What is in a Step

A. Pnueli, M. Shalev

Dept. of Applied Mathematics & Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel

May 1988

This paper presents a proposal for the definition of a step in the execution of a statechart. The proposed semantics maintains the synchrony hypothesis, by which the system is infinitely faster than its environment, and can always finish computing its response before the next stimulus arrives. However, it corrects some inconsistencies present in previous definitions, by requiring global consistency of the step.

The research reported here has been partially supported by ESPRIT project 937 (DESCARTES) granted to AdCad Israel. The research of the second author was done as part of her M.Sc. thesis at the Weizmann Institute.

1. Introduction

The language of Statecharts has been proposed by D. Harel ([H]) as a visual language for the specification and modeling of *reactive systems*. While the (graphical) syntax of the language has been firmed up quite early, the definition of its formal semantics proved to be more difficult than originally expected. These difficulties may be explained as resulting from several requirements that seem to be desirable in a specification language for reactive systems, but yet may conflict with one another in some interpretations. Below, we list and shortly discuss each of these basic requirements.

To illustrate the discussed points we will use a restricted subset of the statechart syntax. The basic reaction of the system to external stimuli (events), are performed by *transitions*. Transitions in the system are graphically represented by arrows connecting one state to another, and are labeled by a *label* which typically has the form e/a. In such a label, the *event* e *triggers* (enables) the transition, i.e., allows it to be taken. The optional *action* a is performed when the transition is actually taken. Typically, the action has the form $f!$, which means that it *generates* the event f, or may be an assignment, assigning a value to a variable. Two transitions may be *parallel* to one another (also referred to as *orthogonal*), which means that they can be performed in the same step. Alternately, two transitions may be *conflicting*, e.g., if they depart from the same state, and then, at most one of them can be taken at any given step.

Synchrony Hypothesis

One of the main requirements one may wish to associate with a specification language for reactive systems is the *synchrony hypothesis*. This hypothesis assumes that the system is infinitely faster than the environment, and hence the response to an external stimulus is always generated in the same step that the stimulus is introduced. We may view this hypothesis as stating that the response is always simultaneous with the stimulus. We remind the reader that a long sequence of internal communications may be required to generate the outgoing response.

For example, we may have a set of parallel transitions with the following labels:

$$t_0 : a/e_1! \quad t_1 : e_1/e_2! \quad \ldots \quad t_n : e_n/b!$$

An incoming stimulus represented by the event a causes the transition t_0 to be activated and generate the event e_1. In turn, the transition t_1 responds to e_1 by generating e_2. This chain reaction continues until the transition t_n responds to e_n by generating b, which may be the final response of the system. According to the synchrony hypothesis b (and $e_1, \ldots e_n$) are all generated in the same step in which

the event a is presented to the system.

The synchrony hypothesis is an abstraction that limits the interference that may occur in the time period separating the stimulus and the response, and hence provides a guaranteed response as a primitive construct. In later stages of the development of the system, a more realistic modeling of the actual implementation can be done by introducing explicit delay elements if necessary.

Yet, Retaining Causality

In spite of the simultaneity abstraction, we should retain the distinction between cause and effect.

Consider for example the case that no external stimulus is given, and the system has two ready parallel transitions, with the following labels

$$t_1 : a/b! \quad t_2 : b/a!$$

In principle one may consider a semantics in which both transitions are taken while generating the events a and b. The justification for taking the transition t_1 with the trigger a is that a is generated by the transition t_2 at the same step. Similarly, the generation of b by t_1 justifies taking t_2.

This is a situation we like to exclude. The principle of causality requires that there is a clear causal ordering among the transitions taken in a step, such that no transition t relies for its activation on events generated by transitions appearing later than t in that ordering.

Expressing Priorities

An important feature of specification formalisms for real-time and reactive systems is the ability to assign *priorities* to responses.

Assume, for example, a system with two conflicting transitions with labels

$$t_1 : a \quad t_2 : b$$

We may consider t_1 to be a response to the event a while t_2 is a response to the event b. In the (probably infrequent) case that both a and b occur in the same step, the response is chosen non-deterministically. In many cases we may want to stipulate that, in the case both a and b occur, the response to b should have the higher priority. In Statecharts, this is expressed by using the *negation* of events. The general syntax of labels allows a *triggering expression* which is a boolean expression over events. The labels

$$t_1 : a \wedge \neg b \quad t_2 : b$$

ensure that if a or b occur exclusively, then as before, t_1 or t_2 are taken, respectively. However, if a and b occur together in the same step, then only t_2 is taken.

The three requirements listed above, i.e., *synchrony, causality*, and *priorities*, led to the semantics defined in [HPSS].

The basic approach presented in [HPSS] is that the behavior of a statechart is described as a sequence of *steps*, each step leading from one stable configuration to the next. The environment may introduce new external events at the beginning of each step. The response of the system to these input events is built up of a sequence of *micro-steps*. The first micro-step consists of all the transitions that are triggered by the input events. Subsequent micro-steps consist of all the transitions triggered by the set of events containing the input events, as well as all the events generated by previous micro-steps. Since there are only finitely many transitions that can be taken in a step, the sequence of micro-steps always terminates when there are no additional enabled transitions. This concludes a single step, and the set of events generated in any of the micro-steps is defined as the events generated during this step. It is not difficult to see that all the three requirements of *synchrony, causality* and expressing *priorities* via negations of events, are satisfied by the [HPSS] semantics. It also has the distinct advantage of being computationally feasible, which implies existence of an efficient implementation.

Unfortunately, the approach presented in [HPSS] has several deficiencies. The main ones are that it is highly operational, strongly depends on the ordering between micro-steps, and does not possess the property of *global consistency*. To illustrate the last point, consider two parallel transitions with the labels

$$t_1 : \neg a / b! \quad t_2 : b / a!$$

The semantics of [HPSS] constructs for this case the step $\{t_1, t_2\}$, even when there are no input events. This step generates the events $\{a, b\}$. We may complain that this step is not globally consistent, because it includes both the event a and the transition t_1, whose triggering condition requires that a is not generated during the current step. We refer to this phenomenon as *global* inconsistency, since the sequence of micro-steps, consisting of $\{t_1\}$ first, followed by $\{t_2\}$, is *locally* consistent. It is justified to take t_1 in the first micro-step and t_2 in the second micro-step, since they are both enabled at these points. It is only when we sum the effect of the complete sequence, that the inconsistency is discovered.

Well defined programming and specification languages usually possess two types of semantics. An operational semantics defines the behavior of a program or a specification in terms of a sequence of simple and atomic *operations*. It usually provides important guidelines for the implementor of the execution engine of such programs or specifications (compiler or interpreter). The other type of semantics is a *declarative* one, which bases the definition of the meaning of a program

on some kind of equational theory (using fixpoints for iteration and recursion), and attempts to ignore operational details such as order of execution, etc. We have intentionally avoided using the term *denotational* semantics which implies *compositionality* in addition to declarativity.

The declarative semantics, being based on simpler mathematical principles, is the one that underlies formal reasoning about programs and specifications, such as comparing two programs for equivalence or inclusion.

A proven sign of healthy and robust understanding of the meaning of a programming or a specification language is the posession of both an operational and declarative semantics, which are consistent with one another. Based on this criterion, our first attempt was to define a declarative semantics consistent with the operational semantics of [HPSS]. We soon found out that one of the main requisites for a declarative semantics is global consistency which is absent from [HPSS].

The present paper attempts to achieve the goal of assigning mutually consistent operational and declarative semantics to the specification language of Statecharts. For that purpose we had to impose some restrictions on the syntax of statecharts, and somewhat modify the operational approach presented in [HPSS].

As seen below, the declarative semantics is based on fixpoints as is often the case. However, in the case considered here, the situation is more complex because, due to the presence of negations, the basic operator is in general non-monotonic.

Anticipating the formal development below, the basic fixpoint equation associated with a step is

$$T = En(T)$$

In this equation, T is a set of transitions which are candidates for being taken together in a step. The function $En(T)$ yields the transitions which are enabled by the events generated by the set of transitions T. We can translate each of the requirements listed above into a condition on the solution T_a which is an acceptable set of transitions that can be jointly taken in a step.

- **Synchrony Hypothesis.** This requirement can be represented by the condition

  $$En(T_a) \subseteq T_a,$$

  which states that all the transitions enabled by the events generated by T_a are already in T_a. This implies that T_a is *maximal* in the sense that no additional transitions can be taken in this step.

- **Causality.** This is represented by the requirement of *inseparability* expressed by the requirement that there exists no $T \subset T_a$, such that

  $$En(T) \cap (T - T_a) = \phi$$

This means that if we try to stop at any subset T which is strictly contained in T_a, there is always an additional enabled transition in $T - T_a$ that can be added to T. In the usual case of monotonic operators this corresponds to *minimality*.

- **Expressing Priorities.** This is provided by allowing negations of events in the triggering expressions.
- **Global Consistency.** This is represented by the condition

$$T_a \subseteq En(T_a),$$

which states that each transition in T_a is enabled on the *complete* set of transitions T_a.

The main concerns considered in this paper are not unique to Statecharts, and have to be faced by any language intended for the specification of reactive systems, such as ESTEREL (see [BC], [BG]) and LUSTRE (see [BCH]). These two languages have also adopted the principles of *synchrony, causality* and *global consistency*. However, they avoid the complex interplay between non-determinism and priorities, present in Statecharts, by ruling out any program giving rise to these problems as illegal.

We should realize that this paper considers only the *micro-semantics* of statecharts, by defining the fine structure of a single step. Single steps can be combined into an operational semantics, following the treatment of [HPSS]. We refer the reader to [HGR], where a denotational semantics for some versions of the operational semantics is considered, and to [HG] for a comparative discussion of the different factors determining the semantics of a reactive language.

2. Syntax

In this section we present the syntax and semantics of statecharts. In addition, we define some notations that are used in the following sections. The syntax presented here is based on the syntax that was introduced in [HPSS] with necessary modifications due to the special approach we adopted here to the semantics of statecharts. The modifications provide further restrictions on the structure of the event expressions, so that the concavity property, introduced in a following section, will hold.

A Statechart is a structure

$$SC = (S, r, \rho, \theta, \delta, V, \Pi, T)$$

where

- S is a set of *states*, r is the *root* state, and ρ, θ and δ are functions which describe some relations between the states.
- V is a set of *variables*.
- Π is a set of *primitive events*.
- T is a set of *transitions*.

The detailed definitions of the structure of states and of the structure of the transitions follow.

States and Their Structure

The set of states S represents both basic states and composite-states which contain other states as substates. The *hierarchy* function ρ, the *type* function θ and the *default* function δ, represent the ancestry relations among states as follows.

The *hierarchy function* $\rho : S \rightarrow 2^S$ defines the direct descendants (substates) of each state. If $\rho(x) = \rho(y)$ then it is required that $x = y$. There exists a unique state $r \in S$ such that $\forall s \in S$, $r \notin \rho(s)$. This state r is the *root* of the statechart. A state s is called *basic* if $\rho(s) = \phi$. Otherwise it is called *composite*. We define ρ^*, ρ^+, the transitive closures of ρ, by:

$$\rho^* = \bigcup_{i \geq 0} \rho^i(s), \quad \rho^+ = \bigcup_{i \geq 1} \rho^i(s).$$

The *type function* $\theta : S \rightarrow \{\text{AND}, \text{XOR}\}$ is a partial function that assigns to each state its type, and identifies it as either an *or*-state or an *and*-state. If $\rho(s) \neq \phi$ and $\theta(s) = \text{XOR}$ then $\rho(s)$ is a *xor* decomposition of s, i.e., when the system is in the state s it is in one and only one of its immediate substates. If $\rho(s) \neq \phi$ and $\theta(s) = \text{AND}$ then $\rho(s)$ is an *and* decomposition of s, i.e., when the system is in the state s it is simultaneously in all of its immediate substates.

The *default function* $\delta : S \rightarrow 2^S$ defines for a composite state s a set of states which are contained in s. If $x \in \delta(s)$, then $x \in \rho^+(s)$, and $\delta(s)$ is the *default set* for s. The intended meaning of the default set $\delta(s)$, is that if some transition names s as its target, then on taking this transition we enter the default states $\delta(s)$, as well as s itself. The typical situation is that s is an *or*-state and $\delta(s)$ is a singleton. In the case that $|\delta(s)| > 1$, a non-deterministic choice is implied. For simplicity, we assume that $|\delta(s)| = 1$ for each *or*-state s.

In this paper we do not consider *history symbols* H which are discussed in [HPSS]. This detail is not essential for the understanding of the approach we present, and using the functions that were defined in [HPSS] we need only slight modifications to add history to the syntax and the semantics considered here.

Terms

The set of terms $\mathcal{T}$ is defined by

1. If $n \in N$ is a numeral, then $n \in \mathcal{T}$.
2. If $v \in V$ is a variable, then both v, $new\ (v) \in \mathcal{T}$.
3. If op is a k-ary operation, and $t_1, \ldots, t_k \in \mathcal{T}$, then $op\ (t_1, \ldots, t_k) \in \mathcal{T}$.

In general we allow an arbitrary number of data domains over which we allow constants, typed variables and terms. For simplicity we consider here only the data domains of the integers, and of the booleans, which are considered next.

The notation $new\ (v)$ refers to the newly assigned value in the current step. Thus, the test $new\ (x) = x + 1$ checks whether the value of x at the end of the step is greater by one than its value in the beginning of the step.

Boolean Terms

The set of boolean terms B is defined by

1. $true$, $false$ $\in B$.
2. If $s \in S$ is a state, then $in\ (s) \in B$.
3. If $t_1, t_2 \in \tau$, then $(tRt_2) \in B$ for each $R \in \{=, >, <, \neq, \leq, \geq\}$.
4. If $b, b_1, b_2 \in B$ are boolean terms, then $\neg b$, $b_1 \vee b_2$, $b_1 \wedge b_2 \in B$.

Boolean terms are expressions that should evaluate to truth values. The expression $in\ (s)$ is true if currently state s is active, i.e., the system is in state s. A boolean term which does not contain a subterm of the form $new\ (v)$ is called an *old boolean term.*

Event Expressions

Event expressions are similar to boolean terms, in testing whether certain conditions hold in the current configuration, and yielding a boolean value as a result. However, while boolean terms are restricted to the examination of variables (either in their old or in their new version), event expressions can also test for the presence or absence of events.

We define the sets of *positive event expressions* E^+ and *negative event expressions* E^-. These two sets are then combined to form the set of *event expressions* E.

Positive Event Expressions E^+

1. The null event, $\lambda \in E^+$.

2. The primitive events $\Pi \subseteq E^+$.
3. If e_1, e_2 are positive event expressions, then so are $e_1 \wedge e_2$, and $e_1 \vee e_2$.
4. If $s \in S$ is a state, then *entered*(s), *exited*$(s) \in E^+$.
5. If $v \in V$ is a variable, then *assigned*$(v) \in E^+$.
6. If $e \in E^-$ is a negative event expression, then $\neg e \in E^+$.

The intended meaning of the positive event expressions, are that these expressions only become "more true" as we add more events to the set of events present in the current step. This means that they can only change from $\perp$ (undefined) to **T**, or from **F** to **T**, but never from **T** to **F**. The expression $e \in \Pi$ tests for the presence of the event e in the current step. The particular case of λ, tests for the presence of the null event in the current step. By definition, the test for the null event is always true.

By clause 3, any *positive* boolean combination of positive event expressions is also positive. The special events *entered*(s) and *exited*(s) are considered to occur as soon as a transition, which respectively enters or exits the state s, is taken.

The event *assigned*(v) is caused by the execution of an action which assigns a value to the variable v. This action also causes the term *new* (v) to be defined.

Negative Event Expressions

1. If $e \in E^+$ is a positive event expression, then $\neg e \in E^-$.
2. If $e_1, e_2 \in E^-$ are negative event expressions, then so are $e_1 \wedge e_2$, and $e_1 \vee e_2$.

Negative event expressions are intended to capture those expressions that can become "less true" as more events are generated. They can only change from $\perp$ to **F** or from **T** to **F**.

The expression $\neg e$ tests for the absence of the event e in the current step. Any positive boolean combination of negative events yields a negative event.

Event Expressions

1. $E^+ \cup E^- \subseteq E$.
2. If $e_1, e_2 \in E$ are event expressions then so are $e_1 \wedge e_2$, and $e_1[c]$, where c is a boolean term.

The event $e[c]$ is caused whenever e happens while the condition c is true. It can be viewed as the conjunction $e \wedge c$, requiring both e and c to hold.

Note that the class of event expressions is closed under conjunction but not under disjunction or negation. This shows that an event expression is either a

positive expression, or a negative expression, or a conjunction of a positive and a negative expression. Thus, $e_1 \wedge (\neg e_2)$ is an admissible event expression, but $e_1 \vee (\neg e_2)$ is not.

Actions

The set of actions A is defined inductively as follows:

1. The *null* action. $\epsilon \in A$.
2. If $e \in \Pi$ is a primitive event, then $e!$ is an action.
3. If $u \in V$ is a variable and t is an old term of compatible type, then $u := t$ is an action, to which we refer as an *assignment*.
4. If a_1, a_2 are actions then so is (a_1, a_2), provided a_1 and a_2 do not contain assignments to the same variable.

The actions are the instantaneous responses of the system to the external stimuli in addition to the internal change of state. They include generation of events (2), assignment to variables (3). Actions can be combined into sets of actions (4).

3. Orthogonal Sets and Configurations

We introduce here some notations and definitions from [HPSS] that are used in the following.

States

- We define *Basic* $\subseteq S$ to be the set of basic states, i.e., states s such that $\rho(s) = \phi$.
- For a set of states X, the *Lowest Common Ancestor* of X, denoted by $lca(X)$ is defined to be the state x such that
  (a) $X \subseteq \rho^*(x)$.
  (b) $\forall s \in S, \quad X \subseteq \rho^*(s) \Rightarrow x \in \rho^*(s)$
- For a set of states X, the *strict Lowest Common* OR*-Ancestor* of X, denoted by $lca^+(X)$ is defined to be the state x satisfying
  (a) $X \subseteq \rho^+(x)$.
  (b) $\theta(x) = \text{OR}$
  (c) $\forall s \in S \ \ [\theta(s) = \text{OR} \ , \ X \subseteq \rho^+(s)] \Rightarrow x \in \rho^+(s)$

- Two states x, y are *orthogonal*, denoted by $x \perp y$, if either $x = y$ or their *lca* is an AND state, that is, $\theta(lca(\{x, y\})) = \text{AND}$.
- A set of states X, is an *orthogonal set* if for every $x, y \in X$, $x \perp y$. Note that the singleton set $\{x\}$ is always an *orthogonal* set. In the more general case, not considered here, that $|\delta(s)| > 1$, it is required that $\delta(s)$ be an orthogonal set.
- A set X is *an orthogonal set relative to* $s \in S$ if:
  (a) $X \subseteq \rho^*(s)$.
  (b) X is an orthogonal set.
- A set X is *a maximal orthogonal set relative to* $s \in S$ if:
  (a) X is an orthogonal set relative to s.
  (b) $\forall y \in \rho^*(s), y \notin X \Rightarrow X \cup \{y\}$ is not orthogonal.

  Note that $\forall s \in S$, $\{s\}$ is a maximal orthogonal set relative to s.

Upwards and Downwards Closures

Given a set of states X, we define the set X' to be the *upwards closure* of X if it is the *smallest* set satisfying the following conditions:

- $X \subseteq X'$
- The ancestor of any state in X' is also in X', i.e., if $\rho(s) \cap X' \neq \phi$, then $s \in X'$.

We denote the upwards closure of a state set X by $up(X)$. A set X is defined to be *upwards closed* if $X = up(X)$.

The *downwards closure* of a set X is defined to be the *smallest* set X' satisfying the conditions:

- $X \subseteq X'$
- For each composite *and*-state $s \in X'$, $\rho(s) \subseteq X'$, i.e., X' contains *all* the descendants of s.
- For each composite *or*-state $s \in X'$, $\delta(s) \in X'$, i.e., X' contains the default descendant of s.

We denote the downwards closure of a state set X by $down(X)$. A set S is defined to be *downwards closed* if $X = down(X)$.

Labels

The set of labels L is the set of pairs $E \times A$. For $l = (e, a)$ we write e/a. Informally, if e/a is a label of a transition t, then t is triggered by e and a is executed when t is taken.

Transitions

The set of transitions T given by a set of triples $2^S \times L \times 2^S$. A transition $t = (X, l, Y)$ consists of a *source set* X, a *target set* Y and a label l, where X and Y are orthogonal sets of states. Informally, if $l = e/a$, the system is in X and e occurs, then t is enabled and can be taken. If t is taken then a is executed and the system is then at Y. We denote the sets X and Y by $source(t)$ and $target(t)$ respectively.

Configurations

- A *state configuration* of $s \in S$ is an orthogonal set relative to s, all of whose members are basic states.
- A *maximal state configuration* X of $s \in S$ is a maximal orthogonal set, relative to s, all of whose members are basic states. In the case s is the root r, we refer to X simply as a maximal state configuration.
- A *store* St is a mapping from all the variables to values. The mapping can be partial and then we say that the value of the variable is $\perp$ interpreted as being undefined.

  Let Nat be the natural numbers domain and let $Bool$ be the boolean domain. A *partial store* is a function $St : V \rightarrow (Nat + Bool)_\perp$. A *total store* is $St : V \rightarrow Nat + Bool$.
- A *system configuration* $C = (S, St)$ consists of S, a maximal state configuration, and a total store St.
- Without loss of generality, we may assume that the target set $target(t)$ of each transition t consists of (orthogonal) basic states. In case it is not so originally, we can always replace Y by $down(Y) \cap Basic$, which will obey the simplifying restriction without changing the bahavior of the statechart.
- The *initial state configuration* X_0, is defined to be $down(\{r\}) \cap Basic$ where r is the root state.

Relations Between Transitions

- For a transition $t \in T$, we define the *arena* of t to be the lca^+ of the source and target of t. That is if $t = (X, l, Y)$ then $arena(t) = lca^+(X \cup Y)$.

Graphically, the arena of t is the lowest OR-state which fully contains the arrow representing t.

- Two different transitions $t_1 = (X_1, e_1/a_1, Y_1)$ and $t_2 = (X_2, e_2/a_2, Y_2)$ are said to be *structurally consistent* if:

  (a) $arena(t_1) \perp arena(t_2)$.

  (b) The set $a_1 \cup a_2$ contains at most one assignment to each variable $v \in V$.

  Otherwise, t_1 and t_2 are said to be *structurally conflicting.*

  Note that every transition is structurally consistent with itself.

- A set T of transitions is said to be a *structurally consistent set* if $\forall t_1, t_2 \in T$, t_1 and t_2 are structurally consistent.

- A transition $t = (X, e/a, Y)$ is *structurally relevant* to a state configuration S, if $\forall x \in X \ \rho^*(x) \cap S \neq \phi$. An equivalent statement of the same fact is $X \subseteq up(S)$.

4. The Evaluation Function m

We introduce here the evaluation function of terms, boolean terms, and event expressions.

Additional Notations

Let A be a consistent set of assignments. We consider here a more general case than the one allowed in the restricted statechart syntax. The restricted syntax requires that the term t appearing in the assignment $x := t$ is an *old* term, i.e., contains no subterms of the form $new(y)$. The definition below covers the case that t may contain new subterms.

We can associate with A a set of equations over $V \cup V'$ (where $V' = \{x' \mid x \in V\}$), denoted by $Eq(A)$, by generating for each assignment $a: \ x := t$ an equation $eq(a): \ x' := t'$. The right hand side t' of this equation, is obtained from t by replacing each occurrence of $new(y)$ by y'.

Let $St: V \rightarrow nat + Bool$ be a total store, and $St': V' \rightarrow (Nat + Bool)_\perp$, a partial store. We say that the pair (St, St') *satisfies* the set of assignments A, if the equations Eq (A) are satisfied over the interpretation (St, St'), i.e.,

$$(St, St') \models \text{Eq (A)}.$$

This means that when we evaluate both sides of each equation $x' = t'$, using the values provided by (St, St'), they evaluate to equal values, possibly $\perp$.

We say that a set of assignments A is *consistent* over a store St, if there exists some partial store St', such that (St, St') satisfies A. For the restricted case that all terms are old, we have that $t' = t$, and hence a sufficient condition for consistency is that, for each variable x, A contains at most one assignment assigning values to x.

Given a total store $St : V \to Nat+Bool$, and a set of assignments A consistent over St, we define $new\_store(St, A)$ to be the store $St' : V' \to (Nat+Bool)_\perp$ which is the *minimal* store of that type, such that (St, St') satisfies A.

Example : Let

$$A = \{x := 5 + y,\ y := new(u) + 1,\ u := new(y) - 1\}$$

then,

$$Eq(A) = \{x' := 5 + y,\ y' := u' + 1,\ u' := y' - 1\}$$

consider a store St such that $St[y] = 0$. There are many solutions to $(St, St') \models Eq(A)$. For example, both $St'_0 : \{x' : 5,\ y' : 1,\ u' : 0\}$ and $St'_1 : \{x' : 5,\ y' : 2,\ u' : 1\}$ are solutions. However, there exists only one minimal solution, which for the above case is:

$$new\_store(St, A) = St'_{min} : \{x' : 5,\ y' : \perp,\ u' : \perp\}.$$

- Let T be a set of transitions. We denote by $assgn(T)$ the set of all assignments appearing in the action part of the transitions in T. We denote by $Ev(T)$ the set of all events *generated* by actions of the transitions in T of the form $e!$

We now define the evaluation function m corresponding to $K = (C, T) = (S, St, T)$, where C is a system configuration and T is a set of transitions. We refer to K as an *extended configuration*. An extended configuration represents an intermediate situation, where we have already decided to take the transitions in T, starting from $C = (S, St)$. The set T defines the set $Ev(T)$ of events generated by T, and a partial new store, generated by the assignments in $assgn(T)$.

Evaluation of Terms

$m_\tau : \mathcal{T} \to K \to Nat_\perp$

- For a numeral $n \in N$,

$$m[\![n]\!](K) = nat(n)$$

i.e., the arithmetical value denoted by the numeral.

- For a variable $v \in V$,

$$m[\![v]\!](K) = St[v].$$

- For a variable $v \in V$,

$$m[\![new(v)]\!](K) = St'[v'],$$

where $St' = new\_store(St, assgn(T))$. Note that $m[\![new(v)]\!](K)$ may yield $\perp$.

- If t is a term and op is a unary algebraic operation, then

$$m[\![op(t)]\!](K) = \mathbf{op}(m[\![t]\!])(K),$$

where **op** is the semantic operation corresponding to op.

- If t_1, t_2 are terms and op is a binary algebraic operation, then

$$m[\![op(t_1, t_2)]\!](K) = \mathbf{op}(m[\![t_1]\!](K), m[\![t_2]\!](K)),$$

where **op** is the semantic operation corresponding to op.

All the arithmetic operations are assumed to be *strict* with respect to their arguments.

Evaluation of Boolean Terms

$m_B : B \to K \to Bool_\perp$

- $m[\![true]\!] = \mathbf{T}$, $m[\![false]\!] = \mathbf{F}$.
- If $s \in S$ is a state, then

$$m[\![in(s)]\!](K) = if\ \rho^*(s) \cap S \neq \phi\ then\ \mathbf{T}\ else\ \mathbf{F}.$$

- If $t_1, t_2 \in \tau$ are terms and $R \in \{=, <, >, \neq, \leq, \geq\}$ is a relation, then

$$m[\![(t_1\ R\ t_2)]\!](K) = if\ (m[\![t_1]\!](K)\ R\ m[\![t_2]\!](K))\ then\ \mathbf{T}\ else\ \mathbf{F}.$$

It is assumed that all relations R are strict.

- If $b \in B$ is a boolean terms, then

$$m[\![\neg b]\!](K) = \neg(m[\![b]\!](K)),$$

where for $b \in Bool_\perp$,

b	$\neg b$
T	**F**
F	**T**
$\perp$	$\perp$

- If $b_1, b_2 \in B$ are boolean terms, then

$$m[\![b_1 \wedge b_2]\!](K) = m[\![b_1]\!](K) \bigwedge m[\![b_2]\!](K)$$

$$m[\![b_1 \vee b_2]\!](K) = m[\![b_1]\!](K) \bigvee m[\![b_2]\!](K),$$

where for $b_1, b_2 \in Bool_\perp$

b_1	b_2	$b_1 \bigwedge b_2 = b_2 \bigwedge b_1$
–	**F**	**F**
T	$\perp$	$\perp$
T	**T**	**T**

b_1	b_2	$b_1 \bigvee b_2 = b_2 \bigvee b_1$
–	**T**	**T**
F	$\perp$	$\perp$
F	**F**	**F**

Note that the boolean operations of conjunction and disjunction are not strict.

Evaluation of Event Expressions

$m_E : E \to K \to Bool_\perp$

- $m[\![\lambda]\!](K) = \mathbf{T}$.
- If $s \in S$ is a state, then

$$m[\![entered(s)]\!](K) = \mathbf{T} \text{ } \textit{iff} \text{ for some } t \in T, s \in \rho^+(arena(t)), \text{ and } \rho^*(s) \cap target(t) \neq \phi$$

$$m[\![exited(s)]\!](K) = \mathbf{T} \text{ } \textit{iff} \text{ } \rho^*(s) \cap S \neq \phi, \text{ and for some } t \in T, s \in \rho^+(arena(t))$$

This definition says that a transition t generates the event *entered*(s) if s is a state (strictly) contained in the arena of t, and contains some states in the target of t. The transition t generates the event *exited*(s) if s is currently on (has some basic descendants in S), and is strictly contained in the arena of t. Thus if s is any state, and t a self-loop transition, connecting s to itself, i.e., $source(t) = target(t) = \{s\}$, then t generates both the events *entered*(s) and *exited*(s).

- If $x \in V$ is a variable, then

$$m[\![assigned(x)]\!](K) = \mathbf{T} \textit{ iff} \text{ there exists some assignment } [x := t] \in assgn(T).$$

- For $e \in \Pi$ a primitive event expression,

$$m[\![e]\!](K) = \mathbf{T} \textit{ iff } e \in Ev(T).$$

- For e, e_1 and e_2 event expressions,

$$m[\![e_1 \wedge e_2]\!](K) = m[\![e_1]\!](K) \bigwedge m[\![e_2]\!](K).$$

$$m[\![e_1 \vee e_2]\!](K) = m[\![e_1]\!](K) \bigvee m[\![e_2]\!](K).$$

$$m[\![\neg e]\!](K) = \neg m[\![e]\!](K).$$

- If c is a boolean term, then

$$m[\![e[c]]\!](K) = m[\![e]\!](K) \bigwedge m[\![c]\!](K).$$

where $\neg$, $\bigwedge$ and $\bigvee$ are the semantic boolean operators that were used in the evaluation of boolean terms.

Enabling A Transition

Let $K = (C, T) = (S, St, T)$, where $C = (S, St)$ is a system configuration and T is a set of transitions. Let $t = (X, e/a, Y)$ be a transition. We say that (C, T) *enables* t if:

(a) t is structurally relevant to S, i.e., $X \subseteq up(S)$.

(b) t is structurally consistent with every $t' \in T$.

(c) $m[\![e]\!](C, T) = \mathbf{T}$.

We define $En(C, T) = \{t \mid (C, T) \textit{ enables } t\}$.

5. The Concavity Property

In this section we formulate the notion of concavity and show that the restrictions imposed on the syntax of statecharts guarantee that all the transitions are concave. The concavity property is a property on transitions, which states that a transition can not be enabled, disabled, and then enabled again as a result of adding more transitions to the extended configuration. The concavity property holds because the syntax of event expressions is restricted in a way that, once an event expression e evaluates to $\mathbf{T}$ in some phase of the step construction, and then changes to a different value at a later phase ($\mathbf{F}$ or $\perp$), it can never regain the value of $\mathbf{T}$. Concavity is essential for the main result of this paper, which is the equivalence of the operational and declarative definitions of a step, and may be viewed as a weaker version of monotonicity.

5.1 Properties of Terms, Boolean Terms and Event Expressions

Lemma 1 : Given a term t and a set of transitions T, then if $m[\![t]\!](C,T) \neq \perp$ then $T \subseteq T' \Rightarrow m[\![t]\!](C,T')=m[\![t]\!](C,T)$ where m is the evaluation function defined above.

Proof : trivial - by structural induction.

This shows that terms are monotone with respect to the argument T.

Definition : We say that an event expression e (a boolean term c) *follows* a path $\pi = (b_1, b_2, ..., b_n)$ where $\forall i \in [1,n]$, $b_i \in Bool_\perp$, if there exist sets of transitions $T_1, T_2, \ldots, T_n$ such that $T_1 \subseteq T_2 \subseteq \ldots \subseteq T_n$, $m[\![e]\!](C,T_i)=b_i$ ($m[\![c]\!](C,T_i)=b_i$), for $i = 1, \ldots, n$. We will represent a set of paths by a regular expression over the set $\{\mathbf{T}, \mathbf{F}, \perp\}$.

Lemma 2 : A boolean term can only follow paths in $\perp^*\mathbf{T}^* + \perp^*\mathbf{F}^*$.

Proof : The proof is by induction on the structure of the boolean term.
Base case :

- *true* passes only through paths in $\mathbf{T}^*$.
- *false* passes only through paths in $\mathbf{F}^*$.
- $in(s)$, where s is a state, passes only through the paths $\mathbf{T}^* + \mathbf{F}^*$.

- If t_1, t_2 are terms, then $(t_1 R t_2)$ for $R \in \{=, >, <, \neq, \leq, \geq\}$ passes only through paths in $\perp^*\mathbf{T}^* + \perp^*\mathbf{F}^*$, because R is strict and by lemma 1.

Inductive step : We assume that b, b_1 and b_2 only follow paths in $\perp^*\mathbf{T}^* + \perp^*\mathbf{F}^*$. It is trivial to show that $\neg b$ also can follow only these paths. We show that the lemma holds also for $(b_1 \vee b_2), (b_1 \wedge b_2)$.
There are three subcases to consider :

1. b_1 follows $\perp^{i_1}\mathbf{T}^{j_1}$, b_2 follows $\perp^{i_2}\mathbf{T}^{j_2}$, and w.l.o.g. we assume that $i_1 \leq i_2$:
   - $(b_1 \wedge b_2)$ follows $\perp^{i_2}\mathbf{T}^{j_2}$.
   - $(b_1 \vee b_2)$ follows $\perp^{i_1}\mathbf{T}^{j_1}$.
2. b_1 follows $\perp^{i_1}\mathbf{T}^{j_1}$, and b_2 follows $\perp^{i_2}\mathbf{F}^{j_2}$. Then,
   - $(b_1 \wedge b_2)$ follows $\perp^{i_2}\mathbf{F}^{j_2}$.
   - $(b_1 \vee b_2)$ follows $\perp^{i_1}\mathbf{T}^{j_1}$.
3. b_1 follows $\perp^{i_1}\mathbf{F}^{j_1}$, and b_2 follows $\perp^{i_2}\mathbf{F}^{j_2}$, and w.l.o.g. we assume that $i_1 \leq i_2$:
   - $(b_1 \wedge b_2)$ follows $\perp^{i_1}\mathbf{F}^{j_1}$.
   - $(b_1 \vee b_2)$ follows $\perp^{i_2}\mathbf{F}^{j_2}$.

Lemma 3 : A positive event expression passes only through paths in $\mathbf{F}^*\mathbf{T}^*$, and a simple negative event expression passes only through paths in $\mathbf{T}^*\mathbf{F}^*$.

Proof : The proof is by induction on the structure of the positive and negative event expressions.
Base case :

- The positive event expression λ always follows the path $\mathbf{T}^*$.
- The primitive events $e \in \Pi$, and the events *entered*(s), *exited*(s), *assigned*(v), become true as soon as T_i contains a transition that generates them. Since the set T_i is monotonically increasing with i, these events obviously follow a path in $\mathbf{F}^*\mathbf{T}^*$.

Inductive step : We assume that the lemma holds for e, e_1 and e_2 and show that it holds also for $(e_1 \vee e_2), (e_1 \wedge e_2)$ and $\neg e$.

- The proof for $\neg e$ is trivial.

- In the case of positive event expressions, assume that e_1 passes through $\mathbf{F}^{i_1}\mathbf{T}^{j_1}$, e_2 passes through $\mathbf{F}^{i_2}\mathbf{T}^{j_2}$ and that w.l.o.g. $i_1 \leq i_2$. Then it follows that
  (a) $(e_1 \wedge e_2)$ passes through $\mathbf{F}^{i_2}\mathbf{T}^{j_2}$.
  (b) $(e_1 \vee e_2)$ passes through $\mathbf{F}^{i_1}\mathbf{T}^{j_1}$.
- In the case of negative event expressions, assume that e_1 passes through $\mathbf{T}^{i_1}\mathbf{F}^{j_1}$, e_2 passes through $\mathbf{T}^{i_2}\mathbf{F}^{j_2}$ and that w.l.o.g. $i_1 \leq i_2$. Then it follows that
  (a) $(e_1 \wedge e_2)$ passes through $\mathbf{T}^{i_1}\mathbf{F}^{j_1}$.
  (b) $(e_1 \vee e_2)$ passes through $\mathbf{T}^{i_2}\mathbf{F}^{j_2}$.

Lemma 4 : An event expression passes only through paths in $(\perp + \mathbf{F})^*\mathbf{T}^*\mathbf{F}^*$.

Proof : The proof is by induction on the structure of the event expression.
Base case : For the case of event expressions which are either positive or negative event expressions, the lemma follows from lemma 3 and the obvious inclusions

$$\mathbf{F}^*\mathbf{T}^* \subseteq (\perp + \mathbf{F})^*\mathbf{T}^*\mathbf{F}^*$$
$$\mathbf{T}^*\mathbf{F}^* \subseteq (\perp + \mathbf{F})^*\mathbf{T}^*\mathbf{F}^*$$

Inductive step : We assume that the lemma holds for e, e_1 and e_2 and show that it holds also for $(e_1 \wedge e_2)$ and $e[c]$, where c is a boolean term. By lemma 2 and because $\perp^*\mathbf{T}^* + \perp^*\mathbf{F}^* \subseteq (\perp + \mathbf{F})^*\mathbf{T}^*\mathbf{F}^*$, it is sufficient to show that the lemma holds for $(e_1 \wedge e_2)$.
Assume that e_1 passes through a path in $(\perp + \mathbf{F})^{i_1}\mathbf{T}^{j_1}\mathbf{F}^*$, e_2 passes through a path in $(\perp + \mathbf{F})^{i_2}\mathbf{T}^{j_2}\mathbf{F}^*$ and let $i_3 = max(i_1, i_2)$ and $j_3 = min(j_1 + i_1, j_2 + i_2)$. Then

- If $i_3 \leq j_3$ then $(e_1 \wedge e_2)$ passes through a path in $(\perp + \mathbf{F})^{i_3}\mathbf{T}^{j_3 - i_3}\mathbf{F}^*$.
- If $i_3 > j_3$ then $(e_1 \wedge e_2)$ passes through a path in $(\perp + \mathbf{F})^{j_3}\mathbf{F}^*$.

5.2 The Concavity Property

Theorem *(The Concavity Property)* : given a system configuration C, if T_1, T_2 and T_3 are three sets of transitions such that $T_1 \subseteq T_2 \subseteq T_3$, then there is no transition t such that $t \in En(C, T_1), t \notin En(C, T_2)$ and $t \in En(C, T_3)$.

Proof : Assume to the contrary that t is a transition such that $t \in En(C,T_1), t \notin En(C,T_2)$, and $t \in En(C,T_3)$. Let $t = (X,e/a,Y)$, and $C = (S,St)$. Then $t \notin En(C,T_2)$ only if one of the following holds

1. $\exists x \in X\ \rho^*(x) \cap S = \phi$. But then, we have a contradiction to the fact that $t \in En(C,T_1)$.
2. $\exists t' \in T_2$ such that t and t' are in a structural conflict. But then, because $t' \in T_3$ it cannot be that $t \in En(C,T_3)$.
3. $m[\![e]\!](C,T_2) \neq \mathbf{T}$. But then because $m[\![e]\!](C,T_1) = \mathbf{T}$ and $m[\![e]\!](C,T_3) = \mathbf{T}$, we have that e passes through $(\mathbf{T},\mathbf{F},\mathbf{T})$ or through $(\mathbf{T},\perp,\mathbf{T})$ which contradicts lemma 4.

6. Definition of a Step

In this section we introduce two definitions of the sets of transitions that are considered to be admissible steps from a given system configuration. The first definition is declarative and is based on particular solutions of a fixpoint equation. The second definition is constructive and suggests an algorithm for computing all the possible steps. We then show that under the property of concavity, which the syntax guarantees, the two definitions coincide.

Separable Sets

A set of transitions T is defined to be separable if there exists a subset $T' \subset T$, such that

$$En(C,T') \cap (T - T') = \phi.$$

A set T is called *inseparable* if it is not separable.

A Declarative Definition

A set of transitions T which is inseparable and satisfies the equation

$$T = En(C,T)$$

is called an *admissible step* of the system from configuration C.

We refer to such a set also as an *inseparable solution.*

The *Yield* of a Step

Let C be a configuration and T an admissible step from C. We define the configuration following the application of the step T to the configuration C, as follows:

$$\mathit{Next}\colon \mathit{Config} \times 2^{\mathit{Trans}} \rightarrow \mathit{Config}.$$

$$\mathit{Next}((S, St), T) = (S', St')$$

where

$$S' = \left(S - \bigcup_{t \in T} \rho^*(\mathit{arena}^+(t))\right) \cup \left(\bigcup_{t \in T} \mathit{target}(t)\right)$$

and

$$St' = \mathit{update}(St, \mathit{new\_store}(St, \mathit{assgn}(T))).$$

Given two stores St and St', the function $\mathit{update}\ (St, St')$ yields a store St'' defined by

$$St''[v] = \ \mathit{if}\ St'[v] = \perp\ \mathit{then}\ St[v]\ \mathit{else}\ St'[v]$$

A Constructive Definition

The definition presented above for admissible steps is declarative. An alternative definition can be given by a non-deterministic procedure that builds a step T by adding one transition at a time.

1. Initially $T = \phi$.
2. Compare $En(C, T)$ to T.
   (a) If $T = En(C, T)$ terminate and report success.
   (b) If $T \subset En(C, T)$, pick a transition $t \in En(C, T) - T$ and add it to T. Repeat step 2.
   (c) Otherwise, i.e., $T \not\subseteq En(C, T)$, report failure.

Proposition : A set of transitions T is an inseparable solution to the equation $T = En(C, T)$ iff it can be constructed by the above procedure.

Proof : Assume that T_0 is an inseparable solution to the equation $T_0 = En(C, T_0)$.

We apply the procedure as specified, with the modification that the only new transitions to be added to T are picked from $(En(C, T) - T) \cap T_0$. We show:

- The procedure cannot fail.
  The procedure can fail only by having $T \subset T_0$, $T \subset En(C,T)$, $t \in (En(C,T) - T) \cap T_0$ and $t' \in T \cup \{t\}$ such that $t' \in En(C,T)$ but $t' \notin En(C, T \cup \{t\})$.

Consider the different reasons of why t' has become disabled when adding t to T.

1. $source(t') \not\subseteq up(S)$, but C has not changed by adding t to T.
2. t' is in structural conflict with some transition t'' in $T \cup \{t\}$. Since $T \subset En(C,T)$ we cannot have both t' and t'' in T. Hence one of them must be the newly added transition t. But then t would not have been enabled on (C,T), contradicting $t \in En(C,T) - T$.
3. t' has the label e/a and while $m[\![e]\!](C,T) = \mathbf{T}$, $m[\![e]\!](C, T \cup \{t\}) \neq \mathbf{T}$. Since $t' \in T_0 = En(C,T_0)$, we have $m[\![e]\!](C,T_0) = \mathbf{T}$, where $T \cup \{t\} \subseteq T_0$. This contradicts the concavity property of event expressions, when applied to the sets $T \subseteq T \cup \{t\} \subseteq T_0$.

It follows that the procedure cannot fail.

- The procedure cannot stop at $T \subset T_0$.
  The procedure can stop at $T \subset T_0$, only if $En(C,T) \cap (T_0 - T) = \phi$, contradicting the fact that T_0 is inseparable.

Let T be a set obtained by the construction. We show that T is an inseparable solution. Let the transitions in T be ordered in a sequence $t_1, t_2, \ldots, t_n$, according to the order of their addition to T. Clearly, since the construction stopped at T, we have that T satisfies

$$T = En(C,T).$$

It only remains to show that T is inseparable.

Consider any $T' \subset T$. Let t_k be the first transition, in the above ordering, which belongs to $T - T'$. We claim that $t_k \in En(C,T')$ which leads to the fact that $En(C,T') \cap (T - T') \neq \phi$. Assume to the contrary, that $t_k \notin En(C,T')$. Then we have three sets

$$T_1 = \{t_1, \ldots, t_{k-1}\}, \qquad T_2 = T', \qquad T_3 = T,$$

such that $T_1 \subseteq T_2 \subseteq T_3$, and yet

1. $t_k \in En(C, \{t_1, \ldots, t_{k-1}\})$
2. $t_k \notin En(C,T')$
3. $t_k \in En(C,T)$.

Claim 1 follows from the fact that the constructive algorithm picks transitions

to be added only if they are enabled under the current approximation. Claim 2 is our contrary assumption. Claim 3 follows from the fact that $t_k \in T = En(C,T)$.

We thus obtain a contradiction to the concavity property.

We must conclude that $En(C,T') \cap (T - T') \neq \phi$ for any $T' \subset T$, and hence T is inseparable.

7. Extending the Syntax

The syntax of event expressions, with its careful construction out of the subclasses of positive and negative event expression, was specially designed to guarantee the property of concavity. Let us consider several possible extensions of the syntax and show that they lead to non-concave behaviors.

Examples : Assume that $e, f, g \in \Pi$ are primitive event expressions, $X, Y, X', Y' \subseteq S$ are pairwise orthogonal sets and $v \in V$ is a variable.

- Allowing disjunction of a positive and negative event expression.
  The expression $e \vee \neg f \notin E$. passes through the path $(\mathbf{T}, \mathbf{F}, \mathbf{T})$ in the case that $T_1 = \phi$, $T_2 = \{(X, g/f!, Y)\}$ and $T_3 = \{(X, g/f!, Y), (X', g/e!, Y')\}$.
- Allowing conditions in negative event expressions.
  The expression $(\neg e)[new(v) = 0] \vee \neg f \notin E$ passes through the path $(\mathbf{T}, \perp, \mathbf{T})$ in the case that $T_1 = \phi$, $T_2 = \{(X, g/f!, Y)\}$ and $T_3 = \{(X, g/f!, Y), (X', g/v := 0, Y')\}$.
- Allowing conditions in positive event expressions.
  The expression $\neg(e[new(v) < 0]) \notin E$ passes through the path $(\mathbf{T}, \perp, \mathbf{T})$ in the case that $T_1 = \phi$, $T_2 = \{(X, g/e!, Y)\}$ and $T_3 = \{(X, g/e!, Y), (X', g/v := 0, Y')\}$.

7.1 Extensions by Transition Duplication

It is still possible to accommodate more general event expressions. Let us define first the most general syntax for event expressions. We define the class of *extended event expressions* $\mathcal{E}$ by

- Every event expression is an extended event expression. That is, $E \subseteq \mathcal{E}$.
- If e, e_1 and e_2 are extended event expressions, then so are $\neg e$, $e_1 \vee e_2$, $e_1 \wedge e_2$ and $e[c]$, where c is a boolean term.

It is straightforward to see that the evaluation function m also defines evaluation of extended event expressions over extended configurations $K = (C, T)$.

Lemma 5 : Every extended event expression is equivalent to a disjunction of event expressions.

Proof : Given an extended event expression, we can bring it to a disjunctive normal form. In performing this transformation we have to manipulate expressions involving events as well as boolean terms. Some of the rules for manipulating such combinations are given by the following equivalences

$$\begin{aligned} \neg e[c] &= \neg e \vee \lambda[\neg c] \\ \lambda \wedge e &= e \wedge \lambda = e \\ (e_1 \wedge e_2)[c] &= e_1 \wedge (e_2[c]) \\ e_1[c_1] \wedge e_2[c_2] &= e_1 \wedge e_2[c_1 \wedge c_2] \\ (e_1 \vee e_2)[c] &= e_1[c] \vee e_2[c] \end{aligned}$$

Using these and additional rules, we can bring every extended event expression to the form

$$E_1 \vee E_2 \vee \ldots \vee E_n,$$

Each disjunct E_i in this presentation is an event expression of the form

$$(e_1 \wedge \ldots \wedge e_m \wedge \neg f_1 \wedge \ldots \neg f_k)[c],$$

where $e_1, \ldots, e_m, f_1, \ldots, f_k$ are either primitive events or simple events of the form *entered*(s), *exited*(s), *assigned*(v).

Consider a statechart whose author wanted to have a transition t consisting of $(X, e/a, Y)$, where e is an extended event expression. Let e have the presentation $E_1 \vee E_2 \vee \ldots \vee E_n$ as a disjunction of event expressions. Then we suggest to the author of the original statechart to replace the single transition t by the n transitions

$$t_1 : (X, E_1/a, Y) \quad t_2 : (X, E_2/a, Y) \ \ldots \ t_n : (X, E_n/a, Y).$$

In this representation, the labels are event expressions and the overall effect is the same. In fact, the author may still prefer to present a single transition labeled by e, which is *interpreted* as the set of n transitions as shown above.

Another extension of the syntax we may allow is the reference to new values of variables on the right hand side of assignment actions. Such a reference, of the form $new(v)$, is allowed, provided the trigger of that transition includes the conjunct $assigned(v)$. This conjunct ensures that the transition will be taken only if the term $new(v)$ is defined. Thus the following is an admissible label

$$e \wedge assigned(y) \wedge assigned(z)/x := x + \ new\ (y) + \ new\ (z)$$

Another extension is the inclusion of ***conditional actions*** with boolean terms containing new subterms. These are actions of the form

$$if\ c\ then\ a_1\ \ else\ a_2,$$

where c is a boolean term that may contain subterms of the form $new(v)$ and a_1, a_2, are actions. The interpretation of such conditional actions is again obtained by transition splitting. Thus, we interpret the following transition

$$t : (X, e/a \cup \{\ if\ c\ then\ a_1\ \ else\ a_2\}, Y),$$

as though it was presented by the two transitions

$t_1 : (X, e[c]/a \cup \{a_1\}, Y)$, and
$t_2 : (X, e[\neg c]/a \cup \{a_2\}, Y)$.

This interpretation transforms any transition with conditional actions into a set of transitions, all of whose actions are unconditional.

The previous version of the syntax, as presented in [HPSS], allowed event expressions of the form $tr[c]$, $fs[c]$, which are considered to occur when the boolean term c changes from false to true, or true to false, respectively. These can be expressed in terms of the events $assigned(v)$ and references to $new\ (v)$. For example, the event $tr[x = y]$, can be expressed as the disjunction

$$\begin{aligned} &\neg assigned(x) \wedge assigned(y)[(x \neq y) \wedge (x = \ new\ (y))] \vee \\ &\ assigned(x) \wedge \neg assigned(y)[x \neq y) \wedge (\ new\ (x) = y)] \vee \\ &\ assigned(x) \wedge assigned(y)[(x \neq y) \wedge (\ new\ (x) = \ new\ (y))] \end{aligned}$$

Obviously, this expression is not a standard event expression but rather an extended expression. Its interpretation calls for splitting the transition into three copies, each labeled by one of the disjuncts.

Acknowledgements

We wish to thank Rivi Sherman for many constructive comments and identification of bugs in previous versions of the manuscript. We gratefully acknowledge many enjoyable discussions with W.P. de Roever, R. Gerth, C. Huizing,

J. Hooman, R. Koymans and R. Kuiper, which significantly contributed to our understanding of the tradeoffs between the various requirements expected from a good semantics. We thank Carol Weintraub and Sarah Fliegelmann for typing the numerous versions of the manuscript.

References

[BC] G. Berry, L. Cosserat – The Synchronous Programming Language ESTEREL and its Mathematical Semantics, Proceeding of CMU Seminar on Concurrency, Springer Verlag, *LCNS* **197** (1985), 389–449.

[BCH] J.-L. Bergerand, P. Caspi, N. Halbwachs – Outline of Real-Time Dataflow Language, Proceeding of IEEE-CS Real-Time Symposium, San Diego (1985).

[BG] G. Berry, G. Gonthier – The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation, Technical Report, Ecole Nationale Superieure des Mines de Paris (1988).

[H] D. Harel – Statecharts: A Visual Approach to Complex Systems, *Science of Computer Programming* **8** (1987) 231–274.

[HG] C. Huizing, R. Gerth – On the Semantics of Reactive Systems, Eindhoven University of Technology (1988).

[HGR] C. Huizing, R. Gerth, W.P. De Roever – Modelling Statecharts in a Fully Abstract Way, Colloquium on Algebras of Trees and Programs, Springer-Verlag *LCNS* **299** (1988).

[HPSS] D. Harel, A. Pnueli, J.P. Schmidt, R. Sherman – On the Formal Semantics of Statecharts, Symposium on Logic in Computer Science, (1987) 54–64.

[S] M. Shalev – On the Operational Semantics of Statecharts, M.Sc. thesis, Weizmann Institute (1988).