Higher Order Process-Algebraic Axiomatizations of Statecharts Variants

James H. Andrews

Dept. of Computer Science
University of Western Ontario
London, Ontario, Canada N6A 5B7

Abstract. Axiomatizations are given for some prominent variants of the semantics of the statecharts formalism. The axiomatizations highlight the similarities and differences amongst the semantics. In particular, all the semantics rely on the same notion of "microstep", but different notions of "step sequence".
A process-algebraic approach is taken, and an executable variety of higher order logic is used, allowing test runs to be performed concerning the effects of the semantics. It is shown that the higher order logic setting facilitates the addition of complex features not previously studied in process-algebraic approaches, including state-exiting transitions and history transitions. Other desirable features are also added, such as state name scoping.

1 Introduction

Despite being a relative latecomer to the concurrent systems scene, the statecharts formalism [Har87] has achieved some notable successes. Closely related formalisms have been used in safety-critical projects in aeronautics [LHHR94], with the effect that statecharts are now becoming widespread in the aeronautics and aviation industry [ADJ97]. A version of it has also been integrated into the Unified Modelling Language (UML), one of the most important object-oriented design tools [HG97]. Its success can be attributed, in part, to the fact that every statechart has a diagrammatic representation which is easy even for non-specialists to read.

However, its youth is not the only hurdle the statecharts formalism has faced. There has also been considerable confusion regarding its semantics. Its originator, David Harel, intended it to be a "visual formalism" with both a visual component and a formal definition. In spite of this, the only rigorous early semantics [HPSS87] is widely regarded as having deficiencies. In 1991 Pnueli and Shalev produced a semantics which seemed to correct the problems [PS91]. However, in 1996 Harel and Naamad [HN96] described a different semantics to Pnueli and Shalev's – the semantics of the Statemate tool, which was the original implementation of statecharts. Harel and Naamad's description is informal, and is imprecise on some points. Mikk et al. [MLPS97] give a more formal version, but of course can only argue informally that it is equivalent to Harel and Naamad's.

Added to the confusion about the "official" semantics is the proliferation of statecharts variants described in the literature. Many of these variants omit features of statecharts included in the early descriptions, such as history or state-exiting transitions; many add features, such as prioritized transitions or timing constraints. Day [Day93] and von der Beeck [vdB94] describe some of the variants, their properties and their problems. It is unclear when or whether any resolution will be achieved amongst all these semantic models.

This paper therefore takes a different tack: it provides a high-level and partially executable semantics for a variety of statecharts variants, thus highlighting their similarities and their differences, by providing modularized axiomatizations of those variants in an executable variety of higher order logic. It is hoped that with similar semantics, present and future researchers can express succinctly and homogeneously the precise nature of their view of the meaning of a statechart, and test the effects of their semantics directly.

1.1 Process-Algebraic Presentations

We take, as a starting point, previous work on process-algebraic presentations of statecharts. Early work on statechart semantics [HPSS87, PS91] gave mathematical definitions of statechart diagrams which resembled definitions of finite state machines; that is, with sets of states, sets of transitions over those states, a "substate" relation and so on. This tended to produce complex semantics with many auxiliary definitions. It also allowed nonsensical statechart diagrams to be built, which had to be disallowed in separate definitions.

Uselton and Smolka [US94] were the first to realize that the elegant and parsimonious process algebra style of Milner's CCS [Mil80] could be used to express simply the syntax and operational semantics of statecharts. The main differences between statecharts and such systems as CCS are that statecharts use broadcast communication and have hierarchically structured states. Uselton and Smolka's semantics took account of these differences.

Unfortunately, broadcast communication can lead to unpleasant causal paradoxes such as self-triggering (in which a process makes a set of transitions not triggered by events in the environment) and inconsistency (in which a transition leads to actions which negate its own condition). Levi [Lev96] showed that Uselton and Smolka's semantics leads to self-triggering and does not sufficiently distinguish between idle and active processes. She gave another process-algebraic semantics, which she proved equivalent to Pnueli and Shalev's [PS91]. However, both Uselton and Smolka's and Levi's semantics depend on a complex structure of events auxiliary to the main structured operational semantics; neither attempts to capture either variant of the Statemate semantics [HN96]; and neither considers such features as state-exiting transitions.

1.2 The Approach of this Paper

In this paper, we use higher order logic [Chu40] to axiomatize several statecharts semantics. The main advantages of higher order logic are that (a) it is a general

and widely accepted logical framework, (b) it has a built-in mechanism (lambda-abstraction) for the variable scoping constructs we need to express such things as loops in diagrams, and (c) its type system forces us to be clear about the types of objects worked with in the semantics. Using higher order logic, in contrast to a higher order functional programming language such as ML, allows us to express more naturally the *relation* between a nondeterministic process and its possible outputs.

Some languages built on higher order logic, such as the specification language S [JDD94], have the further advantages of allowing specifications to be written in ASCII characters (encouraging portability), and being equipped with typecheckers. Indeed, Day [Day93] has given a version of statechart semantics in higher order logic for use in model checking and theorem proving.

Another advantage of higher order logic is that there is at least one executable variant of it. Lambda Prolog [NM94, Mil96] is a logic programming language based on higher order logic. In this paper, we use Lambda Prolog notation to express the axiomatization, giving us one further side benefit: we can use the Lambda Prolog system to execute test cases for the semantics. (Miller has used Lambda Prolog to define the pi-calculus [Mil96] in a somewhat similar manner.) The ability to actually run test cases is especially useful, and such a capability may have allowed earlier researchers to avoid problems in their semantics without having to buy commercial tools. All the semantics referred to in this paper are available freely on the World Wide Web [And98], as is the Lambda Prolog system itself [Mil96].

The three top-level step semantics we define here are in the style of Harel and Naamad's synchronous and asynchronous semantics [HN96], and Pnueli and Shalev's semantics [PS91]. Our goal here is not to achieve, for its own sake, equivalence with previously-published semantics. Rather, we wish to present semantics which avoid the paradoxes of broadcast communication (such as self-triggering) and maintain the essence of the solutions embodied in the published semantics, while retaining the elegance and simplicity of the process algebra presentation.

We also use the framework we establish to add standard features to the semantics which are absent from earlier process-algebraic treatments, such as state-exiting transitions, transition conditions on state inhabitation, and a "history" facility. We also add features, such as parameterized states and event renaming, which are not part of the original statecharts descriptions but are nevertheless useful.

The remainder of this paper is structured as follows. Section 2 gives some basic definitions. Section 3 describes the encoding of statecharts as terms, and the axiomatization of the notion of microstep, common to all the semantics. Section 4 describes the axiomatization of the three high-level semantics we consider. Section 5 describes the axiomatization of the additional features. Finally, section 6 discusses some issues to do with the execution of the semantics by Lambda Prolog, and section 7 presents some conclusions and discusses possible future work.

2 Definitions

Here we define some of the basic notions of this paper. We begin with a traditional but informal graph-theoretic description of statecharts, and then introduce the Lambda Prolog notation for higher order logic.

2.1 Statecharts

We assume given sets of *events*, *states*, *transitions*, and *state names*.

A statechart consists of a set of states (possibly with names) and a set of transitions, together with various relations amongst them. Each transition has a source state and a target state, one or more *trigger events*, a *condition*, and zero or more *actions*. Trigger events and actions are events. Conditions are of the form `in(`*statename*`)` or `not in(`*statename*`)`.

A state is either a *basic state*, an *or-state*, or an *and-state*. Or-states and and-states contain substates. When a statechart is in an or-state, it also is (and must be) in exactly one of its immediate substates; when it is in an and-state, it also is (and must be) in all of its immediate substates. Thus or-states act at their top level like conventional state-transition systems, and the immediate substates of and-states act as concurrent processes. Each or-state has a default substate, and the statechart as a whole has a root state. When a transition is made to an or-state, the substate it is in is its default substate. A statechart *configuration* consists of a set of states of the statechart consistent with the above, and describes a particular condition which the statechart can be in.

Informally, a transition is taken when its trigger event is active and its condition is true; it performs its actions on the transition, and these actions can then trigger other transitions elsewhere in the statechart. All events are broadcast to all states (the key difference with formalisms such as CCS).

The top of Figure 1 gives an example of a statechart. The statechart is an "and-state" consisting of three concurrent components (which are separated by dashed lines). The top state is a state which can make a transition on event `a`, yielding event `d1` as an action, and then make a transition on event `b`, yielding `d2` as an action. The middle state can make one of two transitions if event `c` is *not* a current event: one producing `c` itself, or another producing `d3`. The bottom state can make a transition on event `c`, producing `d4`.

2.2 Lambda Prolog Notation

A Lambda Prolog program contains *kind declarations*, *type declarations* and *clauses*. The only kind declarations we will use here are of the form

`kind` *lci* `type.`

where *lci* is an identifier (sequence of alphanumeric characters and underscores) starting with a lower case letter. This declaration declares *lci* to be a type, suitable for mentioning in type expressions. The predefined types are `o`, the type of propositions, and `int`, the type of integers.

Lambda Prolog has an LF-style [HHP93] type system. The type expressions we will use are the user-defined or predefined types, and expressions of the form (`list` *type-expr*) or (*type-expr* `->` *type-expr*). Type declarations are of the form

`type` *lci type-expr*.

Thus, for example, the standard `map` predicate on integer lists, which maps a binary relation over the list to produce another list, can be declared as follows:

```
type map (int -> int -> o) -> (list int) -> (list int) -> o.
```

In higher order logic, a term is a constant, a lambda-abstraction $\lambda var.term$, or an application ($term\ term$). Terms of the form $(\ldots((t_1\ t_2)\ t_3)\ldots\ t_n)$ can be written $(t_1\ t_2\ t_3\ \ldots\ t_n)$. In Lambda Prolog, free variables are identifiers which start with upper-case letters, but lambda-abstraction variables are written as constants (lower case identifiers). The lambda-abstraction $\lambda x.term$ is written `x \` *term*. The usual typing rules of higher order logic [HHP93] apply. The special variable "_", called the *anonymous variable*, is a variable which is distinct each time it is used; we usually use it to hold the place of variables whose values we don't care about. Terms of the type (`list` T`)` can also be of the form `nil` (the empty list) or of the form ($expr_1$ `::` $expr_2$), where $expr_1$ is of type T and $expr_2$ is of type (`list` T`)`.

A *clause* in Lambda Prolog is of the form "*atom*." or of the form "*atom* `:-` $atom, atom, \ldots, atom$.", where each *atom* is a term of type `o`. As in Prolog, the `:-` is interpreted as a backwards implication arrow $\leftarrow$, the comma is interpreted as conjunction, and the free variables are implicitly interpreted as being universally quantified over the whole clause. Thus the `map` predicate can be defined using the following clauses:

```
map _ nil nil.
map R (X::Xs) (Y::Ys) :-
  R X Y,
  map R Xs Ys.
```

Note the use of the anonymous variable "_" in the first clause to indicate the irrelevance of the relation being mapped.

3 Statechart Terms and Microsteps

3.1 Statechart Terms

Following Uselton and Smolka [US94] and Levi [Lev96], we define statecharts as terms, in which a statechart together with its current configuration are encapsulated in a single term. We begin with five basic types, `event`, `name`, `cond`, `state` and `basic_state`.

Following Day [Day93], we generalize conditions (type `cond`) as being built up from basic conditions of the form `(ev Event)` or `(in Statename)`, using the operators `andc`, `orc` and `notc`. `(ev Event)` means that the given `Event` has been raised by the environment or by a concurrent state, and `(in Statename)` means

that the statechart is currently in the state labelled by `Statename`. We define the types of the operators to conform to this usage.

In statecharts, a superstate with transitions leading from it can enclose an arbitrarily complex substate. It is therefore necessary to preserve a distinction between basic and non-basic states: basic states can perform transitions and can act as the enclosing state of a superstate, while non-basic states can contain, for example, parallel components. We define basic states as being of one of the following forms.

- `null`, the state that can do nothing.
- `(reg_trans Cond Actions State)`, the state which can make a regular transition to `State` if `Cond` is true, performing the list of events `Actions`. (This form of state is similar to CCS processes of the form $a.P$.)
- `(alt B C)`, the state which can act either like `B` or like `C` (similar to $B + C$ of CCS).

So we declare, for instance,

```
type reg_trans  cond -> list event -> state -> basic_state.
```

(Non-basic) states are of one of the following forms.

- `(basic B)`, where `B` is a basic state.
- `(and S T)`, the state which consists of states `S` and `T` operating in parallel and possibly communicating (similar to $S \mid T$ of CCS).
- `(super S B)`, the state which contains the substate `S`, but which makes a transition of the basic state `B` if that transition is enabled.
- `(named Name S)`, the state `S` labelled with the name `Name`.
- `(loop LxS)`, where `LxS` is of type `state -> state`. `(loop x\S)` acts as the fixpoint, on the state variable `x`, of the process `S`. (We use the keyword "loop" rather than the more common "fix" to emphasize the use of the construct to encode looping paths in statecharts.)

As an example, the state depicted on the top of Figure 1 is represented by the statechart on the bottom.

Levi [Lev96] gives a similar language, and shows how to derive a term from a statechart diagram in a meaning-preserving fashion.

3.2 Microsteps

The major published semantics agree on the operation of a "microstep", the group of transitions and actions immediately triggered by a set of events. They differ largely concerning how the triggered actions feed back into the statechart and interact with external events.

We define the relation `microstep` of type `state -> (list event) -> (list name) -> outcome -> o` as a relation between:

- An initial state `S`;

- A list of input events;
- A list of state names (intended to be the names of all the named substates that `S` is currently inhabiting); and
- One possible outcome of the execution of that state, given those events and inhabiting state names.

Outcomes are of the type `outcome`, and are of one of two possible forms:

- `(idle S)`, indicating that the original process has stayed in the state `S` without making any transitions; or
- `(moves T Actions)`, indicating that the original process has made some internal transitions to state `T`, performing the list of events in `Actions`.

For now we consider only transitions which stay within the boundaries of a superstate.

Figure 2 gives the axioms for the `microstep` relation. `microstep` passes some of its work off to `basic_microstep`, which defines the outcomes of basic states. `basic_microstep` in turn refers to `cond_holds`, a straightforward predicate which defines when a condition holds. `microstep` defines for itself the meaning of `loop` states; for all other kinds of states, `microstep` and `basic_microstep` determine the outcomes of any constituent states and then pass those outcomes to an "outcome" predicate (`and_outcome`, `super_outcome`, etc.).

The axioms for the outcome predicates are shown in Figure 3. We will discuss two of them. `alt_outcome` gives the outcome of a basic state `(alt B C)` given the outcomes of `B` and `C`. The state is idle if both `B` and `C` are idle; it makes a transition deterministically if only one of `B` and `C` are idle; and if both `B` and `C` can make a transition, the transition made is nondeterministic. `super_outcome` states that if a superstate can make a transition, that transition is taken, even if the substate can also make a transition; and that otherwise the superstate remains, enclosing its substate, which may have itself taken a transition. The other outcome predicates are similar.

4 Three High-Level Step Semantics

The published semantics for statecharts all attempt to avoid the paradoxes of broadcast communication, such as self-triggering, while still allowing the rich expressiveness of statecharts. Each semantics can be criticized on some grounds, but each attempts to solve the paradoxes and retain expressiveness in its own way.

Here we will study axiomatizations of three semantics. The first is in the style of Harel and Naamad's synchronous semantics, and the second is in the style of their asynchronous semantics [HN96]. The third is in the style of Pnueli and Shalev's semantics [PS91].

The basic task, in all the axiomatizations, is to define the precise meaning of the predicate `step_sequence`, of type `state -> (list (list event)) -> state -> (list (list event)) -> o`, which gives the final outcome of a sequence of steps. This is a relation between:

- An initial state `S`;
- A script, which is a list of lists of events, each list of events representing the events happening externally at each step of a sequence of steps;
- A final state, which is the configuration of `S` after the entire script has been processed; and
- A trace, which is also a list of lists of events, each list representing the set of actions performed by the statechart at each step.

Consider the statechart on the top of Figure 1, whose term representation is shown at the bottom of the figure. Each of the semantics produces a different final result when given the script `(a::nil)::(b::nil)::nil` (that is, a script of two steps in which the first one receives only event `a` from the environment, and the second receives only event `b`).

4.1 Statemate Synchronous Style

In the Statemate synchronous style, the actions performed on a microstep are taken as affecting the statechart *at the next whole step*. A step *is*, in some sense, a microstep, and the actions of a microstep are added to the environment's events to produce the set of events affecting the next step.

The axioms for this style of semantics are given in Figure 4. (From this point on, we omit type declarations in the figures.) The predicate `current_statenames` simply determines which named substates the statechart is currently in, and is defined much as in Levi [Lev96]. The auxiliary predicate `after_microstep` analyzes the outcome of a microstep and defines the final state and trace based on it.

On being run on the differentiating statechart and the script `(a::nil)::(b::nil)::nil`, the Statemate synchronous semantics produces two possible traces: `(d1::c::nil)::(d2::d4::nil)::nil` and `(d1::d::nil)::(d2::nil)::nil`. The two traces correspond to the two possible transitions which can be taken by the middle concurrent substate. Note that the transition $\neg c/c$ is allowed to be followed, even though its effect negates its condition, because the effect is taken as occurring later.

4.2 Statemate Asynchronous Style

The Statemate synchronous semantics is perhaps the simplest, but is very sensitive to the precise sequence of events from the environment and how they interact with "internal" events. In the Statemate asynchronous style, a microstep again feeds its actions back into the statechart without regard for whether its actions have negated the conditions on the transitions. However, before the next set of events from the environment is processed, the statechart continues to make feedback microsteps until it is finally idle.

Harel and Naamad [HN96], in their informal semantics, do not make precise whether the events from the environment persist throughout the entire step in the asynchronous semantics. (Mikk et al. [MLPS97] do not give details of their

formal version of the asynchronous semantics.) Here we assume that environment events persist only for the first microstep, but the axioms could easily be changed.

To axiomatize this semantics, we give an overall definition of `step_sequence` which depends on a specific predicate `step`. `step` in turn performs a microstep and feeds back the results until quiescence. Figure 5 gives these axioms. The predicate `outcome_state` simply takes an outcome and returns the result state and the list of events (if any) it has performed.

On being run on the differentiating statechart and the script `(a::nil)::(b::nil)::nil`, the Statemate asynchronous semantics also produces two possible traces: `(d1::c::d4::nil)::(d2::nil)::nil` and `(d1::d3::nil)::(d2::nil)::nil`. The first trace shows that the two-transition cascade caused by the condition $\neg c$ being true has now happened all on the first step, leaving only `d2` to occur on the second. The second trace is the same as the second trace from the synchronous semantics, since no transition cascades are involved.

4.3 Pnueli-Shalev Style

A possible criticism of both the Statemate semantics is that the firing of transitions with negated conditions is still sensitive to the *sequence* of events generated internally. Furthermore, it is still possible for a transition to fire which eventually brings about a situation in which its own condition does not hold.

Pnueli and Shalev [PS91] instead take the view that a transition with condition $\neg a$, for instance, cannot logically happen on the same step as a transition which produces a. In their view, the upper transition of the middle substate of the statechart in Figure 1 should never be followed, since its action negates its own condition. They therefore define "what is in a step" as a *set* of transitions, each of which is triggered by one of the events from the environment or by some other transition, but none of which negate the conditions of any of the others. The constructive definition of this set involves taking the fixpoint of a transformation, or iterating a nondeterministic microstep operation. The price we pay for this semantics is less clarity in what transitions are allowed in certain circumstances, and greater nondeterminism in other circumstances [LHHR94].

Space does not permit us to describe our axiomatization of the Pnueli-Shalev semantics fully, but here we touch on the main points. The interested reader can access the full axiomatization [And98].

We define "tagged" versions of most of the main predicates, such as `microstep` and the outcome predicates. These tagged predicates do much the same as the originals, but also produce a version of the state being processed with tags on the transitions which have been followed. When these tagged states are fed back into the predicates, they ensure that tagged transitions can be followed and that they are not overridden, for instance by superstate transitions.

The axiomatization of the Pnueli-Shalev notion of step is contained in Figure 6. The axioms for `step_sequence` and `outcome_state` are exactly as in the Statemate asynchronous semantics. Note that `tmicrostep` is run repeatedly on the same state, adding more and more events to the current event set and more

and more tags to transitions. This continues until the process reaches a fixpoint at which the actions produced are a subset of the events input. The outcome of the original state is then taken to be the outcome of the fixpoint tagged state. If a causal paradox arises, such as a transition in a substate triggering a transition from a superstate, then a fixpoint does not exist. Following Pnueli and Shalev, we declare such a state to be idle.

On being run on the differentiating statechart and the script `(a::nil)::(b::nil)::nil`, the Pnueli-Shalev semantics produces only one possible trace: `(d1::d3::nil)::(d2::nil)::nil`. Because the upper transition of the middle concurrent state can never be followed, event `c` cannot be produced and the lower state must remain idle.

5 Adding Features

We have already described one feature not dealt with in previous process-algebraic treatments: general transition conditions including state inhabitation. Here we discuss the axiomatization of some other features of published statecharts systems, as well as some useful features which the higher order logic setting helps to axiomatize. We show that it is even possible to model state history and persistent variables with these features.

Unless otherwise stated, the features described here involve only more constants, more axioms for the `microstep` predicate, and more auxiliary predicates. The full axiomatization of these features can be found on the associated Web site [And98].

State Name Scoping, Action Renaming. It is often useful, when building large systems, to be able to allow several similar states to be active concurrently. Having a global namespace for states and actions inhibits this, since (for instance) if there are several states with the same name, it is impossible to distinguish between them in transition conditions.

We provide a state name scoping mechanism via a new form of state, `(name_scope LnS)`, where `LnS` is a term of type `name -> state`. Event scoping is more problematic because of the broadcast nature of statechart communication and the global feedback of events required for many of the semantics.

We therefore provide a bidirectional event renaming mechanism via a new form of state. `(event_rename Extevent Intevent S)` represent the state which renames any incoming event `Extevent` as `Intevent` for use internally, and any outgoing action `Intevent` as `Extevent`. This does not, however, preclude the state `S` sending out `Extevent` and having it re-enter as `Intevent` via one of the feedback loops. Alternatives include some mechanism such as Levi's event structures or a more local definition of the effect of an action.

Prioritized Transitions. Distinctions in priority between transitions can be modelled easily in a process-algebraic setting. We add a new form of basic state, `(prio B C)`, to represent the state which makes a transition of `B` if one is enabled, and otherwise makes a transition of `C` if one is enabled.

`(prio B C)` is similar in effect to `(super (basic C) B)`, which would lead one to question whether priorities are necessary. However, a transition of `C` causes us to leave `B` in `(prio B C)`, whereas a transition of `C` in `(super (basic C) B)` makes us retain B as an enclosing state.

State-Exiting Transitions. Previous process-algebraic treatments have not considered the modelling of state-exiting transitions – that is, transitions which begin in one state and end in a state outside that state's superstate. Such transitions are useful, since it may only be within a substate that one has the information necessary to trigger a transition.

We model state-exiting transitions using a new basic state and a new outcome. The basic state `(exit_trans Name Cond Actions S)` is a state which can make a transition to `S`, exiting the state named `Name`, and performing the events in `Actions`, if `Cond` is true. The rule we follow is that if more than one transition exiting a state is enabled, the one which is taken will be the one exiting the highest enclosing named state. The term `(exits Exitlist)` represents the outcome of a state with transitions which potentially exit more than one enclosing state. `Exitlist` is a term of type `(list basic_state)`, each of whose elements is a basic state of the form `(exit_trans Name Cond Actions S)`.

The provision of state exiting involves more axioms for the outcome predicates. The augmented outcome predicates ensure that an *exits* outcome always takes priority over an idle outcome or a regular move; that competing exits have their exit lists combined; and that at each named state exited, the exits from that state are weeded out as of lower priority, until the only exit transitions remaining are those exiting one state. At that point, if more than one transition remains, the choice is nondeterministic.

We have provided state-exiting transitions only for the Statemate-style semantics; the extension to the tagged predicates of the Pnueli-Shalev semantics should be straightforward.

State History. History transitions were not considered by Pnueli and Shalev or in process-algebraic approaches, but are included in the statecharts described by Harel. We model the keeping of history as something which happens on a state-exiting transition, rather than as an inherent property of a state. This allows the state moved to to be parameterized by the current configuration of the history state. It also allows history to be a relatively simple extension of state-exiting transitions.

We allow another form of basic state, `(histexit_trans Name Cond Actions LxS)`, where `LxS` is of type `state -> state`. On exiting the outermost named state with the normal exit transition `(exit_trans Name Cond Actions S)`, we move to `S`; in contrast, on exiting the outermost named state with `(histexit_trans Name Cond Actions LxS)`, we simply move instead to `(LxS T)`, where `T` is the term representing the former configuration of the state.

The version of history thus implemented is the "deep history" from Harel's original paper [Har87]. We do not attempt to axiomatize the "shallow history"

which retains only history of the top-level state, or any "history clearing" mechanism.

Value-Passing and Parameterization. We add a simple form of value-passing. We define a *parameter* as an event of the form `(param Tag Value)`, where `Tag` and `Value` are both events. We then define a new form of state, `(param_trans Cond Event Tag LeS)`, where `LeS` is of type `event -> state`. Upon receiving an event of the form `(param Tag Value)`, a state of this form makes a transition to the state `(LeS Value)`. In this way, components of a state are able to communicate values to concurrent components via parameters.

This simple form of value-passing does not permit states which both loop and depend on parameters. For this, we define another form of state, `(parloop LmLvS Initval)`, where `Initval` is of type `event` and `LmLvS` is of type `(event -> state) -> event -> state`. The one added axiom needed for this state is:

```
microstep (parloop LmLvS Val) Es Ns Outcome :-
  microstep (LmLvS u\(parloop LmLvS u) Val) Es Ns Outcome.
```

For example, let the term `Persistent_var` have as its value the term

```
m\ v\ (basic
  (alt (param_trans truec nil newval m)
       (reg_trans (ev query) (currval v :: nil) (m v))))
```

(assuming that `newval` and `query` are of type `event`, and `currval` is of type `event -> event`). Then calling `microstep` on the term `(parloop Persistent_var Initval)` is equivalent to calling it on the term

```
(basic
  (alt (param_trans truec nil newval
         (u\ (parloop Persistent_var u))
       (reg_trans (ev query) (currval a :: nil)
         (parloop Persistent_var a))))
```

That is, the term represents a persistent variable which responds with its current value when queried, and sets itself to a new value when a new value is given.

6 Execution Issues

When viewed as a Lambda Prolog program, the axiomatization in this paper, including the added features described above, comprises about 700 lines of code. The code is modularized in order to prevent undue duplication of predicate definitions. It typechecks and has been subjected to a number of tests.

Test cases can be "executed" in the sense that the query `(step_sequence State Script Final_state Trace)` can be posed to the system, with any of the three top-level semantics and various desired combination of added features. Here `State` is a ground (closed) term of type `state`, and `Script` is a ground

term of type `(list (list event))`. The system returns the final configuration of the state, and the trace of the state when run on the script.

Negation is used in the axiomatization, for instance to tell when a condition is not satisfied. Lambda Prolog implements negation as failure (NAF) [Cla78]. In general, the form of NAF it uses is unsafe, in that it can return invalid results; but it is safe if no uninstantiated variables are used within negations [And93, Stä94]. We claim that for `step_sequence` queries such as those above, this condition is always satisfied.

Thus we can compute the outcome of a state given an event script; but we cannot, for instance, run a query of the form `(step_sequence State Script Final_state Trace)` where `Script` is not instantiated but `Trace` is, and expect Lambda Prolog to correctly find a script leading to that trace. Nevertheless, the execution facility is useful for checking the correctness of the axiomatization. Furthermore, the effect of such queries as those mentioned can be achieved by standard logic programming techniques such as generate and test.

7 Conclusions and Future Work

We have given axiomatizations based on three of the most important semantics for statecharts, using notions from process algebra and an executable version of higher order logic. We have also described how further desirable features of statecharts can be axiomatized. The process-algebraic framework has proven valuable by allowing us to condense the axiomatization and focus on the behaviour of individual language features. Higher order logic has proved valuable for easily allowing us to express relationships and variable abstraction constructs.

In the future, we would like to extend this work to encompass other features. A particularly desirable feature would be event scoping, which comes so naturally in CCS and may be easier to express using Levi-style event structures than in the framework advocated here. It would also be interesting to approach the Statemate semantics more closely by adding the many features of that tool, such as actions taken on state entry and exit.

While Levi [Lev96] has given a method for translating a statechart diagram to a term as she defines it, it is not completely clear how the features we have added would affect the translation. This issue is important if we want to retain the diagrammatic representation which has proven so successful.

We would also like to study how far we can go with Lambda Prolog queries in the analysis and proof of temporal properties of states. The CCS and CSP communities have gotten a head start on statecharts in this area because they established a fixed semantics earlier. If statecharts are to be a major part of industrial object-oriented design, they will have to catch up in this regard.

8 Acknowledgments

This paper comes out of work the author did while funded by the FormalWare project. Thanks to Jeff Joyce, Nancy Day and Michael Donat for many vital

discussions about statecharts during the course of the project. Thanks also to Francesca Levi for further clarifications on her papers.

The FormalWare project is financially supported by the BC Advanced Systems Institute (BCASI), Hughes Aircraft of Canada Limited Systems Division (HCSD), and MacDonald Dettwiler Limited (MDA). While working on the project, the author derived half his funding from the project and half from the generous support of Dr. Paul Gilmore of UBC Computer Science, via his grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[ADJ97] James H. Andrews, Nancy Day, and Jeff Joyce. Using a formal description technique to model aspects of a global air traffic telecommunications network. In *Formal Description Techniques and Protocol Specification, Testing and Verification: FORTE X / PSTV XVII*, pages 417–432, Osaka, Japan, November 1997. Chapman and Hall.

[And93] James H. Andrews. A logical semantics for depth-first Prolog with ground negation. In *Proceedings of the International Logic Programming Symposium*, Vancouver, October 1993. MIT Press.

[And98] James H. Andrews. Web page for higher order process-algebraic axiomatizations of statecharts variants. `http://www.csd.uwo.ca/faculty/andrews/www/software/statecharts-lambda/index.html`, March 1998.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cla78] K. L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, New York, 1978. Plenum Press.

[Day93] Nancy A. Day. A model checker for statecharts. Master's thesis, Department of Computer Science, University of BC, Vancouver, BC, Canada, 1993.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HG97] David Harel and Eran Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, July 1997.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1), January 1993.

[HN96] David Harel and Amnon Naamad. The Statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[HPSS87] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 54–64, Ithaca, NY, June 1987.

[JDD94] Jeffrey J. Joyce, Nancy A. Day, and Michael R. Donat. S: A machine readable specification notation based on higher order logic. In *Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop*, volume 859 of *LNCS*. Springer-Verlag, 1994.

[Lev96] Francesca Levi. A process language for statecharts. In *Proceedings, Analysis and Verification of Multiple-Agent Languages*, number 1192 in LNCS, pages 388–403. Springer, 1996.

[LHHR94] Nancy G. Leveson, Mats P. E. Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[Mil96] Dale A. Miller. Lambda Prolog home page. `http://www.cis.upenn.edu/ dale/lProlog/index.html/`, 1996.

[MLPS97] Erich Mikk, Yassine Lakhnech, Carsta Petersohn, and Michael Siegel. On formal semantics of statecharts as supported by Statemate. In *Proceedings of the BCS-FACS Northern Formal Methods Workshop*, Ilkley, UK, July 1997. Springer.

[NM94] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford, 1994. Oxford University Press.

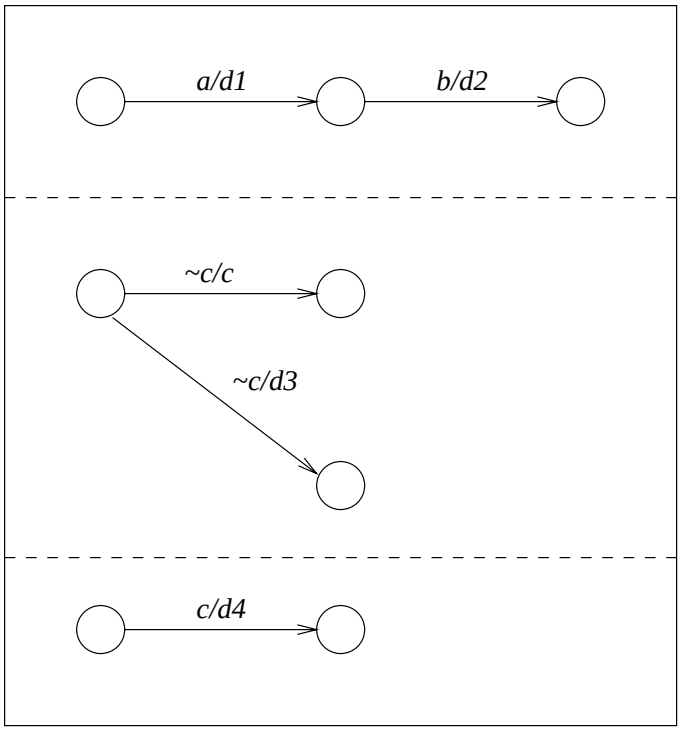
[PS91] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Proceedings of the Symposium on Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, pages 244–264, Berlin, 1991. Springer.

[Stä94] Robert Stärk. The declarative semantics of the Prolog selection rule. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS '94)*, Paris, 1994. MIT Press.

[US94] Andrew Uselton and Scott Smolka. A compositional semantics for statecharts using labelled transition systems. In *Proceedings of CONCUR 94*, volume 836 of *LNCS*, pages 2–17. Springer, 1994.

[vdB94] Michael von der Beeck. A comparison of statecharts variants. In *Proceedings, Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, Lübeck, Germany, September 1994. Springer.

This article was processed using the LaTeX macro package with LLNCS style




```
( and
  ( basic ( reg_trans (ev a) (d1::nil)
            (basic (reg_trans (ev b) (d2::nil) (basic null)))
  ) )
  ( and
    ( basic ( alt
      (reg_trans (notc (ev c)) (c::nil) (basic null))
      (reg_trans (notc (ev c)) (d3::nil) (basic null))
    ) )
    (basic (reg_trans (ev c) (d4::nil) (basic null)))
  )
)
```

Fig. 1. An example statechart, and the term which encodes it.

```
type    basic_microstep basic_state -> (list event) -> (list name) ->
                        outcome -> o.

basic_microstep null _ _ (idle (basic null)).
basic_microstep
    (reg_trans Cond Actions S) Es Ns
    (idle (basic (reg_trans Cond Actions S))) :-
  cond_holds (notc Cond) Es Ns.
basic_microstep (reg_trans Cond Actions S) Es Ns (moves S Actions) :-
  cond_holds Cond Es Ns.
basic_microstep (alt B C) Es Ns Outcome :-
  basic_microstep B Es Ns BO,
  basic_microstep C Es Ns CO,
  alt_outcome BO CO Outcome.

type    microstep       state -> (list event) -> (list name) ->
                        outcome -> o.

microstep (basic B) Es Ns Outcome :-
  basic_microstep B Es Ns Outcome.
microstep (and S T) Es Ns Outcome :-
  microstep S Es Ns SO,
  microstep T Es Ns TO,
  and_outcome SO TO Outcome.
microstep (super S B) Es Ns Outcome :-
  microstep S Es Ns SO,
  basic_microstep B Es Ns BO,
  super_outcome SO BO Outcome.
microstep (named Name S) Es Ns Outcome :-
  microstep S Es Ns SO,
  named_outcome Name S SO Outcome.
microstep (loop Lx_S) Es Ns Outcome :-
  microstep (Lx_S (loop Lx_S)) Es Ns Outcome.
```

Fig. 2. Axioms for microsteps for all semantics.

```
type    alt_outcome     outcome -> outcome -> outcome -> o.

alt_outcome (idle (basic B)) (idle (basic C)) (idle (basic (alt B C))).
alt_outcome (moves S Actions) (idle _) (moves S Actions).
alt_outcome (idle _) (moves T Actions) (moves T Actions).
alt_outcome (moves S Actions) (moves _ _) (moves S Actions).
alt_outcome (moves _ _) (moves T Actions) (moves T Actions).

type    and_outcome     outcome -> outcome -> outcome -> o.

and_outcome (idle S) (idle T) (idle (and S T)).
and_outcome (idle S) (moves T Actions) (moves (and S T) Actions).
and_outcome (moves S Actions) (idle T) (moves (and S T) Actions).
and_outcome (moves S S_Actions) (moves T T_Actions)
            (moves (and S T) Actions) :-
  append S_Actions T_Actions Actions.

type    super_outcome   outcome -> outcome -> outcome -> o.

super_outcome (idle S) (idle (basic B)) (idle (super S B)).
super_outcome (idle _)    (moves T Actions) (moves T Actions).
super_outcome (moves _ _) (moves T Actions) (moves T Actions).
super_outcome (moves S Actions) (idle (basic B)) (moves (super S B) Actions).

type    named_outcome   name -> state -> outcome -> outcome -> o.

named_outcome Name _ (idle S) (idle (named Name S)).
named_outcome Name _ (moves S Actions) (moves (named Name S) Actions).
```

Fig. 3. Axioms for "outcome" predicates.

```
step_sequence Current_state nil Current_state nil.
step_sequence
    Current_state (Events::Rest_script) Final_state Trace :-
  current_statenames Current_state Names,
  microstep Current_state Events Names Outcome,
  after_microstep Outcome Rest_script Final_state Trace.

after_microstep (idle S) Rest_script Final_state (nil::Trace) :-
  step_sequence S Rest_script Final_state Trace.
after_microstep (moves S Actions) nil S (Actions::nil).
after_microstep
    (moves S Actions) (Events::Rest_script)
    Final_state (Actions::Rest_trace) :-
  append Events Actions Combined,
  step_sequence S (Combined::Rest_script) Final_state Rest_trace.
```

Fig. 4. The axioms for the Statemate synchronous style semantics.

```
step_sequence Current_state nil Current_state nil.
step_sequence
    Current_state (Events::Rest_script)
    Final_state (Actions::Trace) :-
  step Current_state Events Outcome,
  outcome_state Outcome Next_state Actions,
  step_sequence Next_state Rest_script Final_state Trace.

outcome_state (idle S) S nil.
outcome_state (moves S Actions) S Actions.

step Current_state Events Outcome :-
  current_statenames Current_state Names,
  microstep Current_state Events Names Interim_outcome,
  after_microstep Interim_outcome Outcome.

after_microstep (idle S) (idle S).
after_microstep (moves S Actions) Outcome :-
  current_statenames S Names,
  microstep S Actions Names Interim_outcome,
  after_move Interim_outcome Actions Outcome.

after_move (idle S) All_actions (moves S All_actions).
after_move (moves S Actions) Actions_sofar Outcome :-
  current_statenames S Names,
  microstep S Actions Names Interim_outcome,
  append Actions_sofar Actions All_actions,
  after_move Interim_outcome All_actions Outcome.
```

Fig. 5. The axioms for the Statemate asynchronous style semantics.

```
step Current_state Events Outcome :-
  current_statenames Current_state Names,
  tmicrostep Current_state Events Names Tagged_outcome,
  til_fixpoint Tagged_outcome Events Names Outcome.
step Current_state Events (idle Current_state) :-
  current_statenames Current_state Names,
  tmicrostep Current_state Events Names Tagged_outcome,
  not (exists_fixpoint Tagged_outcome Events Names).

exists_fixpoint Tagged_outcome Events Names :-
  til_fixpoint Tagged_outcome Events Names Outcome.

til_fixpoint (tidle S) _ _ (idle S).
til_fixpoint (tmoves NewS Actions _ _) Events _ (moves NewS Actions) :-
  subset Actions Events.
til_fixpoint (tmoves NewS Actions _ TaggedS) Events Names Outcome :-
  not (subset Actions Events),
  memb Added_event Actions,
  not (memb Added_event Events),
  Next_events = (Added_event::Events),
  tmicrostep TaggedS Next_events Names Tagged_outcome,
  til_fixpoint Tagged_outcome Next_events Names Outcome.
```

Fig. 6. The axioms for the Pnueli-Shalev style semantics.