Modelling Statecharts behaviour in a fully abstract way

Modelling Statecharts behaviour in a fully abstract way

by

C. Huizing
R. Gerth
W.P. de Roever

88/07

April 1988

COMPUTING SCIENCE NOTES

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB Eindhoven
The Netherlands
All rights reserved
editor: F.A.J. van Neerven

Modelling Statecharts behaviour in a fully abstract way

Conference version

C. Huizing
R. Gerth
W.P. de Roever
Eindhoven University of Technology[†,‡]

ABSTRACT

We present a denotational, strictly syntax-directed, semantics for Statecharts, a graphical, mixed specification/programming language for real-time, developed by Harel [H]. This requires first of all defining a proper syntax for the graphical language. Apart from more conventional syntactical operators and their semantic counterparts, we encounter unconventional ones, dealing with the typical graphical structure of the language. The synchronous nature of Statecharts makes special demands on the semantics, especially with respect to the causal relation between simultaneous events, and requires a refinement of our techniques for obtaining a denotational semantics for OCCAM [HGR]. We prove that the model is fully abstract with respect to some natural notion of observable behaviour. The model presented will serve as a basis for a further study of specification and proof systems within the ESPRIT-project DESCARTES.

1. Introduction

Statecharts belong together with Esterel [B], LUSTRE [BCH], SIGNAL [GBBG] and an unknown number of local industrial concoctions to the group of mixed specification/programming languages used in development of real-time embedded systems.

Some of these languages (LUSTRE, SIGNAL, Esterel) have no internal notion of time. An external signal must be provided as a clock and the system can use it as it likes to. Hence, various clock operations can be specified. The disadvantage of this approach is that time constraints and other specifications w.r.t. the time are not clearly visible in the specification/ program. Statecharts adopts the view that these specifications should be visible and hence has an internal notion of time.

Statecharts adopts, like Esterel, the synchrony hypothesis as formulated by Berry [B]. This means that output occurs simultaneously with the input that caused it. If applied without care, this hypothesis can lead to casual paradoxes, such as events disabling their own cause. In Esterel, these paradoxes are circumvented by *syntactically* forbidding situations in which they can arise[1]. In Statecharts, they are *semantically* impossible, because

†) Eindhoven University of Technology, Department of Mathematics and Computing Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands. E-mail: mcvax!eutrc3!wsinkees.UUCP or: wsdckeesh@heithe5.BITNET
‡) This research was carried out in the context of ESPRIT-project 937, DESCARTES (Debugging and Specification of Ada Real-Time Embedded Systems).
1) Recently, the semantics of Esterel has been changed towards a more semantical check upon paradoxes [G].

there the influence of an event is restricted to events that did not cause it. We expect that the semantics of Esterel and Statecharts coincide in the situations that are allowed by Esterel. The problem is to model causality between events that have no precedence in time. In the operational semantics of [HPSS], this is done by introducing the notion of *micro-steps*. Every time step is subdivided into micro-steps between which only a causality relation holds and no timing relation. On the level of the denotational semantics this is done by applying a partial order on the events that occur simultaneously. This order describes in which direction events influence each other.

Another problem that arises in giving a compositional semantics of Statecharts, is its graphical nature. For textual languages, defined by means of a proper syntax, it is clear what is demanded of a syntax-directed semantics. It has to be compositional (a homomorphism) with respect to the syntactical operators. For a graphical language, without a proper syntax, this is not so clear.

Hence, in chapter 3 we first define a syntax of Statecharts that makes use of a restricted set of natural operators and primitive objects. These objects and the immediate results of applications of operators slightly generalise statecharts, by allowing transitions to be incomplete, i.e., to have no origin states or no target states yet.

Some syntactical operators lack a clear counterpart in conventional languages. This is because in the graphical representation of Statecharts, the notion of area plays an important role, as it defines a hierarchy of states. Subareas of states are associated with alternative activities or concurrent activities. Transitions leaving a superstate influence the behaviour in all its substates (which are lower in hierarchy). This leads to a semantics in which it is possible to extend the behaviour of some subchart with the behaviour of the state that is put higher in hierarchy.

Unlike Esterel, Statecharts does not have a restricted kernel of operations, in terms of which all other features are defined. The designers of Statecharts adopt the view that handy operations should be provided as long as they can be built in. As a consequence, we had to study a restricted version of Statecharts. A next version of this paper will include the use of variables.

The semantics that we develop in chapter 4 is compositional w.r.t. the above syntax. The domain in which Statecharts acquire meaning basically records computations as functions that associate to every time point a record, $(F, C, \leq)$, that represents the activity at that time. Such a record states the *claims*, C, (or assumptions) about which events are generated, both in the statechart and in its environment; it specifies the *fact* that the events in F $(\subseteq C)$ are generated by the Statechart itself and, finally, it records in the partial order $\leq$ on C which events influence the occurrence of which other events.

This semantics turns out to be fully abstract relative to a notion of observation that observes about any statechart only the events that are generated by that statechart. The full abstraction proof is sketched in chapter 5.

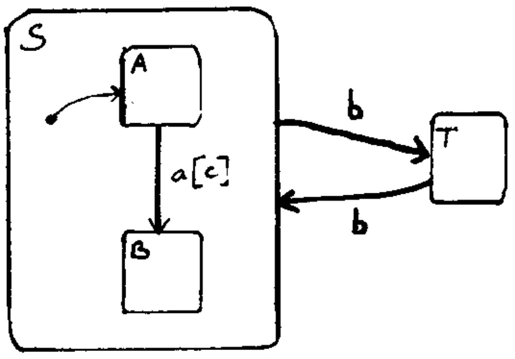
2. Informal introduction to Statecharts

We give a short description of the language Statecharts and an intuitive semantics. For a more basic treatment of this, one is referred to [H] and [HPSS].

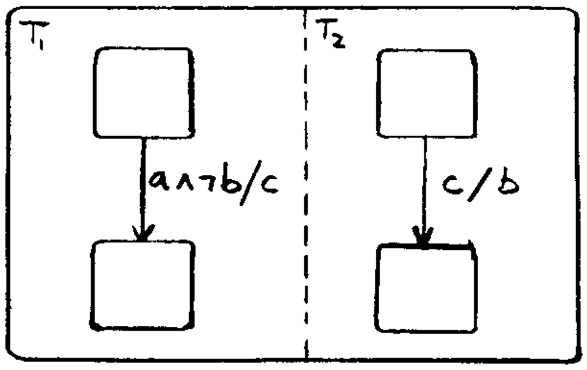
Statecharts is a formalism designed to describe the behaviour of *reactive systems* [HP]. A reactive system is a mainly event-driven system, continuously reacting to external and internal stimuli. In contrast to *transformational systems*, that perform transformations on inputs thus producing outputs, reactive systems engage in continuous interactions, *dialogues* so to say, with their environment. As a consequence, a reactive system cannot be modelled by giving its input and output alone. It is necessary to model also the timing or causality relation between input and output events.

Statecharts generalise Finite State Machines (FSM's), or rather Mealy machines [HU], and arise out of a conscious attempt to free FSM's from two serious limitations: the absence of a notion of hierarchy or modularity and

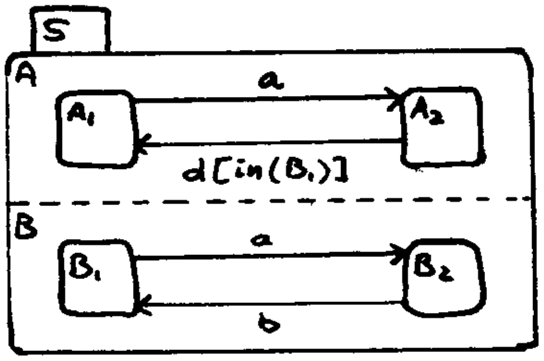
Example 1



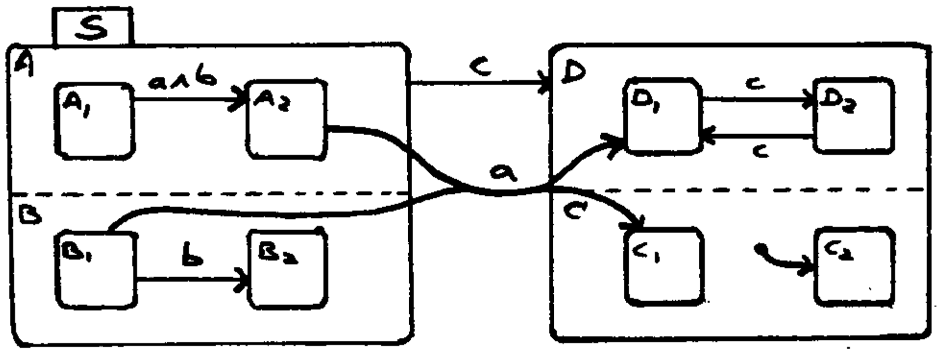
Example 5



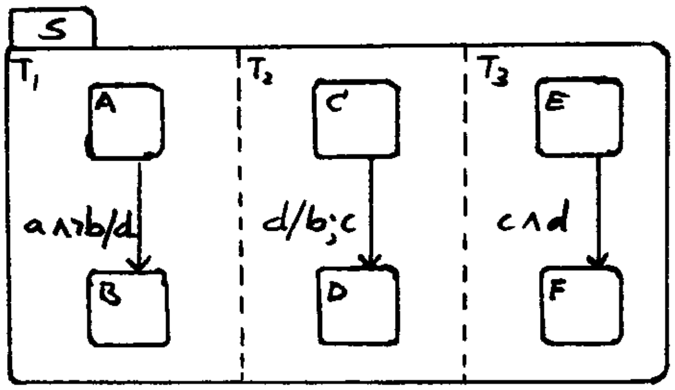
Example 2



Example 3



Example 4



the ability to model concurrent behaviour in a concise way. The external and internal stimuli are called *events* and they cause transitions from one state to the other. We introduce the basic conceptions now.

2.1. States

In contrast to FSM's, states can be structured as a tree. We call the descendants in such a tree *substates*. A state can be of two types: AND or OR. Being in an OR-state implies being in one of its immediate substates, being in an AND-state implies being in all of its immediate substates at the same time. The latter construction describes concurrency.

Example 1 (see overleaf)

In this picture S is an OR-state with substates A and B. Being in state S implies being in A or B, but not in both. A, B and T have no substates, a and b stand for events that trigger transitions and c is a condition. E.g., the transition from A to B is triggered when event a occurs and condition c is true. These events are called *primitive events*, because they have no further structure. They can be generated outside the system, but also by the system itself.

When the system is in A and event a happens and condition c is true, A will go to state B, and also stay in S. Whenever it is in A or B and b happens it will go to T. The transition to A is a *default* transition. When the system is in T and b happens, it will go to S and hence to A.

Example 2 (see overleaf)

Now, S is an AND-state with immediate substates A and B. A and B are OR-states with substates A_1 and A_2 respectively B_1 and B_2. Being in S implies being in A and B simultaneously. When the system is in A_1 and B_2 (and hence also in A, B and S) and b happens it will go to B_1 and also stay in A_1.
Now, if a happens, it will go *simultaneously* to A_2 and B_2. Notice also the condition $in(B_1)$ on the transition from A_2 to A_1. This transition can only be taken if and when the system is in A_2 and B_1 and event d occurs.

2.2. Transitions

In the examples above we used simple transitions from one state to another like in FSM's.
They can be more complicated, however, going from a set of states to a set of states.

Example 3 (see overleaf)

When the system is in A_2 and B_1 and a happens, it will go to T, and in particular to G and D_1. This is the general case. In this version of the paper, however, we don't allow transitions leaving more than one state. We do allow, however, transitions *entering* more than one state.

Notice the *compound event* on the transitions from A_1 and A_2. Only when a and b occur simultaneously this transition will be triggered.

2.3. Actions

In the label of a transition one can specify some events that are generated when the transition is performed. This is called the *action* of a transition. These events immediately take effect and can trigger other transitions.

Example 4 (see overleaf)

When the system is in A, C and E and a occurs, a chain reaction of transitions will be performed. The transition in T_1 will generate event d; this event will trigger the T_2-transition, which, on its turn, will generate b and c and thus trigger the T_3-transition.

All transitions that are triggered by such a chain reaction are considered to happen at the same time. So, in this example, the next state configuration after (A, C, E) is (B, D, F). But see the section on causality.

2.4. Events

In general, the event in the label of a transition has the form of a logic proposition, using conjunction, disjunction and negation. A transition labelled $a \wedge b$ can be taken when a and b occur in the same time step; if the label is $a \vee b$, it can be taken as soon as a or b occurs; a transition labelled with $\neg a$ can be taken at any time step in which a does not occur. In these formulae, one can use primitive events $a, b, c...$, but also the structured events *enter(S)* and *exit(S),* denoting the event of entering respectively exiting state S.

Another structured event is the *time-out* event. The expression *time-out(e,n)* stands for the time-out of n time units on event e. A transition labelled with this expression will be triggered when the last occurrence of e was exactly n times ago. One time unit stands for the time that it costs to take one transition or one chain reaction of transitions. In this version of Statecharts a specification should go with an additional specification relating time units and physical time.

Events are instantaneous and transient of nature, such in contrast to the conditions, which represent a more continuous situation. E.g., the event *enter*(S) can only be sensed at the time unit when state S is entered, but the condition *in*(S) is true throughout the time that the system is in the state S, in other words between the occurrence of *enter*(S) and *exit*(S).

2.5. Causality

As already mentioned above, transitions can trigger other transitions and all these transitions occur simultaneously. Together with the possibility of negation of events and conditions, this can raise causal paradoxes.

Example 5 (see overleaf)
The transition labelled with $a \wedge \neg b$ will be triggered when a occurs and b does not occur. This transition generates an event, c, that triggers another transition which, in its turn, generates b. All transitions in this chain reaction are considered to be happening *at the same time*. So b *did* happen and the first transition could not occur, hence the whole chain reaction did not occur, hence... . These kinds of paradoxes are avoided by giving the following operational interpretation to chain reactions. This is taken from [HPSS].

Every time step is subdivided into *micro-steps,* each of which corresponds to the execution of one transition. The events that are generated by a transition can only influence transitions in the following micro-steps. So in the example above, the T_1-transition takes place in the first micro-step, triggering the T_2-transition in the second micro-step. This one generates the events b and c, but these cannot prevent the T_1-transition any more, because the latter has taken place in a previous micro-step.

We stress that the micro-steps have nothing to do with time. Their sequential occurrence is only related to the way they can influence each other – no order in time is implied. Maximal sequences of micro-steps are called *macro-steps;* a macro-step corresponds to one step in time. Here, maximal means that the sequence of micro-steps cannot be expanded without additional input from the environment. Hence, in example 4 above, the sequence consisting only of the T_1-transition is not maximal, because the T_2-transition is still possible.

3. Syntax

In this chapter we give a non-graphical syntax of statecharts. According to this syntax any statechart is built up from primitive objects and some operators. These operators have a natural relationship with the pictures. The intermediate objects to which the operators are applied are the so-called *Unvollendetes* or *Unvs*. These are incomplete statecharts with transitions without source state(s) or target state(s). Two operators, *Concatenation* and *Connection*, can tie these dangling arrows together, thus creating complete transitions.
Concatenation makes a complete transition between two Unvollendetes and resembles sequential composition. Connection makes a complete transition *within* one subchart, thus possibly creating loops.

In Statecharts, there are two types of states: the AND-type and the OR-type. Being in an AND-state means being in all of its immediate substates simultaneously. We call these immediate substates and their interior the *orthogonal components* of that AND-state. Being in an OR-state means being in exactly one of its substates. The Unv that builds the interior of an AND-state respectively OR-state is called an *AndChart* respectively *OrChart*.

Statification is the operator that builds the hierarchical structure of statecharts. It puts an Unv inside a primitive state, i.e., a state without substates, thus creating a structured AND- or OR-state. Semantically, it means executing the subchart inside, with the possibility of interrupting this execution when one of the (incomplete) transitions leaving the superstate are triggered.

AndCharts and OrCharts are built using the operators *Anding* and *Orring*. Anding corresponds to parallel composition in conventional programming languages. Orring can be compared to non-deterministic choice.

Finally, *Closure* gives the events that are considered internal for the particular subchart, which means that the statechart will ignore such events whenever they are generated by its environment. *Hiding* makes the events that are generated inside a statechart or Unvollendete invisible to the outside world. In this sense they are dual. Neither operator has a graphical counterpart in the language as defined in [HPSS].

In the Appendix we give the formal relationship between the objects generated by the syntax and the formal objects representing statecharts as defined in [HPSS].

3.1. Transition Labels

We define the labels that can be associated with transitions. Let a set of *elementary events* E_e and a *set of states* Σ be given. Define the set of *primitive events* $E_p = E_e \cup \{enter(S), exit(S) \mid S \in \Sigma\}$

Definition.
The set of **events** E is inductively defined by

$\lambda \in E$, the *null* event;
$e \in E_p \rightarrow e \in E$;
$e_1, e_2 \in E_p \rightarrow e_1 \wedge e_2,\ e_1 \vee e_2 \in E$;
$e \in E \rightarrow \neg e \in E$;
$n \in \mathbb{N} \setminus \{0\}, e \in E \rightarrow time-out(e,n) \in E$ $\square$

Remarks: $\neg e$ is here considered as an *event*, in contrast to [HPSS] where it is a condition. Semantically they are the same, i.e. we also have the "not yet" interpretation.

We abbreviate $enter(S)$, $exit(S)$ and $time-out(e,n)$ by respectively $en(S)$, $ex(S)$ and $tm(e,n)$.

Definition.
The set of **conditions** C is inductively defined by

$true, false \in C$;
$c_1, c_2 \in C \rightarrow c_1 \wedge c_2, c_1 \vee c_2 \in C$;
$c \in C \rightarrow \neg c \in C$;
$S \in \Sigma \rightarrow in(S) \in C$ $\square$

Definition.
The set of **actions** A is inductively defined by

$\mu \in A$, the *null* action;
$e \in E_p \rightarrow e \in A$;
$a_i \in A$ for $i=1,\ldots,n \rightarrow a_1,\ldots,a_n \in A$

□

Definition.

Lab $= \{ e[c]/a \mid e \in E, c \in C, a \in A \}$

If $e = \lambda$, $c = true$ or $a = \mu$, we often omit that part of the label. □

3.2. Unvollendetes

Providing a syntax for Statecharts is done using a notion of ***incomplete statechart*** or ***Unvollendete,*** abbreviated as *Unv*. This is a statechart in the process of being built up. It differs from a complete statechart in that it need not have a unique root state (i.e. a state of which all other states are direct or indirect substates) and that it may have so-called incomplete transitions. Incomplete transitions are transitions either without source or without target state(s). These transitions are pictured as dangling arrows. Any statechart can be broken up into Unvollendetes and in Chapter 4 we will give the semantics of these Unvollendetes.

We distinguish two kinds of Unvollendetes. The basic Unvs that cannot be decomposed are called *Primitives*. They consist of one state with some incomplete transitions. They are, together with the operators the terminal symbols of the syntax. We denote them by

$$Prim(I, O, A)$$

where A is the name of a state, I and O a set of incoming respectively outgoing transitions. The other three types of Unvs form the non-terminal symbols of the syntax. *PrimCharts* are Unvs with one root state. A complete statechart is an example of a PrimChart without incoming or outgoing transitions. *AndCharts* form the interior of an AND-state. The operators *Connection* and *Concatenation* cannot be applied to them. *OrCharts* form the interior of OR-states and furthermore all Unvs that are not the interior of an AND-state. Apart from *Anding*, all operators can be applied to them. The structure of the non-terminals is for all three types the same:

$$PrimChart(I, O), \quad AndChart(I, O), \quad OrChart(I, O)$$

where I and O again denote a set of incoming respectively outgoing transitions.

Definition

Let T_I respectively T_O be the set of all incoming respectively outgoing transitions; $T_I \cap T_O = \emptyset$.
Let $e \in E_e \cup \Sigma$, $I, \ldots \subseteq T_I$, $i, \ldots \in T_I$, $O, \ldots \subseteq T_O$, $o, \ldots \in T_O$ and $L : T_O \to$ **Lab**.
Then the set of **Statecharts** is defined by

Stch $= \{ V \mid B \overset{*}{\to} V \}$

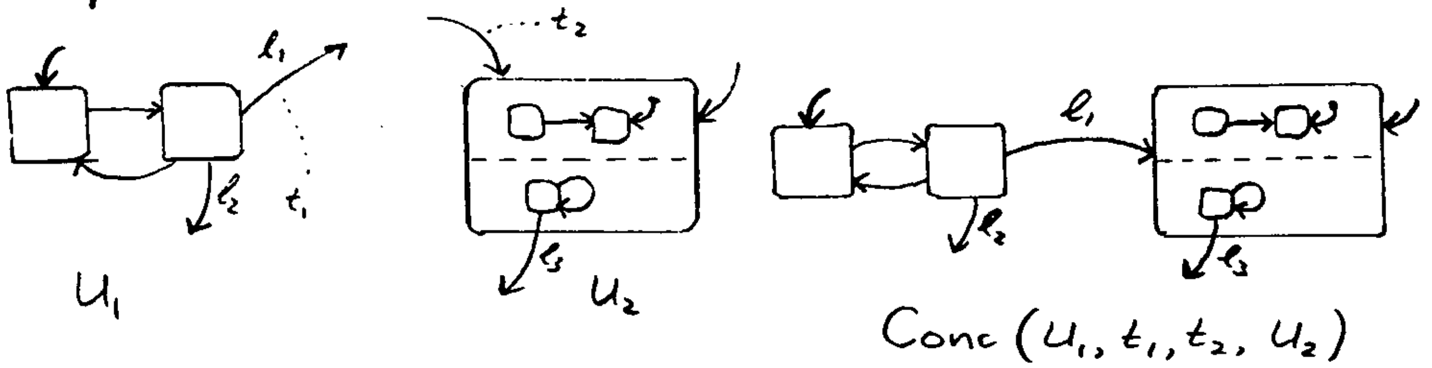
and $\overset{*}{\to}$ is the derivability relation for the following set of rules:

$B \to PrimChart(\emptyset, \emptyset)$

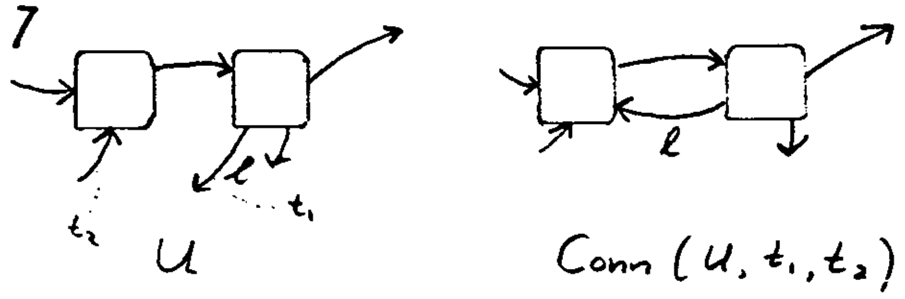
$PrimChart(I, O) \to$ **Prim**(I, O, A)
$PrimChart((I_1 \cup I_2) \setminus \{i\}, O_1 \cup O_2) \to$ **Stat**(**Prim**$(I_1, O_1, A), OrChart(I_2, O_2), i)$ with $i \in I_2$
$PrimChart(I_1 \cup I_2, O_1 \cup O_2) \to$ **Stat**(**Prim**$(I, O, A), AndChart(I_2, O_2))$

$OrChart(I, O) \to PrimChart(I, O)$
$OrChart(I, O) \to$ **Close**$(OrChart(I, O), e)$
$OrChart(I, O) \to$ **Hide**$(OrChart(I, O), e)$
$OrChart(I_1 \cup I_2, O_1 \cup O_2) \to$ **Or**$(OrChart(I_1, O_1), OrChart(I_2, O_2))$
$OrChart((I_1 \cup I_2) \setminus \{i\}, (O_1 \cup O_2) \setminus \{o\}) \to$ **Conc**$(OrChart(I_1, O_1), o, i, OrChart(I_2, O_2))$
with $o \in O_1$ and $i \in I_2$

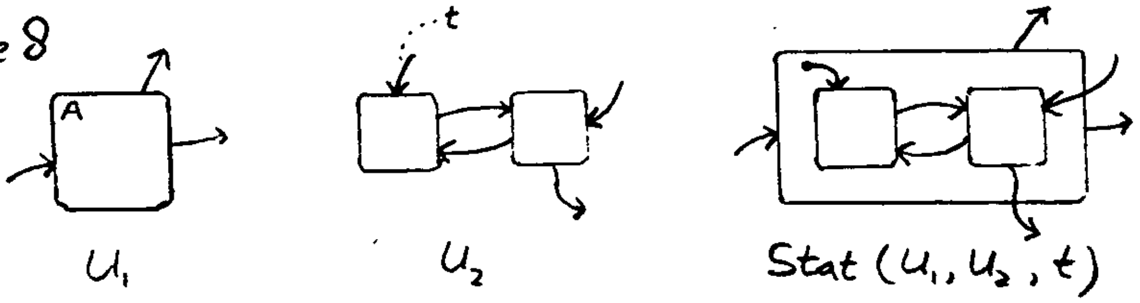
Example 6



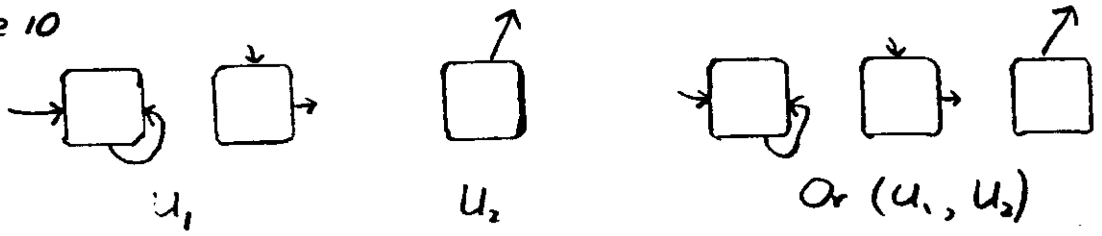
Example 7



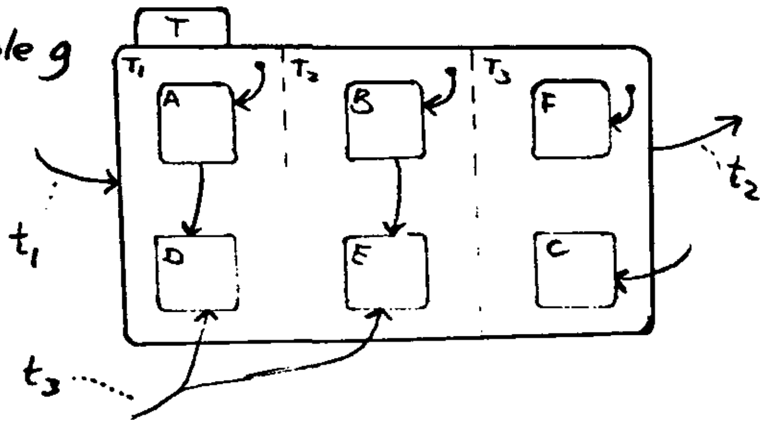
Example 8

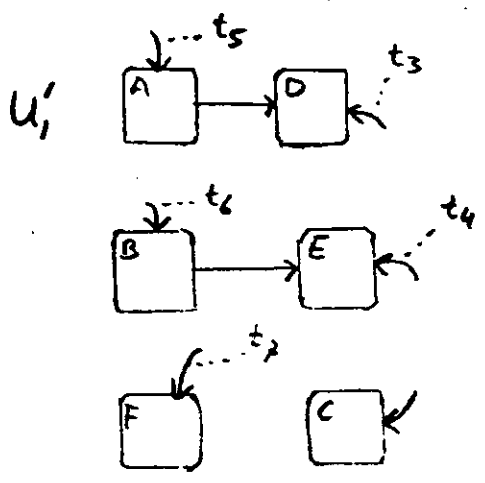


Example 10



Example 9





$Stat(Prim(\{t_1\}, \{t_2\}, T)$

$U = And(And(U_1, U_2, \{(t_3, t_4)\}), U_3, \phi)$

$U_1 = Stat(Prim(\phi, \phi, T_1), U_1', t_5)$
$U_2 = Stat(Prim(\phi, \phi, T_2), U_2', t_6)$
$U_3 = Stat(Prim(\phi, \phi, T_3), U_3', t_7$

$OrChart(I \setminus \{i\}, O \setminus \{o\}) \rightarrow$ **Conn**$(OrChart(I,O), o, i)$ with $o \in O$ and $i \in I$

$AndChart(I,O) \rightarrow PrimChart(I,O)$

$AndChart(I,O) \rightarrow$ **Close**$(AndChart(I,O), e)$

$AndChart(I,O) \rightarrow$ **Hide**$(AndChart(I,O), e)$

$AndChart((I_1 \cup I_2) \setminus \{i_1', \ldots, i_n'\}, O_1 \cup O_2) \rightarrow$ **And**$(AndChart(I_1,O_1), AndChart(I_2,O_2), \{(i_1,i_1'), \ldots, (i_n,i_n')\})$
with $i_i \in I_1$ and $i_i' \in I_2$ □

3.2.1. Explanation of the operators

Concatenation $Conc(U_1, o, i, U_2)$

By concatenation two OrCharts are "sequentially composed". An outgoing transition, o, of U_1 is connected to an incoming one, i, of U_2, thus creating a complete transition. (See example 6 overleaf).

Connection

Connection only differs from concatenation by taking just one chart and making the new transition somewhere inside. In fact we don't need concatenation if we have connection and orring (see below), but from the semantic point of view, concatenation is more basic. (See example 7 overleaf).

Statification

This is the hierarchy operator; it has no counterpart in conventional programming languages. It puts an OrChart (U_2) inside a state A (the state of a primitive U_1). An explicitly mentioned incoming transition, i, from U_2 becomes the default of A. (See example 8 overleaf). If U_2 is an AndChart, the default is left out, because an AND-state needs no default starting point: execution is started in all immediate substates simultaneously. Of course, these substates can have defaults associated with them.

Anding

Anding in Statecharts corresponds to parallel composition in conventional programming languages. Two *AndCharts* are put in parallel. Through *Statification*, they will become the orthogonal components of an AND-state. (See example 9 overleaf). Anding is a binary operator, so if there are to be more than two orthogonal components, it must be applied repeatedly. The semantic counterpart of Anding is associative and commutative. Note that the orthogonal components of an And-state are always *PrimCharts* (charts with one root state). Forked transitions are made by specifying which of the incoming transitions of the operands should be combined. Repeatedly applying *Anding* and combining forked transitions creates forks with more than two target states.

Orring

This is the counterpart of Anding. It puts some *OrCharts* together in non-orthogonal composition, with the intention of statification by an OR-state and can be compared to non-deterministic choice. (See example 10 overleaf).

Closure

In [HPSS], the set of primitive events is divided into internal and external events. External events can be generated outside the statechart itself, internal events cannot. For a compositional semantics this distinction is not useful, because events that are internal to the complete statechart, can be external to some subchart.

Therefore, we introduce an operator that declares some events to be internal to a subchart meaning that such a subchart will not react if one of its internal events is generated outside. This is not the same as hiding since these events are still observable.

Hiding

The hiding operator makes the specified events invisible for the outside world. *Hiding* and *Closure* are in a sense dual. Hiding restricts the influence of the operand on the environment, and maintains the influence of the environment on the operand, whereas *Closure* restricts the influence of the environment on the component, but maintains the influence of the component on the environment. If *Hiding* is muting, then *Closure* is deafening. They can be seen as a consequence of the broadcast communication mechanism. The conventional hiding operation, i.e., making an event or variable fully local, can be obtained by applying both Closure and Hiding to a component.

4. Semantics

This chapter presents a denotational semantics of statecharts or rather of Unvs. This semantics is compositional (syntax-directed) with regard to the operators defined in Chapter 3.

The maximality of the sequences of micro-steps as described in Chapter 2 corresponds with the notion of maximal parallelism as modelled in [HGR,GB] (see also [SM]). The techniques of those papers also apply here.

As Statecharts describes a set of configurations (as any digital system), a discrete model of time is adequate. Since it is intended to make global time specifications, we use a global notion of time. The simplest domain that gives these properties is $\mathbb{N}$, but for reasons that will be explained later, we use $\mathbb{Z}$.

At first sight, Statecharts are quite different from ordinary programming languages. Simplest to characterise are sequential languages without jump-like constructs. Once jumps enter the picture we have to abandon the idea of giving state transformations for each command in isolation. Traditionally, this is solved using the idea of continuations [SW,M].

It is our aim to give a *compositional* semantics of Statecharts. The semantics of [SW] is only given for full program blocks in which all labels of gotos appear. In our solution jumps (transitions) are made in two stages. In the first stage we have only half jumps, in which the place where we are jumping to or where we come jumping from is not specified. These are the incomplete transitions in the syntax.

In the semantics, we record the behaviour of a subchart only between such jumps. And we specify for each history the incomplete transition by which it starts and by which it ends. This specification is just the syntactical identification of the transition.

In the second stage, by concatenation or connection these half jumps are made into full jumps by identifying an incoming and an outgoing transition. Now we can also give the full semantics of the jump, as we know where we come from and where we go to. This semantics is just the concatenation of the history that ends in one half of it and the history that starts with the other half. In case of connection, loops can arise since we jump to the same subchart. Consequently, the semantics of this construct will be characterised by a fixed-point equation.

Now there is a difference between gotos in conventional languages and transitions in Statecharts, namely, in Statecharts the place where a jump can occur is not completely syntactically determined. Transitions from a superstate can be triggered when execution is anywhere inside that state. Our solution is to give two options at any moment during execution inside a state: exiting by the outside transition or continuing the history generated by the semantics of the interior of the state.

4.1. The semantic domain

The semantics of an (incomplete) statechart, i.e., its denotation, will be a set of *history-triples*, each triple corresponding to one possible execution.

The set of history-triples is defined by

$$(T_I \cup \{*\}) \times \mathbb{H} \times (T_O \cup \{\perp\})$$

where T_I and T_O are the sets of incoming respectively outgoing transition identifiers from the syntax and $\mathbb{H}$ denotes the set of *histories*, defined below.

A history-triple consists of three components. The first component is the incoming transition of the chart by which the execution starts, the third component either equals the outgoing transition by which the execution ends, or equals "$\perp$" in case of an incomplete computation. It is possible that there is no starting transition, indicated by $*$. This is the case when we have the root state of the complete statechart, or a component of an AND-state that can be started implicitly by an incoming transition of another component (see fig).

The second component of the triple is a *partial* function that associates to each time unit, a so-called clock record. Execution starts at time unit 0 and ends at the last time unit where the function is defined. The records associated to negative time values contain information about the past, i.e., before the execution of this subchart started. We will need this to describe the occurrence of time-out events. The functions are total on $\mathbb{Z}^{<0}$.

$$\mathbb{H} = \{f \in \mathbb{Z} \to \mathbb{C} \mid \exists i \forall j : j<i \to f(j) \text{ defined} \wedge j \geq i \to f(j) \text{ undefined}\}$$

The precise structure of clock-records, $\mathbb{C}$, is defined later.

Notation:

Let $f : \mathbb{Z} \to X$ be a partial function. Then:

- $|f| = \max(\{i \mid f(i) \text{ is defined}\}) + 1$
  If $f \in \mathbb{H}$, then $|f|-1$ is the time at which the outgoing transition, if there is one, of this execution occurs.
- the *shift* operator changes the time in a history; it shifts each clock record j time units to the future.
  $shift(f, j)(i+j) = f(i)$
  $|shift(f, j)| = |f| + j$
- $f \upharpoonright n : \mathbb{Z} \to X$ is defined by
  $(f \upharpoonright n)(i) = f(i)$ if $i < n$
  $=$ undefined otherwise
- For $f \in \mathbb{H}$ define the *projections* f^F, f^C, $f^{\leq}$ and f^S by:
  $f(i) = (f^F(i), f^C(i), f^{\leq}(i))$ and $f^S(i) = (f^C(i), f^{\leq}(i))$ □

In order to use fixed-point definitions, our domain will be a complete partial order (cpo). In fact, we will use the standard Hoare ordering as in [K&] and represent it, as usual, as inclusion of prefix-closed sets.

We distinguish *extendable* and *finished* history-triples. Extendable triples correspond with incomplete computations and are characterised by a bottom outgoing transition ($\perp$). We define the following partial order on history-triples:

Definition

$(t_1, f, t_2) \leq (t'_1, f', t'_2)$ iff $t_1 = t'_1 \wedge ((t_2 = \perp \wedge |f| \leq |f'|) \vee (t_2 = t'_2 \wedge |f| = |f'|)) \wedge \forall i < |f| : f(i) = f'(i)$ □

If $h_1 \leq h_2$ we say that h_1 is a *prefix* of h_2

Definition

- A set of history-triples H is *"prefix-closed"* iff $\forall h \in H : h' \leq h \rightarrow h' \in H$
- The function $\bullet^{CL}$ maps a set of history-triples, H, into the smallest prefix-closed set that contains H. ▯

The semantical domain is defined as follows:

Definition

The domain is $(\mathbf{D}, \leq, \perp_{\mathbf{D}})$, where $\mathbf{D} = \{H \subseteq T_I \cup \{*\} \times \mathbf{H} \times T_O \cup \{\perp\} \mid H$ is prefix-closed$\}$ and $\perp_{\mathbf{D}} = \emptyset$ ▯

Theorem

$(\mathbf{D}, \leq, \perp_{\mathbf{D}})$ is a cpo. ▯

Proof:

Standard. ▯

The set of *clock records* is defined as follows:

Definition:

$\mathbb{C} = \{(F, C, \leq) \mid F \subset C \subseteq E_p, \leq$ a partial order on $C\}$ ▯

For one particular time-step, a clock record describes the behaviour of the total system (component *and* environment) by C and $\leq$, a partial ordering on C. The contribution of the component is contained in the set F.

F is the set of events that are generated by the component and C is the set of events that are assumed to be generated somewhere in the total system (including the component).

Unfortunately this information is not sufficient. A transition can influence other transitions in the same time step either by triggering them or by preventing them from being triggered. This influence, however, is restricted. A transition can only influence transitions that occur in subsequent micro-steps. This is the way causal paradoxes are avoided.

We have to record this restricted influence, too. This leads to the following additional information.

> A partial order on the events that occur in the same time step representing the way such events can influence each other. E.g., if event a causes transition t, then we have $a<b$ for all events b that are generated by transition t. This means that t can never influence transitions that caused a. These relationships can also arise from negative causes: if a transition labelled $a \wedge \neg b$ is taken, we have $b < a$, because taking such a transition is only possible if a occurs when b has not been generated (yet).

Example11 (see figure in chapter 5).

> If the two transitions occur simultaneously, we have $b<a$ in all behaviours. This means that the T_2-transition cannot trigger the T_1-transition, even though it generates b. The trigger of the latter transition has to come from somewhere else.

The relationship between the partial order and the micro-steps is as follows.

$a<b$ if and only if a occurs in a micro-steps previous to that in which b occurs.

$a \sim b$ (abbreviation for $a \not\leq b \wedge b \not\leq a$) if and only if a and b occur in the same micro-step.

If an event is generated in more than one micro-step in the same time-step, we only take the first occurrence into account, since an event is effective during all micro-steps following the micro-step in which it is generated.

4.2. Semantics of transitions

Before we define the semantics of subcharts, we define a function that gives the semantics of *transitions*. All behaviours that are consistent with taking some transition, are expressed by the function T.

Definition

$T_0 : E \rightarrow 2^{2^{E_p}}$ is defined recursively as follows:

$T_0(\lambda) = 2^{E_p}$

$T_0(a) = \{C \subseteq E_p \mid a \in C\}$ for $a \in E_p$

$T_0(\neg e) = \{C \subseteq E_p \mid C \notin T_0(e)\}$

$T_0(e_1 \wedge e_2) = T_0(e_1) \cap T_0(e_2)$

$T_0(e_1 \vee e_2) = T_0(e_1) \cup T_0(e_2)$

$T_0(tm(e,n)) = T_0(\lambda)$ □

$T_0(e)$ gives all sets of events that may occur when a transition labelled with $e/...$ takes place. This is not sufficient for time-out events, for which the past is also relevant. Therefore we extend T_0 to the function T that also gives all past histories that are consistent with the transition taking place.

Definition

$T : E \rightarrow 2^{\mathbb{Z} \rightarrow 2^{E_p}}$ is defined recursively as follows:

$T(e) = \{f \mid f(0) \in T_0(a) \wedge |f| = 1\}$ for $e \in E_p$ or $e = \lambda$

$T(\neg e) = \{f \mid f \notin T(e) \wedge |f| = 1\}$

$T(tm(e,n)) = \{f \mid shift(f,n) \restriction 1 \in T(e) \wedge \forall 0 < i < n : shift(f,i) \restriction 1 \in T(\neg e)\}$

$T(e_1 \wedge e_2) = T(e_1) \cap T(e_2)$

$T(e_1 \vee e_2) = T(e_1) \cup T(e_2)$ □

A time-out expression $tm(e,n)$ is satisfied if the last occurrence of e was exactly n time steps ago. This is expressed by $shift(f,n) \restriction 1 \in T(e)$ (e occurred n steps ago) and by $shift(f,-i) \in T(\neg e)$ (e did not occur later, i.e., the occurrence at $-n$ was the last occurrence). In Statecharts, it does not matter whether e occurs at the moment of the time-out, hence, no claims about the present are made. This is expressed by $T(tm(e,n)) = \{C \mid C \subseteq E_p\}$.

The semantics of conditions is defined as follows:

Definition

$T_C : \mathbf{C} \rightarrow 2^{\mathbb{Z} \rightarrow 2^{E_p}}$

$T_C(true) = T(\lambda)$

$T_C(false) = T(\neg true)$

$T_C(in(S)) = \{f \mid \exists n \leq 0 : en(S) \in f(n) \wedge \forall n < i \leq 0 : ex(S) \notin f(i)\}$ □

The semantics of actions is as follows:

Definition

$\mathbf{A} : A \rightarrow 2^{E_p}$

$\mathbf{A}(\mu) = \varnothing$

$\mathbf{A}(a) = \{a\}$ for $a \in E_p$

$\mathbf{A}(a_1 ; a_2) = \mathbf{A}(a_1) \cup \mathbf{A}(a_2)$ for $a_{1,2} \in A$ □

Now we extend the domain of T to the set of complete labels, **Lab**, and the codomain to sets of histories.

Definition

Let $(F, C, \leq) \in \mathcal{C}$ and $D, E \subseteq C$. Then

- D is an *initial segment* of C iff $\forall a \in C \forall b \in D$: $a \le b \vee a \sim b \rightarrow a \in D$
- $D < C$ iff $\forall a \in D, b \in C: a < b$

An initial segment contains exactly the events that are generated in some prefix of the sequence of micro-steps.

Definition

T: **Lab** $\rightarrow 2^{H}$ is defined as follows:

$f \in T(e[c]/a)$ iff $|f|=1$ and there exist $f' \in T(e) \cap T_C(c)\}$ and F, C, $\le$ such that

(i) $\forall i<0\ \exists$ partial order $\le'$: $f(i)=(\emptyset, f'(i), \le')$

(ii) $F \subseteq \mathrm{A}(a)$, $\mathrm{A}(a) \subseteq C$

(iii) there exists an initial segment D of C s.t. $D \in f'(0)$ and $D < F$ and $\forall f_1, f_2 \in F: f_1 \sim f_2$. □

ad(i) The past of f has nothing to do with the causality relation. It only depends on whether some event did or did not occur at a particular time step.

ad(ii) We only record the *first* occurrence of an event, hence not all the events in the action part of the label have to occur in F. It can well be the case that the environment will generate an event *before* (in the sequence of micro-steps) this transition generates it. E.g., if the transition is labelled a/a it is clear that a should not occur in F, since the transition is still depends on a being generated outside.

ad(iii) The set D contains all the events that occurred before the transition. These should be consistent with $e[c]$, which is guaranteed by the clause $D \in f'(0)$ (remember that the co-domain of f' consists of *sets* of sets of events, each set representing a consistent behaviour for the associated time unit). All events generated (for the first time) by this transition occur in the same micro-step; this is expressed by $f_1 \sim f_2$.

4.3. Definition of the semantics

A fundamental aspect of the semantics is that it describes ***any behaviour*** of the system that is consistent with the behaviour of the component. So, when two components are combined, only those behaviours should be combined that agree totally on the behaviour of the system. In other words, the C- and $\le$-components of the clock records should be equal. The F-components, however, describe only the local contributions and hence these should be unified.

Definition

$(F_1, C_1, \le_1) \| (F_2, C_2, \le_2) = (F_1 \cup F_2, C_1, \le_1)$ if $C_1 = C_2$ and $\le_1 = \le_2$

= undefined otherwise

For $f_{1,2} \in I\!H$, $f_1 \| f_2$ is only defined if $f_1^S = f_2^S$. In that case,

$(f_1 \| f_2)(i) = f_1(i) \| f_2(i)$ for all $i < |f_1|$ □

We define the semantic function

$$[\![.]\!]: \textbf{Stch} \rightarrow I\!D$$

by induction on the structure of **Stch**:

4.3.1. Primitives

A primitive has only one state and no complete transitions. Hence, a possible execution consists of some incoming transition, possibly waiting in the state until some outgoing transition is triggered and then executing this transition. Incomplete executions have no outgoing transitions (but a $\perp$ instead) and the case that the state is never left is expressed, as usual, by having arbitrary long incomplete executions. The semantics of the outgoing transition is given by the function T, the semantics of waiting is given by a set W. Since waiting is only allowed if none of the outgoing transitions can be taken, W is the complement of the set of all behaviours corresponding to taking one of the transitions. No semantics is given for the incoming transition, only an identification. At a later stage, this transition will be connected to an outgoing transition of another (or the same) chart. There, the outgoing transition will have a semantics.

Definition

Let $g_{end} = shift(g, -(|g|-1)) \restriction 1$ and $W = \{f \mid |f|=1 \wedge \neg\exists i: f \in T(e_i) \cap T_C(c_i)\}$ where $e_i[c_i]/a_i$ are the labels of the outgoing transitions in $\boldsymbol{O}$. Then

$(u, f, v) \in [\![\mathbf{Prim}(I, O, S)]\!]$ iff there exists a $g \in \mathbb{H}$ such that

$u \in I \cup \{*\} \wedge v \in \boldsymbol{O} \cup \{\bot\}$ and

$\forall 0 \le i < |g|-1: shift(g, -i) \restriction 1 \in W$ and

$v = \bot \rightarrow g_{end} \in W$ and $v \ne \bot \rightarrow g_{end} \in T(L(v))$ and $|g| = |f|$ and $g^S = f^S$ and

$$
\begin{aligned}
f^F(i) &= g^F(i) \cup \{en(S)\} \text{ if } i = -1\\
&= g^F(i) \cup \{ex(S)\} \text{ if } i = |f|-1 \text{ and } v \ne \bot\\
&= g^F(i) \text{ otherwise}
\end{aligned}
$$

[]

4.3.2. Concatenation

By concatenating two subcharts, new computations become possible. Namely, by entering the first chart, performing a computation that ends in the connecting transition, entering the second chart by this transition and performing a computation there. In our semantics, this corresponds to simply concatenating the histories from the first chart and those from the second chart that end respectively start with the connecting transition.

It is still possible however, to perform a computation in one of the charts in isolation, provided that it doesn't start or end with one of the connecting transitions, because these are no entering or leaving points anymore.

Hence, the semantics of the concatenation of two subcharts consists of the concatenation of their respective histories together with their own histories (performed by the function *conc*), from which the histories that start or end in a connecting transition are deleted (performed by the function *delete*). We have split this definition into two functions because we need these functions again in the semantics of *Connection* (below).

Definition

$[\![\mathbf{Conc}(U_1, t_1, t_2, U_2)]\!] = delete_{t_1,t_2}(conc\ ([\![U_1]\!], t_1, t_2, [\![U_2]\!]))^{CL}$

where $delete_{i,j}(D) = \{(u, f, v) \mid (u, f, v) \in D \wedge u, v \notin \{i, j\}\}$

and $conc(D_1, t_1, t_2, D_2) = \{(u, f_1\hat{\ }f_2, v) \mid (u, f_1, t_1) \in D_1 \wedge (t_2, f_2, v) \in D_2\} \cup D_1 \cup D_2$

and the concatenation $f_1\hat{\ }f_2$ is defined by:

$$
\begin{aligned}
(f_1\hat{\ }f_2)(i) &= f_2(i+|f_1|) \text{ if } i \ge |f_1|\\
&= f_1(i) \cup f_2(i-|f_1|) \text{ if } i < |f_1|
\end{aligned}
$$

[]

4.3.3. Connection

Since connection creates a transition from an OrChart to itself, the semantics is a fixed point of the concatenation operator. Note, however, that the deletion of histories starting or ending with the connecting transitions can only be applied *after* the fixed-point operation, because these connection points are needed for the repeated application of concatenation. Because of the cpo structure on our domain and the continuity of the function *conc*, this least fixed point exists and is unique.

Definition

$[\![\mathbf{Conn}(U,t_1,t_2)]\!] = \mathit{delete}_{t_1,t_2}(\mu X.\mathit{conc}([\![U]\!],t_1,t_2,X))^{CL}$

where μ is the least fixed-point operator □

4.3.4. Anding

Anding two Unvs means executing them in parallel. This means that for each time step the behaviour of both components at that time step should be combined. The definition of this combination can be found in section 4.3.1. Now we define how two given history-*triples* should be combined.

Definition

Let $v, v_1, v_2 \in T_O$ and $f, f_1, f_2 \in I\!H$. We define the predicate *MERGE* as follows:

$MERGE(v_1, f_1, v_2, f_2, v, f) \Leftrightarrow$

(i) $[v \neq \bot \rightarrow \exists j: v = v_j \wedge |f_j| < |f_{3-j}| \wedge |f| = |f_j| \wedge f \upharpoonright |f|-1 = (f_1 \| f_2) \upharpoonright |f| \wedge f(|f|-1) = f_j(|f|-1)]$ and

(ii) $[v = \bot \rightarrow v_1 = v_2 = \bot \wedge f = f_1 \| f_2]$ □

Case (i) treats complete computations. The computation can only exit the construct via an outgoing transition of exactly one of the components (no forks on outgoing transitions are allowed). Hence, at such a moment the other component must be performing some internal computation. This is expressed by $|f_j| < |f_{3-j}|$, where j is the index of the component from which the outgoing transition is performed. All computations in f_{3-j} beyond and including $|f_j|-1$ (this is the time at which the exiting transition is performed) are discarded by the merge. The remaining ones are combined. If the computation is incomplete, we simply merge the histories of the two components (ii).

Note that $f_1 \| f_2$ is a partial function: if f_1 and f_2 do not agree on the behaviour of the total system at some time step, the function is undefined and the predicate equals false.

Definition

$[\![\mathbf{And}(U_1, U_2, \{(t_1, w_1), \ldots, (t_n, w_n)\})]\!] = \{(u, f, v) \mid \exists (u_i, f_i, v_i) \in [\![U_i]\!]:$

$((\exists 1 \leq j \leq 2: u = u_j \wedge u_{3-j} = *) \vee (\exists 1 \leq j \leq n: u = t_j \wedge u_1 = t_j \wedge u_2 = w_j)) \wedge MERGE(v_1, f_1, v_2, f_2, v, f)\}$ □

The execution starts either explicitly by a forked transition ($u = t_j$) or explicitly by a transition to one of the two components as a result of which execution in the other component is implicitly started ($u = u_j$ and $u_{3-j} = *$).

4.3.5. Statification

There are two types of *Statification*, one for OR-states and one for AND-states. Syntactically and semantically they only differ in that the first one has a default whereas the other one has none.

Definition

$[\![\mathbf{Stat}(U_1, U_2, d)]\!] = \{(u, f, v) \mid \exists (u_i, f_i, v_i) \in [\![U_i]\!]:$

$((u = u_1 \wedge u_2 = d) \vee (u = u_2 \wedge u_2 \neq d \wedge u_2 \neq *)) \wedge MERGE(v_1, f_1, v_2, f_2, v, f)\}$

$[\![\mathbf{Stat}(U_1, U_2)]\!] = \{(u, f, v) \mid \exists (u_i, f_i, v_i) \in [\![U_i]\!]:$

$((u = u_1 \wedge u_2 = *) \vee (u = u_2 \wedge u_2 \neq *)) \wedge MERGE(v_1, f_1, v_2, f_2, v, f)\}$ □

There are two ways to start the execution of an *OR−state* with inner structure. One can either take a transition explicitly to some state(s) inside the outer state ($u = u_2$) or take a transition to the outer state and enter some state(s) inside by default ($u = u_1$).

An AND-state has no defaults associated to it, since execution always starts simultaneously in all of its immediate substates. So, execution starts either by taking a transition to the outer state and start execution inside implicitly ($u = u_1 \wedge u_2 =$) or by entering the inner structure explicitly ($u = u_2 \wedge u_2 \neq$). The way the components of this inner structure are started, is taken care of by the semantics of U_2. Combining the histories from the two components and exiting the construct is not different from *Anding* and hence this definition can be found in the previous section.

4.3.6. Closure

Closure with respect to an event a makes the closed statechart insensitive to all a-events generated outside the chart. However, the environment stays sensitive to a-events generated *within* the closed statechart. Consequently, all histories should be deleted in which a is claimed to occur ($a \in C$), but in which a is not generated. The predicate OK yields falsehood for such clock records. The insensitivity of the closed statechart also means, however, that histories depending on the fact that a did *not* occur are legal if a is generated outside. Therefore, some new histories should be added to the denotation, representing the behaviours the statechart is consistent with as a result of the closure. For a given clock record, the function SAT produces all clock records that are consistent with this behaviour after Closure.

Definition

$OK(F, C, \leq) \Leftrightarrow a \in C \rightarrow a \in F$

$SAT(F, C, \leq) = \{(F, C', \leq') \mid C \subseteq C' \subseteq C \cup \{a\} \wedge \leq' \upharpoonright C = \leq\}$

$[\![\mathbf{Close}(U, a)]\!] = \{(u, f, v) \mid \exists g : (u, g, v) \in U \wedge \forall i : OK(g(i)) \wedge f(i) \in SAT(g(i))\}$ for $a \in E_e$

$[\![\mathbf{Close}(U, s)]\!] = [\![\mathbf{Close}(\mathbf{Close}(U, en(s)), ex(s))]\!]$ for $s \in \Sigma$ $\square$

4.3.7. Hiding

Hiding is in a sense dual to *Closure*. It makes the environment of a statechart insensitive to the occurrence of a particular event. So, at applying *Hiding* of event a to a statechart, all occurrences of a should be deleted from the histories. This is performed by the function $DELETE$. This insensitivity, however, makes some behaviours legal that were not legal before the hiding of the statechart. If a is generated inside, then after hiding the component is consistent with those behaviours of the environment in which a occurred later or even not at all in that macro step. It does *not* become consistent with behaviours in which a occurred *before* it was generated, because in these behaviours a is actually generated by the environment and on such occurrences hiding does not apply. The operator that extends the denotation with the appropriate clock records is $DELAY$.

Definition

$DELAY(F, C, \leq) = \{(F, C. \leq)\}$ if $a \notin F$

$= \{(F \cap C', C', \leq') \mid (C' = C \setminus \{a\} \wedge \leq' = \leq \setminus \{a\}) \vee$
$(C' = C \wedge \leq' \setminus \{a\} = \leq \setminus \{a\} \wedge \forall b \in C : b < a \rightarrow b <' a\}$ otherwise

$DELETE(F, C, \leq) = (F \setminus \{a\}, C, \leq)$

$[\![\mathbf{Hide}(U, a)]\!] = \{(u, f, v) \mid \exists g : (u, g, v) \in U \wedge |g| = |f| \wedge \forall i \exists d \in DELAY(g(i)) : f(i) = DELETE(d)\}$

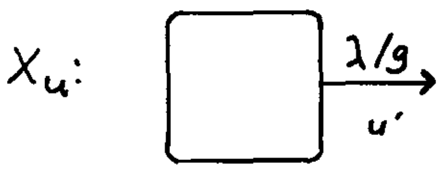
$[\![\mathbf{Hide}(U, s)]\!] = [\![\mathbf{Hide}(\mathbf{Hide}(U, en(s)), ex(s))]\!]$ for $s \in \Sigma$ $\square$

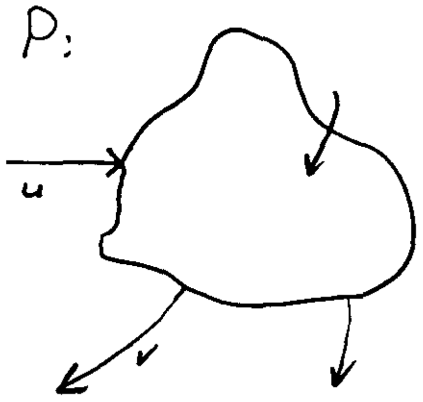
5. Full Abstraction

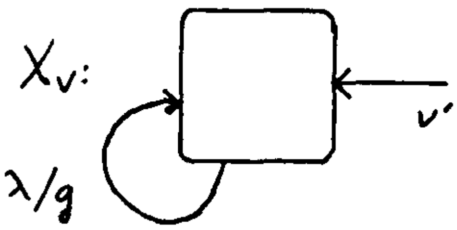
In this chapter we give a notion of *observable behaviour* for Statecharts and prove that the semantics is fully abstract with respect to this notion of observable behaviour. We refer to [HGR,HePl] for a further explanation about full abstraction.

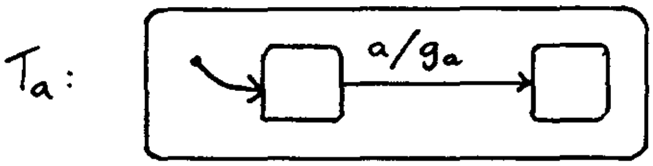
A **context** is a program with a "hole" in it. If $C[.]$ is a context and P a program then $C[P]$ is the program that results from plugging P into the hole of C. Let $O(P)$ give the observable behaviour of program P.

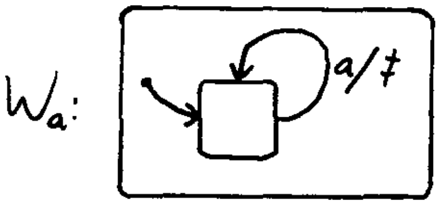
figure

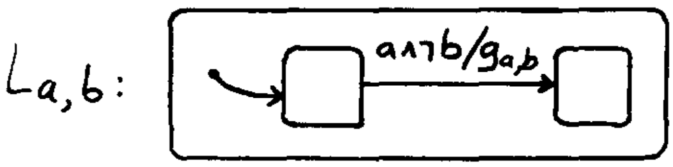


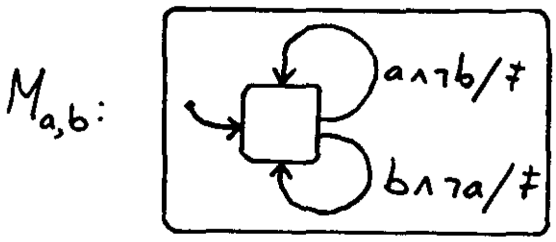




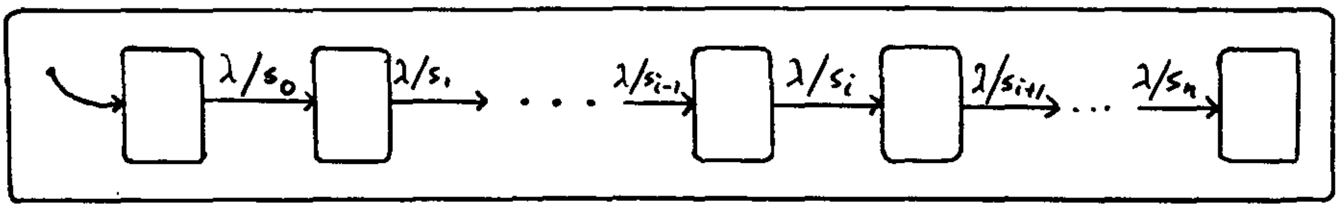




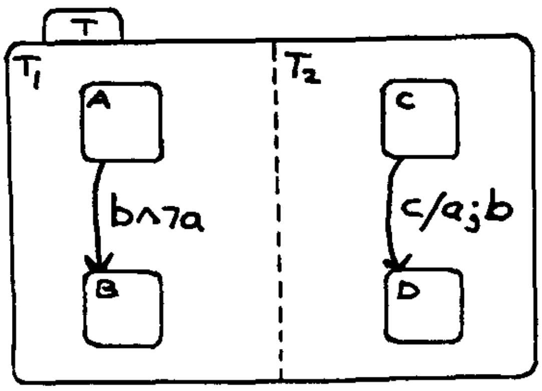




synchroniser: $(n = |k| - 1)$



Example 11



Definition

A semantics $[\![.]\!]$ is **fully abstract** with respect to O iff:

for all programs P, Q: $[\![P]\!]=[\![Q]\!] \Leftrightarrow$ for all contexts C: $O(C[P])=O(C[Q])$ □

For Statecharts, we choose as the observable behaviour of a statechart or Unv the events that are generated by that statechart at every time unit. So we define

$$O(P)=\{f^F \mid \exists u,v: (u,f,v)\in [\![P]\!]\}$$

Theorem

$[\![.]\!]$ = fully abstract w.r.t. O

Proof

For reasons of simplicity we do not consider *enter* and *exit* events, conditions or *time–out* events. The proof can be easily extended to cover these features, too.

$\Rightarrow$: follows from the compositionality of $[\![.]\!]$.

$\Leftarrow$:

Suppose $[\![P]\!] \neq [\![Q]\!]$. We want to find a context X, such that $O(X[P]) \neq O(X[Q])$. Let E be the set of primitive events that occur in P and Q. Assume without loss of generality that there exists a history-triple $(u,k_1,v)\in [\![P]\!] \setminus [\![Q]\!]$.

If the entrance transition $u \neq *$ we signal this by concatenating P to a chart X_u (see fig.). If $u=*$ we don't have to signal this, because any computation can start this way. Let us assume that P and Q are OrCharts, otherwise we apply Statification first. Likewise, if the exit transition $v \neq \perp$, we signal this by concatenating P to a chart X_v (see fig.). Again, we don't have to signal the case that $v=\perp$.

Define X_C = Conc(Conc(X_u,u',u,<*hole*>), v,v',X_v) and $P'=X_C[P]$ and $Q'=X_C[Q]$. Now $[\![P']\!] \neq [\![Q']\!]$ and we know that this difference cannot be based on incoming or outgoing transitions. So we can restrict ourselves to *histories* instead of history-triples. We will write "$f\in [\![P]\!]$" etc. instead of "there exist u,v such that $(u,,f,v)\in [\![P]\!]$" etc. So we assume that there exists a history $k\in [\![P']\!] \setminus [\![Q']\!]$. On basis of this history we build our context X. We use a set of events $G\cup\{\ddagger\}$ that do not occur in P' or Q'. Events in G witness behaviour in accordance with k, the event $\ddagger$ signal a violation of such.

We now construct for each time unit i a program that allows behaviour as described by $k(i)$ and that will try to catch as many aberrations as possible.

Let $k(i)=(F,C,\leq)$. In the construction below, $g_a, g_{a,b}\in G$ will generically stand for some fresh, i.e. unused events.

- For each $a\in E$ with $a\notin C$ we build the chart W_a and for each $a\in C$ we build the chart T_a, (see fig).
- For each pair $a,b\in C$ with $a<b$ and a and b *successors*, i.e. $\neg\exists c: a<c<b$, we build the chart $L_{a,b}$ with g fresh (see fig).
- For each pair $a,b\in C$ with $a\sim b$, we build the chart $M_{a,b}$ (see fig).

Now we put all these charts in parallel by *Anding* and apply *Closure* on the events in $G\cup\{\ddagger\}$. This AndChart defines X_i; the set of events $g\in G$ that it uses is G_i.

Lemma 1

If $(F',C',\leq')\in [\![X_i]\!]$, $G_i \subseteq F$ and $\ddagger\notin F$ then $(F',C',\leq')\restriction E=(\varnothing,C,\leq)$.

Proof

1. $F'\cap E=\varnothing$, because events in E are not in the action part of any label in X_i.
2. $(E\setminus C)\cap C'=\varnothing$, because $\ddagger\notin F'$ implies that none of the transitions in the W_as were taken.
3. $C\subseteq C'$, because $G_i\subseteq F$ implies that all transitions in the T_as were taken.
4. $G_i\subseteq F'$ implies that all $L_{a,b}$ transitions have been taken and hence that any $\leq$-successors are also $\leq'$-successors.
5. Furthermore, none of the $M_{a,b}$ transitions have been taken ($\ddagger\notin F'$) and hence for any $a,b\in C$ with $a\sim b$ we have neither $a<'b$ nor $b<'a$, or, in other words $a\sim'b$.

From 4. and 5. we can infer that for any $a,b\in C$ we have $a\leq b \iff a\leq' b$, which is equivalent with saying that $\leq = \leq'\restriction C$. This, together with 1, 2 and 3, gives the desired property. $\square$

To make sure that the AndChart X_i is only active at time step i, we build a synchroniser that generates at time step j the special event s_j for each $0\leq j\leq |k|$ (see fig). To each label in X_i we add the conjunct s_i. For each time step j we build an AndChart X_j and put these in parallel with the synchroniser and this AndChart we call $\overline{X}$.

Lemma 2

If $l\in [\![\overline{X}]\!]$ and $|l|=|k|$ then
$\forall i<|k|: G_i\subseteq k^F(i)$ and $\ddagger\notin k^F(i)$ iff $\forall i<|k|: l(i)\restriction E=(\varnothing,C,\leq)$.
Furthermore, such an l exists.

Proof
From the construction of $\overline{X}$ and Lemma 1. $\square$

Let $X[.]$ be the context $\mathbf{And}(\overline{X},\langle hole\rangle,\varnothing)$. Now we can prove that $O(X[P'])\neq O(X[Q'])$. Let the history l be as in Lemma 2. Then we define $\overline{l}$ by

$$\overline{l}(i) = (l^F(i)\cup k^F(i), l^C(i), l^{\leq}(i))$$

It is clear that $\overline{l}\in X[P']$ and hence that $\overline{l}^F\in O(X[P'])$.

Suppose that $\overline{l}^F\in O(X[Q'])$. Then there must be some $\overline{m}\in [\![X[Q']]\!]$ with $\overline{m}^F=\overline{l}^F$. By the semantics of And-ing, there must be $k'\in [\![Q']\!]$ and $m\in [\![X]\!]$ such that

$$m^F\restriction G = l^F\restriction G \text{ and } k'^F\restriction E = k^F\restriction E$$

By lemma 2, we know that $m(i)\restriction E=(\varnothing,C,\leq)$ for all i, hence $k'(i)\restriction E=(F_i,C,\leq)$ for some F_i, because the merge could not have taken place otherwise. But since the only events that P' and Q' can generate are in E, F_i must be empty. Applying this argument to all time steps, yields

$$k' = k$$

which is in contradiction with our initial assumption, so that $\overline{l}\notin O(X[Q'])$. $\square$

6. Discussion

In this chapter we discuss the future extension of statecharts with variables, and a possible other definition of the semantics with respect to causality between micro-steps.

6.1. Variables

The full version of this paper will include the use of variables in the labels of transitions (in conditions and in actions as assignments). This will not involve an essential extension of the model. The same technique used for the condition $in(S)$ can be applied here. All changes to variables are signalled in the form of events and the satisfaction of conditions is checked by an inspection of the history.

6.2. Other definition on causality

In the semantics of [HPSS], the influence of a transition is restricted to the transitions that follow it in the sequence of micro-steps building the macro-step. In our compositional semantics, this restricted influence is modelled by the partial order in the clock record. This solves the causal paradox of the transition annulling its own cause (see example 5 in chapter 2), but this solution is not fully satisfactory. E.g., a transition labelled $\neg a$ can always be taken, even if a happens during that time unit. (It only differs from a transition labelled by λ in as much that it need not be taken when a happens.) Furthermore, the semantics depends heavily on the relative order in which the micro-steps occur, whereas the micro-steps are definitely not observable - they are only introduced to solve the causal problems.

A new version of the operational semantics is under study by Pnueli and others, in which global contradictions are not allowed. A global contradiction occurs when two transitions with conflicting labels take place in the same macro-step. E.g., a transition labelled $\neg a$ can never take place in the same macro-step with a transition labelled $.../a$, even if the latter occurs in a later micro-step. This leads to a simpler and more intuitive semantics. The main drawback, however, is that causal paradoxes such as the one in example now lead to a run time error. There is no acceptable behaviour anymore to associate to these situations and there is no way to detect them syntactically.

We can easily adapt the compositional semantics to model this new operational semantics. The only thing that has to change is the definition of the semantics of a label. Instead of demanding that the triggering event should only be satisfied by some *initial segment* of the macro-step, we demand that it should be satisfied by the *complete* macro-step. The partial order is only used to guarantee that there are no circularities in the triggering of transitions by other transitions. In fact, we could do with a linear order instead of a partial order, because there is no need anymore to distinguish events generated in the same micro-step from the same events generated in arbitrary order.

7. Conclusion

We presented a compositional semantics for the graphical specification/programming language Statecharts, as described in [HPSS]. For this, we had to define a proper generative syntax. The operators in this syntax have simple graphical counterparts as well as a natural semantics. The model extends the model of [HGR,GB] to deal with broadcast and, specifically, with the micro-step semantics of Statecharts as described in [HPSS]. This is a subtle operational notion to deal with the consequences of the synchrony of action and reaction (called the *synchrony hypothesis* by Berry [B]). The compositional semantics does not model the micro-steps directly, but records only the occurrence relationship between the generated events as imposed by the order of micro-steps. After fixing the notion of observable behaviour, we prove that the semantics is fully abstract with respect to this notion of observability.

This work serves as a basis for extending the work of Hooman on proof-systems for Real-Time languages [H] and that of Zwiers [Z].

Acknowledgement

The authors want to thank Amir Pnueli and Jozef Hooman for the stimulating discussions with them and their useful suggestions. Special thanks go to S. Ramesh who found a lot of bugs in earlier versions and who helped very much in finding the definitions for *Hiding* and *Closure*.

Furthermore, we want to thank Edmé van Thiel and Inge van Drunen for typing this paper.

References

[B] Berry G., Cosserat L. (1985), The Synchronous Programming Language ESTEREL and its Mathematical Semantics, *in* "Proc. CMU Seminar on Concurrency", LNCS **197**, pp. 389-449, Springer-Verlag, New York.

[BCH] Bergerand J.-L., Caspi P., Halbwachs N. (1985), Outline of a real-time dataflow language, *in* "Proc. IEEE-CS Real-Time systems Symposium", San Diego.

[DD] Damm W., Döhmen G. (1987), An axiomatic approach to the specification of distributed computer architectures, *in* "Proc. PARLE Parallel Architectures and Languages Europe, Vol I", LNCS **258**, Springer Verlag, Berlin.

[G] Gonthier G., (1988), Ph.D. Thesis, Institute Nationale de Récherche en Informatique et en Automatique, Sophia-Antipolis, to appear.

[GB] Gerth R., Boucher A., A Timed Failures Model for Extended Communicating Processes (1986), *in* "Proc. 14th Colloquium Automata, Languages and Programming ICALP", LNCS **267**, pp. 95-114, Springer Verlag, Berlin.

[GBBG] Le Guernic P., Beneviste A., Bournal P., Ganthier T. (1985), SIGNAL: A Data Flow Oriented Language For Signal Processing, IRISA Report 246, IRISA, Rennes, France.

[H] Harel D. (1987), Statecharts: A visual Approach to Complex Systems, *Science of Computer Programming*, Vol. **8-3**, pp. 389-449, pp. 231-274.

[HePl] Hennessy M., Plotkin G. (1979), Full Abstraction for a Simple Programming Language, *in* "Proc. Math. Foundat. of Comput. Science", LNCS **74**, pp. 108-120, Springer Verlag, New York.

[HGR] Huizing C., Gerth R., De Roever W.P., (1987), Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like language, *in* "Proc. 14th ACM Symposium on Principles of Programming Languages POPL", pp. 223-237.

[Ho] Hooman J. (1987), A compositional proof theory for real-time distributed message passing, *in* "Proc. PARLE Parallel Architectures and Languages Europe, Vol II", LNCS **259**, pp. 315-332.

[HP] Harel D., Pnueli A. (1985), On the Development of Reactive Systems, Logic and Models of Concurrent Systems, *in* "Proc. of the NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems", NATO ASI Series F, Vol. 13, pp. 477-498 Springer Verlag, Berlin.

[HPSS] Harel D., Pnueli A., Pruzan-Schmidt J., Sherman R. (1987), On the Formal Semantics of Statecharts, *in* "Proc. Symposion on Logic in Computer Science (LICS)", pp. 54-64.

[HU] Hopcroft J.E., Ullman J.D. (1979), **Introduction to automata theory, languages, and computation**, Addison-Wesley, Reading.

[K&] Koymans R., Shyamasundar R.K., De Roever W.P., Gerth R., Arun-Kumar S. (1988), Compositional Semantics for Real-Time Distributed Computing, *Information and Control*, to appear.

[M] Mazurkiewicz A., Proving algorithms by tail functions, *Information and Control*, **18**, (1971), pp. 220-226.

[SM] Salwicki A., Müldner T. (1981), On the Algorithmic Properties of Concurrent Programs, *in* "Proc. Logic of Programs", LNCS **125**, Springer Verlag, New York.

[SW] Strachey C., Wadsworth C.P., Continuations: A Mathematical Semantics for Handling Full Jumps, Technical Monograph PRG-11, Oxford University Computing Laboratory, Oxford.

[Z] Zwiers J. (1988), Compositionality and dynamic networks of processes: Investigating verification systems for DNP, Ph.D. Thesis, Eindhoven University of Technology, to appear.

Appendix

In [HPSS] the set of statecharts is not defined by a generative grammar, but in a more direct way. We shall call these objects **H-statecharts** and define the formal relationship between H-statecharts and the elements of the sets **Stch**, the expressions generated by the syntax as defined in chapter 3.

Definition (taken from [HPSS], adapted):

Let a set of states Σ and a set of labels **Lab** be given.
A **H–statechart** is a quintuple (S,ρ,ψ,δ,T) where

$S \subset \Sigma$ is the set of states;

$\rho: S \rightarrow 2^S$ is the hierarchy function;

$\psi: S \rightarrow \{\text{AND}, \text{OR}\}$ is the type function;

$\delta: S \rightarrow 2^S$ is the default function;

$T \subseteq S \times \mathbf{Lab} \times S$ is the set of transitions,

with the following restrictions:

(i) $\forall s \in S : s \notin \rho^+(S)$

(ii) $\forall s_1, s_2 : s_1 \neq s_2 \rightarrow \rho(s_1) \cap \rho(s_2) = \varnothing$

(iii) $\forall s \in S : \delta(s) \subseteq \rho^+(s)$

(iv) $\exists! r \in S : \rho^*(r) = S \wedge \forall t \in T : r \notin {}^{<}t \wedge r \notin t^{>}$

(v) $\forall s \in S : (\exists x \in X : s \in \rho(x) \wedge \psi(x) = \text{AND}) \rightarrow \forall t \in T : s \notin {}^{<}t \wedge s \notin t^{>}$

The set of H-statecharts is called **HS** []

Notation: if $t \in T$ and $t=(s_1,s_2)$, then ${}^{<}t = s_1$, $\hat{t}=1$ and $t^{>}=s_2$
where ρ^* and ρ^+ are the reflexive respectively irreflexive transitive closure of ρ.

We define a function $R : \mathbf{HS} \rightarrow \mathbf{Stch}$ as follows:
Let $\sigma = (s,\rho,\psi,\delta,T)$ be given.
Define a function $E : \mathbf{S} \rightarrow \mathbf{Unv}$ that gives for each statechart with one root state and its interior the associated Unvollendete (PrimChart). Then we can define:

$R(\sigma) = E(r)$ where r is the root state of σ, i.e., $\forall s \in S : s \in \rho(s)$.

Define: $T_I=\{(t,s)\in T\times S \mid s\in t^>\}\cup\{(s_1,s_2)\in S\times S \mid s_2\in\delta(s_1)\}$

$T_O=\{(s,t)\in S\times T \mid s\in {}^<t\}$

$L:T_O\rightarrow \mathbf{Lab}$

$L(s,t)=\hat{t}$

Notation: if $i\in T_I$ and $i=(t,s)$, then $i^>=t^>$ and $tr(i)=t$ if $t\in T$, $i^>=\delta(s_1)$ if $s_i\in S$.

if $o\in T_O$ and $o=(s,t)$, then ${}^<o={}^<t$ and $tr(o)=t$.

T_I and T_O will serve as the set of incoming respectively outgoing transitions for the Unvollendetes we are going to use. Since defaults are made out of incoming transitions, we need some of these for this purpose.

Define an auxiliary function $E_p:S\rightarrow\mathbf{Unv}$:

$E_p(s)=Prim(I,O,s)$ with $I=\{(t,s)\in T_I \mid s\in t^>\}$ and $O=\{(s,t)\in T_O \mid s\in {}^<t\}$

For $U\in\mathbf{Unv}$, define

$Inc(U)=I$ if $U=<I,O>$

$Outg(U)=O$ if $U=<I,O>$

We need a function E_i: $S\rightarrow\mathbf{Unv}$ that gives for each state the Unvollendete that should be associated to the *interior* of that state. It depends on whether it is an AND-state or an OR-state. We define E and E_i recursively:

$E(s)=E_p(s)$ if $\rho(s)=\varnothing$

$E(s)=\mathbf{Stat}(E_p(s),E_i(s),s,s')$ if $\rho(s)\neq\varnothing$ and $\psi(s)$=OR and $(s,s')\in Inc(E_i(s))$ for some s';

$=\mathbf{Stat}(E_p(s),E_i(s))$ if $\rho(s)=\varnothing$ and $\psi(s)$=AND.

$E_i(s)$ is defined by two cases.

Let $s\in S$ be given and $\rho(s)=(s_1,\ldots,s_n)$, $n>1$.

Distinguish two cases:

(i) $\psi(s)=$ AND

Define a sequence of Unvollendetes $A_1,\ldots,A_n$ as follows:

$A_1=E(s_1)$

$A_j=\text{And}(A_{j-1},E(s_j),a_j)$ for $2\leq j\leq n$

and $a_j=\{(i_1,i_2)\in I_1\times I_2 \mid s_j\in i_2^>\wedge\exists\, 1\leq k<j: s_k\in i_1^>\}$

and $I_1=Inc(A_{j-1})$, $I_2=Inc(E(s_j))$.

Then $E_i(s)=A_n$

(ii) $\psi(s)=$ OR

Let $U=Or(..Or(E(s_1),E(s_2)),\ldots,E(s_n))$

Let $\{t_1,\ldots,t_n\}=\{t\in T \mid LCA(t)=s\}$ Here, LCA is a function defined in [HPSS]; $LCA(t)$ gives the smallest state that encloses transition t:

Let $R={}^<t\cup t^>$, then $LCA(t)=x$ iff

1. $R\subseteq\rho^+(x)$
2. $\psi(x)=$ OR
3. $\forall s\in R$: if $\psi(s)=$OR then $R\subseteq\rho^+(s)\rightarrow x\in p^*(s)$

Define a sequence of Unvollendetes $B_0,\ldots,B_n$ as follows.

$B_0=U$

$B_j=Conn(B_{j-1},o_j,i_j)$ for $1\leq j\leq n$,

where $tr(i_j)=tr(o_j)=t_j$

Then $E_i(s)=B_n$ □

86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben J.Paredaens	A formal approach to distributed information systems
87/04	T.Verhoeff	Delay-insensitive codes - An overview
87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeterministic automata.
87/11	P.Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.
87/12	K.M. van Hee and A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. van der Woude	Playing with patterns, searching for strings.
87/14	J. Hooman	A compositional proof system for an occam-like real-time language

87/15	C. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas
87/17	K.M. van Hee G.-J.Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples
87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces
87/19	A.J.Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples
88/04	G.J. Houben J.Paredaens D.Tahon	The Nested Relational Algebra: A Tool to handle Structured Information
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling

88/10	J.C. Ebergen	A Formal Approach to Designing Delay Insensitive Circuits