}
\]

\[
\text{end}
\]

\[
\text{is_completed_state } \text{path } \theta \rho :-
\]

\[
\theta(path) = [[\text{path} : ((a_e, a_d, a_x), T, C))].
\]

\[
\text{is_complete_composition } \text{C} \theta \rho.
\]

**4.3 Stateflow Instantiation of the Core Semantics**
Now we describe the Stateflow instantiation of the rules. We describe in slightly more detail the declaration structure for charts, describe the dynamic environment used for computation, and provide implementations for the interface rules.

**Stateflow Declarations**

For the dynamic semantics, we do not care about the typing information for variables or constants. However, we do care about the typing information regarding input events, as we see below in the definition of the is_activeEvt rules. Therefore, we will define the types of events as follows:

\[
\text{InputEventType} \ etype ::= \text{RisingEdge} \mid \text{FallingEdge} \mid \text{EitherEdge}.
\]

\[
\text{InputEventTypeDef} \ ed ::= \text{id : etype}
\]

**Stateflow Environment**

The components of the environment (\(\rho\)) are the common environment \(\text{common}\) and the call stack \(\text{stk}\). The global environment \(\sigma\) contains the current state \(\rho\) and also the input environment from the previous step, in order to deal with Stateflow’s baroque input event semantics.

\[
\begin{align*}
\text{Diff} \ diff & = \text{stk} \\
\text{Env} \ \rho & ::= (\text{common}, \ diff) \\
\text{GEnv} \ \sigma & = (\rho, \ ie)
\end{align*}
\]

**Instantiation of The Interface Rules**

Below we use the Stateflow environment to implement the environment interface rules. We use NullEvt for Stateflow transitions that do not have a triggering event; for Stateflow, this trigger always succeeds.

\[
is\_trigger\_evt \ (\theta ((v, \ evt, \ b), \ stk)) \ path \ trigger \rightarrow \text{TRUE} :-
(\text{trigger} = \text{evt} \lor \text{trigger} = \text{NullEvt}).
\]

\[
is\_trigger\_evt \ (\theta ((v, \ evt, \ b), \ stk)) \ path \ trigger \rightarrow \text{FALSE} :-
\text{trigger} \neq \text{evt},
\text{trigger} \neq \text{NullEvt}.
\]

\[
\text{set\_transition\_occurred} \ \rho \rightarrow \rho.
\]

There are only a couple of interface rules remaining to complete the common rules: choose_elem and \(S_d\).

\[
\text{choose\_elem} \ (\text{hd} :: \text{tl}) \rightarrow (\text{hd}, \ \text{tl}).
\]

\[
S_d [[\text{St}]] \theta \rho \rightarrow \text{res} : S_d\_\text{out\_in} [[\text{St}]] \theta \rho \rightarrow \text{res}.
\]

**Implementing Local Events**

We do not describe most of the expressions and actions used in Stateflow; they are obvious, if tedious to describe. However, it is worth describing how local events are generated and propagated as actions. Stateflow, unlike UML Statecharts and Rhapsody, has a preemption semantics for local events rather than a function call semantics for local events. This means that a send action interrupts the evaluation of the current event to broadcast another event. However, it is quite trivial to model using our approach. We can send an event to the root (\(\phi\)) or a state path. If the state is active (note that the root composition is always
A [[send(path, evt)]] θ ρ → ρ'' :-
get_trigger_evt ρ → old,
set_trigger_evt ρ evt → ρ,
((path = hd::tl, is_in_state(ρ, path) → TRUE, S_d ((θ(L)) ρ’ → ρ’)) ∨
(path = ϕ, C_d ((θ(root))) ρ’ → ρ’)),
set_trigger_evt ρ old → ρ''.

A [[send(hd::tl, evt)]] θ ρ → ρ :-
is_in_state(ρ, path) → FALSE.

Top-Level Rules

Stateflow actually can react to multiple events within a single step, and it uses a somewhat bizarre
activation semantics for events. As described earlier, it allows events to be of type RisingEdge,
FallingEdge, or EitherEdge. Therefore, we must compare events to their value in the previous step in order
to see whether they ‘fire’. For brevity, we represent the is_active_evt rule as a function:

is_active_evt(e, etype, curr, prev) =
if (et = Rising_Edge) then ((curr e) ∧ ¬(prev e))
else if (et = Falling_Edge) then (¬(curr e) ∧ (prev e))
else if (et = Either_Edge) then (curr e) ≠ (prev e)

Top-Level Rules

Stateflow actually can react to multiple events within a single step, and it uses a somewhat bizarre
activation semantics for events. As described earlier, it allows events to be of type RisingEdge,
FallingEdge, or EitherEdge. Therefore, we must compare events to their value in the previous step in order
to see whether they ‘fire’. For brevity, we represent the is_active_evt rule as a function:

is_active_evt(e, etype, curr, prev) =
if (et = Rising_Edge) then ((curr e) ∧ ¬(prev e))
else if (et = Falling_Edge) then (¬(curr e) ∧ (prev e))
else if (et = Either_Edge) then (curr e) ≠ (prev e)

is_active_evt: Id → EType → IEnv → IEnv → Bool.

is_active_evt(e, etype, curr, prev) =
if (et = Rising_Edge) then ((curr e) ∧ ¬(prev e))
else if (et = Falling_Edge) then (¬(curr e) ∧ (prev e))
else if (et = Either_Edge) then (curr e) ≠ (prev e)

EL: InputDefList → Chart → Env → EV → EV → Env
EL [[(InputEvt(e : etype))::e]] θ ρ curr prev → ρ'':
is_active_evt(e, etype, curr, prev) = TRUE,
Evt θ ρ e → ρ',
EL [[e]] θ ρ curr prev → ρ'.'

EL [[(e : etype) ::e]] θ ρ curr prev → ρ'':
is_active_evt(e, etype, curr, prev) = FALSE,
EL [[e]] θ ρ curr prev → ρ'.'

EL [[InputVar(v) ::e]] θ ρ curr prev → ρ'':
EL [[e]] θ ρ curr prev → ρ'.'

Additionally, it is possible to define a Stateflow chart with no input events. In this case, we generate a
single NullEvt each step.

step: Chart → GEnv → IEnv → GEnv

step θ (ρ, v) ie → ((ρ', v), extract_outputs(ρ)) :-
no_input_evt_chart(θ),
Evt θ (set_inputs(ρ, ie)) NullEvt → ρ'.

step θ (ρ, prev) ienv → ((ρ', ienv), get_outputs(ρ)) :-
θ = (root, srcs, ind, outd locd)
EL [[ind]] θ ρ curr prev → ρ'.'

A run then generates a list of output environments given a list of input environments as follows:
run_internal: Chart $\rightarrow$ GEnv $\rightarrow$ (IEnv list) $\rightarrow$ (OEnv list).

run_internal $\theta$ genv $\phi$ $\rightarrow$ $\phi$.

run_internal $\theta$ genv $\{ienv∷tl\}$ $\rightarrow$ $oenv∷oenv_d$ $\Rightarrow$
step $\theta$ genv ienv $\rightarrow$ (genv', oenv),
run_internal $\theta$ genv'tl $\rightarrow$ oenv_d.

run: Chart $\rightarrow$ (IEnv list) $\rightarrow$ (OEnv list).

run $\theta$ inputs $\rightarrow$ outputs $\Rightarrow$
-initialize_statemachine $\theta$ $\rightarrow$ $\rho$,
run_internal ($\rho$, $\phi$) $\rightarrow$ outputs.

This concludes the presentation of semantics for Stateflow.

### 4.4 Rhapsody Instantiation of the Core Semantics

**Rhapsody Declarations and Environment**

For the purposes of the dynamic semantics, we do not need to further elaborate the declarations in Rhapsody.

The components of the environment ($\rho$) are the common environment $common$ and a pair consisting of the list of generated events ($genEvts$) and a Boolean $trns$, which describes whether any transition has occurred in the current step.

$$Diff \ diff ::= \ (genEvts, \ trns)$$

$$Env \ \rho ::= \ (common, \ diff)$$

**Instantiation of The Interface Rules**

We use the Rhapsody environment to implement the environment interface rules. The only interesting rule is the is_trigger_evt, because of the behavior of completion events. We use the is_completed_state function to determine whether we can take transitions without triggering events.

$$is\_trigger\_evt: \ Chart \rightarrow \ Env \rightarrow \ Path \rightarrow \ Event \rightarrow \ Bool$$

$$is\_trigger\_evt \ \theta \ ((v, \ evt, \ init), \ d) \ path \ trigger \rightarrow \ TRUE :\ -(trigger = \ evt).$$

$$is\_trigger\_evt \ \theta \ (common, \ d) \ path \ trigger \rightarrow \ TRUE :\ -(\ (is\_completed\_state \ path \ \theta \ common) = \ TRUE).$$

$$is\_trigger\_evt \ \theta \ ((v, \ evt, \ init), \ d) \ path \ trigger \rightarrow \ FALSE :\ (\trigger \neq \ evt) \lor (is\_completed\_state \ path \ \theta \ ((v, \ evt, \ init), \ d)) = \ FALSE).$$

$$set\_transition\_occurred \ (c, \ (evts, \ trns)) \rightarrow \ (c, \ (evts, \ TRUE)).$$

$$get\_transition\_occurred \ (c, \ (evts, \ trns)) \rightarrow \ trns.$$  

$$queue\_evt(c, \ (evts, \ trns)) \ evt \rightarrow \ (c, \ (evts\_evts, \ trns)).$$

$$dequeue\_evt\_nondet(c, \ (evts, \ trns)) \rightarrow \ (elem, \ (c, \ (rest, \ trns))) :\ -(choose\_elem\_nondet \ evts \rightarrow \ (elem, \ rest)).$$
There are only a couple of interface rules remaining to complete the common rules: choose_elem and \( S_d \).

\[
\text{choose_elem \( L \rightarrow \) res :: choose_elem_nondet \( L \).}
\]

\[
S_d ([\lbrack St \rbrack]) \theta \rho \rightarrow \text{res} :: S_d\text{\_in\_out} ([\lbrack St \rbrack]) \theta \rho \rightarrow \text{res}.
\]

**Implementing Local Events**

Rhapsody has a *queueing* semantics for events; any events generated during a step are placed on a queue for later extraction. This makes their specification quite straightforward. In the current document, we only support events sent to the current statechart.

A \([\lbrack \text{send}(\text{evt})\rbrack] \theta \rho \rightarrow \rho' :: \text{queue\_evt} \rho \text{ \( \text{evt} \).}

**Implementing Fork and Join**

Fork and Join are not directly supported in the pseudostates in the abstract syntax tree. However, support for these constructs can be straightforwardly added by implementing them in terms of constructs already in the abstract syntax tree.

First, to support *join*, it is enough to add boundary-crossing transitions from each of the parallel states involved in the *join* to the destination of the original *join*. These transitions from the parallel regions have a guarding condition asserting that the other parallel regions are in the other states that originate the *join*. For Rhapsody, there is a possible digression from the informal semantics described by Harel in [Harel01:rhapsody] in that it is possible for a transition to enter and exit a substate within a parallel machine simultaneously depending on the interleaving chosen by the code generation scheme. An example with this behavior is shown in Figure 9.

![Figure 9](image_url)

**Figure 9:** (a) Original model with join, (b) translated model without join

Given the informal Rhapsody semantics, one would expect that in Figure 9 (a) two sequential emissions of ‘Evt’ would be required to reach state_6. The first would transition from state_4 to state_5, and the second would enable the junction transition to reach state_6. However, if the parallel machines are interleaved
left-to-right, then using our described translation, shown in Figure 9 (b) it is possible that only one emission of Evt is required to reach state_6. The sequence is as follows. Initially state_4 and state_7 are entered. Subsequently Evt is received by the chart. The parallel region on the left is evaluated first, causing a transition from state_4 to state_5. The parallel region in the right is then evaluated. Now, the transition from state_7 is enabled because the trigger event Evt is true and the left-hand parallel machine is in state 5, so the system exits the parallel machine.

On the other hand, the translation described in Figure 9 (b) is exactly what the current IBM Rhapsody tool suite (7.5.2) does to implement joins in the generated C/C++ code, so this choice matches the reference implementation semantics of Rhapsody. Note also that for model checking, we consider both interleavings, and the interleaving in which the right side is evaluated first will match the “expected” informal behavior of the original model. With the non-deterministic semantics, we expect that users will want to prove properties of all possible interleavings. In this case, some properties that may be provable in the “informal” semantics may not be provable in our translated semantics, but it will not lead to unsound analysis results.

UML has the additional restriction on join transitions that they may only be completion transitions, that is, they have no trigger events or conditions. These restrictions ensure that, for UML, there are no machines that exhibit unexpected behavior (such as the Rhapsody example in Figure 9) after translating join transitions.

Fork is straightforward to implement in our AST using the Rhapsody semantics but requires the use of DynamicChoice in UML Statecharts to ensure equivalent execution sequences. The procedure is as follows:

1. Create a “fresh” Boolean variable (call it $f_v$) for the fork.
2. Set the value of $f_v$ to FALSE as an exit action for the parallel state.
3. Remove the fork and create a transition from the same source to the boundary of the parallel state; assign $f_v$ to TRUE on this transition.
4. Modify the initial transitions for each of the parallel machines in the fork to target a new static conditional branch (Rhapsody) or dynamic branch (UML Statecharts) such that if $f_v$ is true, then the destination of the branch will be the state referenced in the fork; if it is false, then the destination will be the original initial state of the machine.

It would be possible to implement fork directly by generalizing our notion of enter_state to allow sets-of-states to be entered at a particular level of the hierarchy rather than a single state.

**Top-Level Rules**

Rhapsody has a fairly straightforward semantics at the top level. An event is chosen from the event queue and executed. Following this event, completion events occur until the machine reaches a fixpoint in which the completion event does not cause a transition to occur. Rhapsody and UML Statecharts do not actually generate output traces as cause their system states to evolve.

\[
\text{run_completion}: \text{Chart} \rightarrow \text{Env} \rightarrow \text{Env}.
\]

\[
\text{run_completion} \theta \rho \rightarrow \rho''; \\
\text{get_transition_occurred} \rho \rightarrow \text{TRUE}, \\
\text{Evt} \theta (\text{set_trigger_evt} \rho \text{NullEvt}) \rightarrow \rho', \\
\text{run_completion} \theta \rho' \rightarrow \rho''.
\]

\[
\text{run_completion} \theta \rho \rightarrow \rho; \\
\text{get_transition_occurred} \rho \rightarrow \text{FALSE}.
\]

\[
\text{step}: \text{Chart} \rightarrow \text{Env} \rightarrow \text{Env}.
\]

\[
\text{step} \theta (c, (\phi, \text{trns})) \rightarrow (c, (\phi, \text{trns})).
\]
This concludes the presentation of semantics for Rhapsody.

### 4.5 UML Statecharts Instantiation of the Core Semantics

#### UML Declarations and Environment

For the purposes of the dynamic semantics, we do not need to further elaborate the declarations in UML Statecharts.

The components of the environment are much the same as the environment for Rhapsody, with the addition of an `initEnv` argument, which records the initial state of the common environment when the step began executing.

\[
\text{Diff} \; \text{diff} ::= ((\text{genEvts}, \text{trns}), \text{initEnv}) \\
\text{Env} \; \rho ::= (\text{common}, \text{diff})
\]

The interface rule definitions, other than `is_trigger_evt`, are (tediously) the same as Rhapsody, but must be re-defined to account for the `initEnv` argument. The `is_trigger_evt` rule is exactly the same as Rhapsody and will be omitted.

\[
\text{set_transition_occurred} (c, (\text{evts}, \text{trns}, \text{initEnv})) \rightarrow (c, (\text{evts}, \text{TRUE}, \text{initEnv})). \\
\text{get_transition_occurred} (c, (\text{evts}, \text{trns}, \text{initEnv})) \rightarrow \text{trns}. \\
\text{queue_evt}(c, (\text{evts}, \text{trns}, \text{initEnv})) \rightarrow (c, (\text{evts}, \text{initEnv})). \\
\text{dequeue_evt_nondet}(c, (\text{evts}, \text{trns}, \text{initEnv})) \rightarrow (\text{elem}, (c, (\text{rest}, \text{trns}, \text{initEnv}) ::- \\
\quad \text{choose_elem_nondet} \; \text{evts} \rightarrow (\text{elem}, \text{rest})).
\]

Additionally, we add an `update_init_env` to copy over our initial environment with the “current” environment to support dynamic choice operations.

\[
\text{update_init_env} (\text{common}, ((\text{genEvts}, \text{trns}), \text{initEnv})) \rightarrow (\text{common}, ((\text{genEvts}, \text{trns}), \text{common})).
\]

The UML fixed-step vs. the Rhapsody microstep is controlled by using `initEnv` as the environment for evaluation of all `conditions` within the chart and using the `common` environment to store all of the updates that occur as a result of actions. This can easily be achieved in the definitions of actions and conditions by choosing the `common` environment for action updates and the `initEnv` environment for all references to conditions. On the other hand, the `dynamic choice` construct is difficult to reconcile with this view of the semantics.

#### Implementing Dynamic Choice

UML `dynamic choice` junctions are perhaps the single most ill-defined feature of UML Statecharts. The meaning of such transitions is straightforward in the case in which no parallel execution is present, but extraordinarily difficult to define in the presence of parallel statemachines, as discussed in Section 2.5. For the current document, we make the simplest possible choice from our semantic perspective, which is that a dynamic choice unifies the previous step and current step environments. Alternate possible variations are
described in Section 2.5. To follow the UML semantics, we require that one of the outgoing transitions from the dynamic choice reaches a state.

\[
D[[\text{path}]] \theta \rho P \text{transact} \rightarrow \text{res} \quad :- \\
\text{\theta(path) = } [[p : \text{Pseudo(DYNAMIC_CHOICE, TL)}]]. \\
\text{A } [[\text{transact}]] \theta \rho \text{src} \rightarrow \rho'; \\
\text{update_init_env(\rho') } \rightarrow \rho''; \\
T [[TL]] \theta \rho P \text{transact} \rightarrow \text{res}, \\
\text{res} = (\rho''', \text{Succeeded(STATE)}).
\]

**Top-Level Rules**

Rhapsody has a fairly straightforward semantics at the top level. An event is chosen from the event queue and executed. Following this event, completion events occur until the machine reaches a fixpoint in which the completion event does not cause a transition to occur. Rhapsody and UML Statecharts do not actually generate output traces as cause their system states to evolve.

\[
\text{run_completion: Chart } \rightarrow \text{Env } \rightarrow \text{Env}. \\
\text{run_completion } \theta \rho \rightarrow \rho''' \quad :- \\
\text{get_transition_occurred } \rho \rightarrow \text{TRUE}, \\
\text{update_init_env } \rho \rightarrow \rho'; \\
\text{Evt } \theta (\text{set_triggerEvt } \rho \text{NullEvt}) \rightarrow \rho'; \\
\text{run_completion } \theta \rho' \rightarrow \rho''; \\
\text{run_completion } \theta \rho \rightarrow \rho \quad :- \\
\text{get_transition_occurred } \rho \rightarrow \text{FALSE}.
\]

\[
\text{step: Chart } \rightarrow \text{Env } \rightarrow \text{Env}. \\
\text{step } \theta (c, (\phi, \text{trns})) \rightarrow (c, (\phi, \text{trns})). \\
\text{step } \theta \rho \rightarrow \rho'''' \quad :- \\
\text{update_init_env } \rho \rightarrow \rho'; \\
\text{deque_elem_nondet } \rho' \rightarrow (\text{elem}, \rho'''), \\
\text{Evt } \theta (\text{set_triggerEvt } \rho'' \text{elem}) \rightarrow \rho'''', \\
\text{run_completion } \theta \rho'''' \rightarrow \rho''''.
\]

This concludes the presentation of semantics for UML Statecharts.

**5. Related Work**

There are at least dozens and perhaps hundreds of different Statecharts dialects [vonderBeek94:statecharts]. A handful of these dialects were created with a formal semantics, including *Esterel SyncCharts* [Andre03:synccharts], *RSML* [Heimdahl96:analysis], and *RSML*” [Whalen00:ms-thesis], but for the most part they are defined informally; semantics have been ascribed to these notations after the fact. The three dialects considered here: Stateflow, UML Statecharts, and Rhapsody, all fit in the category of informally defined semantics.

For the informal notations in widespread use, multiple formalizations have been proposed. For UML models, formalizations have been provided in a variety of notations, including ASMs, structural operational semantics (SOS) and rewriting to simplified machines.
There are several formalizations of UML Statecharts using ASMs. The most complete formalization for the single chart case is [Borger00:UML]. Borger provides a nice, modular description of the UML semantics. The treatment in Borger does not match the informal semantics of [OMG:UML] in an important way, however, involving the interleaving of actions and conditions. In [OMG:UML] the set of all transitions to be executed is considered (including junctions and nested initial states) prior to any actions (including assignments) on any of those transitions. However, in Borger, the sequence of evaluation is interleaved. For example, when evaluating the model in Figure 6 from Section 2.8 using the semantics in [Borger00:UML] the interpretation will coincide with the Rhapsody semantics rather than the UML semantics. This interleaving is also present in Borger’s proposal for boundary crossing transitions into AND-states.

We also model completion events as a sequence of null transitions as in Rhapsody rather than as a sequence of events generated sequentially. This allows us to discard these “implicit” events –they can be reconstructed by examining the terminated predicate for an active state – and allows for a uniform treatment of events at the top level. We provide support for a handful of constructs not supported in Borger, notably dynamic choice pseudostates and better support for multiple fork and join transitions from the same parallel state set. The work of [Jin02:diagrams] builds on [Borger00:UML] by adding support for multiple concurrent state machines. A less complete UML Statecharts formulation in ASM is provided by Compton et al [Compton00:UML].

Our work is perhaps most similar to the operational semantics described by von der Beek [vonderBeek03:UMLStatecharts]. Von der Beek considers a subset of the UML semantics than is considered in this report: namely, it does not consider initial, junction, fork, join, choice, or final pseudostates, completion transitions or guards, and also unifies the static and dynamic information in a state machine to one structural description. It also uses negative premises, which are problematic for SOS rules as the rule system may no longer be monotonic. The work of von der Beek is built on earlier work by Latella [Latella99:UMLStatecharts] that considers a smaller subset.

Semantics for relatively small subsets of UML statecharts are provided using several different formal frameworks by several groups. For example, Gogolla et al. [Gogolla98:UML] through translation to an abstract machine using graph rewriting and Reggio et al. [Reggio00:UML] through translation to the algebraic specification language CASL.

There is no work that we are aware of that specifically formalizes Rhapsody semantics.

Stateflow semantics have been formalized by Hamon twice: once as an operational semantics [Hamon04:stateflow] and later as a denotational semantics [Hamon05:stateflow]. As mentioned earlier, we base our operational semantics on the denotational rather than the operational semantics. The reason for this choice is both that the denotational semantics is more complete and more modular, and there was little difficulty in moving from Hamon’s continuations to an alternate backtracking form that explicitly returned success or failure. Moreover, we have fixed several small errors in Hamon’s semantics as follows:

**Transition action sequencing:** For transition actions, actions associated with a later transition segment should be performed after actions on previous segments but before state entry actions. In the EMSOFT semantics, transition actions are always pushed before the passed-in success continuation. If multiple path segments have transition actions, then they are executed in reverse order of the informal semantics of [Stateflow].

**AND-state boundary-crossing transitions:**
The EMSOFT semantics assume that boundary crossing entry transitions only traverse OR-states. If an AND-state boundary is crossed, only the explicitly entered child (due to the boundary crossing transition) is entered - none of the children are entered.
Multi-segment boundary crossing transitions:
To compute the scope of a transition in Stateflow, the source state, target state, and position of the intermediate junctions are used. The EMSOFT semantics does not include intermediate junctions in its scope computation leading, in some cases, to the scope of a transition being too "small", not including ancestor states that should be exited and entered.

Flowcharts in states with no substates:
The EMSOFT semantics has a small error in that it does not evaluate OR-compositions with no substates, even if there are default transitions for the state (as occurs with states containing the Stateflow flowchart idiom).

Border-crossing initial transitions
The entry actions for states whose borders were crossed are not called in the EMSOFT semantics, nor are the ancestor states entered.

Internal transitions
The execution of internal transitions (that is, transitions whose initial segments do not exit the active state’s border) incorrectly do not execute ‘during’ actions.

External Loop Transitions
External loop transitions do not evaluate entry / exit actions for the state involved in the external loop.

Additionally there is a body of work that ascribes semantics to multiple dialects. Perhaps the most ambitious is the template-based semantics of Jianwei Niu, Nancy Day, and Jo Atlee [Niu03:template-semantics] that attempts to provide a generic template semantics for arbitrary Statecharts variants as well as for process algebras such as CSPs and CCS. These semantics split evaluations into macrosteps, which correspond to the evaluation of an external stimulus. The macrosteps are split into microsteps which describe a sequence of internal transitions within the model. The semantics allow for parameterization along several axes such as the relative priorities of transitions, the macro semantics that compose microsteps, how to update states, events, and variables at the beginning of macrosteps. Although the template mechanism in [Niu03:template-semantics] is quite flexible, it is not sufficient to describe the behavior of any of the three notations considered here. In particular, there is no distinction between states and pseudostates, so it is not possible to describe the different behaviors (such as backtracking and looping in Stateflow junctions and dynamic vs static choices in UML) that distinguish states from pseudostates. Also, there is no mechanism to distinguish condition actions from transition actions (as in Stateflow), so it is not possible to correctly model (one of) condition or transition actions. Other aspects that would be difficult to model in this framework include: valued events (Rhapsody and UML) and deferred events (UML).

Future Work

- We can define error rules that yield an error value when execution does not satisfy the structural obligations of a chart (e.g., if an OR-state is active, exactly one child of the OR-state is active). These call out the various ways that execution can ‘go wrong’ and can be used to autogenerate well-formedness properties for tools such as model checkers.
- We could re-formalize in the style of modular SOS or implicit modular SOS to greatly simplify the rules.
- The non-determinism of UML Statemachines and Rhapsody is probably less deterministic than any tool implementing the semantics. Perhaps a better representation would be a denotational semantics with an unspecified choice function.
- [MORE HERE?]
Bibliography


