Timed and Hybrid Statecharts and their Textual Representation *

Y. Kesten A. Pnueli
Weizmann Institute of Science †

Abstract.

A structured operational semantics is presented for *Timed* and *Hybrid Statecharts*, which are generalizations of the visual specification language of Statecharts intended to model real-time and hybrid systems. In order to study some of the basic features of Statecharts and the extensions necessary to treat real-time and continuous behaviors without being distracted by the graphical representation, we introduce a concurrent real-time language that can be viewed as a textual representation of Statecharts.

The language contains statements for delays, preemption, and timeouts. A structured operational semantics of the language and an illustrative example of its use for specification are presented. Extensions to the specification of hybrid systems are obtained by allowing a differential equation as a statement of the extended language. Structured operational semantics is also given for the hybrid version.

The same extensions are then applied to the visual Statechart language, and similar compositional semantics are defined.

Keywords: Real-time, timed transitions system, hybrid systems, Statecharts, structured operational semantics.

Contents

*This research was supported in part by the France-Israel project for cooperation in Computer Science, and by the European Community ESPRIT Basic Research Action Project 3096 (SPEC).

†Department of Applied Mathematics, Weizmann Institute, Rehovot, Israel

1 Introduction

The visual formalism of *Statecharts* has been proposed by Harel in [Har87]. Since then it has been used for the specification of complex reactive systems ([HLN+90], [HP85]), and extensively studied from semantic ([HPSS86],[HGdR88], [PS89], [HK89], [Mar91], [Hui91]), and expressive complexity ([DH89], [Har89], [HH90]) points of view.

Two interesting recent developments in the formal specification and analysis of reactive systems are extensions of the methodology to deal with real time and continuously varying elements. The extension to real time adds to the model of reactive systems the element of metric time. This addition has an effect on the set of behaviors a reactive system may generate, as well as on the specification of requirements and properties of a system.

The association of time durations with the basic actions of a system imposes new constraints on the possible executions of the system. As a result, the number of possible computations becomes smaller, and several behaviors, for example computations in which the activation of an enabled action can be arbitrarily delayed, are no longer possible. Consequently, a timed reactive system usually has fewer computations and therefore more valid properties than the corresponding untimed version of the same system. This suggests that more powerful methods of analysis and verification are necessary to deal with the additional properties that are caused by the time dependent restrictions.

Naturally, with the introduction of time, many new properties can now be stated and eventually verified. In many cases, we can take important properties of untimed reactive systems and form their timed version. A case in point is the property of responsiveness, which states that every p should eventually be followed by a q. Within the timed framework, we can be more exacting and instead of requiring that q *eventually* happens, we can demand that q follows p within no more than 5 time units.

Many suggestions for appropriate extensions of the reactive framework to deal with the extension to real time have been made [RR87], [PH88], [Ost89], [AH89], [NRSV90], [HMP91], [AL91].

Another extension in the direction of adding more realism to the formal modeling of real life reactive systems is the consideration of hybrid systems. These are systems consisting of a mixture of discrete components, which are the main topic of study of traditional formal methods, with continuous components which are traditionally described by continuous formalisms such as differential equations. We refer the reader to [MMP91] and [NSY91], where the motivation for such an extension and some proposals for appropriately extended formalisms are discussed.

It appears that both extensions can conveniently and convincingly be applied to the specification language of Statecharts. In fact, [MMP91] uses appropriately extended Statecharts for the most natural illustration of nontrivial but yet readable examples of timed and hybrid systems. However, since Statecharts are not the main topic of that paper, the precise nature of these extensions and their full semantics is not presented there.

This is the main topic of the present paper. We consider the extensions to the language of Statecharts that are necessary in order to represent real time and hybrid systems.

One of our goals in defining the semantics of the extended languages has been to provide a compositional operational semantics. As previous experience shows, this is not as easy a matter when dealing with graphical languages such as Statecharts. To make the extensions easier to understand, we have decided to design first a textual language, called *Statext*, which captures most of the essential elements of Statecharts, excluding its visual representation. *Statext* already contains the timing extension. As a first step, we present structured operational semantics for *Statext*, that includes the representation of real time. The construction of this semantics has been influenced by the development of the timed calculus ATP [NRS90].

As a second step, we consider timed Statecharts, which are Statecharts extended by the addition of time bounds on edges. In building compositional semantics for this language, we have adopted some of the ideas recommended in [Mar91] for the abstract syntax underlying the graphical representation of Statecharts. Similar ideas were proposed in [HGdR88]. However, a major departure from the semantics of either [Mar91] or [HPSS86] and [PS89], is that the notions of macro- and micro-steps no longer play such an important role in the semantics. For the untimed steps transitions are freely interleaved, and the reaction to the generation of an event can come several steps later, provided it is still within the same time stamp. This is supported by the fact that any generated event persists as long as time does not progress, and time cannot progress until all transitions that can respond to this event have been taken.

Thus, the progress of time forms a natural boundary for any chain reaction that consists of a sequence of transitions, each responding to events generated by their predecessors in the chain.

Following, we consider the extension of the *Statext* language to accommodate continuous components represented by differential equations. On the syntactic side the extension is simple. Differential equations are allowed as any other statement of the language. The compositional semantics is more involved and calls for more complex labels for transitions. For the case of a continuous step, these labels describe the continuous evolution of the state variables over the step. Parallel composition is done in almost the standard way: possible steps of two concurrent continuous components can be combined only if the two components agree between themselves on the evolution of variables that they have in common.

The semantics of hybrid Statecharts is not worked out in full detail. We bring a typical example and rely on the analogy between *Statext* and Statecharts to convey the general direction such a definition will take. This will be presented in a fuller version of this paper.

2 The *Statext* Language

As any other textual language, *Statext* is constructed out of *statements*, which manipulate *data* and send and receive messages. Below, we specify first the data and expressions of the *Statext* language, then we list its statements.

Data, Expressions and Interval Specifications

- *Variables* and *Channels*
  Similar to *Statecharts*, the *Statext* language is based on *shared variables* and *communication channels*. Both variables and channels are typed.

  It is required that the domain of each channel contains the special value $\perp$, implying no current message (an empty channel).

- *Expressions*
  Expressions are formed by applying operations to constants, variables and channels.

- *Interval specifications*
  An *interval specification* I, is used to specify real time constraints. It is one of

  $$[l,u],\ \ [l,u),\ \ (l,u],\ \ (l,u)$$

  where l and u are real numbers, $0 \leq l \leq u \leq \infty$, called *lower* and *upper* bounds respectively. We write $\{l,u\}$ to specify any of the above four cases. We say that t belongs to the interval I, denoted by $t \in I$, according to the following cases:

  - $t \in [l,u]$ iff $l \leq t \leq u$
  - $t \in (l,u]$ iff $l < t \leq u$
  - $t \in [l,u)$ iff $l \leq t < u$
  - $t \in (l,u)$ iff $l < t < u$

  We say that t *does not exceed* I, denoted by $t \preceq I$, if there exists a $t' \in I$ such that $t \leq t'$.

Simple Statements

We begin by listing the *simple* statements, that is, statements that do not have substatements.

- **skip**

This statement serves as a filler. It does nothing and terminates in a single execution step.

- **idle**

Like the **skip** statement, the **idle** statement does not change the data state. However, unlike the **skip** statement, the **idle** statement never terminates. The only way to get out of an **idle** statement is by *preemption* which is a major feature of *Statext* (and of *Statecharts*).

- **ascertain**

An *ascertain* statement has the general form

ascertain c **for** I,

where c is a boolean expression (*condition*) and I is an *interval specification*.

Like **skip** and **idle** the *ascertain* statement does not change the data state. It waits until the condition c is *continuously true* for time $t \in I$, at which point it terminates. We write

ascertain c **for** d	for	**ascertain** c **for** $[d, d]$,
await c	for	**ascertain** c **for** 0,
delay I	for	**ascertain** $true$ **for** I

- *Assignment*

For $\overline{y}$ a list of variables and $\overline{e}$ a list of expressions of the same length and corresponding types,

$\overline{y} \ := \ \overline{e}$

is an *assignment* statement. Its execution evaluates the list of expressions $\overline{e} = e_1, \ldots, e_m$ and *then* assigns them to the variables $\overline{y} = y_1, \ldots, y_m$, respectively. An assignment is enabled whenever it is *ready*, i.e., control is in front of the assignment, and it terminates in a single step.

- *Send*

For a channel α and expression e,

$\alpha!e$

is a *send* (also called a *signal*) statement. Its execution broadcasts the message consisting the value of e over channel α, to be received by whoever is listening at this point. This statement can be viewed as an assignment of e to the channel α.

A special type of channel is the *single message* channel, i.e., a channel that ranges over the domain $\langle \perp, 1 \rangle$. For a single message channel α, we may write $\alpha!$ as an abbreviation for $\alpha!1$. This represents a frequent case in which we are not interseted in the value of the message but merely in the fact that a message (or a signal) has been broadcasted on channel α.

- *Receive*

For a channel α, and a variable y of compatible type,

$\alpha?y$

is a *receive* statement. The intended meaning of this statement is that it can be taken from a state in which the statement is ready, and at time t, such that some message $v \neq \perp$ has been broadcasted on α at time t, i.e., channel α is nonempty. When executed, the statement places the value v in variable y.

In the case that channel α is single message and we do not intend to examine variable y, we may write $\alpha?$ as an abbreviation for $\alpha?y$. This statement can be viewed as an equivalent to **await** $\alpha \neq \perp$.

Reference to a single message channel α within a boolean expression (*condition*), is equivalent to the atomic formula $(\alpha \neq \perp)$.

Compound Statements

Compound statements consist of a *controlling frame* applied to one or more simpler statements, to which we refer as the *body* of the compound statement. Typically, execution of a compound statement requires several computational steps which are often non-consecutive.

• *Conditional*

For S_1 and S_2 statements and c a boolean expression,

if c **then** S_1 **else** S_2

is a *conditional* statement. Its intended meaning is that the boolean expression c is evaluated and tested. If the condition evaluates to T, statement S_1 is selected for subsequent execution; otherwise, S_2 is selected. Thus, the first step in an execution of the *conditional* statement is the evaluation of c and the selection of S_1 or S_2 for further execution. Subsequent steps continue to execute the selected substatement.

When control reaches a *conditional* statement, the first step in its execution can always be taken. This is because c always evaluates to either T or F and the first step in an execution of the statement is therefore defined for both values of the condition c.

We abbreviate **if** c **then** S_1 **else skip** to

if c **then** S_1.

• **while**

For c a boolean expression and S a statement,

while c **do** S

is a *while* statement. Its execution begins by evaluating c. If c is found to be false, execution of the statement terminates. If c is found to be true, subsequent steps proceed to execute S. If S terminates, c is tested again. Thus, the first step in the execution of a *while* statement is the evaluation of guard c and either finding it true and committing to execution of at least one more repetition of the body S at subsequent steps, or finding the guard c false and terminating execution of the *while* statement. Statement S is called the *body* of the *while* statement.

We write **loop forever do** S as an abbreviation for **while** T **do** S.

• *Concatenation*

For S_1 and S_2 statements,

$S_1;\ S_2$

is a *concatenation* statement. Its intended meaning is sequential composition. First S_1 is executed, and when it terminates S_2 is executed. Thus, the first step in an execution of $S_1; S_2$ is the first step in an execution of S_1. Subsequent steps continue to execute the rest of S_1, and when S_1 terminates, proceed to execute S_2.

More than two statements can be combined by concatenation to form a *multiple concatenation* statement S:

$S_1;\ S_2;\ \ldots;\ S_n$.

• *Selection*

For S_1 and S_2 statements,

$S_1 \bigsqcup S_2$

is a *selection* statement. Its intended meaning is that, as a first step, one of S_1 and S_2, which is currently enabled, is selected and the first step in the selected statement is executed. Subsequent steps proceed to execute the rest of the selected substatement. If both S_1 and S_2 are enabled, the selection is non-deterministic. If both S_1 and S_2 are currently disabled, then so is the *selection* statement.

Note that all steps in the execution of the *selection* statement are attributed to its body. This is because the first step in the execution of $S_1 \sqcup S_2$ corresponds to a step in the execution of either S_1 or S_2.

More than two statements can be grouped into a *multiple selection* statement

$$S_1 \sqcup S_2 \sqcup \ldots \sqcup S_n,$$

which may be abbreviated to

$$\bigsqcup_{i=1}^{n} S_i.$$

The *selection* statement is often applied to concatenations whose first statement is an *await* or *receive* statement. This combination leads to *conditional selection*. For example, the general conditional command of the *guarded command* language (proposed by Dijkstra), of the form

$$\mathbf{if}\ \ c_1 \rightarrow S_1 \ \square\ c_2 \rightarrow S_2 \ \square\ \ldots \ \square\ c_n \rightarrow S_n \ \ \mathbf{fi}$$

can be represented in our language by a multiple *selection* statement formed out of *await* statements:

$$[\mathbf{await}\ c_1\,;\,S_1] \ \sqcup\ [\mathbf{await}\ c_2\,;\,S_2] \ \sqcup\ \cdots\ \sqcup\ [\mathbf{await}\ c_n\,;\,S_n],$$

or in shorter form:

$$\bigsqcup_{i=1}^{n} [\mathbf{await}\ c_i\,;\,S_i].$$

The first step in the execution of this multiple *selection* statement consists of arbitrarily choosing an i such that c_i is currently true, and *passing the guard* c_i. This implies commitment to execute the selected S_i in subsequent steps. The order in which the alternatives are listed is immaterial and does not imply a higher priority to the alternatives appearing earlier in the list.

Note that we do not require that the c_i's be exclusive, i.e., that $c_i \rightarrow (\neg c_j)$ for every $j \neq i$. Nor do we require that the c_i's be exhaustive, i.e., that $\bigvee_{i=1}^{n} c_i$ is true. Non-exclusivity allows non-determinism, while non-exhaustiveness allows the possibility of deadlock, e.g., being at a *selection* statement with all conditions false (which requires waiting until one becomes true).

The *receive* statement can be used similarly for conditional selection:

$$\bigsqcup_{i=1}^{n} [\alpha_i ? v_i ; S_i]$$

where each $\alpha_i ? v_i$ acts as a guard.

- *Cooperation*

For S_1 and S_2 statements,

$$S_1 \parallel S_2$$

is a *cooperation* statement. It calls for parallel execution of S_1 and S_2. Parallel execution is modeled by *interleaving*. Steps in $[S_1 \parallel S_2]$ are interleaved steps from S_1 and S_2. The cooperation statement terminates when both S_1 and S_2 have terminated. Thus, in the combination $[S_1 \| S_2]; S_3$, execution of S_3 will not start until both S_1 and S_2 are terminated.

Similar to the *selection* statement, more than two statements can be grouped into a *multiple cooperation* statement

$$S_1 \parallel S_2 \parallel \ldots \parallel S_n \qquad \text{or, equivalently,} \qquad \mathop{\Big\|}_{i=1}^{n} S_i.$$

- *Preemption*

For statements S_1 and S_2,

$S_1 \ \mathcal{U} \ S_2$,

is a *preemption* statement. Steps in the execution of this statement are either steps in the execution of S_1 or any *first* step in the execution of S_2. Any step leading to the termination of S_1 also causes the termination of $S_1 \ \mathcal{U} \ S_2$. Execution of a step of S_2 causes the rest of S_1 to be discarded, and proceeds to execute steps of S_2 until it terminates. Thus, the intended meaning of the *preemption* statement is

Execute S_1 until a first step of S_2 can be taken.

Consider, for example, the statement

$$\left[\begin{array}{l} \textbf{while } \mathrm{T} \textbf{ do} \\ \quad \left[\begin{array}{l} \textbf{delay } I \\ y := y + 1 \end{array}\right] \end{array}\right] \quad \mathcal{U} \quad \textbf{await } y > 2$$

This statement terminates as soon as y grows above 2, even though the *while* statement, when standing alone, does not terminate.

- *Block*

The *block* statement is intended to allow local declarations and hiding of local variables and channels. In this version of the paper we will omit the presentation of the *block* statement.

3 An Example

Before presenting the formal semantics of the *Statext* language and *Timed Statecharts*, we consider an example.

In Fig. 1 we present a *Timed Statecharts* specification of two machines which communicate by a buffer that can hold only one item at a time. This buffer can represent a receptacle that travels back and forth between the two machines.

The specification consists of three automata: M_1, M_2, and *Buffer*, which act concurrently. These components may represent a machine that does the initial processing of a part, a second machine which applies more advanced processing to the part, and a conveyer device which moves the part from one to another.

A general label of a transition in a *Timed Statecharts* specification has one of the following forms:

- $name : e[c]/a$.
- $name : ([c] \ for \ I)/a$.

In these labels, *name* stands for the transition name (has no semantic meaning), e is an *event expression*, (conjunction of events or negated events), $[c]$ is a *condition* (boolean expression over shared variables), a is an *action* which can generate events or assign values to shared variables when the transition is taken, and I is a *time interval*. All the transition label components are optional.

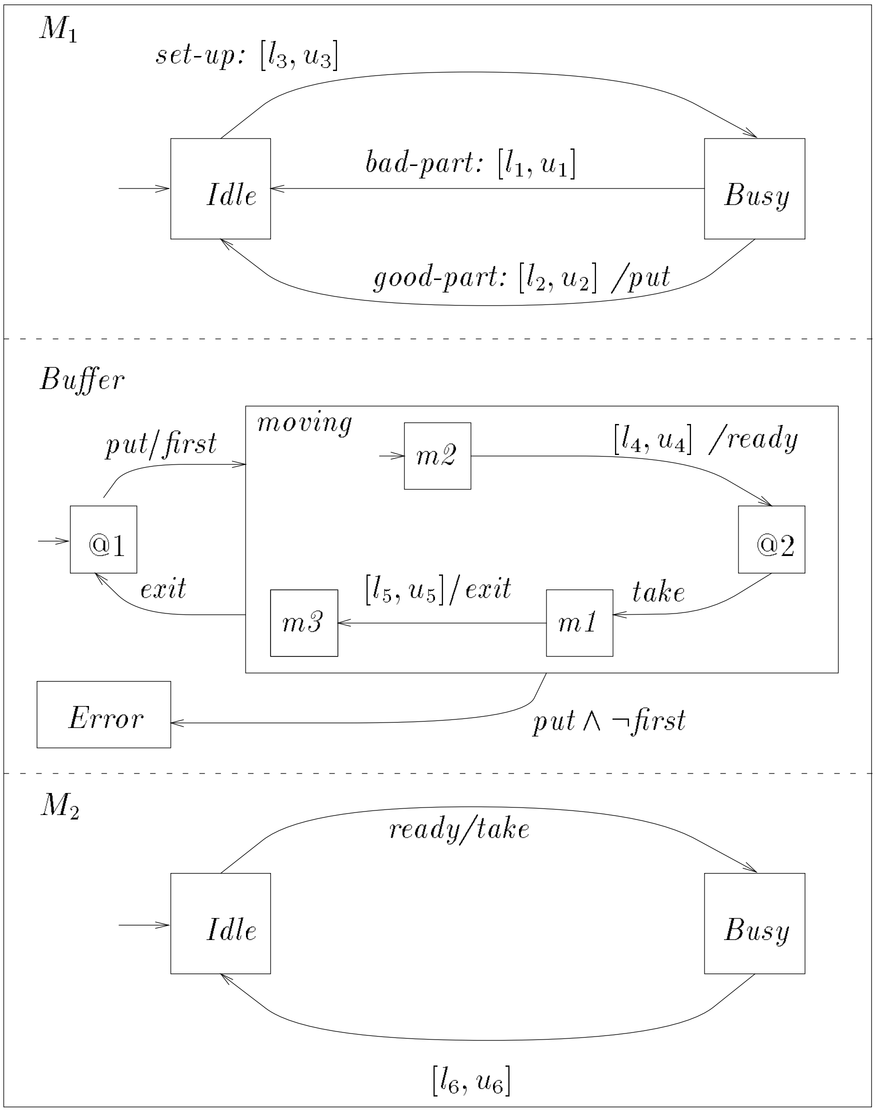


Figure 1: Specification of Two Machines with a Buffer.

For the case that $c = true$, we abbreviate the label

$$name : ([c] \text{ for } I)/a$$

to

$$name : I/a$$

When a transition has no triggering event or condition to be satisfied, such as transition *good-part* in the diagram, the transition is enabled whenever the state from which it departs (state *Busy* in the diagram) is active.

The *time interval* $I = \{l, u\}$ specifies the lower and upper bounds on the continuous enabledness interval of the transition, before it is taken. Transitions with time bounds [0,0] are called *immediate*. We require that all transitions which have a triggering event be immediate. Transitions which are not explicitly labeled are considered to have the time bounds $[0, 0]$, and are therefore immediate. A transition is called *relevant* if the state from which it departs is currently active. Transitions that have no triggering event and no condition, are enabled whenever they are relevant. A transition with a triggering event e (condition $[c]$) is enabled if it is relevant and the event e has just occurred (the condition $[c]$ is satisfied).

Some states are *compound*. For example, the state encompassing basic states *m2*, @2, *m1* and *m3* in the automaton *Buffer* is compound. It is considered active whenever one of these basic states is active. A transition departing from a compound state, such as the transition leading into *Error*, is relevant whenever the compound state is active and, when taken, it makes the compound state and all of its substates inactive and activates the state to which the transition leads (*Error* in the example above). A transition entering a compound state, enters the *default state* within that compound state, denoted by small arrow (see *m2* in *moving*).

When a transition which generates an event e is taken, one or more immediate transitions that have e as a triggering event and are currently relevant are taken. This is the mechanism by which the concurrent automata synchronize. For example, if M_1 takes transition *good-part* while *Buffer* is in state @1, then the transition labeled by *put* can be taken next. Note that, since this transition is immediate and ready, it must be taken before time can progress.

Consider for example the transition connecting state *m2* to state @2. Assume that it is taken at position j while, at the same time, machine M_2 is in state *Idle*. On being taken, this transition generates event *ready*. Machine M_2 responds to this generation by taking the transition leading into state *Busy*, generating the event *take*. Since at this point *Buffer* is at state @2, it responds immediately by moving to state *m1*. These three transitions are taken before the clock advances.

Another important element of the behavior of Statecharts is that events that are generated at a certain step persist until time progresses. This allows more than one transition triggered by an event e to respond to the generation of e before time progresses.

The given specification describes the following possible scenarios. Both machines start at an idle state. After some time ranging between l_3 and u_3, machine M_1 concludes its set-up procedure and moves to the busy state. While being busy, M_1 may either take time between l_1 and u_1 to produce a bad part, or take time between l_2 and u_2 to produce a good part. In both cases it moves to state *Idle* where it completes another set-up procedure. If a good part is produced the event *put* is generated which triggers the transition departing from state @1 in *Buffer*, if it is ready. This transition represents the initiation of movement of the conveyer between M_1 and M_2. The movement itself may take time between l_4 and u_4 to reach M_2. Reaching M_2 is represented by the transition connecting to state @2 which also generates the event *ready*. This event is sensed by M_2 which removes the part from the conveyer and starts processing it in its *Busy* state.

If all timing are right, M_1 should never issue the *put* signal when the conveyer is at state *moving* (i.e. not at state @1). The diagram represents this expectation by having a transition that moves to state *Error* if that erroneous situation arises.

Note the reason for the conjunct $\neg$*first* in the triggering expression of the transition leading from *moving* to *Error*. State *moving* can be entered from state @1 only when signal *put* is emitted. According to the semantics defined in this paper, several transitions in the same component can be taken at the same time instant. Consequently, if the transition from *moving* to *Error* were labeled only by *put*, it would have been taken immediately following the transition entering *moving*. To prevent this, the transition entering *moving* emits the signal *first*, and the error transition requires that *put* is on, but also that *first* is off.

An interesting analysis question is what should be the relation between the various time constants to ensure that this never happens. Various verification tools, based on algorithms similar to the one proposed in [ACD90], can answer such questions algorithmically.

Now in Text form

The top level of the text form of the specification is

$$\mathit{Spec} :: \quad \left[M_1 \parallel \mathit{Buffer} \parallel M_2 \right]$$

M_1, *Buffer*, and M_2 are processes defined as follows:

$$M_1 :: \quad \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{ll} \mathit{Idle}: & \textbf{skip} \\ \text{set-up}: & \textbf{delay}\ [l_3, u_3] \\ \mathit{Busy}: & \left[\begin{array}{lll} \textbf{skip}; & \mathit{bad\text{-}part}: & \textbf{delay}\ [l_1, u_1] \\ & \sqcup & \\ \textbf{skip}; & \mathit{good\text{-}part}: & \textbf{delay}\ [l_2, u_2]; \quad \mathit{put}! \end{array} \right] \end{array} \right] \end{array} \right]$$

Process *Buffer* is given by

$$\mathit{Buffer} :: \quad \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} @1: \ \textbf{await}\ \mathit{put};\ \mathit{first}! \\ \left[\begin{array}{l} \left[\begin{array}{ll} \mathit{m2}: & \textbf{delay}\ [l_4, u_4];\ \mathit{ready}! \\ @2: & \textbf{await}\ \mathit{take} \\ \mathit{m1}: & \textbf{delay}\ [l_5, u_5];\ \mathit{exit!} \\ \mathit{m3}: & \textbf{idle} \end{array} \right] \\ \mathcal{U} \\ \left[\begin{array}{lll} \textbf{await} & \mathit{put} \wedge \neg \mathit{first}; & \mathit{Error}: \textbf{idle} \\ & \sqcup & \\ \textbf{await} & \mathit{exit} & \end{array} \right] \end{array} \right] \end{array} \right] \end{array} \right]$$

Process M_2 is given by

$$M_2 :: \quad \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{ll} \mathit{Idle}: & \textbf{await}\ \mathit{ready}\ ; \quad \mathit{take}! \\ \mathit{Busy}: & \textbf{delay}\ [l_6, u_6] \end{array} \right] \end{array} \right]$$

One may complain that when process *Buffer* detects an error situation, it is the only process affected while M_1 and M_2 continue their activity as though nothing has happened. An alternative specification may require the whole system to stop its activity in such a case. In Fig. 2 we present the Statechart representation of this specification.

The *Statext* representation of this specification is given by

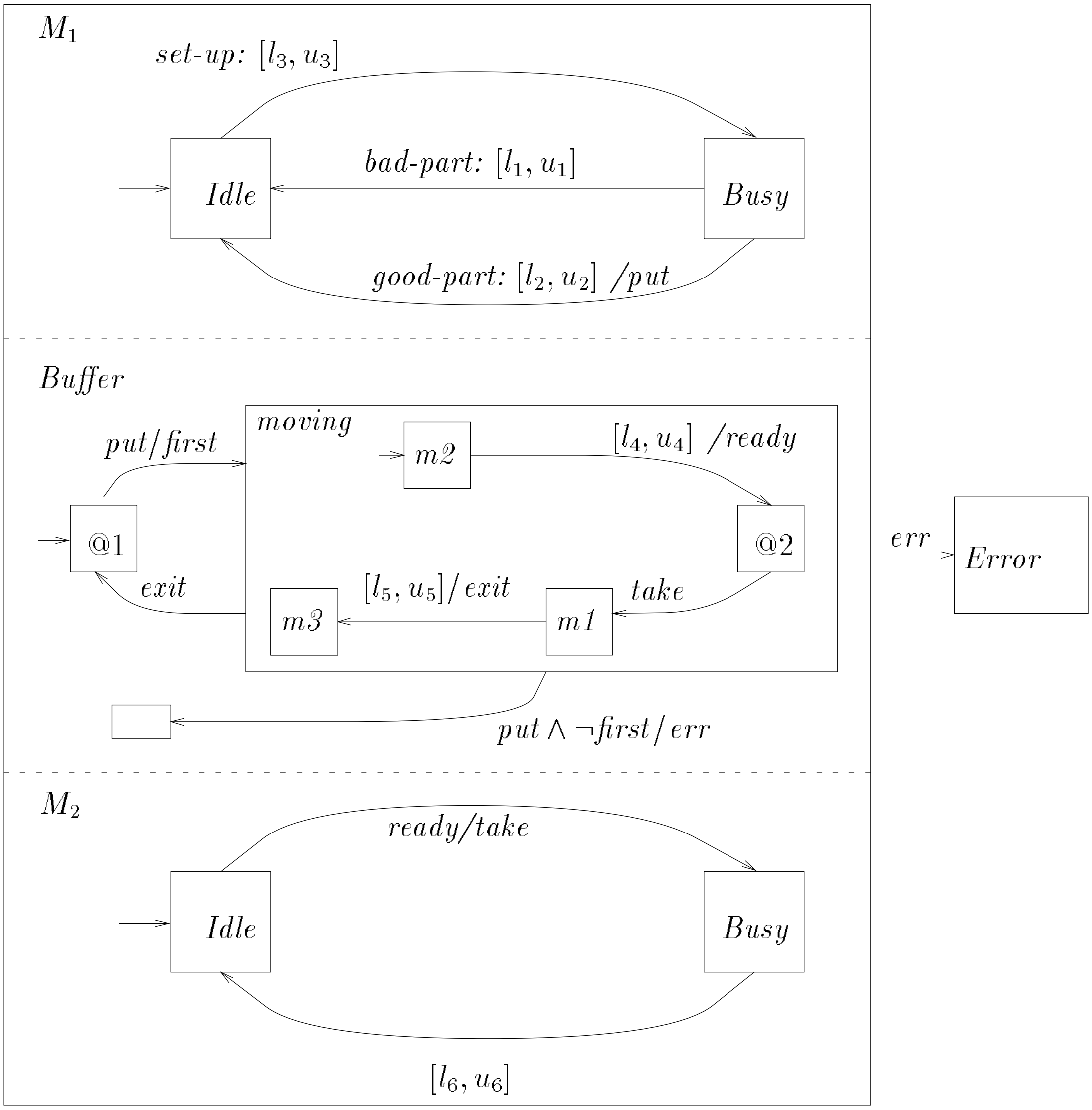


Figure 2: Revised Specification.

$$Spec_1 :: \quad \left[\begin{array}{c} M_1 \parallel Buffer_1 \parallel M_2 \\ \mathcal{U} \\ \textbf{await } err;\ Error: \textbf{idle} \end{array} \right]$$

where M_1 and M_2 are as before, and $Buffer_1$ is given by

$$Buffer_1 :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} @1: \ \textbf{await}\ put;\ first! \\ \left[\begin{array}{l} \left[\begin{array}{ll} m2: & \textbf{delay}\ [l_4, u_4];\ ready! \\ @2: & \textbf{await}\ take \\ m1: & \textbf{delay}\ [l_5, u_5];\ exit! \\ m3: & \textbf{idle} \end{array}\right] \\ \mathcal{U} \\ \left[\begin{array}{lcl} \textbf{await} & put \wedge \neg first; & err!;\ \textbf{idle} \\ & \sqcup & \\ \textbf{await} & exit & \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

This specification uses the auxiliary signal *err* which is emitted by process $Buffer_1$ on detecting an error situation. This signal preempts the parallel composition of M_1, $Buffer_1$, and M_2 and leads to the external *Error* state.

4 Timed Transition System

The semantics of *Timed Statecharts* and *Statext* specifications is defined as sets of *timed behaviors*. This is done in two steps. First, we introduce the computation model of *timed transition system* and identify the possible timed behaviors (computations) of any such system. Then, we show how a *Timed Statecharts* and *Statext* specification can be mapped into this generic model.

The notions of *timed behavior* and *timed transition system* are based on [HMP90, HMP91], to which we have added *explicit* timed transitions and partitioned the set of state variables into *persistent* and *volatile* variables.

As in [HMP90, HMP91], we assume a global clock. A *timed transition* (clock tick) advances the global clock by a positive amount. Timed transitions are synchronous. They require the cooperation of all the concurrent processes, which have to agree on the increment by which the clock is advanced, and perform the transition simultanously. Timed transitions are interleaved with other (untimed) systems transitions.

The partition of the state variables into persistent and volatile variables enables a uniform treatment of communication through both shared variables and channels. Channels are represented by *volatile variables* which are reset by every *clock tick*. Shared variables are represented by *persistent variables*, whose values are unaffected by clock ticks, and can only be modified by an explicit assignment.

To model metric time, we assume a totally ordered time domain $\mathbf{T}$ which contains a zero element $0 \in \mathbf{T}$, and a commutative, associative operation $+$, for which 0 is a unique identity element, and such that for every $t_1, t_2 \in \mathbf{T}$,

$t_1 < t_2$ iff There exists a unique $t > 0$ (denoted by $t_2 - t_1$), such that $t_1 + t = t_2$.

We refer to the elements of $\mathbf{T}$ as *time elements* or sometimes as *moments*. In most cases we will take $\mathbf{T}$ to be either the natural numbers N, or the nonnegative real numbers $\mathsf{R}_{\geq 0}$. We define $\mathbf{T}_{>} = \mathbf{T} - \{0\}$.

With a system to be analyzed we associate

- V : A finite set of *state variables.*
- Σ : A set of *states.* Each state $s \in \Sigma$ is an interpretation of V; that is, it assigns to every variable $x \in V$ a value $s[x]$ in its domain.
- $V_t = V \cup \{t\}$: A finite set of *situation variables.* They are obtained by augmenting V, the set of *state variables*, with t a variable representing the current time in each situation.
- Σ_t : A set of situations. Every situation $s \in \Sigma_t$ is an interpretation of V_t. In particular, $s[t] \in \mathbf{T}$ is the value of the real-time clock at situation s. We denote by $s[V]$ the *state* corresponding to the situation s. It is obtained by restricting the interpretation s to the state variables V.

Timed Traces

A *progressive time sequence* is an infinite sequence of time elements

$$\theta : \; t_0, t_1, \ldots,$$

where $t_i \in \mathbf{T}$, for each $i = 0, 1, \ldots$, satisfying

- $t_0 = 0$.
- Time does not decrease. That is, for every $i \geq 0$, $t_i \leq t_{i+1}$.
- Time eventually increases beyond any bound. That is, for every time element $t \in \mathbf{T}$, $t_i > t$ for some $i \geq 0$. This is called the *Non Zeno* requirement in [AL91].

A *timed trace* is an infinite sequence of situations

$$\sigma : \; s_0, s_1, \ldots,$$

where $s_i \in \Sigma_t$, for each $i = 0, 1, \ldots$. We denote by $t_i = s_i[t]$ the moment at which situation s_i was observed (sampled).

It is required that

- The sequence $t_0, t_1, \ldots$ is a progressive time seqeunce.
- For every $i \geq 0$, state and time do not change at the same time, i.e., either $s_i[V] = s_{i+1}[V]$ or $t_i = t_{i+1}$. This requirement ensures that each state change is precisely timed.

Timed Transition Systems

A *timed transition system* $S = \langle V, \Theta, \mathcal{L}, \mathcal{T} \rangle$ consists of the following components:

- *State variables* V, partitioned into $V = V_p \cup V_v$, where
  - V_p is a set of *persistent variables.* These variables represent data variables. They are not affected by time transitions.
  - V_v is a set of *volatile variables.* These variables represent the communication channels. They are reset by every time transition.

  We denote by Σ the set of all interprations of V.

- The *initial condition* Θ. An assertion that characterizes the states that can appear as initial states in a computation.

- A set of *transition labels* $\mathcal{L}$, partitioned into two disjoint sets $\mathcal{L} = \mathbf{T}_> \cup L$, where a label from $\mathbf{T}_>$ represents a timed step, while a label from L represents an untimed step.

- A *transition relation* $\mathcal{T} \subseteq (\Sigma \times \mathcal{L} \times \Sigma)$, partitioned into $\mathcal{T} = \mathcal{T}_T \cup \mathcal{T}_L$, where
  - $\mathcal{T}_T$ is the *timed transition relation.* For every $(s_i, \mu_i, s_{i+1}) \in \mathcal{T}_T$,

$$\begin{array}{l} \mu_i \in \mathbf{T}_> \\ s_i[V_p] = s_{i+1}[V_p] \\ s_{i+1}[V_v] = \perp \\ s_{i+1}[t] = s_i[t] + \mu_i \end{array}$$

  - $\mathcal{T}_L$ is the *untimed transition relation.* For every $(s_i, \mu_i, s_{i+1}) \in \mathcal{T}_L$,

$$\begin{array}{l} \mu_i \in L \\ s_{i+1}[t] = s_i[t] \end{array}$$

Given the state variables V, we can obtain the corresponding set of situation variables $V_t = V \cup \{t\}$, and the set of situations Σ_t interpreting V_t.
A *computation* of a timed transition system $S = \langle V, \Theta, \mathcal{L}, \mathcal{T} \rangle$ is an infinite sequence of alternating situations and transition labels

$$\sigma : \; s_0, \mu_0, s_1, \mu_1 \ldots,$$

where $s_i \in \Sigma_t$ and $\mu_i \in \mathcal{L}$ for each $i = 0, 1, \ldots$.
It is required that

- [*Time Progress*] The sequence $s_0, s_1, \ldots$ is a timed trace.
- [*Initiality*] $s_0 \models \Theta$.
- [*Consecution*] For all $i \geq 0, \quad (s_i, \mu_i, s_{i+1}) \in \mathcal{T}$.

As shown in [HMP91], the model of timed transition systems is expressive enough to express most of the features specific to real-time programs such as delays, timeouts, preemption, interrupts and multi-programming scheduling.

5 *Statext* Semantics

With every *Statext* specification P, we associate a timed transition system $\langle V, \Theta, \mathcal{L}, \mathcal{T} \rangle$ as follows:

- $V = V_p \cup V_v$, where V_p contains the set of all *shared variables* in P, and the control variable π ranging over statements, and V_v is the set of *communication channels.*
- $\Theta : (\pi = P) \wedge \bigwedge_{\alpha \in V_v} (\alpha = \perp)$
- $\mathcal{L} = \mathbf{T}_> \cup L$, where $L = \{\alpha!e \mid \alpha \in V_v,$ and $e \neq \perp$ is an expression over the domain of $\alpha\} \cup \{\tau\}$. τ represents non communicating internal operations.
- $\mathcal{T} = \mathcal{T}_T \cup \mathcal{T}_L$, where $\mathcal{T}_T$ and $\mathcal{T}_L$ are defined below.

Statext Transitions

The transition relation associated with a *Statext* specification, is defined by structural induction [Plo81].

A state $s \in \Sigma$ is represented by the tuple $\langle S, D, E \rangle$, where S is a statement interpreting π, and D, E are interpretations of $V_p - \{\pi\}$ and V_v respectively. We use Λ to denote the empty statement.

We use derivations of the form $s_i \xrightarrow{\xi} s_j$, where $s_i, s_j \in \Sigma$, $\xi \in \{\alpha!e, \Delta, \lambda, \mu\}$, and
$\Delta \in \mathbf{T}_>$
$\lambda \in L$
$\mu \in \mathcal{L}$

The transition relation is defined by axioms that contain a derivation (possibly with some side conditions) and rules of the form $d_1, d_2, \ldots, d_k \vdash d$, sometimes represented as $\frac{d_1, \ldots, d_k}{d}$.

Untimed Transition Relation

- **skip**

  $\langle \mathbf{skip}, D, E \rangle \xrightarrow{\tau} \langle \Lambda, D, E \rangle$

- **idle**

  The **idle** statement has only associated timed transitions.

- **ascertain**

  To define the semantics of the *ascertain* statement, we introduce a more general form of the statement

  $$\mathbf{ascertain}\ c\ \mathbf{from}\ t_0\ \mathbf{to}\ I$$

  This form specifies an initial value $t_0 \in \mathbf{T}_>$ at which time counting begins. The standard form **ascertain** c **for** I can be viewed as an abbreviation of **ascertain** c **from** 0 **to** I. We define transition semantics for the more general form.

  $\langle \mathbf{ascertain}\ c\ \mathbf{from}\ t_0\ \mathbf{to}\ I, D, E \rangle \xrightarrow{\tau} \langle \Lambda, D, E \rangle$
  provided $t_0 \in I$ and $(D \cup E) \models c$

- *Assignment*

  $\langle \overline{y} := \overline{e}, D, E\rangle \xrightarrow{\tau} \langle \Lambda, D[\overline{y} \leftarrow \overline{e}], E\rangle$

- *Send*

  $\langle \alpha!e, D, E\rangle \xrightarrow{\alpha!e} \langle \Lambda, D, E[\alpha \leftarrow e]\rangle$

- *Receive*

  $\langle \alpha?y, D, E[\alpha \leftarrow v]\rangle \xrightarrow{\tau} \langle \Lambda, D[y \leftarrow v], E[\alpha \leftarrow v]\rangle$, for $v \neq \perp$

  The notation $E[\alpha \leftarrow v]$ denotes any interpretation that can be obtained by sending v on α, i.e. an interpretation in which the value of α is $v \neq \perp$.

- *Conditional*

  $\langle \textbf{if } c \textbf{ then } S_1 \textbf{ else } S_2, D, E\rangle \xrightarrow{\tau} \langle S_1, D, E\rangle$
  provided $(D \cup E) \models c$

  $\langle \textbf{if } c \textbf{ then } S_1 \textbf{ else } S_2, D, E\rangle \xrightarrow{\tau} \langle S_2, D, E\rangle$
  provided $(D \cup E) \not\models c$

- **while**

  $\langle \textbf{while } c \textbf{ do } S, D, E\rangle \xrightarrow{\tau} \langle S; \textbf{while } c \textbf{ do } S, D, E\rangle$
  provided $(D \cup E) \models c$

  $\langle \textbf{while } c \textbf{ do } S, D, E\rangle \xrightarrow{\tau} \langle \Lambda, D, E\rangle$
  provided $(D \cup E) \not\models c$

- *Concatenation*

  $$\frac{\langle S_1, D, E\rangle \xrightarrow{\mu} \langle \Lambda, D', E'\rangle}{\langle S_1; S_2, D, E\rangle \xrightarrow{\mu} \langle S_2, D', E'\rangle}$$

  $$\frac{\langle S_1, D, E\rangle \xrightarrow{\mu} \langle S_1', D', E'\rangle}{\langle S_1; S_2, D, E\rangle \xrightarrow{\mu} \langle S_1'; S_2, D', E'\rangle}, \text{ where } S_1' \neq \Lambda$$

- *Selection*

  $$\frac{\langle S_1, D, E\rangle \xrightarrow{\lambda} \langle S_1', D', E'\rangle}{\langle S_1 \bigsqcup S_2, D, E\rangle \xrightarrow{\lambda} \langle S_1', D', E'\rangle}$$

  $$\frac{\langle S_2, D, E\rangle \xrightarrow{\lambda} \langle S_2', D', E'\rangle}{\langle S_1 \bigsqcup S_2, D, E\rangle \xrightarrow{\lambda} \langle S_2', D', E'\rangle}$$

- *Cooperation*

$$\frac{\langle S_1, D, E\rangle \xrightarrow{\lambda} \langle S_1', D', E'\rangle}{\langle S_1 \| S_2, D, E\rangle \xrightarrow{\lambda} \langle S_1' \| S_2, D', E'\rangle}, \text{ where } S_1' \neq \Lambda$$

$$\frac{\langle S_2, D, E\rangle \xrightarrow{\lambda} \langle S_2', D', E'\rangle}{\langle S_1 \| S_2, D, E\rangle \xrightarrow{\lambda} \langle S_1 \| S_2', D', E'\rangle}, \text{ where } S_2' \neq \Lambda$$

$$\frac{\langle S_i, D, E\rangle \xrightarrow{\lambda} \langle \Lambda, D', E'\rangle}{\langle S_1 \| S_2, D, E\rangle \xrightarrow{\lambda} \langle S_j, D', E'\rangle}, \text{ for } 1 \leq i, j \leq 2 \text{ and } i \neq j.$$

- *Preemption*

$$\frac{\langle S_1, D, E\rangle \xrightarrow{\lambda} \langle S_1', D', E'\rangle}{\langle S_1 \mathcal{U}\ S_2, D, E\rangle \xrightarrow{\lambda} \langle S_1' \mathcal{U}\ S_2, D', E'\rangle}, \text{ where } S_1' \neq \Lambda$$

$$\frac{\langle S_2, D, E\rangle \xrightarrow{\lambda} \langle S_2', D', E'\rangle}{\langle S_1 \mathcal{U}\ S_2, D, E\rangle \xrightarrow{\lambda} \langle S_2', D', E'\rangle}$$

$$\frac{\langle S_1, D, E\rangle \xrightarrow{\lambda} \langle \Lambda, D', E'\rangle}{\langle S_1 \mathcal{U}\ S_2, D, E\rangle \xrightarrow{\lambda} \langle \Lambda, D', E'\rangle}$$

Timed Transition Relation (clock ticks)

- *Ascertain*

  $\langle \textbf{ascertain } c \textbf{ from } t_0 \textbf{ to } I, D, E\rangle \xrightarrow{\Delta} \langle \textbf{ascertain } c \textbf{ from } (t_0 + \Delta) \textbf{ to } I, D, \perp\rangle$
  provided $(D \cup E) \models c$, $\Delta > 0$ and $\Delta + t_0 \preceq I$.

  $\langle \textbf{ascertain } c \textbf{ from } t_0 \textbf{ to } I, D, E\rangle \xrightarrow{\Delta} \langle \textbf{ascertain } c \textbf{ from } 0 \textbf{ to } I, D, \perp\rangle$,
  for every $\Delta > 0$, provided $(D \cup E) \not\models c$

  In these axioms and the following rules, we use $\perp$ to denote the empty channel interpretation that interprets every channel $\alpha \in V_v$ as $\perp$. All timed transitions reset E to $\perp$.

- *Receive*

  $\langle \alpha?y, D, E[\alpha \leftarrow \perp]\rangle \xrightarrow{\Delta} \langle \alpha?y, D, \perp\rangle$, for every $\Delta > 0$

- *Idle*

  $\langle \textbf{idle}, D, E\rangle \xrightarrow{\Delta} \langle \textbf{idle}, D, \perp\rangle$, for every $\Delta > 0$

- *Compound*

  For $\oplus \in \{\bigsqcup, \|, \mathcal{U}\}$

$$\frac{\langle S_i, D, E\rangle \xrightarrow{\Delta} \langle S_i', D, \perp\rangle}{\langle S_1 \oplus S_2, D, E\rangle \xrightarrow{\Delta} \langle S_1' \oplus S_2', D, \perp\rangle}, \text{ for } 1 \le i \le 2$$

The following observations can be made concerning a *Statext* computation:

- Computations alternate between transitions in $\mathcal{T}_{\mathbf{T}}$ that advance the global clock and transitions in $\mathcal{T}_L$ that take zero time.
- Transitions in $\mathcal{T}_{\mathbf{T}}$ are executed *synchronously*, while transitions in $\mathcal{T}_L$ are *asynchronous*.
- A transition in $\mathcal{T}_{\mathbf{T}}$ can advance the global clock only by an amount on which all the enabled transitions agree.

6 Timed Statecharts

Timed Statecharts is an extension to the visual specification language *Statecharts* [Har87], in which each transition is annotated by a time interval $\{l, u\}$, denoting the lower and upper time bounds of that transition.

In the definition of Timed Statecharts, we have adopted the approach taken by *Argos* [Mar90], namely, disallowing:

- Inter-level transitions.
- References to state names.

These modifications are intended to improve the modality of the language.

Different than both Statecharts and Argos, Timed Statecharts are *asynchronous*, with concurrency being modeled by interleaving. The only synchronous steps are those in which the time maintained by a global clock, is incremented.

6.1 Abstract syntax

A *rooted automaton* of degree n is a labeled directed graph (r, V, E, ℓ), where

- V is a set of vertices
- E is a set of directed edges
- ℓ is a labeling function, assigning to each edge $e \in E$ a label $\ell(e)$ in some labeling set $\mathcal{L}$.
- $r \in V$ is the *root* vertex. It is required that no edge enters r, and that r has n edges departing from it.

A *single entry automaton*, denoted *SE-automaton* , is a rooted automaton of degree $n \leq 1$.

Two rooted automata (r_1, V_1, E_1, ℓ_1) and (r_2, V_2, E_2, ℓ_2) can be combined into a single rooted automaton

$$(r, V, E, \ell) = (r_1, V_1, E_1, \ell_1) \bigsqcup (r_2, V_2, E_2, \ell_2)$$

by the operation of *fusion*. The automaton (r, V, E, ℓ) is obtained by identifying r_1 with r_2. It is required that $E_1 \cap E_2 = \emptyset$, $r_1 \notin V_2$ and $r_2 \notin V_1$, but $V_1 - \{r_1\}$ and $V_2 - \{r_2\}$ may have a nonempty intersection.

For every labeled directed graph $G : (V, E, \ell)$ (not necessarily rooted) and a vertex $r \in V$, we can construct a rooted automaton $A : (r', V \cup \{r'\}, E', \ell')$ called the *r-unwinding* of G. We write $A = unwind(r, G)$.

Automata G and A are related by the fact that $\ell_1, \ell_2, \ldots, \ell_k$ is a sequenc of labels appearing on a path departing from $r \in G$ *iff* it is a labeling sequence of a path departing from $r' \in A$.

The operation of unwinding is necessary in order to represent graphically the transformation associated with the **while** statement in *Statext* .

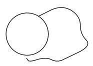
It is not difficult to see that every rooted automaton M of degree n can be obtained as a fusion of n *SE-automata*, i.e., $M = \bigsqcup_{i=1}^{n} B_i$, for some *SE-automata* $B_1, \ldots, B_n$.

We can now give a recursive definition for *Timed Statecharts*.

A Timed Statechart S is one of the following:

- A *rooted automaton* .
- A refined *rooted automaton* .
- A parallel composition of *Timed Statecharts* .

A refined *rooted automaton* is defined as follows. Let $A : (r, V, E, \ell)$ be a *rooted automaton* and $v_1, \ldots, v_k \in V$ a list of vertices. Let $S_1, \ldots, S_k$ be a list of timed statecharts. The refinement of A, encapsulating $S_1, \ldots, S_k$ within $v_1, \ldots, v_k$ is obtained by placing for each $i = 1, \ldots, k$, S_i within the box (circle) that graphically represents v_i.

For the presentation of the *Timed Statecharts* semantics, we use a set of icons as shown in figure 3. We use the icon  to represent a (not necessarily rooted) labeled directed graph.

6.2 Semantics

With every *Timed Statecharts* specification S, we associate a timed transition system $\langle V, \Theta, \mathcal{L}, \mathcal{T} \rangle$, similar to that of *Statext* programs except for the transition relation that will be defined below, and the fact that π ranges over *Timed Statecharts*.

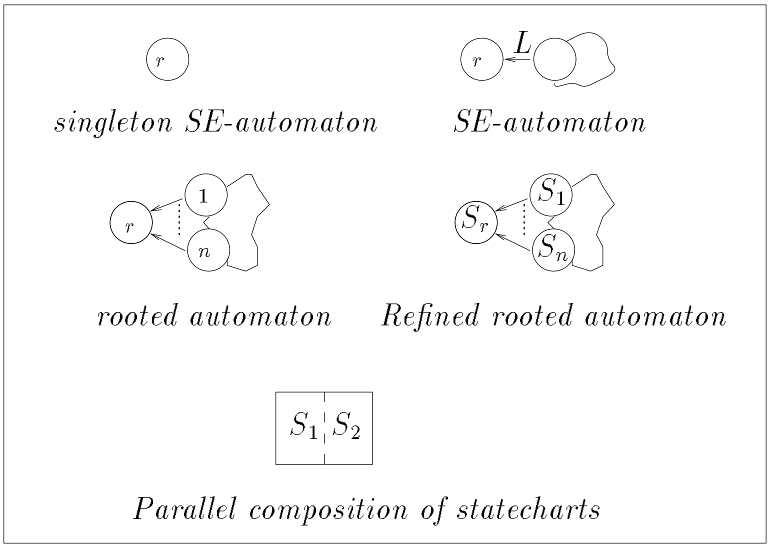


Figure 3: Iconic representation of Statecharts.

Timed Statecharts Transitions

For an action a, data-state D and event set E, we denote by D'_a, E'_a the data-state and event set resulting by executing a.

- For a of the form $y := e$, we have

$$D'_a = D[y \leftarrow e] \qquad \text{and} \qquad E'_a = E$$

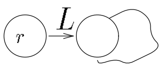
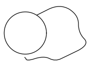
- For a of the form $\alpha!e$, we have

$$D'_a = D \qquad \text{and} \qquad E'_a = E[\alpha \leftarrow e]$$

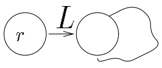
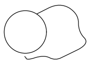
Untimed Transition Relation

- *SE-automaton*

 is equivalent to **idle** in *Statext* , and has no associated *untimed* transitions.

$\langle$  $, D, E\rangle \xrightarrow{\lambda} \langle unwind($  $), D'_a, E'_a\rangle$

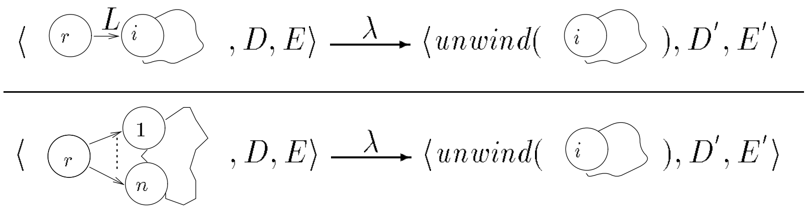
where $L = e[c]/a$, provided $D \cup E \models (e \cup c)$.

$\langle$  $, D, E\rangle \xrightarrow{\lambda} \langle unwind($  $), D'_a, E'_a\rangle$

where $L = ([c]$ *from* t_0 to $I)/a$, provided $D \cup E \models c$ and $t_0 \in I$.

- *Rooted automaton*

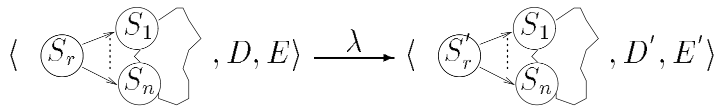
  For every i, $1 \leq i \leq n$



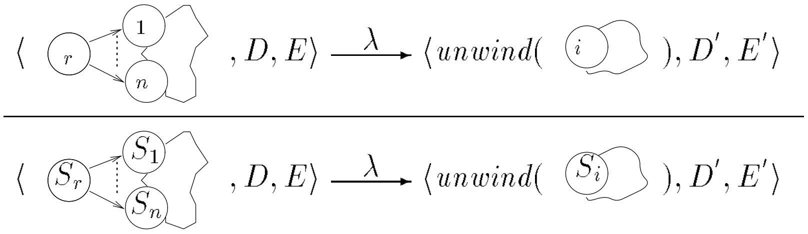
- *Refined rooted automaton*

  For a statechart S_r,

$$\langle S_r, D, E\rangle \xrightarrow{\lambda} \langle S_r', D', E'\rangle$$



For basic states $r, 1, \ldots, n$, statecharts $S_r, S_1, \ldots, S_n$ and every i, $0 \leq i \leq n$,



- *Parallel composition*

$$\frac{\langle S_1, D, E\rangle \xrightarrow{\lambda} \langle S_1', D', E'\rangle}{\langle\ \boxed{S_1 \mid S_2}\ , D, E\rangle \xrightarrow{\lambda} \langle\ \boxed{S_1' \mid S_2}\ , D', E'\rangle}$$

$$\frac{\langle S_2, D, E\rangle \xrightarrow{\lambda} \langle S_2', D', E'\rangle}{\langle\ \boxed{S_1 \mid S_2}\ , D, E\rangle \xrightarrow{\lambda} \langle\ \boxed{S_1 \mid S_2'}\ , D', E'\rangle}$$

Timed Transition Relation

- *SE-automaton*

  For every $\Delta > 0$

$$\langle\ r\ , D, E\rangle \xrightarrow{\Delta} \langle\ r\ , D, \bot\rangle$$

$$\langle\ r \xrightarrow{L} \bigcirc\ , D, E\rangle \xrightarrow{\Delta} \langle\ r \xrightarrow{L} \bigcirc\ , D, \bot\rangle \quad \text{where } L = e[c]/a$$

provided $D \cup E \not\models (e \cup c)$

$$\langle\ r \xrightarrow{L} \bigcirc\ , D, E\rangle \xrightarrow{\Delta} \langle\ r \xrightarrow{L'} \bigcirc\ , D, \bot\rangle$$

where $L = ([c]\ \mathit{from}\ t_0 \text{ to } I)/a$ and $L' = ([c]\ I)/a$,
provided $D \cup E \not\models c$

$$\langle\ r \xrightarrow{L} \bigcirc\ , D, E\rangle \xrightarrow{\Delta} \langle\ r \xrightarrow{L'} \bigcirc\ , D, \bot\rangle$$

where $L = ([c]\ \mathit{from}\ t_0 \text{ to } I)/a$ and $L' = ([c]\ \mathit{from}\ (t_0 + \Delta) \text{ to } I)/a$,
provided $D \cup E \models c$, $\Delta > 0$ and $\Delta + t_0 \preceq I$

- *Rooted automaton*

$$\frac{\langle\ r \xrightarrow{L_i} i\ , D, E\rangle \xrightarrow{\Delta} \langle\ r \xrightarrow{L'_i} i\ , D, \bot\rangle \quad \text{for all } i \in \{1..n\}}{\langle\ r \xrightarrow{L_1} 1, \ldots, r \xrightarrow{L_n} n\ , D, E\rangle \xrightarrow{\Delta} \langle\ r \xrightarrow{L'_1} 1, \ldots, r \xrightarrow{L'_n} n\ , D, \bot\rangle}$$

- *Refined rooted automaton*

$$\frac{\begin{array}{ll} 1. & \langle S_r, D, E\rangle \xrightarrow{\Delta} \langle S'_r, D, E\rangle \\ 2. & \langle\ r \xrightarrow{L_1} 1, \ldots, r \xrightarrow{L_n} n\ , D, E\rangle \xrightarrow{\Delta} \langle\ r \xrightarrow{L'_1} 1, \ldots, r \xrightarrow{L'_n} n\ , D, \bot\rangle \end{array}}{\langle\ S_r \xrightarrow{L_1} S_1, \ldots, S_r \xrightarrow{L_n} S_n\ , D, E\rangle \xrightarrow{\Delta} \langle\ S'_r \xrightarrow{L'_1} S_1, \ldots, S'_r \xrightarrow{L'_n} S_n\ , D, \bot\rangle}$$

- *Parallel composition*

$$\frac{\langle S_i, D, E\rangle \xrightarrow{\Delta} \langle S'_i, D, \bot\rangle}{\langle\ \boxed{S_1 \mid S_2}\ D, E\rangle \xrightarrow{\Delta} \langle\ \boxed{S'_1 \mid S'_2}\ , D, \bot\rangle}, \text{ for } 1 \leq i \leq 2$$

7 Hybrid Systems

We can now extend both the *Statext* and *Timed Statecharts* specification languages, such that they can both be used for the specification of hybrid systems. We first define a *hybrid transition system*. Then, we describe the extended specification languages, and define their compositional semantics.

We conclude with a small example, specified in both *Hybrid Statext* and *Hybrid Statecharts*.

7.1 Hybrid Transition System

A *Hybrid transition system* $S_{\mathcal{H}} = \langle V, \Theta, \mathcal{L}, \mathcal{T} \rangle$ consists of the following components:

- *State variables* V, partitioned into $V = V_p \cup V_v$, where
  - V_p is a set of *persistent variables*. These variables represent data variables. They are not reset by time transitions.
  - V_v is a set of *volatile variables*. These variables represent the communication channels. They are reset by every *time transition*.

  The set of *persistent variables* is further partitioned into $V_p = V_c \cup V_d$, where
  - V_c is the set of *continuous variables*. These variables are modified by continuous activities in the behavior of a hybrid system.
  - V_d is the set of *discrete variables*. These variables are changed by discrete steps.

  We denote by Σ the set of all interprations of V.

- The *initial condition* Θ. An assertion that characterizes the states that can appear as initial states in a computation.
- A set of *transition labels* $\mathcal{L}$, partitioned into $\mathcal{L} = \mathcal{A} \cup L$, where
  - Every $\mu \in \mathcal{A}$ is of the form Δ, A where
    - $\Delta \in \mathbf{T}_{>}$.
    - A is an assignment of differentiable functions over $[0, \Delta]$ to a set of continuous variables $Y_A \subseteq V_c$. Thus, A assigns to each $y \in Y_A$ a function $f_y(t)$ differentiable over $t \in [0, \Delta]$.
  - A label $\mu \in L$ represents an *untimed* step.
- A transition relation $\mathcal{T} \subseteq (\Sigma \times \mathcal{L} \times \Sigma)$, partitioned into $\mathcal{T} = \mathcal{T}_{\mathcal{A}} \cup \mathcal{T}_L$, where
  - $\mathcal{T}_{\mathcal{A}}$ is the *timed transition relation*. For every $(s_i, \mu_i, s_{i+1}) \in \mathcal{T}_{\mathcal{A}}$,

$$
\begin{array}{ll}
\mu_i = (\Delta, A) \in \mathcal{A} & \\
s_i[V_d] = s_{i+1}[V_d] & \\
s_{i+1}[V_v] = \bot & \\
s_{i+1}[t] = s_i[t] + \mu_i[\Delta] & \\
s_{i+1}[y] = s_i[y] & \text{for every } y \in (V_c - Y_A) \\
s_i[y] = f_y[0] \text{ and } s_{i+1}[y] = f_y(\Delta) & \text{for every } y \in Y_A
\end{array}
$$

Timed transitions are allowed to change the values of continuous variables.

– $\mathcal{T}_L$ is the *untimed transition relation.* For every $(s_i, \mu_i, s_{i+1}) \in \mathcal{T}_L$,

$$\begin{array}{l} \mu_i \in L \\ s_{i+1}[t] = s_i[t] \end{array}$$

Given the state variables V, we can obtain the corresponding set of situation variables $V_t = V \cup \{t\}$, and the set of situations Σ_t interpreting V_t.

A *hybrid trace* is an infinite sequence of situations

$$\sigma : \; s_0, s_1, \ldots,$$

where $s_i \in \Sigma_t$, for each $i = 0, 1, \ldots$. We denote by $t_i = s_i[t]$ the moment at which situation s_i was observed (sampled).

It is required that

- The sequence $t_0, t_1, \ldots$ is a progressive time seqeunce.
- For every $i \geq 0$, discrete state variables and time do not change at the same time, i.e., either $s_i[V_d] = s_{i+1}[V_d]$ or $t_i = t_{i+1}$.

A *computation* of a hybrid transition system is defined in a similar way to the computation of a timed transition system, except for the *Time Progress* requirement which will require the sequence $s_0, s_1, \ldots$ to be a *hybrid trace* instead of a *time trace.*

7.2 Hybrid *Statext*

Syntax

To the set of *Statext* statements, we add the *continuous assignment* statement,

$$\dot{y} = R$$

This statement represents a differential equation, where y is a continuous variable, and R is a term over V_p. We denote by $vars_c(R)$ the set of continuous variables appearing in R.

To the *ascertain* statement, **ascertain** c **for** I, we add the following restriction

If $u_I > 0$, the boolean expression c can only range over V_d.

This restriction may be removed, and is imposed here for the sake of simplicity.

Semantics

The semantics of *Hybrid Statext* is defined by associating with each hybrid specification, a hybrid transition system. Following we present the transition relation corresponding to each of the statements.

Transitions

Untimed Transition Relation

The untimed transition relation remains unchanged, since the *continuous assignment* statement has no associated untimed transitions.

Timed Transition Relation

The set of rules defining the timed transition relation associated with a *Statext* specification, is replaced by the following rules:

- *Continuous Assignment*

  for every $\Delta > 0$
  $\langle \dot{y} = R, D, E \rangle \xrightarrow{\Delta, A} \langle \dot{y} = R, D[Y_A \leftarrow F_A(\Delta)], \bot \rangle$,
  where $Y_A = \{y\} \cup vars_c(R)$.

  The notation $[Y_A \leftarrow F_A(\Delta)]$ stands for the substitution $[\ldots y \leftarrow f_y(\Delta) \ldots]$, updating each $y \in Y_A$ to the value $f_y(\Delta)$.

- *Ascertain*

  for $Y_A = vars_c(c)$, and every $\Delta > 0$
  $\langle \textbf{await } c \text{ }, D, E \rangle \xrightarrow{\Delta, A} \langle \textbf{await } c \text{ }, D[Y_A \leftarrow F_A(\Delta)], \bot \rangle$
  provided $D \cup E \not\models c$ and $(D[Y_A \leftarrow F_A(\Delta')] \cup \bot) \not\models c$ for every Δ', $0 < \Delta' < \Delta$.

  The two other timed transitions for *ascertain*, remain as in *Statext*

- *Receive*

  As in *Statext*

- *Idle*

  As in *Statext*

- *Compound*
  Two function assignments A_1 and A_2 are said to be consistent, if for every $y \in (Y_{A_1} \cap Y_{A_2})$, $f_y^{A_1}(t) = f_y^{A_2}(t)$, where $f_y^{A_1}$ and $f_y^{A_2}$ are the functions assigned to y by A_1 and A_2 respectively.

  For $\oplus \in \{\bigsqcup, \|, \mathcal{U}\}$

$$\frac{\langle S_i, D, E \rangle \xrightarrow{\Delta, A_i} \langle S_i', D[Y_{A_i} \leftarrow F_{A_i}(\Delta)], \bot \rangle \qquad \text{for } 1 \leq i \leq 2}{\langle S_1 \oplus S_2, D, E \rangle \xrightarrow{\Delta, A_1 \cup A_2} \langle S_1' \oplus S_2', D[Y_{A_1 \cup A_2} \leftarrow F_{A_1 \cup A_2}], \bot \rangle}$$

provided A_1 and A_2 are consistent.

When A_1 and A_2 are consistent, we can define the assignment $A = A_1 \cup A_2$ by taking $Y_A = Y_{A_1} \cup Y_{A_2}$ and assigning to each $y \in Y_{A_i}$ the function $f_y^{A_i}$.

7.3 Hybrid Statecharts

Timed Statecharts are augmented with a notation that allows to annotate a basic state by a differential equation. The implied meaning is that whenever the state is active, the associated differential equation is operational.

The semantics of *Hybrid Statecharts* is defined by associating with each Hybrid Statecharts specification, a hybrid transition system. The associated transition relation is partitioned into $\mathcal{T}_L$, the untimed transition relation and $\mathcal{T}_{\mathcal{A}}$, the timed relation. As for *Hybrid Statext* , $\mathcal{T}_L$ remains as defined for *Timed Statecharts* , and $\mathcal{T}_{\mathcal{A}}$ has to be redefined, in a way very similar to the definition given for Hybrid *Statext*.

7.4 An Example

To summarize the discussion on Hybrid systems, we give a small example, specified by both a *Hybrid Statecharts* and a *Hybrid Statext.*

The example can be described as follows: at time $t = 0$, a mouse starts running from a certain position on the floor in a straight line towards a hole in the wall, which is at a distance X_0 from the initial position. The mouse runs at a constant velocity V_m. After Δ time units, a cat is released at the same initial position and chases the mouse at velocity V_c along the same path. Will the cat catch the mouse, or will the mouse find sanctuary while the cat crushes against the wall?

The Statechart in Fig. 4 describes the possible scenarios.

The specification (and underlying hybrid transition system) has as continuous variables x_m and x_c, measuring the distance of the mouse and the cat, respectively, from the wall. It refers to the constants X_0, V_m, V_c, and Δ.

The systems starts by the distance variables x_m and x_c being reset to their initial value X_0. Then each of the players proceed in its local behavior. The mouse proceed immediately to the state of running, in which his variable x_m changes continuously according to the equation $\dot{x}_m = -V_m$. The cat waits for a delay of Δ before it enters its running state. Then there are several exits possible. If the event $x_m = 0$ happens first, the mouse wins and moves to its intermediate stage *M.* At this point, the system moves to state *Mouse-Wins.* The other possibility is that the event $x_c = x_m$ occurs first which means that the cat overtook the mouse before the mouse reached sanctuary. In this case they both move to state *Cat-Wins.*

This diagram illustrates the typical interleaving between continuous activities and discrete state changes, which in this example only involved change of control.

In Text form

The top level of the *hybrid Statext* specification is

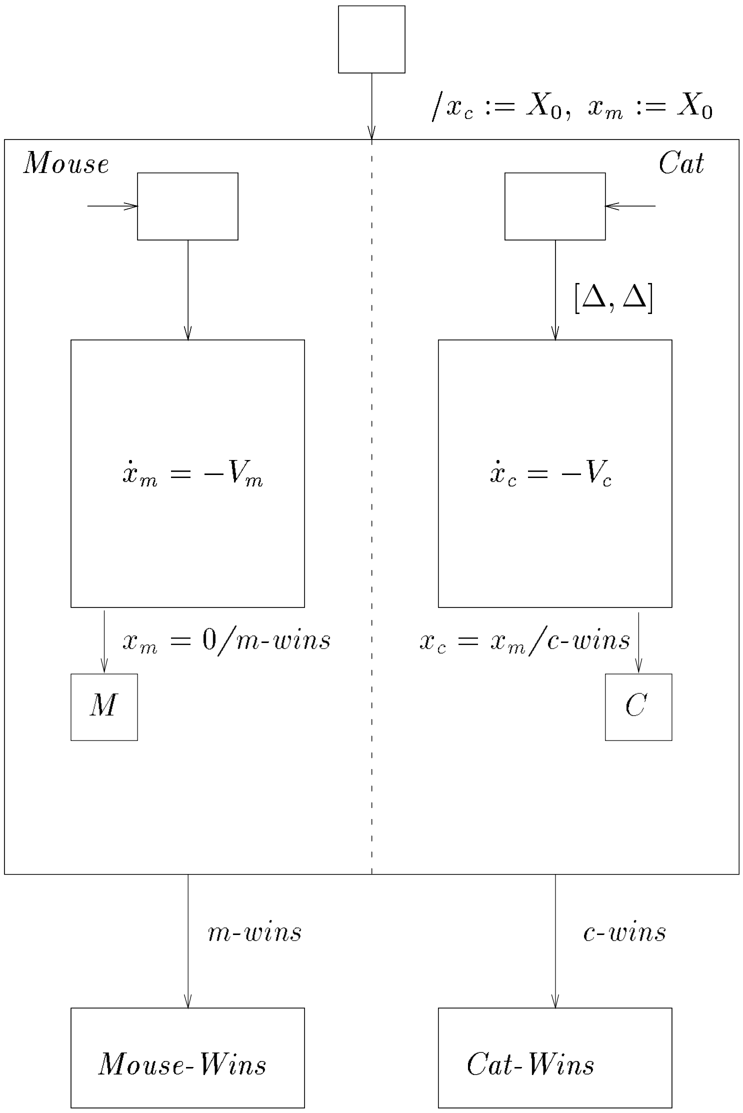


Figure 4: A Hybrid Specification.

$$
Spec :: \quad \left[\begin{array}{c}
\textbf{local } x_c, x_m \textbf{ where } x_c = x_m = X_0 \\[1em]
Mouse \parallel Cat \\
\mathcal{U} \\
\left[\begin{array}{lll}
\textbf{await } c\text{-}wins\ ; & Cat\text{-}Wins: & \textbf{idle} \\
 & \sqcup & \\
\textbf{await } m\text{-}wins\ ; & Mouse\text{-}Wins: & \textbf{idle}
\end{array} \right]
\end{array} \right]
$$

Mouse and *Cat* are processes defined as follows:

$$
Mouse :: \quad \left[\begin{array}{l}
\textbf{skip} \\
\left[\begin{array}{l}
\dot{x}_m = -V_m \\
\mathcal{U} \\
\textbf{await } x_m = 0; m\text{-}wins!\ ; M: \textbf{idle}
\end{array} \right]
\end{array} \right]
$$

$$Cat :: \quad \left[\begin{array}{l} \textbf{delay}[\Delta, \Delta] \\ \left[\begin{array}{l} \dot{x}_c = -V_c \\ \mathcal{U} \\ \textbf{await } x_c = x_m;\, c\text{-}wins! \;;\, C: \textbf{idle} \end{array} \right] \end{array} \right]$$

Note that if $x_m = x_c = 0$, namely, both the cat and the mouse get to the hole simultaneously, the answer to the question who is the winner is nondeterministic.

References

[ACD90] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking for real-time systems. In *Proc. 5th IEEE Symp. Logic in Comp. Sci.*, 1990.

[AH89] R. Alur and T.A. Henzinger. A really temporal logic. In *Proc. 30th IEEE Symp. Found. of Comp. Sci.*, pages 164–169, 1989.

[AL91] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *Real-Time: Theory in Practice*. Lec. Notes in Comp. Sci., Springer-Verlag, 1991.

[DH89] D. Drusinsky and D. Harel. On the power of bounded concurrency I: The finite automata level. submitted, 1989. (Preliminary version appeared as "On the Power of Cooperative Concurrency", in *Proc. Intl. Conf. on Concurrency, Concurrency 88*, Lec. Notes in Comp. Sci. 335, Springer, 1988, pp. 74–103).

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comp. Prog.*, 8:231–274, 1987.

[Har89] D. Harel. A thesis for bounded concurrency. In *Proc. 14th Symp. Math. Found. Comput. Sci.*, pages 35–48. Lec. Notes in Comp. Sci. 379, Springer-Verlag, 1989.

[HGdR88] C. Huizing, R. Gerth, and W.P. de Roever. Modelling statecharts behaviour in a fully abstract way. In *Proc. 13th CAAP*, pages 271–294. Lecture Notes in Comp. Sci. 299, Springer-Verlag, 1988.

[HH90] T. Hirst and D. Harel. On the power of bounded concurrency II: The pushdown automata level. In *Proc. 15th Coll. Trees in Algebra and Programming*. Lec. Notes in Comp. Sci., Springer-Verlag, 1990.

[HK89] D. Harel and H.A. Kahana. On statecharts with overlapping. Technical report, The Weizmann Institute, 1989.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. Software Engin.*, 16:403–414, 1990.

[HMP90] T. Henzinger, Z. Manna, and A. Pnueli. An interleaving model for real time. In *5th Jerusalem Conference on Information Technology*, pages 717–730, 1990.

[HMP91] T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *Proc. 18th ACM Symp. Princ. of Prog. Lang.*, pages 353–366, 1991.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.

[HPSS86] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 54–64, 1986.

[Hui91] C. Huizing. *Semantics of reactive systems: comparison and full abstraction.* PhD thesis, Technical University Eindhoven, 1991.

[Mar90] F. Maraninchi. *Argos: un langage graphique pour la conception, la description et la validation des systèmes réactifs.* PhD thesis, University of Grenoble, 1990.

[Mar91] F. Maraninchi. Languages for reactive systems: a common framework for comparing statecharts and argos. Technical report, LGI-IMAG Grenoble, 1991.

[MMP92] O. Maler, Z. Manna, and A. Pnueli. A formal approach to hybrid systems. In *Proceedings of the REX workshop "Real-Time: Theory in Practice"*, LNCS. Springer Verlag, New York, 1992.

[NRSV90] X. Nicollin, J.-L. Richier, J. Sifakis, and J. Voiron. Atp: an algebra for timed processes. In *Proc. IFIP Working Conference on Formal Description of Programming Concepts, Tiberias, Israel.* North-Holland, 1990.

[NSY91] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. In *Real-Time: Theory in Practice.* Lec. Notes in Comp. Sci., Springer-Verlag, 1991.

[Ost89] J.S. Ostroff. *Temporal Logic for Real-Time Systems.* Advanced Software Development Series. Research Studies Press (John Wiley & Sons), Taunton, England, 1989.

[PH88] A. Pnueli and E. Harel. Applications of temporal logic to the specification of real time systems. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 84–98. Lec. Notes in Comp. Sci. 331, Springer-Verlag, 1988.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical report, Dept. of Comp. Sci., Arhus University, 1981.

[PS89] A. Pnueli and M. Shalev. What is in a step? In J.W. Klop, J.-J.Ch. Meijer, and J.J.M.M. Rutten, editors, *J.W. De Bakker, Liber Amicorum*, pages 373–400. CWI, AMsterdam, 1989.

[RR87] G.M. Reed and V.W. Roscoe. Metric spaces as models for real-time concurrency. In *Mathematical Foundations of Programming*, pages 331–343. Lec. Notes in Comp. Sci. 298, Springer-Verlag, 1987.