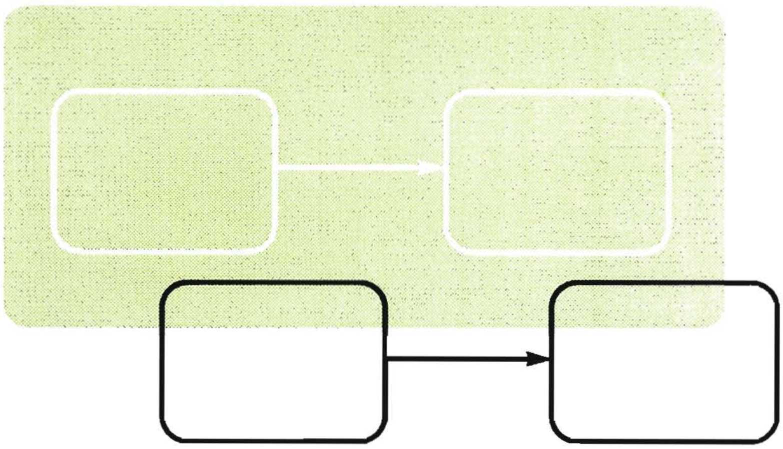
# Semantics of reactive systems : comparison and full abstraction

# SEMANTICS OF REACTIVE SYSTEMS: COMPARISON AND FULL ABSTRACTION



C. HUIZING

# SEMANTICS OF REACTIVE SYSTEMS: COMPARISON AND FULL ABSTRACTION

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op vrijdag 8 maart 1991 om 16.00 uur

door

CORNELIS HUIZING

Geboren te Bergen op Zoom

druk: wibro dissertatiedrukkerij, helmond.

Dit proefschrift is goedgekeurd door de promotoren:

Prof.dr. W.P. de Roever
en
Prof.dr. K.M. van Hee

*To my grandfather*

## Acknowledgements

I thank Willem-Paul de Roever for his guidance in the research and related work for this thesis. His stimulation, always on a moment that you least expected it, was essential for the completion of this thesis.

Rob Gerth has been helping me from my Master's Thesis until now with his sharp insight and his knowledge. His ideas and criticism have shaped my research.

They and my other colleagues at the Eindhoven University of Technology created the open working atmosphere that is so important for the joy in work. I thank them for the many discussions, or sometimes sound-board sessions, and I will never forget that special combination of science and comradry that characterises our joint trips.

Many thanks go to Edmé van Thiel, Marieke Munter, and Annelies Vermeulen-Adolfs for typing various parts of this thesis.

I thank Kees van Hee for his glad willingness to be my second promotor and to help me with the problems that occurred due to Willem-Paul's leave to Kiel. Furthermore, we had some interesting discussions and he gave me a view on my thesis from a new angle.

This research was partly carried out in the context of two ESPRIT projects, DESCARTES and SPEC, financial supported by the European community. I thank Dr. ir. A. Heijligers for his active interest as a director of the faculty in the well-being of these projects.

Since I consider the promotion as the ultimate conclusion of one's education, I thank all the teachers who have contributed to my knowledge or whatever development of the mind I have gone through. My first teachers were my grandfather, my parents and both my brothers and what they learned me I relish as the most important. Of the others, I want to mention Meester van Doorn, Dhr. A. Vermeulen, and Amir Pnueli.

Finally, I thank my wife Inge, *pour tout.*

# Contents

# Introduction

# Introduction

This thesis consists of the five articles below, presenting fundamental aspects of the semantics of reactive systems.

1. C. Huizing, W.P. de Roever, *Everything you always wanted to know about Statecharts but were afraid to ask*, submitted to Information Processing Letters, under the title *Introduction to design choices in the semantics of Statecharts*, 1990.
2. C. Huizing, R. Gerth, W.P. de Roever, *Full Abstraction of a Denotational Semantics for an OCCAM-like Language*. In Proc. POPL 87, 1987, extended version.
3. C. Huizing, R. Gerth, W.P. de Roever, *Modelling Statecharts in a fully abstract way*. In Proc. CAAP, LNCS 299, pp. 217-294, 1988, improved and extended version.
4. C. Huizing, R. Gerth, *On the semantics of reactive systems*.
5. C. Huizing, *Formalisms related to Statecharts*.

These articles provide a semantic basis for reactive systems from which methods of specification, verification, and, ultimately, programming can be developed.

Since the term *reactive system* was introduced by Amir Pnueli and David Harel in [HP85], an increasing amount of research has been dedicated to this topic. Programming reactive systems shows many problems that are in general considered "difficult", and could appear in areas such as real-time and parallel programming, but were never identified as one single concept. This concept of reactivity distinguishes reactive systems from the more conventional transformational systems. The continuous interaction with its environment makes it unrealistic to analyse reactive systems as performing a function from input to output.

As we point out in chapter 1, time plays an essential role in reactive systems. Although a reactive system does not need to be a real-time system in every sense of the word, the relative timing of input and output events plays an essential role. Therefore, we first study real-time as such, without the notion of reactivity. For this purpose, we use a language that is similar to OCCAM and very close to CSP-R [KSR$^+$88]: synchronously communicating parallel processes with a simple notion of real-time.

We based our denotational semantics on the model presented in [KSR$^+$88] and made the necessary changes to make it fully abstract.

For a thorough study of semantics we need the concept of *full abstraction*. An essential aspect of semantics is abstraction: a semantics should abstract from the particular formulation of the program or specification text and only distinguish programs that really have to be distinct. To define when programs have to be considered distinct, we define the notion of *observable behaviour*. The observable behaviour is that part of the behaviour that is of real interest to us (of course, this is a matter of choice). Now one could ask: why not use the observable behaviour as the semantics of a program? The answer to this lies in the other essential aspect of a semantics: the properties of the domain. A semantics maps a program or specification to an object in a domain with certain desired properties. Nowadyas, it is considered very important that this domain is *compositional* with respect to the syntactic operators. This means that the semantics of a compound program $P \circ Q$, where $\circ$ is a syntactic operator, can be described in terms of the semantics of the components $P$ and $Q$. This demand makes it in general impossible to use the observable behaviour as a semantics. Consequently, it is important to find the semantics that is as abstract as possible without violating compositionality. This property is called *full abstraction* [HP79].

With this model as a solid starting point, we studied the semantics of a reactive language: *Statecharts.*

In Chapter 1, we motivate that a reactive system should be able to react instantaneously to stimuli from the environment, i.e., without delay, at least at a conceptual level. This can raise causal paradoxes that have to be taken care of somehow.

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 forbidding programs by a static check of the compiler. In Statecharts, they are *semantically* impossible, because there the influence of an event is restricted to events that did not cause it. The problem is to model causality between events that have no precedence in time. In the operational semantics of [HPPSS87], 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 an order[1] 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.

In chapter 3, we chose the solution that was implemented in Statecharts at that moment. In chapter 4 several ways to tackle this problem are discussed and compared in one semantic framework. This formal treatment makes it possible to formulate three desirable properties, or criteria, to match the various solutions against. Unfortunately, it turns out, and is proved, that it is not possible to combine these three criteria into one semantics. To end with a positive remark, we can tell you that we designed a two-levelled semantics that satisfies the criteria, but on different levels.

Chapter 5 presents some formalisms that are related to Statecharts: the graphical language *Argos* and the process algebra *ATP.*

The thesis is concluded with a bibliography of Statecharts, including articles and books to be published and documentation of the Statemate system, the implementation of Statecharts.

[1]To be specific: it is a total preorder

## References

[HP79] M.C.B. Hennessy and G.D. Plotkin. Full abstraction for a simple parallel programming language. In *Proceedings MFCS '79*, LNCS 74, pages pp. 108–120. Springer, 1979.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. NATO, ASI-13, Springer-Verlag, 1985.

[HPPSS87] D. Harel, A. Pnueli, J. Pruzan-Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proceedings Symposium on Logic in Computer Science*, pages 54–64, 1987.

[KSR+88] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. *Information and Computation*, 79(3):210–256, 1988.

# Chapter 1

# Everything you always wanted to know about Statecharts but were afraid to ask

# Everything you always wanted to know about Statecharts but were afraid to ask [*]

C. Huizing [†] W.P. de Roever [‡]

Eindhoven University of Technology
Postbus 513
5600 MB Eindhoven
The Netherlands
August, 1990

## 1 What are Reactive Systems?

There is a fundamental dichotomy in the analysis of computing systems. This dichotomy crosses all borderlines between sequential and parallel systems, central and distributed systems, and between functional and imperative systems. This is the dichotomy between *transformational* and *reactive systems* [HP85]. Transformational systems are well described by a relation between input and output value. They read some input value, then produce, perhaps non-deterministically, an output value and terminate. They have a *linear structure*, only the initial and the final state are of interest. Reactive systems do not compute a function, but perform a continuous *interaction* with their environment. Whereas a transformational system is compared to a *black box*, a reactive system should be compared to a *black cactus* (terminology from Amir Pnueli), having several input and output channels. Reactive systems can be found everywhere, especially in embedded systems. They include digital watches, television sets, interactive software systems, integrated circuits, etc.

Transformational systems are well studied and an abundant number of sound theories for their formal description and analysis exists. For reactive systems, this is not the case. In this paper we explain what the problems are and why the language Statecharts is a good candidate for specifying and programming reactive systems.

## 2 Why not use transformational description techniques?

One can ask the question: "If transformational systems are so well studied, why don't you consider a reactive system as a transformational one?" Simply gather all the input together and all the output together and say that a reactive system transforms a *sequence* of inputs into a *sequence* of outputs. The objection is called "feedback". In principle, the behaviour

[*]This research is partially supported by BRA project "SPEC", ESPRIT 3096.

[†]e-mail: wsinkees@win.tue.nl or BITNET: wsdckeesh@heitue5

[‡]Current address: Institut für Informatik und Praktische Mathematik II der Christian-Albrechts-Universität zu Kiel, Preußerstraße 1-9, D-2300 Kiel, (West-)Germany; e-mail: wpr@informatik.uni-kiel.dbp.de

of the environment, i.e. the input to the system, depends on the reaction of the system, i.e. previous outputs. Therefore a relation between input sequences and output sequences is not enough to describe a reactive system. This phenomenon is known as the *Brock-Ackermann paradox* ([BA81]). Originally it is in terms of dataflow networks; it gives an example of two systems that perform the same transformation on sequences. When the output of the systems is fed back and merged with their input, they suddenly behave differently, *even if you consider the new systems as transformational ones.*

**Example: Brock-Ackermann paradox for reactive systems.** Suppose we have two systems that read events $a$ and $b$ from the input and then produce a $c$ as an output for every $a$ it has read. The difference is that system $S$ starts producing output only after the second input ($a$ or $b$), whereas system $T$ starts immediately after the first input. Viewed as sequence transformers, they have the same behaviour (assumed that the input is always infinite): both produce a sequence with as many $c$'s as there were $a$'s in the input.

Now suppose we put these systems in an environment that does the following. It produces $a$'s at regular intervals, but as soon as it sees an output of the system occurring, it switches from $a$'s to $b$'s. Now the two systems behave differently: in this environment, system $S$ will produce two $c$'s and system $T$ only one, because the moment that the environment stops sending $a$'s is different.

Apparently, we also have to know *when* an output is produced. This information can include a fully timed description of input and output events, but also a more abstract notion of time is possible, as long as the order of output events *relative* to the input events is specified. A similar observation is made by Jonsson and Kok in the context of dataflow networks ([JK89]).

## 3 Graphical language

Transformational systems have a linear structure and so have conventional languages for the specification and the programming of these systems. The only important states of the system are the initial state and the final state and what one describes is the relation between them or how to go from one to the other.

In a reactive system, however, we have a totally different picture. As we said before, the "moment" at which new input arrives is relevant to the behaviour of the system. In other words, the internal state of the system at the time of the input is important for the reaction. Hence, more attention must be paid to these intermediate states. They are not just a point on the way to the final state, they have a meaning of their own. In many reactive systems there isn't even a final state! In a conventional textual formalism, statements have a natural entry and exit point, namely the beginning and the end of the statement. The sequencing of statements in the language corresponds to the sequencing of states in the transformational system that is described. In reactive systems, however, there is no main sequential flow of control and statements can have several entry and exit points, since in any state, new input from the environment may arrive and change the direction of the execution. Hence, it makes sense to look at other languages than the well-known linear/textual ones.

There exists a graphical formalism that seems quite natural for our purposes. This is the *state diagram* of a finite state machine. For each state, the possible reactions to input that arrives when the machine is in that state is specified by *transitions* to other states. The traditional state diagram gives an intuitively clear picture of this behaviour. States are drawn as dots, possible transitions as arrows labelled with the input associated to it. In figure 1 we have drawn a typical state diagram of a finite state machine with initial state $S_1$ and final state $S_2$.

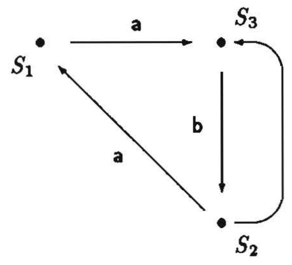


Figure 1: A typical state diagram of a Finite State Machine

The traditional FSM (Finite State Machine) does not fully serve our purposes. The only output it produces is a signal that it has reached its final state. A reactive system, however, may produce output at any time during execution. Hence, we allow our machine to produce an output whenever it takes a transition. This is essentially the formalism of the *Mealy machine* [HU79]. Statecharts is an extension of this formalism. The Israelian company AdCAD[1] has developed a programming environment, called STATEMATE, that uses three graphical formalisms to describe a system. *Activitycharts* describe the decomposition of the system into functional units and the interface between them. A *Modulechart* describes the physical modules where these units are implemented and the communication channels connecting them. The actual behaviour of a functional unit can be described by a *Statechart.* In this paper we only discuss Statecharts, since this is is the most interesting formalism from the semantic point of view. Below we present the essential extensions that David Harel made to this formalism. For a more extensive introduction to the language of Statecharts, we refer the reader to [Har87].

## 4 Hierarchy and Structure

There is an important concept that is not covered by the traditional state diagram. This is the concept of structure. As soon as the system to be described becomes only a little bit complex, the picture gets messed up with arrows and states, because there is no way to structure and summarise information. David Harel developed a clear and effective way to express structure by means of hierarchical states.

We shall introduce these concepts with the help of an example of a reactive system that is probably quite familiar to the reader. This is a television set with remote control. Input events are provided by pressing the buttons on the remote control unit and the system reacts by sending events and changing states.

E.g., there is a button on for activating the set and a button off for putting it back into standby mode. This behaviour is represented by the following picture(fig. 2).

[1]AdCAD is a subsidiary of i-Logix

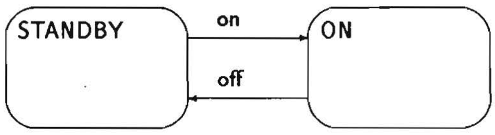


Figure 2:

## 4.1 Depth

The first concept that was added by Harel is hierarchy or depth in states. This is achieved by drawing states as boxes that can contain other boxes as substates. When the television is in state ON, it can be either in normal broadcast mode or in videotext (or teletext) mode. In the latter mode it shows special text pages that are sent along with the television program. Switching between these two modes can be done with the button txt. The sub-machine representing the behaviour is drawn inside the state ON (see fig. 3). This construction can be applied to any state and thus be nested.

As before, the event off will take the system from ON to STANDBY, no matter the system is in the substate NORMAL or VIDEOTXT. This provides a natural way to express refinement. One can also consider it as an interrupt without resume: any computation that is going on inside ON is interrupted by an *off* event. In both views, it is important that the outer transition has priority over the transitions inside. Although not present in earlier versions of Statecharts, this priority mechanism is present in the current implementation. One also encounters it in Esterel.

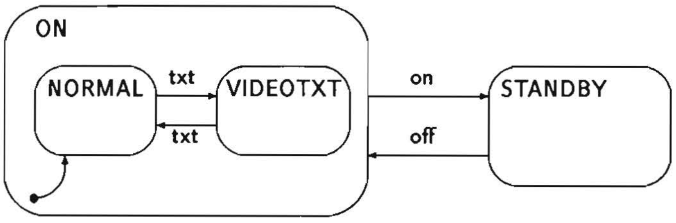


Figure 3:

We call the state ON an OR-state, because being in ON means being in NORMAL or VIDEOTXT. The $\bullet\!\!\rightarrow$ arrow can be used to specify which sub-state should be entered when the higher level state is entered.

Note that the state changes are in principle not observable, only generated events are. In a fully detailed specification, the states NORMAL and VIDEOTXT would contain a lot of substates that generate the observable events; at the very end these would be the glowing of the phosphorus spots on the screen etc.

## 4.2 Orthogonality

One of the drawbacks of the conventional finite state machine as a means of specification is the exponential blowup of the number of states because all possible combinations of states from different components have to be drawn. With the use of *orthogonal states* this is can be avoided. Two independent components can be put together in a so-called AND-state, separated by a dotted line (see fig. 4). Being in an AND-state means being in all of its immediate substates at the same time. In [DH88a] this is thoroughly investigated; it is proved that Statecharts is exponentially more succinct than the traditional formalism of finite state machines, and cumulatively so in several aspects.

In the television set, the operations of sound and image are independent from each other. E.g., switching from normal mode to videotext mode does not affect the sound, see fig. 4. So we refine the state ON into two orthogonal substates IMAGE and SOUND.

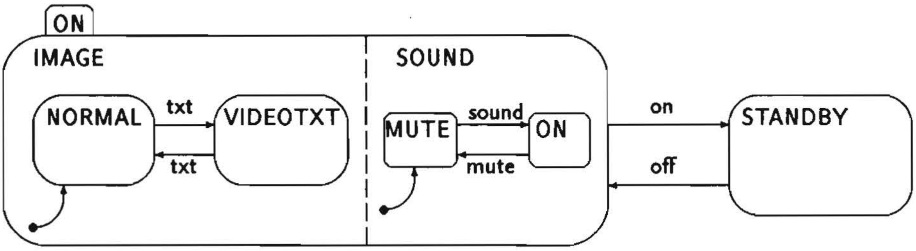


Figure 4:

For simplicity we have only two sound levels, MUTE and ON. Switching between them is done with the mute- and the sound- button. Of course, one could *refine* the state ON inside SOUND into several substates modelling the various levels of the sound.

## 4.3 Broadcast

In general, candidates for orthogonal components are not fully independent. Some mutual influence, or *communication*, should be specified. As in the formalism of the Mealy machine, one can associate an output to a transition. This output is a broadcast that can be sensed anywhere in the system and can trigger other transitions.

When you change in the television example from one channel to another, usually the sound is turned off for one second, probably to avoid unwanted noises. To model this, we add two orthogonal components CHANNELS and SM (for switching mute). When you press a channel button on the remote control, the television switches to that channel and an internal event *sm* is generated. That causes the generation of the event *mute* by a transition in the state SM and the sound will be turned off. After one second the event *sound* is generated to turn it on again. This is done by a special *time-out* event, written as tm(1). One can see this is an abbreviation for a counter that is started when the state MUTE is entered; it counts clock events, e.g. tenth of seconds; when the value 1 second is reached, the time-out event is generated.

For simplicity, we drew only two channels in the statechart (see fig. 5). For those who worry about the cluttering of the picture when more channels are drawn, we can say that there is nice syntactic sugar available to keep specifications like this readable (see [Har87] and [*i-Logix Inc*87]).

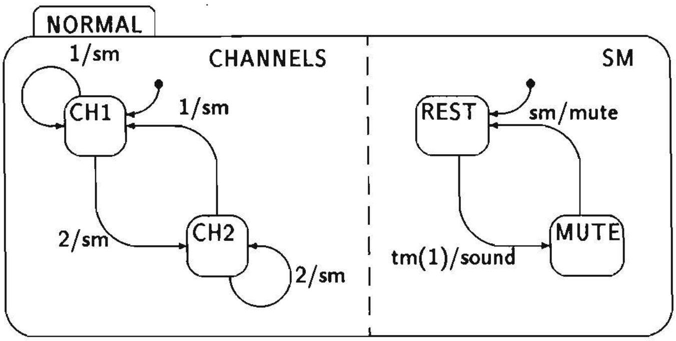


Figure 5:

Summarising, we may say with David Harel[Har88]:

*Statecharts = FSM + depth + orthogonality + broadcast*

These enrichments yield a graphical formalism that is structured and easy to understand, yet lacks the exponential growth of states of conventional finite state diagrams when concurrency is described. In [DH88b], it is showed that Statecharts are double exponentionally more succinct than state machines.

### 4.4 Compound events

We have seen that the label of a transition consists of two parts: a *trigger* that determines if and when a transition will be taken and an *action* that is performed when the transition is taken. This action is in general the generation of a set of events.

In the examples above, the trigger consisted of only one event. In general, this can be any logical proposition of events. E.g., a transition labelled $\neg a \wedge b/e$ can be taken when $b$ is present (either as an input or generated by a transition in an orthogonal component) and at the same time $a$ is not present. These triggers are called *compound events.*

## 5 Time

### 5.1 Events and transitions

The elementary entity of observation of a reactive system is the event. The environment sends events to the system to trigger computations, the system reacts to the environment by sending, or *generating* in the Statecharts terminology, events. Events are also the means of communication between parts of the system. Because we want to specify reactive systems at a high level of abstraction and in a discrete fashion, events are discrete signals, occurring at a point in time. Events have no duration: they either just occur at a particular moment or they don't occur. Consequently, the program construct that generates events should have no duration either. Events are generated at transitions from one state to the other. Hence, these transitions have this discrete, uninterruptable nature.

In [Lam83, Lam89] a specification method for concurrent systems is proposed. In this framework, all transitions from state to state are instantaneous. Only residing in a state has duration. When it comes to refining the specification, the state description becomes more detailed, but the transitions remain instantaneous as the "points of no return" in the computation of the next state. This way of specifying turns out to be very useful and Statecharts has also adopted it. In a state there is literally room for refinement, i.e. drawing substates. Transitions remain the instantaneous change-over from one state to another, no matter whether they are high or low in the hierarchy of states.

In a reactive system, there is another important reason for this choice. Because new input may arrive at any moment, it should always be clear what the current state is. Since transitions have no duration, there are no "transient" periods in between states and the reaction on a possible input is always well-defined. If we have a discrete time domain, the moment of time after the change succeeds immediately the moment before the change, there are no points of time in between.

This is an important advantage when you only have to deal with discrete events and discrete time. Of course, it is an abstraction from reality. When one looks deep enough into the electronic implementation, one will encounter a level where discrete reasoning makes no sense anymore. Statecharts, however, is meant to be a high level specification language that is used on a level where this abstraction can be maintained and is appropriate.

## 5.2 How long is the reaction time of the system?

We have only answered one part of the question about the timing of transitions. We know that they have no duration, but *when* do they take place, relative to the trigger? How long does it take the system to compute a reaction?

In a (not timed) *transformational* system the answer is easy. Any positive amount of time will do, because we are only interested in *whether* the output is produced (and what it is, of course) but not in *when* it is produced. So as far as time is concerned, the only important distinction is between finite and infinite values (corresponding to output or no output).

For *reactive* systems, however, this is not enough. Reactions can interfere with future inputs, so the moment they occur is important. Even if we don't have to quantify time explicitly, we need to know when an output occurs, relative to the events in the input sequence (see the Brock-Ackermann paradox in section 2). So we have to determine what the reaction time of the system is.

One approach is to specify for each situation a concrete amount of time. This is cumbersome and not in accordance to the level of abstraction we are aiming at, since it depends on the implementation. It forces us to quantify time right from the beginning. At this stage one is in most cases only interested in the relative order and the coincidence of events.

Another approach is fixing the reaction time to, say, one time unit (assume we have a discrete time domain). This is simpler, but still not abstract, since specifications using this principle are difficult to refine without changing their high level meaning.

In many applications, one uses several orthogonal components to describe parts of the system that are *conceptually* independent. In the example of the television set, the sound can be muted by pressing a button on the remote control (event *mute*) as well as by generating the internal event *sm* when there is a channel switch (see fig. 5). The latter causes a transition within the state SM and this generates the event *mute*. If we would adopt the approach of a fixed reaction time of 1 unit, muting by remote control would be faster than muting by a channel switch. There is absolutely no inherent reason why this should be so. Of course, this anomaly could be removed by shortening the reaction chain for automatic muting or artificially lengthening the other one, but this is not the point. Even if this would yield a "better" or a "more natural" specification, the problem remains that one has to watch closely the length of reaction chains and the moment in these chains that transitions are taken.

This is the typical clumsiness of a low level programming language: a local change of some statements affects the behaviour of the whole program. Note that such a local change need not be the correction of an error or something alike. It could well be the refinement of an action. E.g., the transition labelled

*question/answer*

one might refine to

*question/consult*

and a database server with a transition labelled

*consult/answer*

So in the process of development, the (syntactic) length of a computation is due to change. *A fixed execution time for syntactic entities* (transitions, statements, etc.) *is therefore not flexible enough.*

This approach has another disadvantage. In practice, a fixed amount of reaction time will be some kind of upperbound upon the execution times of different statements in different situations in the actual implementation. So the implementation will have to be artificially delayed in order to meet its specification. In many cases, however, we want the reaction to be as quick as possible. The delay of 1 time unit was only introduced for uniformity and the implementation is slower than necessary.

A third approach is to leave things open: only say that execution of a reaction takes some positive amount of time and see at a later stage (closer to the actual implementation) how much time things did take. This is also awkward, however, since it introduces a lot of non-determinism, which will make it difficult or even impossible to prove interesting things at an early stage of the development.

From the discussion above, we see that we want the execution time associated to reactions, or statements in general, to have the following three properties.

- It should be accurate, but not depending on the actual implementation.
- It should be as short as possible, to avoid artificial delays.
- It should be abstract in the sense that the timing behaviour must be orthogonal to the functional behaviour.

We believe that the only choice that meets all these wishes is a zero reaction time. It is precise and clearly as short as possible. It is also abstract: changing the functionality of a behaviour, e.g. extending the reaction chain or adding statements, does not affect the timing behaviour, since $0 + 0 = 0$. This is essentially *synchrony hypothesis* of Gérard Berry ([BG88]).

Naturally, the question arises whether this is implementable. Literally speaking, the answer is no because any real computation takes some time. In an actual implementation it means: the reaction comes before the next input arrives, or, so to say, reactions are not infinitely fast but at least fast enough. This is not unrealistic: in many cases the frequency of external events is low relative to the response time of the system. And even in case there may arise problems in the implementation, we believe that these should not interfere with the work of the programmer/designer at the first stages of development.

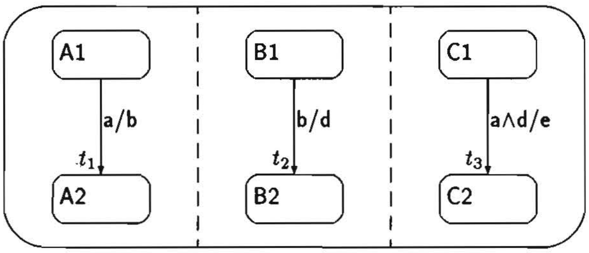


Figure 6:

In fig. 6 you see an example of the consequences of zero response time. When the system is in states $A_1$, $B_1$ and $C_1$, and event $a$ occurs somewhere, a chain reaction of transitions takes place from left to right: $t_1$ triggers $t_2$, $t_2$ triggers $t_3$. Nevertheless, all three transitions are considered to occur at the same time, as are events $a$, $b$, $d$ and $e$. That is the reason why transition $t_3$ will be taken: namely, event $d$ occurs simultaneously with event $a$ – even though it is generated two transitions "later" in the chain – and hence the compound event $a \wedge d$ is true.

## 6 Negations and paradoxes

The idea of immediate reaction works fine as long as transitions are only triggered by primitive events or conjunctions and disjunctions of primitive events. One needs, however, also negations of events in the trigger of a transition.

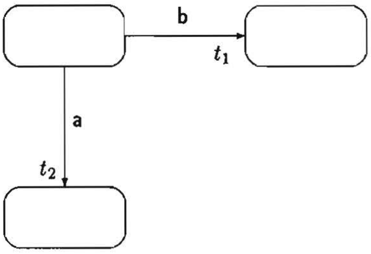


Figure 7:

E.g., to specify in fig. 7 that transition $t_1$ has priority over transition $t_2$, one should change the label of $t_2$ into $a \wedge \neg b$, meaning that the transition should only be taken when $a$ occurs and $b$ does not occur at the same time (in that case $t_1$ should be taken). Now consider the following example (see fig. 8).

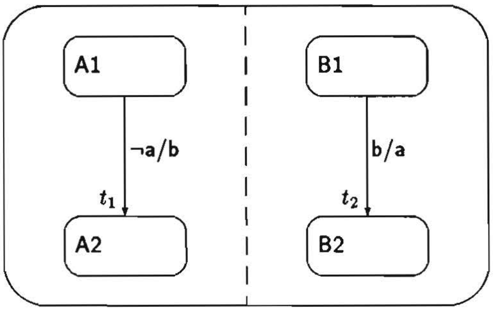


Figure 8: Causal paradox

Suppose at some moment $T$, $a$ and $b$ are not generated outside this statechart. Then transition $t_1$ can be taken, generating the event $b$. This causes $t_2$ to be taken, generating the event $a$. So apparently, transition $t_1$ should not be taken at all, since it is not enabled. But then $t_2$ can not be taken, because does not occur. Hence, $a$ does not occur and $t_1$ should be taken. *Etcetera ad infinitum.*

One might wonder of what use these kind of programs are. In Esterel, for instance, all programs in which the execution of a statement depends, directly or indirectly, on an event that is generated as a result of the exectuion of that statement are considered erroneous.

There are, however, statecharts that use these kind of causal loops in a very natural way. Fig. 9 shows a statechart that specifies in a succinct way that the critical sections $X_1$ and $X_2$ are occupied by at most one process at the time. The equivalent specification in Esterel would be rejected by the compiler.

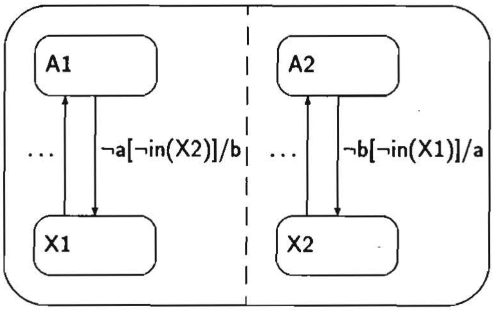


Figure 9: Mutual exclusion

So we are still faced with the problem: what semantics should we give to statecharts as given in fig. 8?

### 6.1 The micro-step approach

One of the solutions that has been adopted by Statecharts is the introduction of two levels of time ([HPPSS87]).

The top level counts time in *macro-steps*, these are the observable time steps. Every macro-step is divided in an arbitrary number of *micro-steps*. If one transition triggers another one, they will be taken in subsequent micro-steps, but within the same macro-step. Thus, the chain of causality inside one macro-step is modelled by a sequence of micro-steps. Although simultaneous at the top level, i.e., on the level of macro-steps, a transition can never affect transitions taken in previous micro-steps. So, in the example above, transition $t_1$ and $t_2$ are taken in two subsequent micro-steps. Transition $t_2$ cannot kill its trigger $t_1$, because the latter took place in an earlier micro-step.

This sequence of micro-steps has only operational meaning. In the semantic model [HGdR88] only the relative order of events are recorded, by means of a *total pre-order* for every macro-step:

$a < b$ means that the occurrence of $a$ is not dependent on $b$ in the current macro-step. Consequently, $a$ may cause transitions that generate $b$, but not the other way round.

$a < b$ and $b < a$ means that $a$ and $b$ cannot cause each other in the current macro-step, e.g., $a$ and $b$ are generated by the same transition.

This leads to a compositional, fully abstract semantics for Statecharts, described in the cited paper.

Compositional means that the semantics of a program is defined in terms of its syntactic components. In this graphical formalism, syntactic decompositions are, a.o., AND- and OR-decomposition. Compositionality is an important property when it comes to developing and verifying programs in a structured way, since it allows for specifying program parts, and hence for programming with specifications, independently of their implementation.

A semantics is fully abstract with respect to some other semantics if it does not distinguish more programs than is necessary for being compositional. The other semantics describes which programs should be distinguished at the least, e.g., on basis of an operational definition. Two programs that cannot be distinguished in any syntactic context must have the same semantics. Hence, relative to a particular notion of context, a fully abstract semantics is "best" since it introduces the least amount of redundancy required for compositional reasoning.

### 6.2 Another approach

A problem of the approach above is that macro-steps are no longer *globally consistent*. By this we mean that sometimes a transition is taken when the set of events that occur in the macro-step does not enable its trigger. In the example above (fig. 8), transition $t_1$ is taken even though event $a$ occurs in the macro-step. The reason that such a transition can nevertheless be taken is that it *was* enabled somewhere half way the chain of micro-steps, but it triggered other transitions and these generated events that disabled it.

In the paper [HG89] we study this problem of global consistency and formulate several desirable properties of a language for reactive systems. The most important of these are

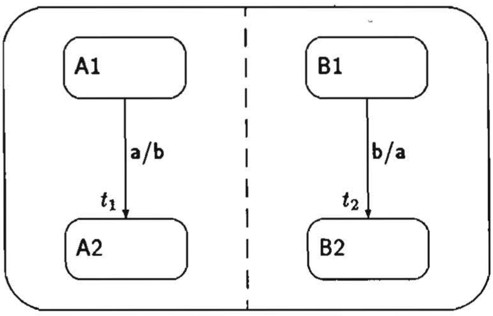


Figure 10:

modularity, causality and responsiveness. We call a semantics modular if the interaction of modules (orthogonal components) can be understood in terms of macro-steps; it implies global consistency. Every transition that is taken is enabled by the full set of events occurring at the current macro-step. Causality says that the behaviour at every macro-step can be decomposed into a sequence of transitions that form a causal chain; no transitions are "triggering each other". In fig. 10 you see an example of this: in some versions of the semantics, both transitions will be taken, even if $a$ and $b$ don't occur in the environment. Responsiveness is in essence the property we discussed in section 5.2, namely that reactions are simultaneous with their triggering actions.

The conclusion of the paper is that these three properties can not be combined into one semantics.

In Esterel ([BC85],[BG88],[Gon88]) this problem is circumvented by disallowing programs that would violate causality. The compiler detects these programs and refuses to compile them. Since this is a static check, some programs in which the paradox can never arise, e.g., because of specific values of variables, are nevertheless rejected.

In the paper [HG89] we follow another approach. We propose a two-level semantics in which the interaction between modules is different from the interaction of components within a module. Within a module, causality is achieved, thus providing an operational way of reasoning on a local scale; between modules modularity holds, thus providing a simple and clean interface that does not presuppose a detailed knowledge of the inside mechanics of the module.

## 6.3 Current Statecharts

Regrettably, the current implementation has left the micro-step approach and treats time in a more primitive fashion. There is never any synchrony between actions and reactions and hence the causal problems disappear. An event generated by a transition is only available during the next (micro-)step. Hence, the behaviour of a program is very dependent on the length of reaction chains. Time proceeds independently from the steps in the computation, but there is no consistency is enforced in what happens during one time step (in contrary to the approach of Esterel).

## 7 Conclusion

We introduced the notion of reactive systems and explained why they are different from the more conventional transformational systems. We introduced the graphical language Statecharts as a description formalism for reactive systems. For the first time the rationale behind the design decisions of Statecharts is explained in relation to the specific nature of reactive systems. In order to be able to react at any moment at an incoming input, transitions are instantaneous, whereas states have duration. To avoid accumulation of time in reaction chains, the reaction time should be zero.

Furthermore, we showed a semantic problem that arises when reactions take no time and we pointed out various solutions to this problem. These solutions were not discussed in detail; the reader is referred to the literature.

Finally, we want to answer the question of one of our reviewers who is afraid that "the beautiful, simple and elegant ideas of Statecharts perish in a muddle of 'transitions', 'events', 'actions', (...) etc. There must be a basis of, say, three basic notions and four clear, unique relations among them. But when trying to derive this, I quickly came into a confusion (...)." Although he has a point here, we can say that some work on this has been done in[HG89] and especially in [Mar90], where an improvement on the notion of refinement has been obtained.

## Acknowledgement

This paper arose out of work for the ESPRIT project DESCARTES (937) on Statecharts. In this project we collaborated intensely with Amir Pnueli and colleagues from AdCAD. Obviously, our paper owes a lot to their many presentations and their views upon Statecharts. We thank Iko Keesmaat for his help in referencing Petri nets and Wolfgang Reisig for his thorough reading and his important comments.

## References

[BA81] J.D. Brock and W.B. Ackerman. Scenarios: a model of non-deterministic computation. In Diaz and Ramos, editors, *Formalization of Programming Concepts*, LNCS 107, pages 252–259. Springer-Verlag, 1981.

[BC85] G. Berry and L. Cosserat. The synchronous programming language esterel and its mathematical semantics. In *Proc. CMU Seminar on Concurrency*, LNCS 197, pages 389–449, 1985.

[BG88] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. Technical report, Ecole Nationale Supérieure des Mines de Paris, 1988. Technical Report.

[DH88a] D. Drusinsky and D. Harel. On the power of cooperative concurrency. In F.A. Vogt, editor, *Proc. Concurrency '88*, LNCS 335, pages 389–449. Springer-Verlag, 1988.

[DH88b] D. Drusinsky and D. Harel. On the power of cooperative concurrency. In *Proceedings of Concurrency 88*, pages 74–103. Springer-Verlag, 1988.

[Gon88] G. Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones; Application à ESTEREL.* PhD thesis, University of Orsay, 1988.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 31:514 – 530, 1988.

[HG89] C. Huizing and R. Gerth. On the semantics of reactive systems. Deliverable in ESPRIT 3096 "SPEC", Eindhoven University of Technology, 1989.

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

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. NATO, ASI-13, Springer-Verlag, 1985.

[HPPSS87] D. Harel, A. Pnueli, J. Pruzan-Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proceedings Symposium on Logic in Computer Science*, pages 54–64, 1987.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley, Reading, 1979.

[*i-Logix Inc*87] *i-Logix Inc. The Languages of* STATEMATE, 1987. *in* Documentation for the STATEMATE System.

[JK89] B. Jonsson and J.N. Kok. Comparing two fully abstract dataflow models. In *Parallel Architectures and Languages Europe*, pages 217–234. LNCS 366, Springer-Verlag, 1989.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM TOPLAS*, 5:190–222, 1983.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[Mar90] F. Maraninchi. *Statecharts: sémantique et application à la spécification de systèmes.* PhD thesis, INP Grenoble, 1990.

# Chapter 2

# Full abstraction for a real-time denotational semantics for an OCCAM-like language

# Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language[1]

*C. Huizing*[2]
*R. Gerth*
*W.P. de Roever*

Department of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands,
October 1986

*ABSTRACT*

We present a fully abstract semantics for real-time distributed computing of the Ada and OCCAM kind in a denotational style. This semantics turns termination, communication along channels, and the time communication takes place, into observables. Yet it is the coarsest semantics to do so which is syntax-directed (this is known as full abstraction). It extends the linear history semantics for CSP of Francez, Lehman and Pnueli. Our execution model is based on maximizing concurrent activity as opposed to interleaving (in which only one action occurs at the time and arbitrary delays are incurred between actions). It is a variant of the maximal parallelism model of Salwicki and Müldner.

## 1. Introduction

Although real-time embedded systems are surrounding us in a growing number of applications, little reflection has been given to the theoretical foundations of their design. Here, one encounters problems of

- language design: what are the right primitives for prescribing real- time computing;
- semantics: what computational models underly real-time computing:
- syntax-directed specification: how does one express the behaviour of real-time systems, so as to allow modular design;
- verification: how does one prove the correctness of real-time programs.

Real-time languages include Ada[A83], OCCAM [Occ84], Chill [BW82], ESTEREL [BC85], LUSTRE [BCH85] and Statelan [Har84]. We are interested in real-time embedded systems, in which the system and the environment interact, yet are autonomous. Therefore, languages such as Esterel and LUSTRE, that express event driven and externally clocked systems, do not serve our purposes. Statelan has a highly developed expressive power as to concurrency and

[1]This paper is based on C. Huizing's M.Sc. Thesis [HGR85]

[2]The authors are working in and partially supported by Esprit Project 937; Debugging and Specification of Real-Time Embedded Systems (DESCARTES).

real-time. However, this very fact causes such problems when defining its semantics that no undisputed results on the meaning of the language exists. Finally, studies of Milner's [Mil83] and ourselves [KSRGA86] seem to indicate that on the level of model building synchronous communication (as in Ada and OCCAM) is more basic than asynchronous communication (as in Chill). This leaves us Ada and OCCAM to concentrate on.

Preceding studies [KVR83, KR85] on specification and verification of real- time systems stress the urgent need for a clear understanding of the underlying model.
*The primary aim of this paper is to find the right model for real-time, synchronously communicating distributed systems, and to prove that it is the right one, indeed, within that context.*

We cannot adopt the usual model based on arbitrary interleaving in order to treat concurrency, because this model allows arbitrary delays between any two actions of a process to occur. For real-time embedded systems, however, where time constraints are the rule, one at least should have an a priori bound on such delays, since otherwise real-time constraints can never be provably met. Our model, based on the notion of *maximal parallelism* [SM81], takes the view that no unnecessary delays are incurred at any time.

Since our ultimate aim is specifying and verifying the timing behaviour of a distributed system from the timing behaviours of its components, the specification language should refer to a global notion of time (cf. the analysis of local clock synchronization algorithms in [HMM85]).

So, our semantic model is based on maximizing activity and a global notion of time. On basis of this characterization, we define a denotational, so-called linear history, semantics along the lines of [FLP84]. In an independent way, we define what should be observable about the behaviour of a program. In principle, the semantics should record exactly this observable behaviour [OH86] in order to be syntax directed. Consequently, we search for the minimal amount of information additional to the observables that makes the semantics syntax-directed. In literature, such a semantics is known as *fully abstract* [Mil83, HP79].

In general, fully abstract, (hence) syntax directed, semantics derive their interest from the fact that they determine exactly the amount of information which must be expressed in a specification language for it to be syntax directed. That is, for allowing the specification of a composite construct to be expressed in terms of the specification of its components — the very basis of modular design.

The semantics of [KSRG86], our starting point, turns out to be not fully abstract. We modify this model and prove that a fully abstract model is obtained indeed. In compliance with the usual definition of full abstraction, we show that any two programs with a different semantics admit a different observable behaviour when embedded in an appropriate context and vice versa.

Basically, the semantics of a program is the set of all histories that can be called forth by an environment. Technically, these histories record the observable information. In our case, the latter expresses that the process is waiting for another process and is required to enforce maximal activity (namely, if *two* processes are waiting for each other, this behaviour is not maximal and hence should be ruled out). Therefore, the history of the denotation of the program $P$ that distinguishes $P$ from a program $Q$ need not be *observably* different from the histories in the denotation of $Q$.

We construct a context for these programs that exploits these non-observable differences. Whenever the history signals waiting, the context should not be waiting and vice versa. In that way, the combined behaviour of the program in this context is maximal, because there is no unnecessary waiting. We can construct the context in such a way that (1) this behaviour is observable and (2) any history with the same observable behaviour but with a different waiting behaviour will not be maximal in this context (because there will always be some unnecessary waiting). Hence, the other program cannot display this combined behaviour in the given context, resulting in the required observable difference.

A number of models are known from literature [RR86, Bro83, BM83] and our own work [KSRGA86]. For classical temporal logic, which treats time qualitatively, finally fully abstract models have been obtained [BKP86]; however, quantitative treatments of time based on temporal logic, such as needed for real-time [BH81, HS86, KR85, Mos83], have not yet reached that level of sophistication. Timed Petri Nets [BM83] display impressive power, but do not support modular design as enforced by Ada or OCCAM. [Bro83] gives a relevant and early study on real-time, in the context of functional languages. The aims of [RR86] are closest to ours, although their approach is based on some different decisions concerning the observability of programs.

The paper is structured as follows. In Section 2 we give the syntax of our programming language and its (intuitive) semantics. Section 3 presents our execution model and section 4 our notion of observable behaviour. A denotational semantics that is not fully abstract, yet is intuitively appealing, is given in Section 5. In Section 6 we give an operational semantics that defines the observable behaviour of a program and relate it to the denotational semantics. Section 7 is the heart of this paper. Herein we define and motivate full abstraction, modify the denotational semantics and prove that it is fully abstract. Section 8 draws some conclusions and states open problems. In the appendix the syntax of our language and it semantics are given.

## 2. The Language DNP-R

In this section the syntax and informal semantics of our OCCAM-like DNP-R are defined. Denotational and operational semantics of this language are given in Sections 5 and 6.

### 2.1. Syntax

**Definition 1.**

- *Var* is the set of program variables, ranged over by $x$.
- *Chan* is the set of channels, ranged over by $\alpha$; $A \subseteq Chan$

- $e$ denotes some expression, $b_i$ some boolean expression, and $n$ some integer-valued expression.
- The context-free syntax of DNP-R is given by the following BNF-grammar:

$$\begin{array}{lll} S & ::= & x := e \mid g \mid S_1; S_1 \mid IOC \mid *IOC \mid S_1||_A S_2 \mid [S]_A \\ g & ::= & \alpha!e \mid \alpha?x \ \textbf{wait}\ n \mid - \\ IOC & ::= & [\Box_{i=1}^{n}\ b_i; g_i \to S_i] \end{array} \qquad \Box$$

Next we impose some context-sensitive constraints. These are needed to ensure that (1) channels are unidirectional, connecting at most two processes and (2) no variable is shared between two processes. For this we need some more notation.

**Definition 2.** For any $S$, generated by the above grammar

- *ichan*($S$) denotes the set of **internal channels** of $S$, which is defined as the union of all sets $A$ occurring in any substatement $S_1||_A S_2$ of $S$.
- *chin*($S$) denotes the (external) **input-channels** of $S$, defined as the set of all channels $\alpha$ occurring in an input command $\alpha?x$ somewhere inside $S$ and not contained in *ichan*($S$).
- *chout*($S$) denotes the (external) **output-channels** of $S_1$, defined likewise.
- *hid*($S$) denotes the **hidden-channels** of $S$, defined as the union of all sets $A$ that occur in a construct $[S_1]_A$ somewhere inside $S$.
- *var*($S$) denotes the **write-variables** of $S$, defined as the set of all variables that occur in the left-hand side of an assignment or an input command somewhere in $S$. □

**Definition 3.** *Stat*, the set of all **DNP-R statements**, is the set of statements generated by the grammar in Definition 1, satisfying:

(i) if $S \in Stat$, then $chin(S) \cap chout(S) = \emptyset$

(ii) if $S_1||_A S_2 \in Stat$, then $S_1 \in Stat$ and $S_2 \in Stat$ and

(ii.1) $var(S_1) \cap var(S_2) = \emptyset$

(ii.2) $(chin(S_1) \cap chout(S_2)) \cup (chout(S_1) \cap chin(S_2)) \subseteq A$

(ii.3) $chin(S_1) \cap chin(S_2) = chout(S_1) \cap chout(S_2) = \emptyset$

(iii) if $[S]_A \in Stat$, then $S \in Stat$ and $A = ichan(S)$

(iv) if $S \in Stat$ and $S_1||_A S_2$ is a substatement of $S$, then none of the channels in $A$ occur anywhere outside $S_1||_A S_2$ in $S$.

□

**Examples.** The following statements are excluded by Definition 3.

ad (i). $\alpha?x; \alpha!0$

A process cannot send values to itself.

ad (ii.1). $x := 0||_{\emptyset}x := 1$
There are no shared variables.

ad (ii.2). $\alpha?x||_{\{\beta\}}\alpha!0$
The index set should at least contain $\alpha$.

ad (ii.3). $\alpha?x||_A\alpha?y$ and $\alpha!1||_A\alpha!2$
A channel connects exactly two processes: one sender and one receiver.

ad (iii). $[\alpha?x||_{\{\alpha\}}\alpha!0]_{\emptyset}$
The index set of the hiding operator should be $\{\alpha\}$.

ad (iv). $[\alpha?x||_{\{\alpha\}}\alpha!1]_{\{\alpha\}};[\alpha?y||_{\{\alpha\}}\alpha!1]_{\{\alpha\}}$
Although hidden, the name $\alpha$ may be used only for one channel. Otherwise, we cannot impose the global maximality constraints in the definition of the operational semantics (see Chapter 6). As we have no procedures, this restriction raises no problems. For the denotational semantics, which we will prove fully abstract, this restriction can be dropped.

## 2.2. Informal semantics

The intuitive meaning of **sequential composition** $(S_1; S_2)$ should be clear.

The **output command** $\alpha!e$ sends the value of expression $d$ along channel $\alpha$. The **input command** receives a value on channel $\alpha$ and stores it in the variable $x$. An input action has to synchronize with an output action and vice versa. Consequently, execution of such an action may involve waiting until a communication partner becomes available. Our execution model will ensure that such waiting is minimized. In the **parallel composition** $S_1||_AS_2$ the components $S_1$ and $S_2$ are executed concurrently and synchronously. $A$ is a set containing the joint channels of $S_1$ and $S_2$ and explicitly gives the communications that have to be synchronized.
**Hiding** $[S]_A$ of statement $S$ has no effect on its execution but changes what can be observed about such an execution: communications along channels in $A$ are internalized and cannot be observed anymore.

The **iterative command** $\star IOC$ stands for repeated execution of the I/O guarded conditional $IOC$ (see below) while at least one of the boolean expressions $b_i$ yields true.

The **empty statement** – is like a skip action but takes zero time. It allows us to have pure boolean guards and empty branches in a guarded conditional.

The **Input/Output guarded Conditional** $[\Box_{i=1}^{n}\ b_i; g_i\ \rightarrow\ S_i]$ allows waiting for a set of I/O-commands, namely, the set of all commands $g_i$ for which the boolean expression $b_i$ yields true. If the guard $g_i$ is empty (–), the branch $S_i$ can be executed if $b_i$ yields true. If none of the booleans yields true, the conditional does not fail, but is skipped. There is no priority of local actions over communications or vice versa.
A conditional may also contain wait-guards, $b_i$; **wait** $n$. Such a wait-guard is passed as soon

as the associated waiting time, $n$, has elapsed (provided $b_i$ evaluates to true). As indicated earlier, such wait-guards allow waiting for I/O-actions to time out. Local actions and communications have priority over passing wait-guards.

## 3. Real-time execution model

As stated in the introduction, our semantics is based on the maximal parallelism model of [SM81]. This model is intended to express the behaviour of a system in which every concurrent process runs on its own dedicated processor. Hence, no unnecessary delays are incurred. More specifically, the model suspends process execution only in case no local action is possible and no partner is available for communication. As soon as an action becomes possible, execution must proceed.

To illustrate the effect of this model of execution, consider the program
$P; \alpha!3 ||_{\{\alpha\}} [\alpha?x \rightarrow - \Box \beta?x \rightarrow -] ||_{\{\beta\}} Q; \beta!4$ .

Here $P$ and $Q$ denote two terminating programs, not containing I/O-actions. Two scenarios are possible:

1. The value 3 is sent along $\alpha$ and the third component gets stuck (deadlock).

2. The value 4 is sent along $\beta$ and the first component gets stuck.

In models that allow finite but unbounded delay of actions, such as interleaving models, both scenarios are always possible. In our model, however, both scenarios are only possible if $P$ and $Q$ terminate at the same time. If $P$ terminates before $Q$, the communication on $\alpha$ will be performed immediately and, hence, the communication on $\beta$ will not occur and vice versa if $Q$ terminates before $P$. Consequently, in our execution model the choice of communication is highly dependent on the timing behaviour of the components.
To obtain a manageable and analyzable semantics, the following idealizations are imposed. Time proceeds in discrete steps. Every elementary action (assignment, communication, passing a guard) takes one time step[3]. In a parallel statement, processes start executing simultaneously.

## 4. Observable behaviour

The decision as to what should be observable about a program and what not, is closely connected to the purpose of the language. As our language should be able to describe real-time reactive systems [Pnu85], which are continuously interacting with the environment and

[3]We take the view that evaluating an expression takes time. Hence, **wait** $n$, even if $n$ evaluates to 0, takes 1 time unit.

often non-terminating, these interactions should be observable. Therefore, the observable behaviour of a program includes the sequences of communications and the time at which they occur. It also includes the program state at start and, in case the program terminates, the final state. Deadlock is deliberately *not* an observable entity. Nevertheless, we *can* observe indefinite suspension of execution, as we can observe the progress of time. Consequently, we can distinguish deadlocked programs from normally terminated ones, but we cannot distinguish them from (internally) divergent programs. This contrasts with [RR86] in which deadlock and non-termination observably differ.

With any program $P$ and starting state $\sigma$ we associate its set of possible behaviours: $\mathcal{O}[\![P]\!]\sigma$. This is formalized by the operational semantics in Section 6. A behaviour part is a pair $<\tau, h>$, where $\tau$ is the end state if $P$ terminates — otherwise $\tau = \infty$ — and $h$ is a, possibly infinite, sequence (also called history) of time records, each time record being a set of communication records; if the value $v$ was sent along channel $\alpha$ at the $t$-th time step, then the communication record $\alpha_v$ is a member of the $t$-th element of $h$. Hence, the length of $h$ corresponds to the time of termination. If several communications occur simultaneously, then this set contains more then one record. The empty set in a sequence implies that one time step passed without anything observable happening. This occurs when every active process was either waiting or doing an internal action (assignment, passing a guard).

**Definition 4:** we adopt the following notation.

- $\lambda$ stands for the empty sequence
- $t^n$ represents the $n$-fold repetition of time record $t$
- $h_1h_2$ represents the concatenation of the sequences $h_1$ and $h_2$
- $|h|$ denotes the length of the sequence $h$
- $h[i]$ represents the $i$-th element of the sequence $h$; if $i > |h|$, we define $h[i] = \emptyset$
- $h \lceil A$ denotes the restriction of sequence $h$ to the set of channels $A$ : $(h\lceil A)[i] = \{\alpha_v \in h[i] | \alpha \in A\}$
- $h_1 < h$ ("$h_1$ is a prefix of $h$") iff there exists a sequence $h_2$ such that $h_1h_2 = h$.

□

## 5. Denotational Semantics

Our denotational semantics, $\mathcal{D}$, is a linear history semantics along the lines of [FLP84]. The domain consists of non-empty, prefix-closed sets of pairs; each pair consisting of a state or bottom ($\perp$) and a finite history leading to that state. A bottom-state indicates that the pair corresponds to an incomplete computation. Infinite behaviours are modeled by their sets of finite approximations (and not by $<\infty, h>$ as in the operational semantics).
To give sense to the notion of approximation, we turn our domain into a complete partial order (cpo) with set inclusion as the ordering relation and $<\perp, \lambda>$ as least element. All

denotations, $\mathcal{D}$, will be **prefix-closed**. This means that for any $<\sigma, h>\in \mathcal{D}$ and $h' < h$ we have $<\perp, h>\in \mathcal{D}$. This cpo structure also allows us to define the semantics of the iterative construct as a least fixed point. For more information on cpos and their use in denotational semantics, see [deB80]. In order to enforce maximal progress, the denotational semantics has to record whether processes are suspended and on which communications they are suspended. This is done by adding so-called "readies"[4] to the sets in the histories. The presence of a ready $R_\alpha$ in a history has as meaning: *some process was waiting during this time for a communication along channel* $\alpha$. E.g., the denotation of the program $P = \alpha!0$ includes the pairs $<\sigma, \alpha_0>$ , $<\sigma, R_\alpha\alpha_0>$ , $<\sigma, R_\alpha R_\alpha \alpha_0>, \ldots$, whereas the denotation of $Q =$ **wait** $1; \alpha?x$ includes $<\sigma', \emptyset\alpha_v>$ , $<\sigma', \emptyset R_\alpha \alpha_v>$ , $<\sigma, \emptyset R_\alpha R_\alpha \alpha_v>, \ldots$ for any value $v$, each pair signifying a longer period of waiting ($\sigma$ is the starting state, $\sigma'$ is defined by $\sigma'(x) = v$ and $\sigma'(y) = \sigma(y)$ for $y \not\equiv x$).
These histories reflect the idea that the semantics must give the meaning of a statement in every environment, since the actual environment is not known. Now, if we execute $P$ and $Q$ in parallel, due to the maximality in our model, communication will happen at the earliest possible time, hence, at time step 2. So, we have to discard all histories that express a longer period of waiting. Thus, in the parallel merge of two denotations we only combine *consistent* histories. I.e., we combine only those histories that

(i) have no common readies at the same time. So, e.g., $R_\alpha R_\alpha \alpha_0$ and $\emptyset R_\alpha \alpha_0$ are *not* consistent. Thus maximality is enforced.

(ii) agree on the communications on the joint channels. So, e.g., $\alpha_0$ and $\emptyset\alpha_0$ or $R_\alpha\alpha_0$ and $\emptyset\alpha_1$ are *not* consistent. Thus synchronization of communications is expressed.
To be more specific, the semantics of the parallel composition is as follows:

$$\begin{aligned}\mathcal{D}[\![P_1||_A P_2]\!]\sigma = Cl\{ \; & <\sigma_1||_\sigma\sigma_2, h_1||_A h_2> \; | \\ & <\sigma_i, h_i>\in \mathcal{D}[\![P_i]\!]\sigma \;, \\ & maximal(h_1, h_2), synchronous(h_1, h_2, A) \;, \\ & comparable(\sigma_1, h_1, \sigma_2, h_2)\} \;.\end{aligned}$$

This definition uses the following operators.

$Cl$ is the closure function that extends a set to the smallest prefix-closed set that contains it.
$\sigma_1||_\sigma\sigma_2$ is a strict function defined by

$$(\sigma_1||_\sigma\sigma_2)(x) \overset{def}{=} \begin{cases} \sigma_1(x) & \text{if } \sigma_1(x) \neq \sigma(x) \\ \sigma_2(x) & \text{if } \sigma_2(x) \neq \sigma(x) \;, \\ \sigma(x) & \text{otherwise} \end{cases}$$

(hence, $\sigma_1||_\sigma \perp = \perp ||_\sigma\sigma_2 = \perp$).
This definition is unambiguous, because $P_1$ and $P_2$ cannot both change $x$ (there are no shared variables).

$h_1||_A h_2$ is defined by

$$(h_1||_A h_2)[j] \overset{def}{=} (h_1[j] \cup h_2[j]) \backslash \{R_\alpha | \alpha \in A\}$$

[4]This terminology comes from the ready-set semantics for TCSP [OH86]. There, a ready also signifies the willingness of a process to communicate in the future. No such willingness is implied here.

(remember the convention that $h[j] = \emptyset$ if $j > |h|$).
This is the pointwise union, except that readies on channels in $A$ are not preserved; they are not needed anymore.
$maximal(h_1, h_2) \overset{def}{\Longleftrightarrow} \forall j, \alpha : R_\alpha \notin h_1[j] \cap h_2[j]$
embodies the maximality constraint of (i) above,
$synchronous(h_1, h_2, A) \overset{def}{\Longleftrightarrow} \forall j, \alpha, v : \alpha \in A \rightarrow (\alpha_v \in h_1[j] \leftrightarrow \alpha_v \in h_2[j])$
expresses synchrony as in (ii) above,
$comparable(\sigma_1, h_1, \sigma_2, h_2) \overset{def}{\Longleftrightarrow} \forall u \in \{1,2\} : \sigma_i = \perp \rightarrow |h_{3-i}| \leq |h_i|$
guarantees that no incomplete history will be merged with a longer one. □
To understand the necessity of this constraint, consider the program $P = \alpha!0 ||_{\{\alpha\}} \alpha?x$

$$\text{Then, e.g., } <\perp, R_\alpha R_\alpha> \in \mathcal{D}[\![\alpha?x]\!]\sigma$$

and $<\perp, x> \in \mathcal{D}[\![\alpha!0]\!]\sigma$
Without the comparability check we would have
$<\perp, \emptyset\emptyset> = <\perp ||_\sigma \perp, R_\alpha R_\alpha ||_{\{\alpha\}} \lambda> \in \mathcal{D}[\![\alpha?x ||_{\{\alpha\}} \alpha!0]\!]\sigma$
This would imply that there exists a possible computation of $P$ that takes at least two time steps. The intended meaning of $P$ is, however, that it should terminate immediately after one time step, during which the successful communication took place.

These three constraints together will be referred to as *consistency*. The full definition of $\mathcal{D}$ can be found in appendix A.

## 6. Operational Semantics

We give an operational semantics $\mathcal{O}$, by defining a syntax-directed transition system along the lines of Plotkin [Plo83] and by imposing, in a second stage, a notion of maximizing progress globally on this system. Thus, maximality is enforced by local constraints during parallel composition in the denotational semantics, and is enforced in the operational semantics by globally constraining the possible behaviours of a program. Hence, $\mathcal{O}$ captures – indeed defines – exactly the observable behaviour in a way that is independent from the denotational semantics.

### 6.1. The Labelled Transition System

As expected, the operational semantics is based on a labelled transition relation that transforms configurations consisting of pairs of statements and states. We write

$$(P, \sigma) \xrightarrow{L} (P', \sigma')$$

if the statement $P$ in state $\sigma$ can be transformed into statement $P'$ in state $\sigma'$ in *one time step*. The *label* $L$ consists of two components: $L^C$, the set of *communications* that take place during this step and $L^N$, a natural number indicating the number of *local actions* that are performed during the step. The second component is needed to define in the second stage the maximality of steps.

In the appendix this transition relation is inductively defined by a set of axioms and rules. Here, we discuss some representative cases:

**assignment:**
$(x := e, \sigma) \xrightarrow{(1,\emptyset)} (-, \sigma\{\sigma(e)/x\})$
The statement $x := e$ terminates in one step; this is indicated by the empty statement on the right-hand side. The state is updated accordingly. The empty set in the label denotes that no communications take place and the number 1 indicates that one local action is performed.

**output command:**
$(\alpha!e, \sigma) \xrightarrow{(0,\alpha_{\sigma(e)})} (-, \sigma)$
$(\alpha!e, \sigma) \xrightarrow{(0,\emptyset)} (\alpha!e, \sigma)$
In the first transition the communication is performed and hence the statement terminates. In the second transition the process waits for a communication. This waiting is not considered to be a local action.
As in Plotkin's operational semantics for CSP [Plo83], the first axiom involves assumptions about the availability of communication partners. These assumptions are validated in the parallel-rule. Unlike Plotkin's semantics, the second axiom involves assumptions about the absence of communication partners. Such assumptions are validated at the second stage, when maximality is imposed.

**parallel statement:**
$$\frac{L_1^c\lceil_A = L_2^c\lceil_A, (P_1,\sigma) \xrightarrow{L_1} (P_1',\sigma_1'), (P_2,\sigma) \xrightarrow{L_2} (P_2',\sigma_2')}{(P_1||_A P_2, \sigma \xrightarrow{L} (P_1'||_A P_2', \sigma')}$$
where $L = (L_1^n + L_2^n, L_1^c \cup L_2^c)$

$$\sigma' \text{is defined by } \sigma'(x) = \begin{cases} \sigma_1'(x) & \text{if } x \in var(P_1) \\ \sigma_2'(x) & \text{if } x \in var(P_2) \\ \sigma' & \text{otherwise} \end{cases}$$

The condition $L_1^c\lceil_A = L_2^c\lceil_A$ guarantees that all communications along channels in $A$ are synchronized.
Note that this rule enforces that in both components time proceeds. This is in accordance with our real-time model.

## 6.2. Imposing Maximality

The above transition system generates non-maximal computation steps, too. For instance, the program

$$P = \alpha?x||_{\{\alpha\}}\alpha!3$$

admits both the transitions
a) $(P, \sigma) \xrightarrow{0,\{\alpha_3\}} (-||_{\{\alpha\}}-, \sigma\{3/x\})$ and
b) $(P, \sigma) \xrightarrow{(0,\sigma)} (P, \sigma)$.
In the latter transition both $\alpha?x$ and $\alpha!x$ are unnecessary idling.

We shall rule out such a transition by imposing an order on transition labels, and by requiring in Definition 7 of our operational semantics below that all transitions are maximal with respect to this order.

**Definition 5.**
Let $I$ be a set of channels, $\leq_I$ is an order relation on labels, defined as follows:

$$(n_1, c_1) \leq_I (n_2, c_2) \Leftrightarrow n_1 \leq n_2 \wedge c_1 \restriction I \subseteq c_2 \restriction I \wedge c_1 \restriction \bar{I} = c_2 \restriction \bar{I}$$

□

where $\bar{I}$ stands for the complement of $I$.

**Definition 6.**
A transition $(P, \sigma) \xrightarrow{L} (Q, \tau)$ is *maximal* iff for every $L', Q', \tau'$ with $(P, \sigma) \xrightarrow{L'} (Q', \tau')$ and $L \leq_I L'$ we have $L = L'$, where $I = ichan(P)$.
Now we see that transition b) in the above example is not maximal, because $(0, \emptyset) \leq_{\{\alpha\}} (0, \{\alpha_3\})$.
This leads to the operational semantics $\mathcal{O}$ (a full definition can be found in appendix B).

**Definition 7.**
Let $P_0$ be a DNP-R program and $\sigma_0$ a state.

$$\begin{aligned} &\mathcal{O}[\![P_0]\!]\sigma_0 = \\ &\{<\sigma, h> \; | \exists \vec{P_i}, \vec{\sigma_i}, \vec{L_i} : \forall 1 \leq i \leq |h| : \\ &\qquad (P_{i-1}, \sigma_{i-1}) \xrightarrow{L_i} (P_i, \sigma_i) \; \textit{is maximal}, h[i] = L_i^C \restriction vis(P_{i-1}) \\ &\qquad \wedge (|h| < \infty \rightarrow \sigma = \sigma_{|h|} \; \wedge \; \textit{terminated} \; (P_{|h|}) \\ &\qquad \wedge (|h| = \infty \rightarrow \sigma = \infty) \\ &\}, \end{aligned}$$

where $vis(P) = Chan \backslash hid(P)$, the *visible* channels of $P$ and *terminated* $(P)$ is a predicate that is true if $P$ consists only of empty statements, combined with $||_A$, ; or $[\cdot]_A$.
The operational and denotational semantics, $\mathcal{O}$ and $\mathcal{D}$, are related by the following.

**Theorem 1.** $\mathcal{O} = \beta \circ \mathcal{D}$, where $\beta$ is an abstraction function. □

Here, $\beta$ deletes all non-observable information, viz. readies, and smoothes away the differences between the two domains; e.g., infinite chains of finite histories are replaced by their limits. The tedious proof of this theorem can be found in appendix B. It is non-trivial, as it proves the equivalence of two completely different ways of expressing maximality.

## 7. A Fully Abstract Semantics

Define a context $C$ as a program with several "holes" in it; let $C(P)$ denote the program obtained by replacing each of these holes by the program text $P$.

**Definition 8**: a semantics $\mathcal{D}$ is **fully abstract** w.r.t. a semantics $\mathcal{O}$ iff for all programs $P$ and $Q$ : $\mathcal{D}[\![P]\!] = \mathcal{D}[\![Q]\!] \Leftrightarrow \forall$ contexts $C$ : $[\![C(P)]\!] = \mathcal{O}[\![C(Q)]\!]$. □

This definition can be found in the literature [HP79]. Its motivation lies in the following

(Folk?) **Theorem 2:** $\mathcal{D}$ is fully abstract w.r.t. $\mathcal{O}$ iff

(i) $\mathcal{D}$ is syntax-directed,

(ii) $\mathcal{D}$ is the coarsest semantics that distinguishes at least as much as $\mathcal{O}$ does. □

This notion of full abstraction is too restrictive for our purposes, as it assumes that the syntax is context-free. In view of the fact that DNP-R has a context-sensitive syntax, we use the following modification.

**Definition 9.** $P, Q \in Stat$ are **syntactically comparable** iff for any context $C$ holds

$$C(P) \in Stat \Leftrightarrow C(Q) \in Stat$$

□

In effect, this boils down to $P$ and $Q$ having the same sets *chin, chout, ichan,* and *var*. We redefine full abstraction as follows:

**Definition 10.** A semantics $S$ is **fully abstract** w.r.t. a semantics $\mathcal{O}$ iff for all *syntactically comparable* programs $P$ and $Q$:
$\mathcal{D}[\![P]\!] = \mathcal{D}[\![Q]\!] \Leftrightarrow \forall$ contexts $C$ : $\mathcal{O}[\![C(P)]\!] = \mathcal{O}[\![C(Q)]\!]$. □
Relative to this modified notion, $\mathcal{D}$ is not fully abstract with respect to $\mathcal{O}$ either. We can show this with the aid of an example — the usual example that shows that the readiness semantics of *CSP* [Plo83] is not abstract. The following programs

$$P = [\mathbf{true} \rightarrow \alpha!0 \Box \mathbf{true} \rightarrow \beta!0 \Box \mathbf{true} \rightarrow [\alpha!0 \rightarrow - \Box \beta!0 \rightarrow -]]$$

$$Q = [\mathbf{true} \rightarrow \alpha!0 \Box \mathbf{true} \rightarrow \beta!0]$$

have different semantics and yet cannot be distinguished by any context.
The solution to this problem is taking the *convex-closure* of program denotations:
if $< \sigma, h_1 R h_2 >, < \sigma, h_1 R_2 h_2' > \in \mathcal{D}$ then all pairs $< \sigma, h_1 R h_2 >$ with $R_1 \subseteq R \subseteq R_1 \cup R_2$ are added to $\mathcal{D}$.
Although this turns the readiness semantics into a fully abstract one, this does not suffice in our case. Consider for instance the two programs

$$P = [\mathbf{true} \rightarrow \mathbf{wait}\ 1; \alpha!0 \Box \mathbf{true}[\rightarrow \alpha!0 \rightarrow - \Box \mathbf{true} \rightarrow -] \Box\ \mathbf{true} \rightarrow \alpha!0]$$

$$Q = [\mathbf{true} \rightarrow \mathbf{wait}\ 1; \alpha!0 \Box \mathbf{true}[\rightarrow \alpha!0 \rightarrow - \Box \mathbf{true} \rightarrow -].$$

These two programs have different denotations, since e.g., the pairs $< \sigma, \emptyset R_\alpha \alpha_0 >$, $< \sigma, \emptyset R_\alpha R_\alpha \alpha_0 >$,... occur in the denotation of $P$, but do not occur in the denotation of $Q$ (the first branch does not generate an $R_\alpha$ in the second time step, neither does the inner guarded statement). However, there is no context $C$ that can *separate* these programs: $\mathcal{O}[\![C(P)]\!] = \mathcal{O}[\![C(Q)]\!]$ for any context $C$. Before we explain this, we introduce a useful notation.

**Definition 11.** A history $h'$ is a **ready-extension** of a history $h$, notation $h' \subseteq_R h$, off $|h'| = |h|$ and for any $i \leq |h|$:

$$h'[i] \setminus h[i] \leq \mathit{Readies},$$

where *Readies* $= \{R_\alpha | \alpha \in \mathit{Chan}\}$. □

Note that a history which is consistent with $h'$ is also consistent with $h$, if $h' \geq_R h$.
A ready can only be observed indirectly as a result of its function in the parallel merge of denotations; it prevents the history in which it occurs from merging with any other history with a ready on the same channel at the same time instant. Now, observe that above, every distinguishing history in the denotation of $P$ is, in fact, a ready-extension of one in the denotation of $Q$.

In order to make the semantics fully abstract, it indeed suffices to add all histories that are ready-extensions of histories in the original denotations.

**Definition 12.** $\mathcal{D}_a[P]\sigma = \{< \sigma', h > | \exists h' :< \sigma', h' > \in \mathcal{D}[P]\sigma \text{ and } h \supseteq_R h'\}$[5] □

**Theorem 3:** $\mathcal{D}_a$ is fully abstract with respect to $\mathcal{O}$:

$$\mathcal{D}_a[\![P]\!] = \mathcal{D}_a[Q] \Leftrightarrow \forall C\ \mathcal{O}[C(P)] = \mathcal{O}[C(Q)]\,.$$

□

We prove this theorem by two lemmas.

**Lemma 1.** $\mathcal{D}_a[P] = \mathcal{D}_a[\![Q]\!] \Rightarrow \forall C : \mathcal{O}[\![C(P)]\!] = \mathcal{O}[\![C(Q)]\!]$ □

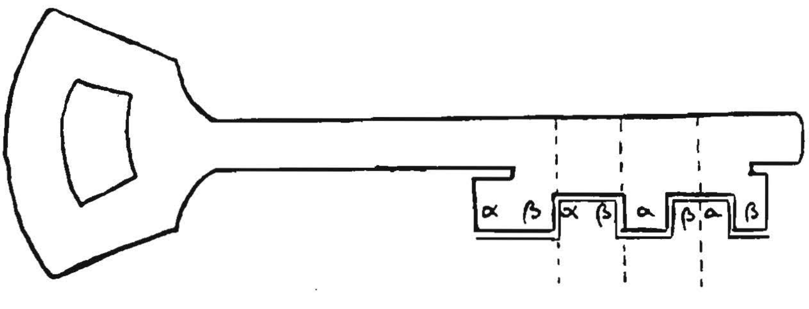
**Lemma 2.** $\mathcal{D}_a[P] \neq \mathcal{D}_a[\![Q]\!] \Rightarrow \exists C : \mathcal{O}[\![C(P)]\!] \neq \mathcal{O}[\![C(Q)]\!]$ □

From these two lemmas, we immediately infer Theorem 2.
*Proof of Lemma 1*: Suppose $\mathcal{D}_a[\![P]\!]\sigma = \mathcal{D}_a[\![Q]\!]\sigma$ and let $C$ be given.
Because $\mathcal{D}_a$ is defined using induction on the structure of DNP-R, we have $\mathcal{D}_a[\![C(P)]\!]\sigma = \mathcal{D}_a[\![C(Q)]\!]\sigma$ and hence $\beta(\mathcal{D}_a[\![C(P)]\!]\sigma) = \beta(\mathcal{D}_a[\![C(Q)]\!]\sigma)$.
From $\mathcal{O} = \beta \circ \mathcal{D}$ we can easily infer $\mathcal{O} = \beta \circ \mathcal{D}_a$ and hence
$\mathcal{O}[\![C(P)]\!]\sigma = \mathcal{O}[\![C(Q)]\!]\sigma$. □

Lemma 2 is the more complicated and interesting one. We first give a sketch of the proof.

Assume $< \tau, k > \in \mathcal{D}_a[\![P]\!]\sigma \setminus \mathcal{D}_a[\![Q]\!]\sigma$. On the basis of this history $k$, we shall construct a program $L$ that produces the "ready-complement" $l$ of $k$ (and some state $v$). E.g., if $k = \{R_\alpha, R_\beta\}\emptyset R_\alpha R_\beta$, and the only channels occurring in $P$ and $Q$ are $\alpha$ and $\beta$, then $l = \emptyset\{R_\alpha, R_\beta\}R_\beta R_\alpha$. (This will be our running example in what follows.) If we see $k$ as a key, then $l$ is a lock that fits as tightly as possible around the key (see figure).

[5]This semantics is also syntax-directed, and, by the nature of $\beta$, also $\mathcal{O} = \beta \circ \mathcal{D}_a$ holds.



Obviously, $l$ is consistent with $k$ when performing a parallel merge (i.e., in our analogy, the key can turn in the lock). For any other history $k'$ that is consistent with $l$, and that has the same observable behaviour as $k$, we have $k \supseteq_R k'$ (i.e. any other key that turns in the lock will fit "more loosely" then $k$). This follows from the construction of $l$ as the tightest lock fitting around $k$. Hence, $< \tau, k' > \notin \mathcal{D}_a[\![Q]\!]\sigma$, since otherwise $< \tau, k > \in \mathcal{D}_a[\![Q]\!]\sigma$, contradicting our initial assumption by definition of $\mathcal{D}_a$. So, $< \tau \,\|_\sigma\, v, k' \,\|\, l > \in \mathcal{D}_a[\![P \,\|\,]\!]\sigma$, but $< \tau \,\|_\sigma\, v, k' \,\|\, l > \notin \mathcal{D}_a[\![Q \,\|\, L\,]\!]\sigma$ for any $k'$ with the same observable behaviour as $k$.

There is one problem left since there might be other histories, *not* observationally equivalent with $k$, which are consistent with $l$, i.e., which can open the lock. E.g., history $\emptyset\emptyset$ is able to merge with $l$.
Although $\emptyset\emptyset$ and $k$ are observably different (there is a difference in termination time), $\emptyset\emptyset \,\|\, l$ is observably the same as $k \,\|\, l$. In fact, any such history has to be of smaller length than $k$. We detect such "forged" keys by making the lock sensitive to the length of the key. If we take as a context $((< \mathit{hole} >; \gamma!0) \,\|\, \gamma?x) \,\|\, L$, where $\gamma$ is a channel not occurring in $P, Q$ or $L$, then the occurrence of the communication along $\gamma$ will serve to indicate termination of the program in the hole ($P$ or $Q$). This makes visible the length of a shorter history that merges with $l$ at the time of which this communication occurs. Hence, this context separates $P$ and $Q$.
Before proving Lemma 2 we introduce some notation and auxiliary lemma's:

**Definition 13.**

- If $h$ is a history, then its **observable reduct** $h^c$ is defined by $h^c[i] = \{\alpha_v | \alpha_v \in h[i]\}$
- If $\mathcal{D}$ is a denotation, then its observable reduct $\mathcal{D}^c$ is defined by $\{< \sigma, h^c > \mid < \sigma, h > \in \mathcal{D}\}$ □

**Lemma 3.** If there is a $< \tau, h >$ such that $< \tau, h > \in (\mathcal{D}_a[\![P_1]\!]\sigma)^c$ and $< \tau, h > \notin (\mathcal{D}_a[\![P_2]\!]\sigma)^c$ then $\mathcal{O}[\![P_1]\!]\sigma \neq \mathcal{O}[\![P_2]\!]\sigma$.

*Proof*: Suppose $< \tau, h >$ is given as in the lemma. There are two cases:

(a) If $\tau \neq \perp$, it is clear that $<\tau, h> \in \beta(\mathcal{D}_a[\![P_1]\!]\sigma) \backslash \beta(\mathcal{D}_a[\![P_2]\!]\sigma)$ and hence $\mathcal{O}[\![P_1]\!]\sigma \neq \mathcal{O}[\![P_2]\!]\sigma$.

(b) If $\tau = \perp$ there must be

  (i) some $<\tau', h'> \in \mathcal{D}_a[\![P_1]\!]\sigma$ with $\tau' \neq \perp$ and $h \leq h'$, or

  (ii) an infinite chain $(h_n)_{n \in N}$ with $<\perp, h_n> \in \mathcal{D}_a[\![P_1]\!]\sigma$ for all $n$ and $h' = h_n$ for some $n$.

This fact is an immediate consequence of the definition of $\mathcal{D}_a$.
If case (i) applies, we see that $<\tau', h'^c> \notin (\mathcal{D}_a[\![P_2]\!]\sigma)^c$, (otherwise $<\tau, h> \in (\mathcal{D}_a[\![P_2]\!]\sigma)^c$ by prefix-closure), and we can apply the lemma, since $\tau' \neq \perp$ and this case has been proven already.
If case (ii) applies, we know that $<\infty, h^\infty> \in \beta(\mathcal{D}_a[\![P_1]\!]\sigma)$ where $h^\infty = lim_{n \to \infty} h_n^c$. Because, for some $n, <\perp, h_n> \notin \mathcal{D}_a[\![P_2]\!]_\sigma, <\infty, h^\infty> \notin \beta(\mathcal{D}_a[\![P_2]\!]\sigma)$ and hence $\mathcal{O}[\![P_1]\!] \neq \mathcal{O}[\![P_2]\!]\sigma$. □

The following is the key-lemma in the proof.

**Lemma 4.** Let a history $k$, a state $\sigma \neq \perp$, a set of input channels $I$, and a set of output channels $\mathcal{O}$ be given. Assume that $k$ has the property that whenever $\alpha_v \in k[i]$ and $\alpha_w \in k[i]$ then $v = w$[6].
Then there exists a program $L$ and a state-history pair $<\nu, l>$ with the following properties:

(i) $<\nu, l> \in \mathcal{D}_a[\![L]\!]\sigma$

(ii) $\nu \neq \perp$ and $|l| = |k|$

(iii) $chin(L) = I, chout(L) = \mathcal{O}$

(iv) for all $1 \leq i \leq |l|$ and all $\alpha \in I \cup \mathcal{O}$:
$\alpha_v \in k[i] \leftrightarrow \alpha_v \in l[i]$
$R_\alpha \in k[i] \leftrightarrow R_\alpha \in l[i]$

(v) for all $<\nu', l'> \in \mathcal{D}_a[\![L]\!]\sigma$:
$\nu' \neq \perp \rightarrow |l'| = |l|$ and $l'^c = l^c \rightarrow l \subseteq_R l'$.

*Proof:* For each channel $\alpha \in I \cup \mathcal{O}$ we construct a parallel component $L_\alpha$. Then $L = L_{\alpha_1} \|_\emptyset \ldots \|_\emptyset L_{\alpha_n}$, where $I \cup \mathcal{O} = \{\alpha_1, \ldots, \alpha_n\}$. Let $n = |k|$. We define $L_\alpha = L_\alpha^{(1)}; \ldots; L_\alpha^{(n)}$,

$$\text{where } L_\alpha^{(i)} = \begin{cases} [\alpha!v \rightarrow - \square \mathbf{wait}\,1 \rightarrow -] & \text{if } \alpha_v \in k[i] \\ \mathbf{wait}\ 1 & \text{if } \alpha_v \notin k[i] \text{and } R_\alpha \in k[i] \\ [\alpha!0 \rightarrow - \square \mathbf{wait}\,1 \rightarrow -] & \text{otherwise} \end{cases}$$

if $\alpha \in \mathcal{O}$.
If $\alpha \in I$ we take $\alpha?x_\alpha$ instead of $\alpha!v$ and $\alpha!0$. Now the history $l$ of length $n$, defined by property (iv) is clearly generated by $L$ in a terminating computation. The other properties can be easily checked. □

[6]All histories generated by a program have this property.

This lemma claims that $L$ has all the required properties of the lock above. I.e., $L$ produces a history $l$ (expressed by (i)) that is the "ready-complement of $k$" (formalized by (iv)). Clause (v) guarantees that lock $L$ does not produce other histories, that could make it possible for $Q$ to "turn the lock" as well. (iii) ensures that both $L \| P$ and $L \| Q$ are syntactically correct programs.

*Proof of Lemma 2:* Suppose $\mathcal{D}_a[\![P]\!] \neq \mathcal{D}_a[\![Q]\!]$. Assume, without loss of generality, that there are $\tau, k$ and $\sigma \neq \perp$, with $<\tau, k> \in \mathcal{D}_a[\![P]\!]\sigma \backslash \mathcal{D}_a[\![Q]\!]\sigma$. It suffices to prove that there are $C, h$ and $\rho$ such that $<\rho, h> \in (\mathcal{D}_a[\![C(P)]\!]\sigma)_c \backslash (\mathcal{D}_a[\![C(P)]\!]\sigma)_c$. (this follows from Lemma 1.) Let $I = chout(P), \mathcal{O} = chin(P)$. Applying Lemma 4 to $k, \sigma, I$ and $\mathcal{O}$ gives us a program $L$, history $h$ and state $\nu$ with the properties (i) to (v) as stated in the lemma. Define $C = (< hole >; \gamma!0 \|_A L) \|_{\{\gamma\}} \gamma?x$ where $A = I \cup \mathcal{O}$ and $x$ is a variable not appearing in $P, Q$ or $L$. Note that $C(P)$ and $C(Q)$ are syntactically correct. There are two cases, depending on whether $\tau = \perp$ or not.

**Case I.** $\tau \neq \perp$.
Let $\rho = (\tau \|_\sigma \nu)\{0/x\}$ and $h = (k\gamma_0 \| l)^c$ and let $n = |k|$. It is clear from properties (i) and (ii) that

$$<\rho, h> = <\tau \|_\sigma \nu, ((k\gamma_0 \|_A l) \|_\gamma \gamma_0)^c> \in (\mathcal{D}_a[\![((P; \gamma!0) \|_A L) \|_\gamma \gamma?x]\!]\sigma)^c \ .$$

Now suppose $<\rho, h> \in \mathcal{D}_a[\![((Q; \gamma!0) \|_A L) \|_\gamma \gamma?x]\!]\sigma)^c$.
By definition of $\mathcal{D}_a$ and $.^c$, there must be $<\tau', k'> \in \mathcal{D}_a[\![Q]\!]\sigma$, $<v', l'> \in \mathcal{D}_a[\![L]\!]\sigma$, $<\varphi_1, g_1> \in \mathcal{D}_a[\![\gamma!0]\!]\sigma$ and $<\varphi_2, g_2> \in \mathcal{D}_a[\![\gamma?x]\!]\sigma$, such that *consistent* $(\varphi_1, k'g_1, \nu', l', A)$, *consistent* $(\varphi_1 \|_\sigma v', k'g_1 \|_A l, \varphi_2, g_2\{\gamma\})$ (since $(k'g_1 \|_A l) \|_\gamma g_2)^c = h$ and $(\varphi_1 \|_\sigma \nu \|_\sigma \varphi_2) = \rho$). Here $k'$ is chosen such that it only contains readies in $I \cup \mathcal{O}$. (all other readies appear only on behalf of ready-closure). Straightforward application of all definitions gives us that $g_1 = \gamma_0, \varphi_1 = \tau'$, $\varphi_2 = \nu'\{0/x\}, g_2 = R^n_\gamma \gamma_0$. Hence,

$$(k'g_1 \|_A L') \|_\gamma g_2 = k'\gamma_0 \|_A L' \text{ and } \rho = (\tau' \|_\sigma v')\{0/x\} \ . \tag{1}$$

$$\text{Because } ((k'\gamma_0) \|_A l')^c = (k\gamma_0 \|_A l)^c = h \ , \tag{2}$$

we see that $|k'| = |k|$ and from properties (ii) and (v) also $|k| = |l| = |l'|$.

Claim 1: $k'^c = k^c$ and $l'^c = l^c$.
*Proof of claim:* suppose $\alpha_v \in k'[i]$, so $\alpha \neq \gamma$, then $\alpha_v \in (k\gamma_0 \|_A L)[i]$, by (1), so $\alpha_v \in k[i]$ or $\alpha_v \in l[i]$. If $\alpha \in A$, then $\alpha_v \in k[i] \cap l[i]$ by consistency. If $\alpha \notin A$, then $\alpha_v \notin l[i]$, because $\alpha$ cannot be in the channels of $L$. So, in both cases $\alpha_v \in k[i]$. All other cases are symmetric.

Claim 2: $\tau' = \tau$
*Proof of claim:* we know that $(\tau' \|_\sigma \nu')(\{0/x\}) = (\tau \|_\sigma \nu)\{0/x\}$ by (3). Let $y \in var$. If $y \in var(P) = var(Q)$, then $\nu'(y) = \nu(y)$ and hence $\tau'(y) = (\tau' \|_\sigma \nu')\{0/x\}(y) = (\tau \|_\sigma \nu)\{0/x\}(y) = \tau(y)$. A similar argument applies if $y \in var(L)$ or $y \notin var(p) \cup var(L)$.

$$\text{From Claim 1 and property (v) we infer } l \subseteq_R l' \ . \tag{3}$$

Now we prove: $k' \subseteq_R k$. Let $R_\alpha \in k'[i]$. Then $R_\alpha \notin l'[i]$ by consistency of $k'$ and $l'$. Hence, by (2), $R_\alpha \notin l[i]$ and by (iv): $R_\alpha \in k[i]$. But now we have a contradiction, because, by ready closure and claim 2: $<\tau, k> \in \mathcal{D}_a[\![Q]\!]\sigma$ and hence $<\rho, h> \in (\mathcal{D}_a[\![C(Q)]\!]\sigma)^c$.

**Case II** $\tau = \perp$.
Choose $\rho = \perp$ and $h = (k \parallel_A l)^c$. Again we prove that $<\tau, k> \in \mathcal{D}_a[\![Q]\!]\sigma$, which leads to a contradiction. If $<\rho, h> \in \mathcal{D}_a[\![((Q;\gamma!0 \parallel_A L) \parallel \gamma?x]\!]\sigma$, then there must be state-history pairs $<\tau', k'> \in \mathcal{D}_a[\![Q]\!]\sigma$ and $<\nu', l'> \in \mathcal{D}_a[\![L]\!]\sigma$ with *consistent* $(\tau', k', \nu', l'A)$, $(k' \parallel_A l'^c = (k \parallel_A l)^c$ and $\tau' = \perp$.
By consistency, in particular comparability, we know that $|k'| \geq |l'|$. By definition of $h_1 \parallel_A h_2$ we have in general:
$|(h_1 \parallel_A h_2)| = \max(|h_1|, |h_2|)$, so $|k'| = \max(|k'|, |l'|) = |(k' \parallel_A l')| = |(k \parallel_A l)| = |k|$.
We also have $|l'| = |l|$. If $\nu' = \perp$, this follows from the same argument as above and if $\nu' \neq \perp$, it is a direct consequence of property (v). Now we can follow the reasoning of case (i) and obtain $k'^c = k^c, l'^c = l^c, \tau' = \tau, L \subseteq_R l', k' \subseteq_R k$ and $<\tau, k> \in \mathcal{D}_a[\![Q]\!]\sigma$ — contradiction. □

**Remark**
In this proof we make essential use of the empty statement $(-)$. With it, the separating context can be defined in an easy and intuitively clear way. Without the empty statement, we still have full abstraction, but the proof becomes more complicated. Obviously, we can remove the empty statements from the context, by substituting $L^{i+1}_\alpha$ for any empty statement in $L^i_\alpha$. This may leave us with an empty statement in $L^n_\alpha$.
Now, if $\alpha_v \in k[n]$, then we can replace $L^n_\alpha$ by $\alpha!v$ or $\alpha?x_\alpha$. If $k^c[n] = \emptyset$, we may replace $L^n_\alpha$ by **wait** 1. Why? Clearly, there are now pairs of denotations which we cannot separate. One can show that such pairs of denotations contain $<\sigma, kt>$ respectively $<\sigma, kt'>$, with $\sigma \neq \perp$ and $\exists \alpha : R_\alpha \in t \wedge R_\alpha \notin t'$. If there is no empty statement, then, the $R_\alpha$-record can only have been generated by ready-closure. This means that $<\sigma, kt'>$ is also part of the first denotation and hence this state-history pair is not separating.
Consequently, although we cannot construct contexts for all histories, we are still able to do so for the separating ones.
We do point out that using an empty statement allows us to prove a slightly more general result than just full abstraction of DNP-R, since in the proof we did not rely on the fact that the separating history, $k$, is generated by a DNP-R program.

## 8. Conclusion and future work

The paper answers the question of what syntax directed semantics is the correct one for prescribing real-time distributed computations. After fixing a language – essentially OCCAM –, fixing a computation model – every concurrent process has its dedicated processor, thus maximizing activity – and fixing a notion of observability – communications at every time instant, the starting state and the termination state (if any) – this question admits an univocal answer: This paper's semantics is indeed the right one, since it is fully abstract and hence is the semantics that for any program respects its observational behaviour and records the least amount of non-observables for it to become syntax directed.
In retrospect, the ideas on which the semantics is based proved to be surprisingly natural. Basically Francez, Lehmann and Pnueli's method of linear history semantics had to be mod-

ified,

1) by making waiting for communications explicit, through adding so-called readies, and

2) by realizing that a ready only serves to make certain behaviours illegal and hence, if such a behaviour is allowed anyway, through other means, the ready is irrelevant. This is the meaning of "ready-closure".

The semantics provides a good starting point for future work. We mention some topics.

1. Develop a syntax-directed specification language and corresponding proof system based on this semantics.
2. Develop a fully abstract temporal logic for real-time distributed computing, thus generalizing [BKP86].
3. Develop decision procedures for the propositional fragment of such a logic.
4. Integrate such a logic into automated specification tools such as Statemate [Har84] in order to obtain machine support for modular design and its verification.
5. Specialize these specification languages and proof systems to a real-time fragment of Ada and to OCCAM (through incorporating local clocks).
6. Use the semantics to extend Lamport's ideas on the implementation of modules [Lam83] to real-time.
7. Develop techniques for the stepwise refinement of real-time programs, possibly along the lines of [Old86].
8. Relax the idealizations, in our computation model, of synchronization, instantaneous communication, and unit duration of any atomic action.
   Presently, we are working on topics 1, 2, 4, and 5 in the context of ESPRIT project no. 937: Debugging and Specification of Real-Time Embedded Systems (DESCARTES).

## References

[A83 ] ADA (1983), *The programming Language Ada Reference Manual*, LNCS **155**, Springer-Verlag, New York.

[deB80 ] Bakker de, J.W. (1980), **Mathematical Theory of Program Correctness**, Prentice Hall, London.

[BC85 ] Berry G., Cosserat L. (1985), The ESTEREL Synchronous Programming Language and its Mathematical Semantics, *Proc. Seminar on Concurrency*, LNCS **197**, pp. 389-449, Springer-Verlag, New York.

[BCH85 ] Bergerand J-L., Caspi P., Halbwachs N. (1985), Outline of a real-time data flow language, *Proc. IEEE-CS Real-Time Systems Symposium*, San Diego.

[BH81 ] Bernstein A., Harter Jr. P.K.(1981), Proving Real-time Properties of Programs with Temporal Logic, *Proc. 8th ACM-SIGOPS Symposium on Operating System Principles*, pp. 1-11.

[BKP86 ] Barringer H., Kuiper R., Pnueli A. (1986), A Really Abstract Concurrent Model and its Temporal Logic, *Proc. 13th ACM Symposium on Principles of Programming Languages*, pp. 173-183.

[BM83 ] Berthomieu, Menasce, (1983), Analysis of Timed Petri Nets, *Proc. IFIP Confer. on Information Processing*, **83**, North- Holland.

[Bro83 ] Broy M. (1983), Applicative Real-Time Programming, *Proc. IFIP Confer. on Information Processing*, **83** (R.A. Mason ed.), pp. 259-264, North-Holland.

[BW82 ] Branquart P., Louis G., Wodon L.P. (1982), An Analytic Description of CHILL, the CCITT High Level Language, LNCS **128**, Springer-Verlag, New York.

[FLP84 ] Francez N., Lehmann D., Pnueli A. (1984), A Linear History Semantics of Distributed Programming, *Theoret. Comput. Science* **32**, pp. 25-46.

[HMM85 ] Halpern J.Y., Megiddo N., Munski A.A. (1985), "Optimal Precision in the Presence of Uncertainty", Technical Report IBM, San Jose.

[HS86 ] Halpern J.Y., Shoham Y. (1986), A Propositional Modal Logic of Time Intervals, *Proc. Symposium on Logic and computer Science*, June 16-18, Cambridge, Mass., IEEE.

[Har84 ] Harel D. (1984), Statecharts: A Visual Approach to Complex Systems, *Proc. of the Advanced NATO Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, NATO ASI Series F, Vol.13, pp. 1-44, Springer-Verlag, Berlin.

[HGR85 ] Huizing C., Gerth R., de Roever W.-P. (1985), Maximal Parallelism, Synchronous Concurrency and Real Time, Dept. of Comput. Science, University of Utrecht.

[HP79 ] Hennessy M., Plotkin G. (1979), Full Abstraction for a Simple Parallel Programming Language, *Proc. Math. Foundat. of Comput. Science*, LNCS **74**, pp. 108-120, Springer-Verlag, New York.

[KR85 ] Koymans R., de Roever W.-P. (1985), Examples of a Real-Time Temporal Logic Specification, *Proc. Analysis of Concurrent Systems*, LNCS **207**, pp. 231-252, Springer-Verlag, New York.

[KSRGA86 ] Koymans R., Shyamasundar R.K., de Roever W.- P., Gerth R., Arun-Kumar S. (1986), Compositional Semantics for Real-time Distributed Computing, *Information and Control*, to appear.

[KVR83 ] Koymans R., Vytopil J., de Roever W.-P. (1983), Real-Time Programming and Asynchronous Message Passing, *Proc. 2nd ACM Symposium on Principles of Distributed Computing*, pp. 187-197.

[Lam83 ] Lamport L. (1983), Specifying Concurrent Program Modules, *ACM Trans. on Progr. Lang. and Systems* **5-2**, pp. 190-222.

[Mil83 ] Milner R. (1983), Calculi for Synchrony and Asynchrony, *Theoret. Comput. Science* **25**, pp. 267-310.

[Mos83 ] Moszkowski B., Manna Z. (1983), Reasoning in Interval Temporal Logic, *Proc. Logics of Programs*, LNCS **164**, pp. 371-383, Springer-Verlag, New York.

[Occ84 ] **The Occam Language Reference Manual**, Prentice Hall, 1984.

[OH86 ] Olderog E.-R., Hoare C.A.R. (1986), Specification-Oriented Semantics for Communicating Processes, *Acta Informatica*, to appear.

[Old86 ] Olderog E.-R. (1986), Process Theory: Semantics, Specification and Verification (1986), *Current Trends in Concurrency - Overview and Tutorials (J.W. de Bakker, W.-P. de Roever, G. Rozenberg, eds.)* LNCS **224**, pp. 442-510, Springer-Verlag, New York.

[Plo83 ] Plotkin G. (1983), An Operational Semantics for CSP, *Proc. IFIP Conference on the Formal Description of Programming Concepts II*, pp. 199-225, North-Holland.

[Pnu85 ] Pnueli A. (1985), Linear and Branching Structures in the Semantics and Logics of Reactive Systems, *Proc. 12th Colloquium Automata, Languages and Programming (ICALP)*, LNCS **194**, pp. 15-33, Springer-Verlag, New York.

[RR86 ] Reed G.M., Roscoe A.W. (1986), A Timed Model for Communicating Sequential Processes, *Proc. 13th Colloquium Automata, Languages and Programming ICALP*, Rennes.

[SM81 ] Salwicki A., Müldner T. (1981), On the Algorithmic Properties Concurrent Programs, LNCS **125**, Springer-Verlag, New York.

[SW79 ] Strunk J.W., White E.B. (1979), **The Elements of Style**, third edition, MacMillan.

[W86 ] Wegman M.N., What it's like to be a POPL Referee or How to write an extended abstract so, that is is more likely to be accepted, *SIGPLAN Notices*, **21-5**, pp. 91-95.

# A Appendix

**Syntax of the language**

$$\begin{array}{lcl} S & ::= & x := e \mid \bar{g} \mid S_1; S_2 \mid GC \mid \star GC \mid S_1 \parallel_a S_2 \mid [S]_A \mid - \\ \bar{g} & ::= & \alpha!e \mid \alpha?x \mid \textbf{wait } n \mid - \\ GC & ::= & [\Box_{i=1}^{n} b_i; \bar{g} \rightarrow S_i] \end{array}$$

$\alpha$ is a channel, $x$ a program-variable, $e$ an expression with values in $V$, $n$ an integer expression, $b_i$ boolean expressions, $A$ a set of channels. In $S_1 \parallel_A S_2, A$ should at least contain the joint channels of $S_1$ and $S_2$. In $[S]_A, A$ should contain the internal channels of $S$. These sets are added to the syntax in order to achieve a compositional semantics.
Parallel processes do not share variables. For every channel, there is at most one process that can output values to the channel and at most one process that can input values from it.

**Denotational Semantics**
$\mathcal{D} = P(State \times His)$
*State*: states, valuating the program variables, or $\perp$.
*His*: sequences of sets of records, consisting of communication records $\alpha_v$ (value from channel $\alpha$) and readies $R_\alpha$.
$\mathcal{D}[\![S]\!] : State \rightarrow \mathcal{D}$
$\mathcal{G}[\![g]\!] : P(g) \rightarrow State \rightarrow \mathcal{D}$
Define some auxiliary functions
$.^{+} : (State \rightarrow \mathcal{D}) \rightarrow (\mathcal{D} \rightarrow \mathcal{D})$ by
$\varphi_{+}(U) = \{<\sigma, h_1 h_2> \mid \exists \sigma' \in State :< \sigma', h_1 > \in U \wedge <\sigma, h_2> \in \varphi(\sigma')\}$
$R(\{b_1;\bar{g}_1, ..., b_n;\bar{g}_n\}, \sigma) = \{R_\alpha \mid \exists i : \sigma(b_i) = tt \wedge (\bar{g}_i = \alpha?x)\}$,

$$waitvalue(b;\bar{g},\sigma) = \begin{cases} 0 & \text{if } \bar{g} = - \wedge \sigma(b) = true \\ \max(n,1) & \text{if } \bar{g} = \textbf{wait } n \wedge \sigma(b) = true \\ \infty & \text{otherwise} \end{cases},$$

$minwait(\mathcal{G}, \sigma) = \min\{waitvalue(g, \sigma) \mid g \in \mathcal{G}\}$.
A set $U$ is *prefix-closed*
iff $<\sigma, h> \in U \wedge h = h_1 h_2 \rightarrow <\perp, h_1> \in U$.
$Cl(U)$ is the smallest prefix-closed set that contains $U$.
$\mathcal{G}[\![\textbf{wait}n]\!]G\sigma = Cl\{<\sigma, R(G,\sigma)^t> \mid t = minwait(G,\sigma) \wedge t = min(\sigma(d), 1)\}$
$\mathcal{G}[\![\alpha!e]\!]G\sigma = Cl\{<\sigma, R(G,\sigma)^t\{\alpha_{\sigma(e)}\}> \mid 0 \leq t < minwait(G,\sigma) \vee t = 0\}$
$\mathcal{G}[\![\alpha?x]\!]G\sigma = Cl\{<\sigma, R(G,\sigma)^t\{\alpha_v\}> \mid 0 \leq t < minwait(G,\sigma) \vee t = 0, v \in Values\}$
$\mathcal{G}[\![-]\!]G\sigma = Cl\{<\sigma, \emptyset>\}$
$\mathcal{G}[\![b;\bar{g}]\!]G\sigma = \textbf{if } \sigma(b) = true \textbf{ then } \mathcal{G}[\![\bar{g}]\!]G\sigma \textbf{ else } \{<\perp, \lambda>\} \textbf{ fi}$
$\mathcal{D}[\![S]\!] \perp = \{<\perp, \lambda>\}$ for any $S$ ($\lambda$ is the empty sequence)
$\mathcal{D}[\![x := e]\!]\sigma = Cl\{<\sigma\{\sigma(e)/x\}, \emptyset>\}$
$\mathcal{D}[\![g]\!]\sigma = \mathcal{G}[\![g]\!]\{g\}\sigma$ if $g \neq -$
$\mathcal{D}[\![-]\!]\sigma = Cl\{<\sigma, \lambda>\}$
$\mathcal{D}[\![S_1; S_2]\!] = (\mathcal{D}[\![S_2]\!])^{+}(\mathcal{D}[\![S_1]\!]\sigma)$

$$\begin{array}{lll} \mathcal{D}[\![[\Box_{i=1}^{n} b_i; g_i \rightarrow S_i]]\!]\sigma & = \cup_{i=1}^{n}(\mathcal{D}[\![S_i]\!])^{+}(G[\![g_i]\!]G\sigma & \text{if } \exists i : \sigma(b_i) = true \\ & = Cl\{<\sigma, \emptyset>\} & \text{otherwise} \end{array}$$

$\mathcal{D}[\![\star GC]\!]\sigma = \mu\varphi.\lambda\sigma.\textbf{if } \exists i : \sigma(b_i) = true \textbf{ then } \varphi^{+}(\mathcal{D}[\![GC]\!]\sigma) \textbf{ else } Cl\{<\sigma, \emptyset> \textbf{ fi}$

$$\begin{aligned}\mathcal{D}[\![P_1 \|_A P_2]\!]\sigma = Cl\{ & < \sigma_1 \|_\sigma \sigma_2, h_1 \|_A h_2 > \mid \\ & < \sigma_i, h_i > \in \mathcal{D}[\![P_i]\!]\sigma, \\ & maximal(h_1, h_2),\ synchronous(h_1, h_2, A), \\ & comparable(\sigma_1, h_1, \sigma_2 h_2)\} \ ,\end{aligned}$$

where
$\sigma_1 \|_\sigma \sigma_2$ is a strict function defined by

$$(\sigma_1 \|_\sigma \sigma_2)(x) \overset{def}{=} \begin{cases} \sigma_1(x) & \text{if } \sigma_1(x) \neq \sigma(x) \\ \sigma_2(x) & \text{if } \sigma_2(x) \neq \sigma(x) \\ \sigma(x) & \sigma(x) \text{ otherwise} \end{cases}$$

$(h_1 \|_A h_2)[j] \overset{def}{=} (h_1[j] \cup h_2[j] \backslash \{R_\alpha | \alpha \in A\}$

$maximal(h_1, h_2,) \overset{def}{\Longleftrightarrow} \forall j, \alpha : R_\alpha \notin h_1[j] \cap h_2[j]$

$synchronous(h_1, h_2, A) \overset{def}{\Longleftrightarrow} \forall j, \alpha, v : \alpha \in A \rightarrow (\alpha_v \in h_1[j] \leftrightarrow \alpha_v \in h_2[j])$

$comparable(\sigma_1, h_1, \sigma_2, h_2) \overset{def}{\Longleftrightarrow} \forall i \in \{1,2\} : \sigma_i = \perp \rightarrow |h_{3-i}| \leq |h_i|$

$\mathcal{D}[\![[S]_A]\!]\sigma = Cl\{< \sigma, h > | \exists h' :< \sigma, h' > \in \mathcal{D}[\![S]\!]\sigma \wedge h' \lceil \bar{A} = h\}$, where $h' \lceil \bar{A}$ is the history that results after deleting all communications and readies on channels in $A$ from $h'$.
Define *terminated* as the least predicate on *Stat* satisfying:

(i) *terminated* $(-)$

(ii) if *terminated*$(S_1)$ and *terminated*$(S_2)$ then *terminated*$(S_1 \|_A S_2)$

(iii) if *terminated*$(S)$ then *terminated*$([S]_A)$

**Operational Semantics**

We do not bother to formally define the transition system but concentrate on the transition relation.
$\rightarrow \subseteq (Stat \times State) \times \mathbf{N} \times P(Chan \times Val)) \times (Stat \times State)$ is defined as the least relation satisfying the following set of axioms and rules:
(Notation: instead of $(P, \sigma, n, c, P', \sigma') \in \rightarrow$ we write $(P, \sigma) \overset{n,c}{\rightarrow} (P', \sigma')$.)

1a) $(\mathbf{wait}\ d, \sigma) \xrightarrow{0,\emptyset} (-, \sigma)$

1b) $(\mathbf{wait}\ d, \sigma) \xrightarrow{0,\emptyset} (\mathbf{wait}\ d', \sigma)$ where $\sigma(d') = \sigma(d) - 1$

2) $(x := e, \sigma) \xrightarrow{1,\emptyset} (-, \sigma\{\sigma(e)/x\})$

3a) $(\alpha!e, \sigma) \xrightarrow{0,\alpha_{\sigma(e)}} (-, \sigma)$

3b) $(\alpha!e, \sigma) \xrightarrow{0,\emptyset} (\alpha!e, \sigma)$

4a) $(\alpha?x, \sigma) \xrightarrow{0,\alpha_v} (-, \sigma\{v/x\})$

4b) $(\alpha?x,\sigma) \xrightarrow{0,\emptyset} (\alpha?x,\sigma)$

5a) $\dfrac{(P,\sigma) \xrightarrow{L} (P',\sigma')}{(P;Q,\sigma) \xrightarrow{L} (P';Q,\sigma')}$

5b) $\dfrac{(Q,\sigma) \xrightarrow{L} (Q',\sigma')}{(P;Q,\sigma) \xrightarrow{L} (Q',\sigma')}$ if *terminated* $(P)$

6a) $([\Box_{i=1}^{n} b_i; g_i \rightarrow P_i],\sigma) \xrightarrow{1,\sigma} (P_i,\sigma)$ if $\sigma(b_i) = true$ and $g_i \equiv 0$

6b) $\dfrac{(g_i,\sigma) \xrightarrow{1,\emptyset} (-,\sigma')}{([\Box_{i=1}^{n} b_i;g_i \rightarrow P_i],\sigma) \xrightarrow{L} (P_i,\sigma)}$ if $\sigma(b_i) = true.$

6c) $\dfrac{\text{for all } i{:}\sigma(b_i)=\text{true} \Rightarrow (g_i,\sigma) \xrightarrow{0,\emptyset} (g_i',\sigma)}{([\Box_{i=1}^{n} b_i;g_i \rightarrow P_i],\sigma) \xrightarrow{0,\emptyset} ([\Box_{i=1}^{n} b_i;g_i' \rightarrow P_i],\sigma)}$ where $g_i' \equiv g_i$ if $\sigma(b_i) =$ false

7a) $\dfrac{(\mathcal{G}S,\sigma) \xrightarrow{L} (P,\sigma')}{(\star\mathcal{G}S,\sigma) \xrightarrow{L} (P;\star\mathcal{G}S,\sigma')}$ if $\sigma(b_i) =$ true for some $i$

7b) $(\star\mathcal{G}S,\sigma) \xrightarrow{1,\emptyset} (-,\sigma)$ if $\sigma(b_i) =$ false for all $i$

8a) $\dfrac{L_1^c\restriction_A = L_2^c\restriction_A, (P_1,\sigma) \xrightarrow{L_1} (P_1',\sigma_1'), (P_2,\sigma) \xrightarrow{L_2} (P_2',\sigma_2')}{(P_1\|_A P_2,\sigma) \xrightarrow{L} (P_1'\|_A P_2',\sigma')}$

8b) $\dfrac{(P_1,\sigma) \xrightarrow{L} (P_1',\sigma'), L^c\restriction A=\emptyset, terminated\ (P_2)}{\begin{array}{c}(P_1\|_A P_2,\sigma) \xrightarrow{L} (P_1'\|_A P_2,\sigma') \\ (P_2\|_A P_1,\sigma) \xrightarrow{L} (P_2\|_A P_1'.\sigma')\end{array}}$

9) $\dfrac{(P,\sigma) \xrightarrow{L} (P',\sigma')}{([P]_A,\sigma) \xrightarrow{L} ([P']_{A'},\sigma')}$ where $A' = ichan(P')$

## B Appendix

**Theorem 1** *There exists a function $\beta$ such that $\mathcal{O} = \beta \circ \mathcal{D}$.*

### B.1 General structure of the proof

Define $\beta$ by

$$\beta(D) = \{\langle \sigma, h\rangle \mid \exists h' : \langle \sigma, h'\rangle \in D \wedge \forall i : h[i] = h'[i] \setminus \mathsf{Readies}\}$$

We prove this in several steps, through defining intermediate semantics $\mathcal{J}_1$, $\mathcal{J}_2$, and $\mathcal{J}_3$, each different from the previous one in one essential aspect. We have the following semantics:

(i). $\mathcal{O}$;

(ii). $\mathcal{J}_1$ forces maximality not by a global constraint on the traces, like in $\mathcal{O}$, but by readies, like in $\mathcal{D}$;

(iii). $\mathcal{J}_2$ performs hiding locally, not by deleting records from the traces, like in $\mathcal{J}_1$;

(iv). $\mathcal{J}_3$ contains visible readies, like $\mathcal{D}$ does;

(v). $\mathcal{D}$ is not defined by a transition relation, like $\mathcal{J}_3$, but is compositionally defined, using fixed points in a cpo of prefix-closed sets.

These semantics are linked one another by the following lemma.

**Lemma 1** *(Linking)*

*(i).* $\mathcal{O} = \mathcal{J}_1$

*(ii).* $\mathcal{J}_1 = \mathcal{J}_2$

*(iii).* $\mathcal{J}_2 = \beta \circ \mathcal{J}_3$

*(iv).* $\mathcal{J}_3 = \mathcal{D}$

From this lemma, the theorem follows immediately. The proof of this lemma can be found in the next sections.

### B.2 Proof of Linking Lemma (i)

Semantics $\mathcal{J}_1$ is based on a similar transition relation as $\mathcal{O}$ is based upon. The difference is that the label of this transition relation consists of three fields:

(i). a number that records how many local steps have been taken; this component is only there for compatibility with $\longrightarrow$.

(ii). a set of communications that have taken place on both internal and external channels.

(iii). a set of channel names that function as the readies $R_\alpha$ in the denotational semantics.

**Notation:** If $L = (n, c, r)$ is a label of this transition relation, we denote $L^N = n, L^C = c, L^R = r$.

**Definition 1** $\rightarrow_1 \subseteq (\mathsf{Stat} \times \mathsf{States}) \times (\mathcal{N} \times \mathcal{P}(\mathsf{Chan} \times \mathsf{Val}) \times \mathcal{P}(\mathsf{Chan})) \times (\mathsf{Stat} \times \mathsf{States})$ *is the smallest relation that satisfies the following axioms and rules:*

**1a.** $\mathsf{wait}d, \sigma \xrightarrow{0,\emptyset,\emptyset}_1 -, \sigma$, *where* $\sigma(d) = 0$

**1b.** $\mathsf{wait}d, \sigma \xrightarrow{0,\emptyset,\emptyset}_1 \mathsf{wait}d', \sigma$, *where* $\sigma(d') = \sigma(d) - 1$

**2.** $x := e, \sigma \xrightarrow{1,\emptyset,\emptyset}_1 -, \sigma\{\sigma(e)/x\}$

**3a.** $\alpha!e, \sigma \xrightarrow{0,\alpha_v,\emptyset}_1 -, \sigma$ *where* $\sigma(e) = v$ [1]

**3b.** $\alpha!e, \sigma \xrightarrow{0,\emptyset,\alpha}_1 \alpha!e, \sigma$

**4a.** $\alpha?x, \sigma \xrightarrow{0,\alpha_v,\emptyset}_1 -, \sigma\{v/x\}$

**4b.** $\alpha?x, \sigma \xrightarrow{0,\emptyset,\alpha}_1 \alpha?x, \sigma$

**5a.** $$\frac{(P, \sigma) \xrightarrow{L}_1 (P', \sigma')}{(P; Q, \sigma) \xrightarrow{L}_1 (P'; Q, \sigma')}$$

**5b.** $$\frac{(Q, \sigma) \xrightarrow{L}_1 (Q', \sigma')}{(P; Q, \sigma) \xrightarrow{L}_1 (Q', \sigma')}$$ *if terminated*$(P)$.

**6a.** $[\mathop{\Box}_{i=1}^{n} b_i; g_i \rightarrow_1 P_i], \sigma \xrightarrow{1,\emptyset,\emptyset} P_i, \sigma$ *if* $\sigma(b_i) = \mathsf{true}$ *and* $g_i = -$

**6b.** $$\frac{(g_i, \sigma) \xrightarrow{L}_1 (-, \sigma')}{([\mathop{\Box}_{i=1}^{n} b_i; g_i \longrightarrow P_i], \sigma) \xrightarrow{L}_1 (P_i, \sigma)}$$ *if* $\sigma(b_i) = \mathsf{true}$.

**6c.** $$\frac{\forall i : [\sigma(b_i) = \mathsf{true} \Rightarrow (g_i, \sigma) \xrightarrow{0,\emptyset,r_i}_1 (g_i', \sigma)]}{[\mathop{\Box}_{i=1}^{n} b_i; g_i \longrightarrow P_i], \sigma \xrightarrow{0,\emptyset,R}_1 [\mathop{\Box}_{i=1}^{n} b_i; g_i' \longrightarrow P_i], \sigma}$$, *where* $R = \cup r_i$ *and* $g_i' = g_i$ *if* $\sigma(b_i) =$ $\mathsf{false}$

[1]Remember that we may leave out the curly brackets for singletons, in this case we write $\alpha_v$ instead of $\{\alpha_v\}$.

**7a.** $$\frac{(GS,\sigma) \xrightarrow{L}_1 (P,\sigma')}{(\star GS,\sigma) \xrightarrow{L}_1 (P;\star GS,\sigma')} \text{ if } \sigma(b_i) = \mathsf{true} \text{ for some } i.$$

**7b.** $\star GS,\sigma \xrightarrow{1,\emptyset,\emptyset}_1 -,\sigma$ *if* $\sigma(b_i) = \mathsf{false}$ *for all* $i$

**8a.** $$\frac{L_1^R \cap L_2^R = \emptyset, L_1^C \upharpoonright A = L_2^C \upharpoonright A, (P_1,\sigma) \xrightarrow{L_1}_1 P_1',\sigma_1'), (P_2,\sigma) \xrightarrow{L_2}_1 P_2',\sigma_2')}{(P_1||_A P_2,\sigma) \xrightarrow{L}_1 (P_1'||AP_2',\sigma_1'||_A\sigma_2')} \text{ where}$$

$$L_1||_A L_2 = (L_1^N + L_2^N, L_1^C \cup L_2^C, (L_1^R \cup L_2^R) \setminus A)$$

**8b** $$\frac{(P_1,\sigma) \rightarrow_1 (P_1',\sigma_1'),\ L^C \upharpoonright A = \emptyset,\ \text{terminated}(P_2)}{(P_1||_A P_2,\sigma) \xrightarrow{L}_1 (P_1'||_A P_2,\sigma_1')\ \ (P_1||_A P_2,\sigma) \xrightarrow{L}_1 (P_1||_A P_2',\sigma_1')}$$

**9** $$\frac{(P,\sigma) \xrightarrow{L}_1 (P',\sigma')}{([P]_A,\sigma) \xrightarrow{L}_1 ([P']_{A'},\sigma')} \text{ where } A' = ichan(P').$$

**Definition 2**

$$\begin{aligned}\mathcal{J}_1[\![P_0]\!]\sigma_0 = \mathcal{C}\ell\ \ \{\ \ & \langle\sigma,h\rangle \mid \forall 1 \le i \le |h| \exists \vec{P}_i, \vec{\sigma_i}, \vec{L_i} : \\ & P_{i-1},\sigma_{i-1} \xrightarrow{L_i}_1 P_i,\sigma_i, \\ & h[i] = L_i^C \upharpoonright vis(P_{i-1}) \\ & |h| < \infty \Rightarrow \sigma = \sigma_{|h|} \wedge terminated(P_{|h|}), \\ & |h| = \infty \Rightarrow \sigma = \infty \\ \}\ \ & \end{aligned}$$

Note that we do not demand maximality of the steps here, since this is already taken care of in rule 8a. We are going to prove that this is the case.

To prove Linking Lemma (i), it suffices to prove

**Lemma 2** *(Consistency 1)*

$$P,\sigma \xrightarrow{n,c} P',\sigma' \text{ is a maximal step} \Leftrightarrow \exists r : P,\sigma \xrightarrow{n,c,r}_1 P',\sigma'$$

Before we prove this lemma, we first prove two fundamental lemmata that establish the exact relationship between readies and maximality of steps.

**Lemma 3** *(Ready) Suppose* $\alpha \in chin(P)$. *Then*

$$P,\sigma \xrightarrow{n,c,r \cup \{\alpha\}}_1 \Leftrightarrow \exists r' \subseteq r, v \in \mathsf{Val} : P,\sigma \xrightarrow{n,c \cup \{\alpha_v\},r'}_1$$

*Proof:*
By structural induction w.r.t. $P$.

For the *primitive statements* only rules 3 and 4 lead to readies or communications in the label and for these the lemma is obviously satisfied.

For the composite statements *sequential composition*, *iteration*, and *hiding*, the rules are not dependent on the ready part and do not change it. Hence, the induction step is obvious for these statements.

This leaves us with only two cases.

**guarded statement** Write $GS = [\mathop{\Box}_{i=1}^{n} b_i; g_i \longrightarrow P_i]$ and let $I = \{i : \sigma(i) = \mathsf{true}\}$.

Assume that $GS, \sigma \xrightarrow{0,\emptyset,R}_1$ and $\alpha \in R$. Then rule 6c must have been applied, hence $R = \bigcup_{i \in I} r_i$ and

$GS, \sigma \xrightarrow{0,\emptyset,R \cup \{\alpha\}}_1 C \Leftrightarrow$ {rule 6c}

$R \cup \{\alpha\} = \bigcup_{i \in I} r_i$ and $g_i, \sigma \xrightarrow{0,\emptyset,r_i}_1 g_i', \sigma$ for any $i \in I$ $\{\Longleftrightarrow\}$

there is an $i \in I$ with $g_i, \sigma \xrightarrow{0,\emptyset,r_i}_1 g_i', \sigma$ and $\alpha \in r_i$ $\{\Longleftrightarrow\}$$g_i$ is of the form $\alpha!e$ or $\alpha?x\}$

$g_i, \sigma \xrightarrow{0,\alpha_v,\emptyset}_1$ for some $v \in \mathsf{Val}$ $\{\Longleftrightarrow\}$

$GS, \sigma \xrightarrow{0,\alpha_v,\emptyset}_1$

**parallel composition** If one of the $P_i$ has terminated, it is trivial. So assume $\neg terminated(P_i)$ for both $i$. Then

$P_1||_A P_2, \sigma \xrightarrow{L \cup \{\alpha\}}_1$ $\{\Longleftrightarrow\}$

$P_i, \sigma \xrightarrow{L_i}_1$ with $combine(L_1, L_2, L \cup \{\alpha\}, A)$ $\{\Longleftrightarrow\}$

$P_i, \sigma \xrightarrow{L_i}_1$ with $combine(L_1, L_2, L \cup \{\alpha\}, A)$ and $\alpha \in L_j^R$ for some $j$ $\{\Longleftrightarrow\}$

$P_j, \sigma \xrightarrow{\tilde{L}_j}_1$ with $\tilde{L}_j^N = L_j^N, \tilde{L}_j^C = L_j^C \cup \{\alpha_v\}, \tilde{L}_j^R \subseteq L_j^R$ and $combine(L_1, L_2, L \cup \{\alpha\}, A)$ and

$P_{3-j}, \sigma \xrightarrow{L_i}_1$ $\{\Longleftrightarrow\}$

$P_1||_A P_2, \sigma \xrightarrow{\tilde{L}}_1$ with $combine(\tilde{L}_j, L_{3-j}, \tilde{L})$
□

From the proof of the ready lemma, it is clear that the same result holds for $\alpha \in chout(P)$, for any $v$ instead of some $v$.

The next lemma we need states that readies do indeed the same job as the global maximality constraint of $\mathcal{O}$. All steps of $\longrightarrow_1$ are maximal. This lemma shows also why a Janus semantics like $\mathcal{J}_1$ is useful: the concept of readies and the maximality constraint can be compared in one semantics.

**Lemma 4** *(maximality) All steps of $\rightarrow_1$ are maximal.*

*Proof:*
We prove this by structural induction w.r.t. the program $P$.

**primitive statements** One can easily check that all transitions of primitive statements are maximal (note that $ichan(P) = \emptyset$ for any primitive statement $P$).

**parallel composition** The case that one of the components is terminated, is treated below.

Assume that $\neg terminated(P_i)$ for $i = 1, 2$. This is the most interesting case. Suppose we have

$$P_1||_A P_2, \sigma \xrightarrow{L}_1 \text{ and } P_1||_A P_2, \sigma \xrightarrow{\tilde{L}}_1$$

with $L <_I \tilde{L}$ and $I = ichan(P_1||_A P_2)$. Now, because both statements are not terminated, we know that there are $P_i$ and $L_i$ with

$$P_i, \sigma \xrightarrow{L_i}_1 \text{ and } P_i, \sigma \xrightarrow{\tilde{L}_i}_1$$

with $L_1||_A L_2 = L$, $\tilde{L}_1||_A \tilde{L}_2 = \tilde{L}$, and, a.o., $L_1^R \cap L_2^R = \emptyset$ (*).

Assume (1) that $\tilde{L}^C = L^C$. Since $chan(P_1) \cap chan(P_2) = A$ and $\tilde{L}_1^C \lceil A = \tilde{L}_2^C \lceil A$, we know that $\tilde{L}_i^C = L_i^C$ for both $i$. Furthermore, we must have $L^N < \tilde{L}^N$ and hence $L_i^N < \tilde{L}_i^N$ for some $i$ and hence the sub-step is not maximal, which violates the induction hypothesis.

So assume (2) that $\tilde{L}^N = L^N$ and hence $L^C \neq \tilde{L}^C$, whence there is $\alpha_v \in \tilde{L}^C \setminus L^C$, where $\alpha \in ichan(P_1||_A P_2)$. If $\alpha \in ichan(P_i)$, we have again a contradiction, because then $P_i, \sigma \xrightarrow{L_i}_1$ is not maximal. If $\alpha \notin ichan(P_i)$, then $\alpha \in A$ and hence $\alpha_v \in \tilde{L}_1 \cap \tilde{L}_2$. By the Ready Lemma, we know that in this case $\alpha \in L_i$ for $i = 1, 2$. This is in contradiction with (*).

**other compound statements** (including parallel composition of which one of the components has terminated) What is left are situations in which

$$P, \sigma \xrightarrow{L}_1 \Leftrightarrow F[P], \sigma \xrightarrow{L}_1$$

in which $F[P]$ is some syntactic operator. In this case, $L^C \lceil ichan(P) = L^C \lceil ichan(F[P])$ and hence

$$L <_{ichan(P)} \tilde{L} \Leftrightarrow L <_{ichan(F[P])} \tilde{L}$$

This implies, of course, that if $P, \sigma \xrightarrow{L}$ is maximal, then also $P, \sigma \xrightarrow{L}$ is maximal, which is exactly the induction step that we had to prove. □

Now we can prove the consistency lemma, using structural induction w.r.t $P$.

The proofs for the cases of the *primitive statements* are obvious, since in these cases, the ready part of the label is not used for the definition of $\rightarrow_1$.

The same holds for *sequential composition, guarded statement, iteration,* and *hiding.*

So let us consider *parallel composition.* Suppose $(P_1||_A P_2, \sigma) \xrightarrow{L}_1 C$ is a maximal step. Then either rule 8a or 8b must have been applied to achieve this. Hence $C = P_1'||_A P_2', \sigma'$, $L = L_1||_A L_2$, and $P_i, \sigma \xrightarrow{L_1}_1 P_i', \sigma'$, for $i = 1, 2$, where $L_1^C \lceil A = L_2^C \lceil A$.

Now assume that $P_1, \sigma \xrightarrow{L_1}_1 P_1', \sigma'$ is not maximal. Then there must exist $\tilde{L}_1$ with $L_1 <_I \tilde{L}_1$, where $I = ichan(P_1)$, and $P_1, \sigma \xrightarrow{\tilde{L}_1}_1$. If $\tilde{L}_1^C = L_1^C$, then $L_1^N < \tilde{L}_1^N$ and $P_1||_A P_2, \sigma \xrightarrow{\tilde{L}}_1$ with $\tilde{L}^C = L^C$ and $\tilde{L}^N = \tilde{L}_1^N + L_2^N > L^N$, so $P_1||_A P_2, \sigma \xrightarrow{L}_1$ is not maximal, cotrary to assumption. Consequently, $\tilde{L}_1^C \neq L_1^C$ and hence, $L_1^C \lceil I \subset \tilde{L}_1^C \lceil I$. Since $L_1 <_I \tilde{L}_1$, we know that $L_1^C \lceil \bar{I} = \tilde{L}_1^C \lceil \bar{I}$ ($\bar{I} = \mathsf{Chan} \setminus I$). Because $A \cap ichan(P_1) = \emptyset$, we have $\tilde{L}_1^C \lceil A = L_1^C \lceil A = L_2^C \lceil A$ and hence $P_1||_A P_2, \sigma \xrightarrow{\tilde{L}}_1$. Since $I \subseteq ichan(P)$ we have $L <_{ichan(P)}$ and $(P_1||_A P_2, \sigma) \xrightarrow{L}_1$ is not maximal. Contradiction.

So $P_1, \sigma \xrightarrow{L_1}_1 P_1', \sigma_1'$ is maximal and likewise we can prove that $P_2, \sigma \xrightarrow{L_2}_1 P_2', \sigma_2'$. Now we can apply the induction hypothesis and get:

$$P_i, \sigma \xrightarrow{\tilde{L}_i}_1 P_i', \sigma_i' \text{ , where } \tilde{L}_i = (L_i^N, L_i^C, r_i)$$

We claim that $r_1 \cap r_2 = \emptyset$.

By contradiction, assume that $\alpha \in r_1 \cap r_2$. By the Ready Lemma, we know that there exists a $v$ such that

$$P_i, \sigma \xrightarrow{L_i^N, L_i^C \cup \{\alpha_v\}, r_i'}_1$$

Again, by the induction hypothesis, we know that

$$P_i, \sigma \xrightarrow{L_i^N, L_i^C \cup \{\alpha_v\}}$$

and consequently,

$$P_1||_A P_2, \sigma \xrightarrow{\bar{L}}$$

where $\bar{L}^N = L_1^N + L_2^N$, $\bar{L}^C = L_1^C \cup L_2^C \cup \{\alpha_v\}$ and hence $L <_I \bar{L}$, which contradicts the initial assumption of maximality. So $r_1 \cap r_2 = \emptyset$ and

$$P_1||_A P_2, \sigma \xrightarrow{\tilde{L}}_1 P_1'||_A P_2', \sigma_1'||_\sigma \sigma_2'$$

where $\tilde{L} = (L_1^N + L_2^N, L_1^C \cup L_2^C, (r_1 \cup r_2) \setminus A)$ and hence $\tilde{L}^N = L^N$, $\tilde{L}^C = L^C$.
□

## B.3 Proof of Linking Lemma (ii)

The next step is to treat hiding. In $\mathcal{D}$, communication records of channels that appear in the scope of a hiding operator are removed as soon as the behaviour of this operator is computed. In $\mathcal{O}$, however, this can not be done, since then the maximality constraint could not be applied anymore. Computations that are not maximal may pass through, because the maximal computation is not visibly different from the non-maximal one. Therefore, hiding is treated globally. When the computation of the full program is known, the communications on channels that appear in the scope of a hiding operator are removed from the traces. In $\mathcal{J}_1$, this is not necessary anymore, since readies are used locally to ensure that all computations are maximal. Hence, we construct $\mathcal{J}_2$, that treats hiding locally and for the rest behaves like $\mathcal{J}_1$.

**Definition 3** $\longrightarrow_2 \subseteq$ $(\mathsf{Stat} \times \mathsf{States}) \times (\mathcal{N} \times \mathcal{P}(\mathsf{Chan} \times \mathsf{Val}) \times \mathcal{P}(\mathsf{Chan})) \times (\mathsf{Stat} \times \mathsf{States})$ *is the smallest relation that satisfies the following axioms and rules: 1 to 8 from the definition of* $\longrightarrow_1$ *and*

9 $$\frac{P,\sigma \xrightarrow{L}_2 P',\sigma'}{[P]_A,\sigma \underset{P'}{\xrightarrow{L'}}{}_2 {}_A,\sigma'} \quad \text{where } A' = ichan(P') \text{ and } L' = (L^N, \{\alpha_v \in L^C \mid \alpha \notin A\}, L^R)$$

**Definition 4**

$$\begin{aligned}\mathcal{J}_2[\![P_0]\!]\sigma_0 = \mathcal{C}\ell \quad \{ \quad & \langle \sigma, h \rangle \mid \forall 1 \le i \le |h| \exists \vec{P}_i, \vec{\sigma_i}, \vec{L_i} : \\ & P_{i-1}, \sigma_{i-1} \xrightarrow{L_i}_2 P_i, \sigma_i, \\ & h[i] = L_i^C \\ & |h| < \infty \Rightarrow \sigma = \sigma_{|h|} \wedge terminated(P_{|h|}), \\ & |h| = \infty \Rightarrow \sigma = \infty \\ \} \quad & \end{aligned}$$

Again, we have a lemma that immediately implies Linking Lemma (ii).

**Lemma 5** *(Consistency 2)*

$$P,\sigma \xrightarrow{L}_2 P',\sigma' \Leftrightarrow P,\sigma \xrightarrow{L \restriction vis(P)}_1 P',\sigma'$$

*Proof:*

For the primitive statements, this is obvious, since for these $vis(P) = chan(P)$.

For sequential composition of which the first statement is not terminated, we know that

$$P,\sigma \xrightarrow{L \restriction vis(P)}_1 P',\sigma' \Leftrightarrow P;Q,\sigma \xrightarrow{L \restriction vis(P;Q)}_1 P';Q,\sigma'$$

because $\alpha_v \in L^C \Rightarrow \alpha_v \in chan(P)$ and $vis(P;Q) \subseteq vis(P)$. We can use the same argumentation for iteration and the guarded statement.

For parallel composition $P_1||_A P_2$, we know that if $P_i,\sigma \xrightarrow{L_i}_1 P_i',\sigma_i'$, then $\alpha \notin vis(P_i) \wedge \alpha \in L_i^C \Rightarrow \alpha \notin L_{3-i}^C$

For hiding we have the following equivalences: $[P]_A, \sigma \xrightarrow[P' \; A]{n,c,r \;\; '}_2 , \sigma' \{\Longleftrightarrow\}$

$P, \sigma \xrightarrow{n,c',r}_2 P', \sigma' \wedge c' \setminus A = c \; \{\Longleftrightarrow\}$

$P, \sigma \xrightarrow{n,c'',r}_1 P'\sigma' \wedge c' \setminus A = c \wedge c' = c'' \cap vis(P) \; \{\Longleftrightarrow\}$

$[P]_A, \sigma \xrightarrow[P' \; A]{n,c'',r \;\; '}_1 , \sigma' \wedge c' \setminus A = c \wedge c' = c'' \cap vis(P) \; \{\Longleftrightarrow\}$

$[P]_A, \sigma \xrightarrow[P' \; A]{n,c'',r \;\; '}_1 , \sigma' \wedge c = c'' \cap vis(P)$

## B.4 Proof of Linking Lemma (iv)

In $\mathcal{J}_2$, the first component of the label, the number $n$, is not used anymore, since there is no global maximality constraint. This makes it possible to move closer to the denotational semantics. In the third Janus semantics, we don't have this component anymore, but we keep readies visible.

**Definition 5** $\rightarrow_3 \subseteq$ (Stat $\times$ States) $\times$ ($\mathcal{P}$(Chan $\times$ Val) $\cup$ $\mathcal{P}$(Readies) $\times$ (Stat $\times$ States) *is defined by:*

$$C \xrightarrow{L}_3 C' \Longleftrightarrow \exists n, c, r : C \xrightarrow{n,c,r}_2 C' \wedge c \cup \{\mathcal{R}_\alpha \mid \alpha \in r\} = L$$

**Definition 6**

$$\begin{aligned}\mathcal{J}_3[\![P_0]\!]\sigma_0 = \mathcal{C}\ell \quad \{ \quad & \langle \sigma, h \rangle \mid \forall 1 \le i \le |h| \exists \vec{P_i}, \vec{\sigma_i}, \vec{L_i} : \\ & P_{i-1}, \sigma_{i-1} \xrightarrow{L_i}_3 P_i, \sigma_i, \\ & h[i] = L_i^C \\ & |h| < \infty \Rightarrow \sigma = \sigma_{|h|} \wedge terminated(P_{|h|}), \\ & |h| = \infty \Rightarrow \sigma = \infty \\ \} \quad & \end{aligned}$$

Linking Lemma (iii) follows directly.

The last step is to prove that $\mathcal{D} = \mathcal{J}_3$ or, more specifically, that $\langle \sigma, h \rangle \in \mathcal{D}[\![P]\!]\sigma \Leftrightarrow \langle \sigma, h \rangle \in \mathcal{J}_3[\![P]\!]\sigma$. We prove this with an intertwining of structural induction to $P$ and natural induction to the length of $h$. The induction hypothesis is:

$|h| < n$ or $P$ a substatement of $\bar{P} \Rightarrow$
$(\langle \sigma, h \rangle \in \mathcal{D}[\![P]\!]\sigma \Leftrightarrow \langle \sigma, h \rangle \in \mathcal{J}_3[\![P]\!]\sigma)$

And the induction step is, assuming this, to prove that

$$\langle \sigma, Lh \rangle \in \mathcal{D}[\![F[P]]\!]\sigma \Leftrightarrow \langle \sigma, Lh \rangle \in \mathcal{J}_3[\![F[P]]\!]\sigma$$

where $L$ is an arbitrary record and $F[.]$ is some syntactic operator.

### B.4.1 Primitive Statements

**1a.**

$$\mathcal{D}[\![\mathsf{wait}d]\!]\sigma = \mathcal{G}[\![\mathsf{wait}d]\!]\{\mathsf{wait}d\}\sigma = \mathcal{C}\ell\{\langle\sigma,h\rangle\} = \mathcal{J}_3[\![\mathsf{wait}d]\!]\sigma$$

since $(\mathsf{wait}d,\sigma) \xrightarrow{\emptyset,\emptyset}_2 (—,\sigma)$. The other cases are as easy.

### B.4.2 Compound Statements

**Induction basis** Let $\langle\sigma,\lambda\rangle \in \mathcal{D}[\![P]\!]\sigma_0$. If $\sigma = \bot$, we are done, since $\{\langle\bot,\lambda\rangle\}$ is the bottom element of the cpo and it is always added by the closure operator $\mathcal{C}\ell$. If $\sigma \neq \bot$, we can easily see that *terminated*$(P)$ must hold and $\sigma = \sigma_0$. Hence, by definition, $\langle\sigma,\lambda\rangle \in \mathcal{J}_3[\![P]\!]\sigma$.

For the induction case, we make use of the following lemma, which is clear from the definition of $\mathcal{J}_3$.

**Lemma 6** *(Induction)* $\langle\sigma,Lh\rangle \in \mathcal{C}\ell(\mathcal{J}_3[\![P]\!]\sigma_0) \Leftrightarrow$ *there exist* $P',\sigma'$ *with* $P,\sigma_0 \xrightarrow{L}_2 P'\sigma'$ *and* $\langle\sigma,h\rangle \in \mathcal{J}_3[\![P']\!]\sigma'$

**Sequential Composition** Assume $\neg$*terminated*$(P)$. Then
$\langle\sigma,Lh\rangle \in \mathcal{D}[\![P;Q]\!]\sigma_0$ $\{\Longleftrightarrow\}$
$\langle\sigma_1,Lh_1\rangle \in \mathcal{D}[\![P]\!]\sigma_0 \wedge \langle\sigma,h_2\rangle \in \mathcal{D}[\![Q]\!]\sigma_1 \wedge h = h_1h_2$ $\{\Longleftrightarrow\}$
$\langle\sigma_1,Lh_1\rangle \in \mathcal{J}_3[\![P]\!]\sigma_0 \wedge \langle\sigma,h_2\rangle \in \mathcal{J}_3[\![Q]\!]\sigma_1 \wedge h = h_1h_2$ $\{\Longleftrightarrow\}$
$P,\sigma_0 \xrightarrow{L} P',\sigma' \wedge \langle\sigma_1,h_1\rangle \in \mathcal{J}_3[\![P']\!]\sigma' \wedge \langle\sigma,h_2\rangle \in \mathcal{J}_3[\![Q]\!]\sigma_1 \wedge h = h_1h_2$ $\{\Longleftrightarrow\}$
$P,\sigma_0 \xrightarrow{L} P',\sigma' \wedge \langle\sigma_1,h_1\rangle \in \mathcal{D}[\![P']\!]\sigma' \wedge \langle\sigma,h_2\rangle \in \mathcal{D}[\![Q]\!]\sigma_1 \wedge h = h_1h_2$ $\{\Longleftrightarrow\}$
$P,\sigma_0 \xrightarrow{L} P',\sigma' \wedge \langle\sigma,h_1h_2\rangle \in \mathcal{D}[\![P'];Q]\!]\sigma' \wedge h = h_1h_2$ $\{\Longleftrightarrow\}$
$P,\sigma_0 \xrightarrow{L} P',\sigma' \wedge \langle\sigma,h\rangle \in \mathcal{J}_3[\![P'];Q]\!]\sigma'$ $\{\Longleftrightarrow\}$
$P;Q,\sigma_0 \xrightarrow{L} P';Q,\sigma' \wedge \langle\sigma,h\rangle \in \mathcal{J}_3[\![P'];Q]\!]\sigma'$ $\{\Longleftrightarrow\}$
$\langle\sigma,Lh\rangle \in \mathcal{J}_3[\![P;Q]\!]\sigma_0$
□
The other cases are analogous, except iteration and guarded statement, which follow now.

**Iteration** Assume $\sigma_0(b_i) = \mathsf{true}$ for some $i$, where $GS = [\overset{n}{\underset{i=1}{\square}}\, b_i; g_i \rightarrow S_i]$. Then
$\langle\sigma,Lh\rangle \in \mathcal{D}[\![*GS]\!]\sigma_0$ $\{\Longleftrightarrow\}$
$\langle\sigma,Lh\rangle \in \mathcal{D}[\![*GS]\!]^+(\mathcal{D}[\![GS]\!]\sigma_0)$ $\{\Longleftrightarrow\}$
$\langle\sigma_1,Lh_1\rangle \in \mathcal{D}[\![GS]\!]\sigma_0 \wedge \langle\sigma,h_2\rangle \in \mathcal{D}[\![*GS]\!]\sigma_1 \wedge h = h_1h_2$ $\{\Longleftrightarrow\}$
$\langle\sigma_1,Lh_1\rangle \in \mathcal{J}_3[\![GS]\!]\sigma_0 \wedge \langle\sigma,h_2\rangle \in \mathcal{J}_3[\![*GS]\!]\sigma_1 \wedge h = h_1h_2$ $\{\Longleftrightarrow\}$
$GS,\sigma_0 \xrightarrow{L} P,\sigma_0' \wedge \langle\sigma_1,h_1\rangle \in \mathcal{J}_3[\![P]\!]\sigma_0' \wedge \langle\sigma,h_2\rangle \in \mathcal{J}_3[\![*GS]\!]\sigma_1 \wedge h = h_1h_2$ $\{\Longleftrightarrow\}$
$GS,\sigma_0 \xrightarrow{L} P,\sigma_0' \wedge \langle\sigma_1,h_1\rangle \in \mathcal{D}[\![P]\!]\sigma_0' \wedge \langle\sigma,h_2\rangle \in \mathcal{D}[\![*GS]\!]\sigma_1 \wedge h = h_1h_2$ $\{\Longleftrightarrow\}$
$GS,\sigma_0 \xrightarrow{L} P,\sigma_0' \wedge \langle\sigma,h_1h_2\rangle \in \mathcal{D}[\![P;*GS]\!]\sigma_0' \wedge h = h_1h_2$ $\{\Longleftrightarrow\}$
$*GS,\sigma_0 \xrightarrow{L} P;*GS,\sigma_0' \wedge \langle\sigma,h\rangle \in \mathcal{J}_3[\![P;*GS]\!]\sigma_0'$ $\{\Longleftrightarrow\}$

$\langle\sigma, Lh\rangle \in \mathcal{J}_3[\![ * GS]\!]\sigma_0$

ad 1. $\Rightarrow$ is by definition of $\mathcal{D}[\![ * GS]\!]$. $\Leftarrow$ holds only if the fixed point from this definition is unique. Since we have strictly growing histories, this is the case. [FLP84]

### B.4.3 Guarded Statement

In the following, $GS$ stands for $GS = [\overset{n}{\underset{i=1}{\Box}} b_i; g_i \rightarrow S_i]$ and $G = \{b_i; g_i \mid 1 \leq i \leq n\}$. If $g_i, \sigma \xrightarrow{\emptyset, r} g_i', \sigma$, then $GS' = [\overset{n}{\underset{i=1}{\Box}} b_i; g_i \rightarrow S_i]$ and $G = \{b_i; g_i' \mid 1 \leq i \leq n\}$. We first prove the following lemma:

**Lemma 7** *If $minwait(G, \sigma) \geq 1$ and $L \subseteq$ Readies, then $\langle\sigma, Lh\rangle \in \mathcal{G}[\![g_i]\!]G\sigma_0 \Longleftrightarrow \langle\sigma, h\rangle \in \mathcal{G}[\![g_i']\!]G'\sigma_0 \wedge L = \mathcal{R}(G, \sigma_0)$*

*Proof:* $\langle\sigma', Lh\rangle \in \mathcal{G}[\![g_i]\!]G\sigma \Leftrightarrow$

**1.** $L = \mathcal{R}(G, \sigma) \wedge h = \mathcal{R}(G, \sigma)^{t-1}\{\alpha_v\} \wedge 0 \leq t < minwait(G, \sigma) \wedge g_i$ is an I/O command

**or 2.** $L = \mathcal{R}(G', \sigma) \wedge h = \mathcal{R}(G, \sigma)^{t-1} \wedge t = minwait(G, \sigma) \wedge t = min(\sigma(d), 1) \wedge g = \mathsf{wait}d$

Then 1 $\{\Longleftrightarrow\}$

$L = \mathcal{R}(G', \sigma) \wedge h = \mathcal{R}(G', \sigma)^{t-1}\{\alpha_v\} \wedge 0 \leq t-1 < minwait(G', \sigma) \wedge g_i, \sigma \xrightarrow{\emptyset, r} g_i, \sigma$

and 2 $\Leftrightarrow$

$L = \mathcal{R}(G', \sigma) \wedge h = \mathcal{R}(G', \sigma)^{t-1} \wedge t-1 = minwait(G', \sigma) \wedge t-1 = min(\sigma(d-1), 1) \wedge g, \sigma \xrightarrow{\emptyset, \emptyset} \mathsf{wait}d-1, \sigma$

And these two cases exactly add up to

$\langle\sigma', h\rangle \in \mathcal{G}[\![g_i']\!]G'\sigma \wedge L = \mathcal{R}(G, \sigma)$

For histories of length zero, the lemma obviously holds.

□

Now we can prove the induction step for the guarded statement.

$\langle\sigma', Lh\rangle \in \mathcal{D}[\![GS]\!]\sigma$ $\{\Longleftrightarrow\}$

$\exists i: \langle\sigma', Lh\rangle \in \mathcal{D}[\![S_i]\!]^+(\mathcal{G}[\![g_i]\!]G\sigma)$ $\{\Longleftrightarrow\}$

$\exists i,: \langle\sigma_1, Lh_1\rangle \in \mathcal{G}[\![g_i]\!]G\sigma \wedge \langle\sigma', h_2\rangle \in \mathcal{D}[\![S_i]\!]\sigma_1$ $\{\Longleftrightarrow\}$

$\exists i,: \langle\sigma_1, h_1\rangle \in \mathcal{G}[\![g_i']\!]G'\sigma \wedge L = \mathcal{R}(G, \sigma) \wedge \langle\sigma', h_2\rangle \in \mathcal{D}[\![S_i]\!]\sigma_1$ $\{\Longleftrightarrow\}$

$\exists i,: L = \mathcal{R}(G, \sigma) \wedge \langle\sigma', h_1h_2\rangle \in \mathcal{D}[\![S_i]\!]^+(\mathcal{G}[\![g_i']\!]G'\sigma)$ $\{\Longleftrightarrow\}$

$g_i, \sigma \xrightarrow{\emptyset, r_i} g_i', \sigma \wedge \cup r_i = L \wedge \langle\sigma' h\rangle \in \mathcal{D}[\![GS']\!]\sigma$ $\{\Longleftrightarrow\}$ $g_i, \sigma \xrightarrow{\emptyset, r_i} g_i', \sigma \wedge \cup r_i = L \wedge \langle\sigma', h\rangle \in \mathcal{J}_3[\![GS']\!]\sigma$ $\{\Longleftrightarrow\}$ $\langle\sigma', Lh\rangle \in \mathcal{J}_3[\![GS]\!]\sigma$

If $L \cap$ Readies $= \emptyset$, the argument is simple.

□

## References

[FLP84] N. Francez, D. Lehman, and A. Pnueli. A linear history semantics for distributed programming. *Theoretical Computer Science*, 32, 1984.

# Chapter 3

# Modelling Statecharts behaviour in a fully abstract way

# Modelling Statecharts behaviour in a fully abstract way

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

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].

†) 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].

In Statecharts, they are *semantically* impossible, because 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 total preorder 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.

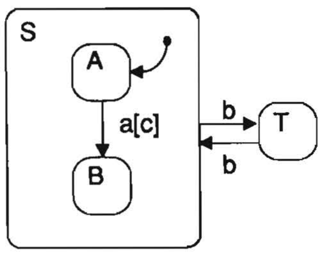
Some syntactical operators lack a clear counterpart in conventional languages. This is because in the graphical representation of Statecharts, the notion of area plays an important role, as it defines a hierarchy of states. Subareas of states are associated with alternative activities or concurrent activities. 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.

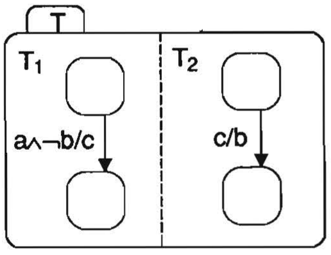
The semantics that we develop in chapter 4 is compositional w.r.t. the above syntax. The domain in which Statecharts acquire meaning basically records computations as functions that associate to every time point a record, $(F, C, \leq)$, that represents the activity at that time. Such a record states 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. This proof has a modular setup, which makes it possible to adapt it easily to new features in the language. As an example of this, we extend Statecharts with variables and show what extensions have to be made to the proof to handle this.

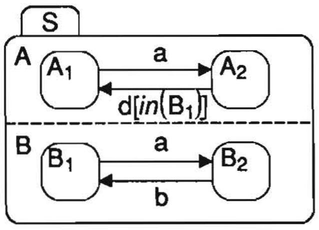
**Example 1**



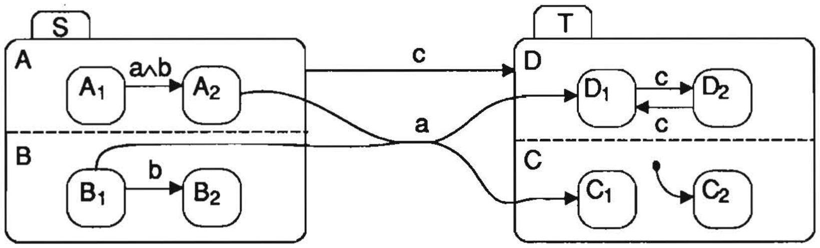
**Example 5**



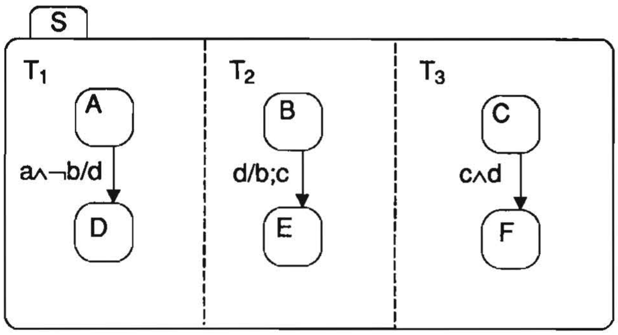
**Example 2**



**Example 3**



**Example 4**



## 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 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 $C_1$ 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, which 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 A 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* $\Sigma$ 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;

$e \in E_p \rightarrow e \in E$;

$e_1, e_2 \in E_p \rightarrow e_1 \wedge e_2,\ e_1 \vee e_2 \in E$;

$e \in E \rightarrow \neg e \in E$;

$n \in \mathbb{N} \backslash \{0\},\ e \in E \rightarrow time-out(e, n) \in E$ ▯

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. This will be explained in section 4.1.

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

**Definition.**

The set of **conditions** $C$ is inductively defined by

$true,\ false \in C$;

$c_1, c_2 \in C \rightarrow c_1 \wedge c_2,\ c_1 \vee c_2 \in C$;

$c \in C \rightarrow \neg c \in C$;

$S \in \Sigma \rightarrow in(S) \in C$ ▯

**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 $i=1,\dots,n \rightarrow a_1, \dots, a_n \in A$ ▯

**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

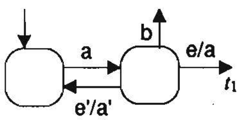
Providing a syntax for Statecharts is done using a notion of *incomplete statechart* or *Unvollendete,* abbreviated as *Unv.* This is a statechart in the process of being built up. It differs from a complete statechart in that it need not have a unique root state (i.e. a state of which all other states 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

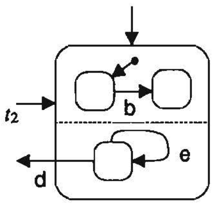
**Prim**$(I, O, A)$

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.

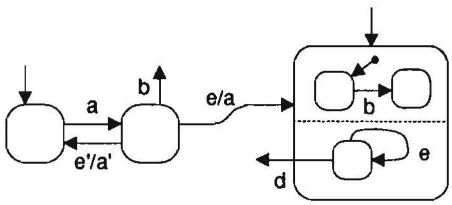
**Example 6**



$U_1$

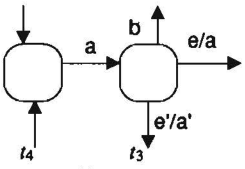


$U_2$

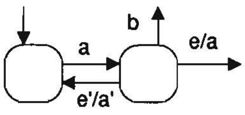


**Conc** $(U_1,t_1,t_2,U_2)$

**Example 7**

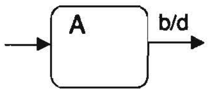


U

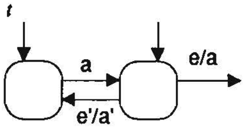


**Conn** $(U,t_3,t_4)$

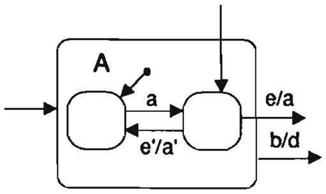
**Example 8**



$U_1$



$U_2$



**Stat** $(U_1,U_2,t)$

*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:

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

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 $L: T_O \rightarrow$ **Lab**.
Then the set of **Statecharts** is defined by

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

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

$B \rightarrow PrimChart(\emptyset, \emptyset)$
$PrimChart(I,O) \rightarrow \mathbf{Prim}(I,O,A)$
$PrimChart((I_1 \cup I_2)\backslash\{i\}, O_1 \cup O_2) \rightarrow \mathbf{Stat}(i, \mathbf{Prim}(I_1, O_1, A), OrChart(I_2, O_2))$ with $i \in I_2$
$PrimChart(I_1 \cup I_2, O_1 \cup O_2) \rightarrow \mathbf{Stat}(\mathbf{Prim}(I,O,A), OrChart(I_2, O_2))$
$PrimChart(I_1 \cup I_2, O_1 \cup O_2) \rightarrow \mathbf{Stat}(\mathbf{Prim}(I,O,A), AndChart(I_2, O_2))$

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

$AndChart(I, O) \rightarrow PrimChart(I, O)$
$AndChart(I, O) \rightarrow \mathbf{Close}(e, AndChart(I,O))$
$AndChart(I,O) \rightarrow \mathbf{Hide}(e, AndChart(I,O))$
$AndChart((I_1 \cup I_2) \setminus \{i_1', \ldots, i_n'\}, O_1 \cup O_2) \rightarrow \mathbf{And}(AndChart(I_1, O_1),$
$AndChart(I_2, O_2), \{(i_1, i_1'), \ldots, (i_n, i_n')\})$
with $i_i \in I_1$ and $i_i' \in I_2$ $\square$

### 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 - Conn($o,i,U$)**

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 - Stat(Prim($I,O,A$),$U$)**

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 optional 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 - And($U_1, U_2, \{(i_1, i_1'), ..., (i_n, i_n')\}$)**

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 - Or($U_1, U_2$)**

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 - Close($e,U$)**

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 - Hide($e, U$)**

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 $\mathbb{N}$.

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 *histories*, each history corresponding to one possible execution.

The set of histories $I\!H$ is defined by

$$I\!H = \{(i,m,f,o,m) \mid i \in T_I \cup \{*\},\ m,n \in dN, f \in (I\!N \to_p \boldsymbol{C}), o \in T_O \cup \{*, \perp\},\ m \leq n,$$
$$Dom(f) = \{0,\ldots,n\}\}$$

where $T_I$ and $T_O$ are the sets of incoming respectively outgoing transition identifiers from the syntax, $\boldsymbol{C}$ denotes the set of so-called *clock–records*, which will be defined later, and $I\!N \to_p \boldsymbol{C}$ denotes the set of partial functions from $I\!N$ to $\boldsymbol{C}$.

A history consists of five components. The first two components give the incoming transition of the chart by which the execution starts and the time at which this occurs. The last two components do the same for the outgoing transition. The outgoing transition equals "$\perp$" in case of an incomplete computation. In the case that there is no actual incoming or outgoing transition taken by the component, we denote this by $*$. For the start of the execution this is the case when we have the root state of the complete statechart, or a component of an AND-state that is started implicitly by an incoming transition of another component (see fig). The execution can end without a transition when it is interrupted by another component taking a transition that exits the complete construct (either an orthogonal composition or a statified state).

The third component of the histrory is a function that associates to each time unit, a so-called clock record. The precise structure of clock-records, $\boldsymbol{C}$, is defined later. The records associated to time values less than the starting time 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 and to evaluate conditions.

**Notation:**

Let $h \in I\!H$, $h = (i,n,f,o,m)$. Then we define:

- The *projections in, out, st, end* and $\bar{\ }$ are defined by:
  $h = (in(h), st(h), \bar{h}, out(h), end(h))$
  If there is no confusion, we will just write:
  - $h$ instead of $\bar{h}$,
  - $h(st)$ instead of $\bar{h}(st(h))$,
  - $h(end)$ instead of $\bar{h}(end(h))$.

□

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

**Definition**

$f \leq f'$ iff $in(f) = in(f') \wedge st(f) = st(f') \wedge out(f) = \perp \wedge end(f) \leq end(f') \wedge \bar{f} = \bar{f'}$ □

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

**Definition**

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

The semantical domain is defined as follows:

**Definition**

The domain is $(\mathbb{D}, \leq, \bot_{\mathbb{D}})$, where $\mathbb{D} = \{H \subseteq \mathbb{H} \mid H \text{ is prefix-closed}\}$ and $\bot_{\mathbb{D}} = \emptyset$ ▯

**Theorem**

$(\mathbb{D}, \leq, \bot_{\mathbb{D}})$ is a cpo. ▯

*Proof:*

Standard. ▯

**Definition**

A **total preorder** on a set $A$ is a binary relation $\leq$ on $A$, such that for all $a,b,c \in A$:

| | |
|---|---|
| (transitivity) | $a \leq b \wedge b \leq c \rightarrow a \leq c$ |
| (reflexivity) | $a \leq a$ |
| (totality) | $a \leq b \vee b \leq a$ ▯ |

If $a \leq b$ and $b \neq a$ we write $a < b$. Notice that $\leq$ induces an equivalence relation $\sim$ on $A$: $a \sim b$ iff $a \leq b \wedge b \leq a$.

**Definition**

$\mathbb{C} = \{(F, C, \leq) \mid F \subseteq C \subseteq E_p, \leq \text{ a total pre-order on } C\}$

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 $f^S(i) = (f^C(i), f^{\leq}(i))$

If there can be no confusion, instead of $(a,b) \in f^{\leq}(i)$ we will write $a \leq_i b$ or even $a \leq b$. ▯

For one particular time-step, a clock record describes the behaviour of the total system (component *and* environment) by $C$ and $\leq$, a total preorder 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 the information provided by $C$ and $F$ 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 total preorder 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. This is the "not yet" interpretation of the not operator from [HPSS].

**Example 11** (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 preorder 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 $a \leq b \wedge b \leq a$) 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 *at the time that* 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 $T_E$ that also gives all past histories that are consistent with the transition taking place.

**Definition**

$\mathbb{G} = \{(f,n) \mid f: \mathbb{N} \rightarrow_p E_p, n \in \mathbb{N}, Dom(f)=\{0,...,n\}\}$

$\mathbb{G}$ is a set of simplified histories. We are only interested in the end point (also referred to as $end(h)$) and in the sets $C$. The *shift* operator changes the end point of such a history.

$shift(h,j) = (h, max(0, end(h)-i))$

$T_E: E \rightarrow 2^G$ is defined recursively as follows:

$T_E(a) = \{h \mid h(end) \in T_0(a)\}$

$T_E(\neg e) = \{h \mid h \notin T(e)\}$

$T_E(tm(e,n)) = \{h \mid end(h) \geq n \wedge shift(h,-n) \in T(e) \wedge \forall 0<i<n: shift(h,-i) \in T(\neg e)\}$

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

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

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 $shift(f,n) \in T(e)$ ($e$ 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 $T_0(tm(e,n)) = \{C \mid C \subseteq E_p\}$.

The semantics of conditions is defined as follows:

**Definition**

$T_C: C \rightarrow 2^G$

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

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

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

The semantics of actions is as follows:

**Definition**

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

$T_A(\mu) = \varnothing$

$T_A(a) = \{a\}$ for $a \in E_p$

$T_A(a_1;a_2) = T_A(a_1) \cup T_A(a_2)$ 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 $\leq$ be an ordering relation on $C$ and $F \subseteq C$. Then the **set of predecessors** of $F$ under $\leq$ is defined by:

$\leq\!-pred(F) = \{a \in C \mid \exists b \in F: b < F\}$

**Definition** In the following, $F$, $C$ and $\leq$ abbreviate $g^F(end)$, $g^C(end)$ and $g^{\leq}(end)$ respectively.

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

$T(e[c]/a) =$

$\{g \in 2^{\mathbb{H}} \mid \exists h \in 2^G: h \in T_E(e) \cap T_C(c), end(h) = end(g),$

$\forall i \neq end(h): g^F(i) = \varnothing, h = g^C,$

$ex(*) \in F, F \setminus \{ex(*)\} \subseteq T_A(a) \subseteq C,$

$\leq\!-pred(F) = h(end),$

$\forall f_1, f_2 \in F: f_1 \sim f_2$

$\}$ □

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 dependent on $a$ being generated outside.

### 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 $\leq$-components of the clock records should be equal. The $F$-components, however, describe only the local contributions and hence these should be unified.

**Definition**

$(F_1, C_1, \leq_1) \| (F_2, C_2, \leq_2) = (F_1 \cup F_2, C_1, \leq_1)$ if $C_1 = C_2$ and $\leq_1 = \leq_2$
$\qquad\qquad\qquad\qquad\qquad\qquad$ = undefined otherwise

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

$\quad (f_1 \| f_2)(i)\colon \mathbb{N} \to_p \mathbb{C}$
$\quad (f_1 \| f_2)(i) = f_1(i) \| f_2(i)$ for all $i \leq min(end(f_1), end(f_2))$
$\qquad\qquad\qquad = (f_j^F(i), f_j^C(i), f_j^{\leq}(i))$ if $end(f_{3-j}) < i \leq end(f_j)$
$\qquad\qquad\qquad$ = undefined otherwise $\qquad\qquad\qquad\qquad$ []

We define the semantic function

$$[\![ . ]\!] \colon \mathbf{Stch} \to \mathbb{D}$$

by induction on the structure of **Stch** in the following sections.

#### 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

$\quad O = \{o_1, \cdots, o_n\}, L(o_i) = e_i[c_i]/a_i,$
$\quad T_i = \{h \mid h \in T(L(o_i)), out(h) = o_i\},$

$$W_i = T(\neg e_i) \cup T(\neg c_i),\ W = \bigcup_{i=1}^{n} W_i.$$

Then

$[\![\mathbf{Prim}(I,O,S)]\!] =$

$\{g \in I\!H \mid \exists h \in I\!H$:

$in(h) \in I \cup \{*\},\ out(h) \in O \cup \{*, \perp\}$

$in(h) \in I \rightarrow st(h) > 0,\ out(h) \in O \rightarrow h \in T(L(out(h))$

$\forall st(h) \leq i < end(h) : shift(h, i - end(h)) \in W$

$st(h) > 0 \rightarrow g^F(st-1) = \{\text{en}\}$

$out(h) \in \{*, \perp\} \rightarrow g^F(end) = \varnothing$

$out(h) \neq \perp \rightarrow g^F(end) = h^F(end)$

$\forall i : f_1, f_2 \in g^F(i) \rightarrow f_1 \sim f_2$

$g^C = h^C,\ g \leq = h \leq$

$\forall i : i \neq st(g) \wedge i \neq end(g) \rightarrow g^F(i) = \varnothing$

$\}^{CL}(ex(S)/ex(*))$

□

## 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 $delete_{i,j}(D) = \{h \in D \mid \wedge in(h),\ out(h) \notin \{i,j\}\}$

and $conc(D_1, t_1, t_2, D_2) =$

$\{h \mid \exists h_i \in D_i$:

$st(h_1) = st(h) \wedge end(h_1) = st(h_2) \wedge end(h_2) = end(h) \ \wedge\ out(h_1) = in(h_2) \ \wedge$

$\bar{h} = h_1 \| h_2 \ \wedge\ \forall f_1, f_2 \in h_1^F(end) : f_1 \sim f_2\ \} \cup D_1 \cup D_2$

□

### 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.

**Definition**

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

where $\mu$ 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 operator, $\|$, can be found in section 4.3.1. Now we define how two given histories should be combined.

**Definition**

Let $f_1, f_2 \in I\!H$. We define the predicate **MERGE** as follows:

$MERGE(h_1,h_2,h) \Leftrightarrow$

(i) $st(h_1) = st(h_2) \wedge end(h_1) = end(h_2) \wedge h = h_1 \| h_2$

(ii) $out(h) \neq \perp \rightarrow \exists j : out(h) = out(h_j) \wedge out(h_{3-j}) = *$

(iii) $out(h) = \perp \rightarrow out(h_1) = out(h_2) = \perp$

(iv) $\forall a,b \in h^F(st): a \sim b \wedge \forall a,b \in h^F(st): a \sim b$ ▯

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**

$[\![\mathbf{And}(U_1, U_2, \{(t_1, w_1),\ldots,(t_n, w_n)\})]\!] = \{h \mid \exists h_i \in [\![U_i]\!]:$

$[(\exists j : in(h) = in(h_j) \wedge in(h_{3-j}) = *) \vee \exists 1 \leq j \leq n : in(h) = t_j \wedge in(h_1) = t_j \wedge in(h_2) = w_j]$

$\wedge\ MERGE(v_1, h_1, v_2, h_2, v, h)\}$

▯

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

## 4.3.5. Statification

There are two types of *Statification*, one with and one without a default.

**Definition**

$[\![\mathbf{Stat}(d, U_1, U_2)]\!] = \{h \mid \exists h_i \in [\![U_i]\!]:$

$((in(h)=in(h_1) \wedge in(h_2)=d) \vee (in(h)=in(h_2) \wedge in(h_1)=*) \wedge MERGE(h_1,h_2,h))\}$

$[\![\mathbf{Stat}(U_1, U_2)]\!] = \{h \mid \exists h_i \in [\![U_i]\!]:$

$[(\exists j: in(h)=in(h_j) \wedge in(h_{3-j})=*) \wedge MERGE(h_1,h_2,h)\}$ □

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 ($in(f)=in(f_2)$) or take a transition to the outer state and enter some state(s) inside by default ($in(f)=in(f_1)$).

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 ($in(h)=in(h_1) \wedge in(h_2)=*$) or by entering the inner structure explicitly ($in(h)=in(h_2) \wedge in(h_2)\neq*$). 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. Hiding and Closure

Hiding and Closure are operators that change the behaviour of an Unv only with respect to one event. In the rest of this section this is the event *a*. All the necesssary operations are first defined on clock records and later applied uniformly to histories. These operations have the form of relations: *aRb* means that *a* can be transformed to *b*.

### Closure

Closure with respect to an event *a* makes the closed statechart insensitive to all *a*-events generated outside the chart. Consequently, all histories should be deleted in which *a* is claimed to occur ($a \in C$), but in which *a* is not generated. This is performed by the relation *OK*. The relation *Id* is the identity relation on clock records.

**Definition**

$(F_1, C_1, \leq_1)\ OK\ (F_2, C_2, \leq_2) \iff a \in C \rightarrow a \in F$ □

With these operation we can define the semantics of Closure.

$$CLOSE = (OK \cap Id)$$

$$[\![\mathbf{Close}(a,U)]\!] = \{g \mid \exists h\ \forall i: h(i)\ CLOSE\ g(i)\}$$

**Hiding**

*Hiding* an Unv with respect to an event *a* makes that event fully internal to the Statechart. This means first of all that any claim of *a* should be satisfied by a generation of *a* inside the Unv. In other words, *Closure* should be applied first. Furthermore, the Unv becomes fully insensitive to generations of the event *a* outside and the environment of the Unv becomes insensitive to generations of *a* inside. This means that the denotation should be *saturated* with histories that display any behaviour as far the event *a* is concerned. The relation *SAT* performs this. At the end, any occurrence of *a* in the *F*-set is removed by the relation *DEL*.

**Definition**

$(F_1, C_1, \leq_1)\, SAT\, (F_2, C_2, \leq_2) \Leftrightarrow$

$\quad a \in F_2 \rightarrow a \in C_2$

$\quad F_1 \setminus \{a\} = F_2 \setminus \{a\}$

$\quad C_1 \setminus \{a\} = C_2 \setminus \{a\}$

$\quad \leq_1 \setminus \{a\} = \leq_2 \setminus \{a\}$

$\quad a \in F_2 \rightarrow a \in F_1$

$(F_1, C_1, \leq_1)\, DEL\, (F_2, C_2, \leq_2) \Leftrightarrow F_2 = F_1 \setminus \{a\} \wedge C_1 = C_2 \wedge \leq_1 = \leq_2$

$$HIDE = (OK \cap Id) \circ SAT \circ DEL$$

$$[\![\mathbf{Hide}(a, U)]\!] = \{ g \mid \exists h\ \forall i : h(i)\ HIDE\ g(i) \}$$

□

Hiding and Closure can not only be applied to primitive events, but also to states. This is tantamount to Closing or Hiding with respect to the events *entered* and *exited*.

$$[\![\mathbf{Close}(S, U)]\!] = [\![\mathbf{Close}(en(S), Close(ex(S), U))]\!]$$

$$[\![\mathbf{Hide}(S, U)]\!] = [\![\mathbf{Hide}(en(S), Hide(ex(S), U))]\!]$$

# 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 $C[X]$ is a context and P a program then $C[P]$ is the program that results from plugging $P$ into the hole, denoted by $X$, of $C$. Let $O(P)$ give the observable behaviour of program $P$.

**Definition**

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

$$\text{for all programs } P,Q\text{: } [\![P]\!]=[\![Q]\!] \Leftrightarrow \text{for all contexts } C: O(C[P])=O(C[Q]) \qquad []$$

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

$$O(P)=\{h^F \mid h\in[\![P]\!]\}$$

**Theorem**

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

Before we give the proof, we introduce some preliminary definitions and lemmas. The main difference between this proof and that of [HGR88] is that the first is much more structured. It can be easily adapted to extensions of the model, e.g., to cover variables in the language, as shown in Section 5.1.

## Domain properties

For the proof of full abstraction, we need some properties of the domain. Namely, not all prefixed-closed subsets of $I\!H$ can be generated by the semantic function $[\![.]\!]$.

**Lemma 1** For all programs $P\in$ **Stch** and histories $h\in[\![P]\!]$:

(i) if $out(h)=\bot$, then there exists $h'\in[\![P]\!]$ with $out(h)\neq\bot$ and $h\leq h'$.

(ii) there exists $h'\in[\![P]\!]$ with $st(h')>st(h)$, $end(h')=end(h)+st(h')-st(h)$, $out(h')=out(h)$, $in(h')=in(h)$ and $\forall st(h')\leq i\leq end(h')$: $h'(i)=h'(i-(st(h')-st(h)))$.

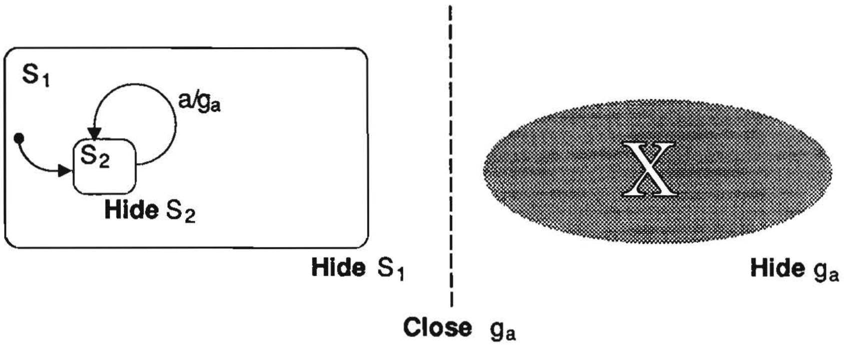
(iii) if $h\restriction A=h'\restriction A$ and $A$ is a set of events that contains all the events occurring in $P$ (including en and $ex(S)$ for states $S$ in $P$), then $h'\in[\![P]\!]$.

*Proof* By structural induction to $P$.

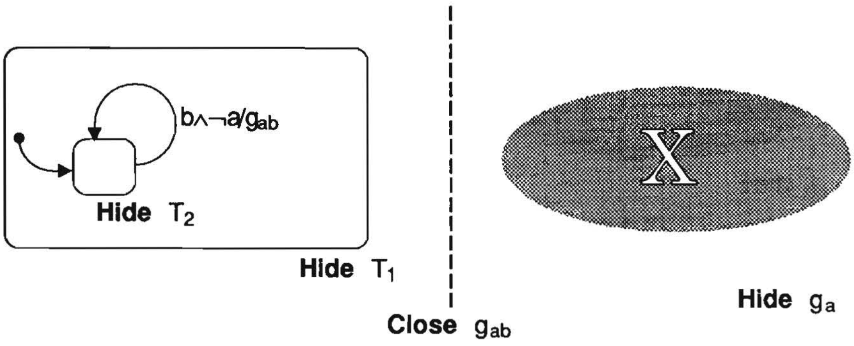
## Properties of histories

In the following definitions, we notate a property of a history by $\phi(h)$, $\psi(h)$, $\cdots$, where $\phi$, $\psi$, $\cdots$ are logical formulas in which $h$ occurs as a free variable.

$\mathbf{F_a}$



$\mathbf{F_{ab}}$

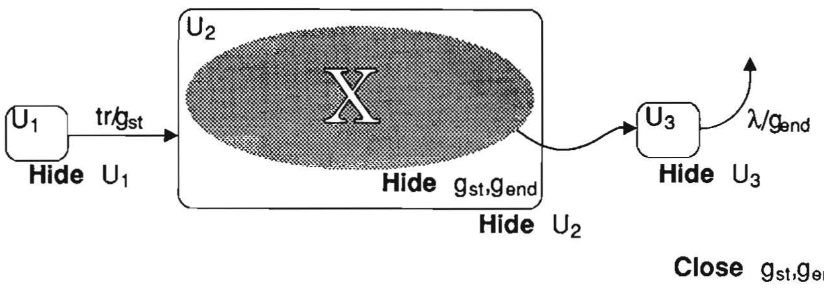


$\mathbf{F_{tu}}$

$t = *$ $\quad$ $tr = tm(en(U_1, m))$

$u \neq *, \perp$ $\quad$ or

$tr = \lambda$



**Definition** A history property $\phi(h)$ is **observable** iff for all histories $h_1, h_2$ holds:

$$\phi(h_1) \leftrightarrow \phi(h_2) \Leftrightarrow h_1^F = h_2^F$$

**Definition** Let $\approx$ be an equivalence relation on $\mathbb{H}$. A formula $\phi$ **characterises** a history $h$ **upto** $\approx$ iff for all histories $h'$ holds:

$$\phi(h') \Leftrightarrow h \approx h'$$

**Filters**

The central notion in the proof is that of the *filter*. A filter is a context, especially designed to make properties of histories observable. For every event, we use a filter that observes the presence of this event in the set of claims $C$ of a particular time-step. For every pair of events we use a filter that observes whether this pair is in the total preorder associated to a particular time unit or not.

**Definition**

Let $\phi(h)$ and $\psi(h)$ be some properties of histories; a **filter for** $\phi$ **by** $\psi$ is a context $F[X]$ such that for all programs $P$

$$\exists h \in [\![P]\!]: \phi(h) \Leftrightarrow \exists h \in [\![F[P]]\!]: \psi(h)$$

**Convention**

In the following we refer to an Unvollendete by the name of the root state, if it exists. So in the example below, $S_1$ denotes the Unvollendete $[<S_1>, \varnothing, \varnothing]$.

Furthermore, we hide all state names to simplify the denotations.

Define $F_a[X]$=**Close**($g_a$, **And**(**Hide**($g_a$,$X$),**Stat**($t_1$, **Hide**($S_1$,$[<S_1>,\varnothing,\varnothing]$),
**Conn**($t_2$, $t_3$, **Hide**($S_2$, $[<S_2>, \{t_1,t_3\}, \{t_2\}]$)))))

where $L(t_2) = a/g_a$.

□

**Lemma 2** Let $i$ be an arbitrary integer; then

(i) $F_a[X]$ is a filter for $st(h) \le i < end(h) \wedge a \in h^C(i)$ by $g_a \in h^F(i)$.

(ii) $F_a[X]$ is a filter for $a \notin h^C(i)$ by $g_a \notin h^F(i) \wedge st(h) \le i < end(h)$.

*Proof*

(a) $[\![S_1]\!] = \{h \mid in(h)=*, out(h)=*, h(i) \in W \cup T \text{ for } in(h) \le i < end(h)\}^{CL}$
where $W = \{(\varnothing, C, \le) \mid a \notin C\}$ and
$T = \{(\{g_a\}, C, \le) \mid \{a, g_a\} \subseteq C \wedge a < g_a\} \cup \{(\varnothing, C, \le) \mid \{a, g_a\} \subseteq C\}$

(b) $[\![\text{And}(S_1, X)]\!] = \{h \mid \exists h_1, h_2 : h_1 \in [\![S_1]\!] \wedge h_2 \in [\![X]\!] \wedge$
$in(h)=* \wedge (out(h)=* \vee (out(h)=t \wedge out(h_2)=t)) \wedge$
$st(h)=st(h_1)=st(h_2) \wedge end(h)=end(h_1)=end(h_2)$

$$h^F = h_1^F \cup h_2^F \wedge h^C = h_1^C = h_2^C \wedge h^{\leq} = h_1^{\leq} = h_2^{\leq} \}$$

(c) Now suppose that $h_1 \in [\![X]\!]$ and $\phi_1(h_1)$. Take $h_2 \in [\![S_1]\!]$, such that $h_2^{C,\leq} = h_1^{C,\leq}$ and $g \in h_2(i)^F$. (ii) shows that this is possible. Then from (b), we see that there exists $h \in [\![\text{And}(S_1,X)]\!]$ with $g_a \in h(i)$ □

**Corollary** Let $h_0$ be an arbitrary history. Then $F_a[X]$ is a filter for

$\phi_a$: $\forall st(h) \leq i < end(h)$: $h^C(i) \restriction a = h_0^C(i) \restriction a$ by

$\psi_a$: $\forall st(h) \leq i < end(h)$: $g_a \in h^F(i) \leftrightarrow a \in h_0(i)$

This filter makes the occurrence of the event $a$ in the $C$-sets of a history observable. The idea of the proof is to combine several of these filters, thus obtaining a big filter that exactly filters out the history we want. To achieve this, we need filters that have a special property that makes it possible to combine them.

**Definition** A filter $F[X]$ for $\phi$ by $\psi$ is **transparent** for $\chi$ iff for all programs $P$:

$$\exists h \in [\![P]\!]: \phi(h) \wedge \chi(h) \Leftrightarrow \exists h \in [\![F[P]]\!]: \psi(h) \wedge \chi(h)$$

**Example** The filter $F_a$ is transparent for any property not involving $a$ or $g_a$.

**Proposition** If F1 and F2 are transparent for $\chi$, so is $F_1[F_2[X]]$.

The idea of the proof is to combine many filters, each filtering out a specific property of a particular history. The combination of all these filters should then make all the relevant properties of the history *observable*. Unfortunately, we cannot directly derive that transparency for two properties implies transparency for the conjunction of these properties. For this, we need somewhat stronger notions of filter and transparency.

**Definition**

A filter $F$ is **transformational** if there are functions $\iota, \kappa: D \rightarrow D$ such that

$$\forall h \in [\![P]\!]: \phi(h) \Rightarrow \iota(h) \in [\![F[P]]\!] \wedge \psi(\iota(h))$$

$$\forall h \in [\![F[P]]\!]: \psi(h) \Rightarrow \kappa(h) \in [\![P]\!] \wedge \phi(\kappa(h))$$

**Definition**

A filter $F$ is **transformationally transparent** for $\chi$ if it is a transformational filter that is transparent for $\chi$ and furthermore:

$$\chi(h) \Leftrightarrow \chi(\iota(h))$$

$$\chi(\kappa(h)) \Leftrightarrow \chi(\kappa(h))$$

**Proposition** If $F$ is a transformational filter for $\phi$ by $\psi$ and it is transformationally transparent for $\chi_1$ and for $\chi_2$, then $F$ is transformationally transparent for $\chi_1 \wedge \chi_2$.

*Proof*
$h \in [\![P]\!]$ and $\phi(h)$ and $(\chi_1 \wedge \chi_2)(h) \Rightarrow h \in [\![P]\!]$ and $\phi(h)$ and $\chi_1(h)$ and $\chi_2(h) \Rightarrow \iota(h) \in [\![F[P]]\!]$ and $\phi(\iota(h))$ and $\chi_1(\iota(h))$ and $\chi_2(\iota(h)) \Rightarrow \iota(h) \in [\![F[P]]\!]$ and $\phi(\iota(h))$ and $(\chi_1 \wedge \chi_2)(\iota(h))$
The reverse is analogous.

**Lemma 3**

(i) If $F_1$ and $F_2$ are filters for $\phi_1$ by $\psi_1$ respectively $\phi_2$ by $\psi_2$, and $\psi_1 \longleftrightarrow \phi_2$, then $F_2[F_1[X]]$ is a filter for $\phi_1$ by $\psi_2$. (ii) If $F_1$ and $F_2$ are filters for $\phi_1$ by $\psi_1$ respectively $\phi_2$ by $\psi_2$, and $F_1$ is transparent for $\phi_2$ and $F_2$ is transparent for $\psi_2$, then $F_2[F_1[X]]$ is a filter for $\phi_1 \wedge \phi_2$ by $\psi_1 \wedge \psi_2$. $\square$

*Proof*(i) Let $h \in [\![X]\!]$ and $\phi_1(h)$. Then there exists $h_1 \in [\![F_1[X]]\!]$ with $\psi_1(h)$, because $F_1$ is a filter for $\phi_1$ by $\psi_1$. Then $\phi_2(h)$ holds, because $\psi_1$ and $\phi_2$ are equivalent. Then there must also exist $h_2 \in [\![F_2[F_1[X]]]\!]$ with $\psi_2$, because $F_2$ is a filter for $\phi_2$ by $\psi_2$.

For the other direction the proof can be reversed.

*Proof*(ii) Let $h \in [\![X]\!]$ and $\phi_1(h) \wedge \phi_2(h)$. Then there exists $h_1 \in [\![F_1[X]]\!]$ with $\psi_1(h) \wedge \phi_2(h)$, because $F_1$ is a filter, transparent for $\phi_2$. Then there must also exist $h_2 \in [\![F_2[F_1[X]]]\!]$ with $\psi_2 \wedge \psi_2$, because of the transparency of $F_2$.

For the other direction the proof can be reversed. $\square$

To observe the ordering between two events we use a filter similar to $F_a$. Remember that a transition labelled $a \wedge \neg b$ *can* be taken in the case that both $a$ and $b$ occur in the same step, but then $b$ must have occurred in a later micro-step than $a$.

**Definition** $F_{ab}[X] =$ **Close**($g_{ab}$, **And**(**Stat**($t_1$, **Hide**($T_1$, [<$T_1$>,$\varnothing$, $\varnothing$]), **Conn**($t_2$, $t_3$, $T_2$)), $X$)),
where $T_2 =$ **Hide**($T_2$,[<$T_2$>,$\{t_2\}$, $\{t_3\}$]) and $L(t_2) = b \wedge \neg a / g_{ab}$ $\square$

**Lemma 4** $F_{ab}$ is a filter for
$st(h) \leq i < end(h) \wedge ((a,b) \notin h^{\leq}(i) \wedge b \in h^{C}(i))$ by $g_{ab} \in h^{F}(i)$

**Corollary** Let $h_0$ be an arbitrary history. Then $F_{ab}[X]$ is a filter for

$\phi_{ab}$: $\forall st(h) \leq i < end(h)$: $((a,b) \notin h^{\leq}(i) \wedge b \in h^{C}(i)) \longleftrightarrow (a,b) \notin h_0^{\leq}(i)$ by

$\psi_{ab}$: $\forall st(h) \leq i < end(h)$: $(a,b) \notin h_0^{\leq}(i) \longleftrightarrow g_{ab} \in h^{F}(i)$

The following filter observes the incoming and outgoing transition, and it enables the other filters $F_a$ and $F_{ab}$ to make their observations also on the *past* of the original histories. This filter observes an incoming transition $t = *$ and an outgoing transition $u \neq *, \perp$. Any other combination can be observed with a similar filter.

**Definition**

$F_{tu} =$ **Close**($\{g_{st}, g_{end}\}$, **Conc**($U_1$, $t_1$, $t_2$, **Conc**(**Stat**($U_2$, **Hide**($\{g_{st}, g_{end}\}$, $X$), $u$, $t_3$, $U_3$)))),
where $U_1 =$ **Hide**($U_1$, [<$U_1$>,$\varnothing$, $\{t_1\}$]),
$U_2 =$ **Hide**($U_2$, [<$U_2$>,$\{t_2\}$, $\varnothing$]),
$U_3 =$ **Hide**($U_3$, [<$U_3$>,$\{t_3\}$, $\{t_4\}$]),
$L(t_1) = tr / g_{st}$,
$L(t_4) = \lambda / g_{end}$,

$tr = tm(en(U_1, m))$ if $m>0$,
$= \lambda$ if $m=0$. □

**Lemma 5** $F_{tu}$ is a filter for

$\phi_{tu}$: $in(h)=t \wedge out(h)=u \wedge st(h)=m \wedge end(h)=n$ by
$\psi_{tu}$: $g_{st} \in h^F(m) \wedge g_{end} \in h^F(n+1) \wedge end(h)=n+1$.
Furthermore, $h \in [\![F_{tu}[P]]\!] \wedge g_{st} \in h^F(m)$ implies $st(h)=0$ □

For the purpose of combining these filters we have the following transparencies. Let

$\overline{\phi}_a$: $\forall 0 \le i \le end(h_0)$: $h^C(i) \restriction a = h_0^C(i) \restriction a$ by
$\overline{\psi}_a$: $\forall 0 \le i \le end(h_0)$: $h^F(i) \restriction a = h_0^C(i) \restriction a$
$\overline{\phi}_{ab}$: $\forall 0 \le i \le end(h_0)$: $((a,b) \notin h^{\le}(i) \wedge b \in h^C(i)) \longleftrightarrow (a,b) \notin h_0^{\le}(i)$
$\overline{\psi}_{ab}$: $\forall 0 \le i \le end(h_0)$: $(a,b) \notin h_0^{\le}(i) \longleftrightarrow g_{ab} \in h^F(i)$
$\overline{\psi}_{tu}$: $g_{st} \in h^F(n) \wedge g_{end} \in h^F(n+1)$
$\chi_a$: $h^F \restriction a = h_0^F \restriction a$.

**Lemma 6** Let the integer $k$, the events $a$, $b$ and the transitions $t$, $u$ be given and let $c$, $d$ and $e$ be arbitrary events not of the form $g_x$, $g_{xy}$, $g_{st}$ or $g_{end}$. Then

$F_a$ is transparent for $st(h)=k$, $\overline{\psi}_{tu}$, $\overline{\phi}_{cd}$, $\overline{\phi}_e$, $\overline{\psi}_e$ and $\chi_F$ for any events $c$ and $d$ and any event $e \ne a$.

$F_{ab}$ is transparent for $st(h)=k$, $\overline{\psi}_{tu}$, $\overline{\psi}_c$, $\overline{\phi}_d$, $\overline{\phi}_e$ and $\chi_F$ for any event $c$ and any two events $(d,e) \ne (a,b)$.

$F_{tu}$ is transparent for $c \in h^C(i)$, $c \notin h^{C(i)}$, $(c,d) \in h^{\le}(i)$, $(c,d) \notin h^{\le}(i)$ and $\chi_F$ for any two events $c$ and $d$ and integer $i$. □

Now we can give the actual proof of full abstraction. Let $k \in [\![P]\!] \setminus [\![Q]\!]$, such that $out(k) \ne \bot$. This is possible because of Lemma 1 (i). Let $A = \{a_1, \ldots, a_k\}$ be the set of events occurring in $Q$, let $t = in(k)$ and let $u = out(k)$.

**Define** $F_C = F_{a_1}[\cdots F_{a_k}[X]\cdots]$ and $F_{\le} = F_{a_1 a_1}[\cdots F_{a_k a_k}[X]\cdots]$,
where $E_p = \{a_1, \cdots, a_k\}$.

Take $k$ for the history $h_0$ as used in the definitions of $\phi_a$, $\psi_a$ etc. Now, by the corollaries of Lemma 2 and 4 and Lemma 5 and 6, and by the observation that $\psi_{tu}(h)$ implies $end(h)=end(h_0)+1$, $F_{\le}[F_C[F_{tu}[X]]]$ is a filter for

$\overline{\phi}$: $\overline{\phi}_{a_1} \wedge \cdots \wedge \overline{\phi}_{a_k} \wedge \overline{\phi}_{a_1 a_1} \wedge \cdots \wedge \overline{\phi}_{a_k a_k} \wedge \phi_{tu} \wedge \chi_{a_1} \wedge \cdots \wedge \chi_{a_k}$ by
$\overline{\psi}$: $\overline{\psi}_{a_1} \wedge \cdots \wedge \overline{\psi}_{a_k} \wedge \overline{\psi}_{a_1 a_1} \wedge \cdots \wedge \overline{\psi}_{a_k a_k} \wedge \psi_{tu} \wedge \chi_{a_1} \wedge \cdots \wedge \chi_{a_k}$

We can write $\overline{\phi}$ and $\overline{\psi}$ instead of $\phi$ and $\psi$ for the following reason. Let $P$ be given and suppose $h \in [\![P]\!]$ and, e.g., $\phi_{\overline{a}}(h) \wedge st(h)=m$. Then, by transparency of $F_{tu}$, there exists $h_1 \in [\![F_{tu}[P]]\!]$ with $\phi_{\overline{a}}(h_1) \wedge g_{st} \in h_1^F(m)$. This implies that $st(h)=0$ and hence $\phi_a \longleftrightarrow \phi_{\overline{a}}$. For the other direction, we use that $F_a$ is transparent for $st(h)=k$. Define $h_1 \approx_A h_2$ iff $h_1 \restriction A = h_2 \restriction A$. We see that $\overline{\phi}$ characterises $k$ upto $\approx_A$. Furthermore, $\overline{\psi}$ is observable.

By the filter property, we know that there exists $l \in [\![F[P]]\!]$ with $\overline{\psi}(l)$ and hence $l^F \in O(F[P])$. This history $l^F$ is the history that makes the observable difference between $P$ and $Q$. In other words, we claim that $l^F \notin O(F[Q])$.

Suppose, by contradiction, that $l^F \in O(F[Q])$. Then there must be $l' \in [\![F[Q]]\!]$ that is observably the same as $l$, i.e., $l'^F = l^F$ and, because $\bar{\psi}$ is an observable property, $\bar{\psi}(l')$ must hold. By the filter property, there exists $k' \in [\![Q]\!]$ with $\phi(k')$. Because $\phi$ characterises $k$ upto $A$, we know that $k' \simeq_A k$. Then, by Lemma 1 (iii), $k \in [\![Q]\!]$, which contradicts our initial assumption. $\square$

### 5.1. Variables

One of the advantages of this modular proof of full abstraction is that it can easily be extended to cover extensions of the language. We illustrate this by extending Statecharts with variables. In this extension, it is possible to specify actions on a transition as *assignments* to variables. In the conditions we can test the values of these variables by arbitrary boolean expressions. Furthermore, for every variable $x$, there is an event *changed*$(x)$ that is generated when $x$ is written to.

#### 5.1.1. Syntax

Let a set of **variables Var** with values in $V$ be given and a set of **Boolean expressions** $B$**(Var)** over these variables as well as a set of **V-valued expressions** $E$**(Var)**. We extend the set of **primitive events** $E_p$ with the set $\{changed(x) \mid x \in \mathbf{Var}\}$. We extend the syntax of **Conditions** with the following clause:

$$b \in B(\mathbf{Var}) \to b \in C;$$

and the syntax of actions:

$$x \in \mathbf{Var} \wedge e \in E(\mathbf{Var}) \to x := e \in A;$$

#### 5.1.2. Semantics

Variables are shared and we disallow that two processes (orthogonal components) try to write to the same variable in the same step. The effect of a change is felt in the next *micro*-step following the action of writing. When a variable is written to, the event *changed*$(x)$, or $ch(x)$ for short, is generated.

To model this, we introduce a set of *internal* events of the form $x_v$, signalling the action of writing the value $v$ to the variable $x$.

$$E_i = \{x_v \mid x \in Var, v \in V\}$$

These events serve only semantic purposes and they can not be used in the labels of transitions. We redefine the clock-records:

$$\begin{aligned}\boldsymbol{C} = \{(F,C,\le) \mid\ & F \subseteq C \subseteq E_p \cup E_i,\ \le \text{ a total pre-order on } C,\\ & \forall x,v,v' : x_v, x_{v'} \in C \to v = v',\\ & \forall x : (ch(x) \in C \leftrightarrow \exists v : x_v \in C) \wedge x_v \sim ch(x)\}\end{aligned}$$

To give the new definitions of $T_C$ and $T_A$, we introduce valuation functions and how to interpret these on histories.

**Definition** **A valuation** is a partial function from **Var** to $V$. We extend it canonically to $B(\mathbf{Var}) \rightarrow \{\text{true, false}\}$ and to $E(\mathbf{Var}) \rightarrow V$.

**Definition** Let $h$ be a history, then $\sigma(h)$: $\mathbf{Var} \rightarrow V$ is a valuation defined by

$$\sigma(h)(x) = v \text{ iff } \exists n \leq end(h): x_v \in h^C(n) \wedge \forall n < k < end(h) \forall w \in V: x_w \notin h^C(k)$$

We extend the definition of $T_C$ with the following clause:

$$T_C(b) = \{h \mid \sigma(h)(b) = \text{true}\}$$

The semantics of an assignment is dependent on the evaluation of the expression. This means that the semantics of actions becomes dependent on the history. So we add the history as a parameter to the semantic function on actions, $T_A$.

$T_A: A \times I\!H \rightarrow 2^{I\!H}$

$T_A(\mu, h) = \varnothing$

$T_A(a,h) = \{a\}$ for $a \in E_p$

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

$T_A(x{:=}e,h)$ is $\{x_v \mid \sigma(h)(x){=}v\} \cup \{ch(x)\}$ □

In the semantics of complete labels, $T_A(a)$ should be changed by $T_A(a,h)$. All other definitions stay the same.

## 5.1.3. Full Abstraction

To maintain full abstraction, we need a new type of filter to observe the events $x_v$ in the $C$-sets. For the events $ch(x)$ no filter is needed, because the set $E_p$ is extended with these events and hence the filters of the type $F_a$ will take care of this.

**Define** $F_{xv}[X] = \mathbf{Close}(g_{xv}, \mathbf{And}(\mathbf{Hide}(g_a, X), \mathbf{Stat}(\mathbf{Hide}(V1, [<V_1>, \varnothing, \varnothing]), t_1, \mathbf{Conn}(t_2, t_3, \mathbf{Hide}(V2, [<V_2>, \{t_1, t_3\}, \{t_2\}])))))$

where $L(t_2) = ch(x)[x{=}v]/g_{xv}$. □

This is a filter for

$st(h) \leq i < end(h) \wedge x_v \in h_C(i)$ by $g_{xv} \in h_F(i)$ and

$x_v \notin h_C(i)$ by $g_{xv} \notin h_F(i) \wedge st(h) \leq i < end(h)$

*Proof*

The reason that the second case works is that our denotations satisfy the following property:

$$x_v \in C \rightarrow ch(x) \in C$$

So $x_v \notin C$ implies either $ch(x) \notin C$ or $x_w \in C$ with $v \neq x$. In both cases the transition cannot be taken.

Applying repeated conjunction to this filter, we get that $F_{xv}$ is a filter for
$\phi_{xv}$: $\forall st(h) \leq i \leq end(h)$: $x_v \in h^C(i) \longleftrightarrow x_v \in h^C_S(i)$ by
$\psi_{xv}$: $\forall st(h) \leq i \leq end(h)$: $x_v \in h^C_S(i) \longleftrightarrow g_{xv} \in h^F(i)$

For our set $A$ of relevant events, we choose

$A = \{a \mid a \textit{ in } Q\} \cup \{en(S), ex(S) \mid S \textit{ in } Q\} \cup \{ch(x), x_v \mid x \textit{ in } Q\}$

where *in* stands for syntactically occurring.

We cannot use a filter for every event in this set, because it is potentially infinite. We must allow that the set of values, $V$, is infinite. So we consider the subset $A'$ of events that occur in the history $k$ that makes the difference between $[\![P]\!]$ and $[\![Q]\!]$.

$$A' = A \cap \{x_v \mid \exists i : x_v \in k^C(i)\}$$

This set is finite, because

(i) the set of variables used syntactically in $Q$ is finite

(ii) for each variable $x$ there is at most one event of the form $x_v$ in the $C$-component of a clock-record.

(iii) the number of clock-records in a history is finite

Our new filter becomes:

$F_{x_1 v_1}[\cdots F_{x_n v_n}[F[X]]\cdots]$,

where $A' = \{x_1 v_1, \cdots, x_n v_n\}$ and $F[X]$ is the filter from the previous proof.

This is a filter for

$\phi'$: $\phi \wedge \bar{\phi}_{x_1 v_1} \wedge \cdots \wedge \bar{\phi}_{x_n v_n}$ by

$\psi'$: $\psi \wedge \bar{\psi}_{x_1 v_1} \wedge \cdots \wedge \bar{\psi}_{x_n v_n}$

where $\bar{\phi}_{xv}$ and $\bar{\psi}_{xv}$ are obtained in the same way as the $\bar{\phi}_a$ and $\bar{\psi}_a$ in the previous proof.

Since $\psi'$ clearly is observable, we only have to show that $\phi'$ characterises $k$ upto $\approx_A$. Because $ch(x) \in E_p$, we have no problems with these events. Furthermore, the ordering of the events of the form $x_v$ is determined by the ordering of the events of the form $ch(x)$, so we only have to show that $\phi'$ is characteristic for the sentence $x_v \in h^C(i)$. So suppose $\phi'$ and $x_v \in k'^C(i)$. Then $ch(x) \in k'^C(i)$ and hence $ch(x) \in k^C(i)$. So there must be some $w$ with $x_w \in k^C(i)$ and, by $\phi'$, $x_w \in k'^C(i)$. This implies that $v = w$, so $x_v \in k^C(i)$. If $\phi'$ and $x_v \in k^C(i)$, then $x_v \in A'$, by definition, and hence $x_v \in k'^C(i)$, by $\phi'$. If $k' \approx_A k$, it is obvious that $\phi'(k)$ holds. The rest of the proof stays the same.

# 6. Discussion

In this chapter we discuss a possible other definition of the semantics with respect to causality between micro-steps.

## 6.1. 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 pre-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 $\lambda$ 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 pre-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. Whether this will yield to a fully abstract model is not sure, since the ordering relation cannot be made observable in the same way as it is done in the proof of this paper, by means of the $F_{ab}$-filter.

## 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 proof introduces the notion of *filter*, a special context that makes one aspect of a history observable. Using filters modularises the proof, since extensions of the language can be handled by adding new filters. This strategy is shown by the example of adding shared variables to the languages.

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. Furthermore, we want to thank Ton Kalker who made an essential suggestion to acquire full abstraction. 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 Edm'e 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., D'ohmen 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 R´echerche 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 Guemic 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. 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 A**

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 $\Sigma$ and a set of labels **Lab** be given.
A **H-statechart** is a quintuple $(S, \rho, \psi, \delta, T)$ 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** $\square$

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

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 $\sigma$, 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 $E_p$:$S \rightarrow$ **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 $U \in$ **Unv**, define

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

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

We need a function $E_i$: $S \rightarrow$ **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 $\rho(s) = \varnothing$

$E(s) =$ **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 $s'$;

$=$ **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) $\psi(s) = \text{AND}$

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) $\psi(s) = \text{OR}$

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. $\psi(x) = \text{OR}$
3. $\forall s \in R$: if $\psi(s) = \text{OR}$ 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 $E_i(s) = B_n$ $\square$

## B Appendix

This appendix contains the proofs of some lemmas in chapter 3.

**Lemma 1 (1)** *For all programs $P \in$* **Stch** *and histories $h \in [\![P]\!]$:*

1. *if $out(h) = \perp$, then there exists $h' \in [\![P]\!]$ with $out(h) \neq \perp$ and $h \leq h'$.*
2. *there exists $h' \in [\![P]\!]$ with $st(h') > st(h)$, $end(h') = end(h) + st(h') - st(h)$, $out(h') = out(h)$, $in(h') = in(h)$, and $\forall st(h') \leq i \leq end(h') : h'(i) = h(i - (st(h') - st(h)))$.*
3. *if $h \upharpoonright A = h' \upharpoonright a$ and $A$ is a set of events that contains all the events occurring in $P$ (including $en(S)$ and $ex(S)$ for states $S$ in $P$), then $h' \in [\![P]\!]$.*

*Proof* By structural induction to $P$.

1. For a primitive chart $P$, there is for every history $h \in [\![P]\!]$ a history $h' \in [\![P]\!]$ that is exactly the same, except that $out(h) = *$. This history suffices.

   For the composite charts, we first prove the following property. For any history $h$ and chart $P$, there is a history $\tilde{h} \in [\![P]\!]$, with $\tilde{h}^S = h^S$ and $st(\tilde{h}) = st(h)$. For the primitive charts, this is guaranteed by the fact that $(T \cup W)^S = \mathbf{H}^S$. For the composite charts, this follows from the nature of the MERGE predicate and the weaker $||$-operator that do not change the $.^S$ components of histories.

   Then, if $out(h) = \perp$ and $MERGE(h_1, h_2, h)$, we know that $out(h_1) = out(h_2) = \perp$ and hence there are $h'_1$, $h'_2$ with $h_i \leq h'_i$ and $out(h_i) \neq \perp$. By the property above, one can always find $\tilde{h}'_1$ and $\tilde{h}'_2$ such that $MERGE(\tilde{h}'_1, \tilde{h}'_2, h')$ and a fortiori $h \leq h'$. The same holds for the $||$-operator.
2. Neither the semantics of primitive charts, nor the semantics of the constructors depend on the particular starting and ending points of the histories.
3. Obvious.

□

### Lemma 4

$F_{ab}$ is a filter for

$$st(h) \leq i < end(h) \wedge (a, b) \notin h^{\leq}(i) \wedge b \in h^C(i) \text{ by } g_{ab} \in h^F(i)$$

*Proof*
Let
$W = \{(\emptyset, C, \leq) \mid b \notin C \vee a \in C\}$
$T = \{(F, C, \leq) \mid b \in C \wedge (a \notin C \vee b < a) \wedge g_{ab} \in C \wedge g_{ab} \in F \wedge b < g_{ab}\}$
Then, if $h \in [\![T_2]\!]$, for every $st(h) \leq i < end(h)$, we know that $h(i) \in W$ or $h(i) \in T$.

Furthermore, for any $h \in \mathbf{H}$, there exists a $h' \in [\![T_2]\!]$ with $h^S \lceil \{\bar{g_{ab}}\} = h'^S \lceil \{\bar{g_{ab}}\}$, because $W \cup T$ covers all possibilities of $C$ and $\leq$, apart from $g_{ab}$.

Hence, if $h_1 \in [\![X]\!]$, there is a $h_2 \in [\![T_2]\!]$ with $h_1^S \lceil \{\bar{g_{ab}}\} = h_2^S \lceil \{\bar{g_{ab}}\}$. Then there is a $\bar{h_1}^S \in [\![\mathbf{Hide}(g_{ab}, X)]\!]$ with $\bar{h_1}^S = h_2^S$ and hence there is $h$ with $MERGE(\bar{h_1}, h_2, h)$.

Now suppose that $(a, b) \notin h_1^{\leq}(i) \wedge b \in h_1^C(i)$ $(\phi(h))$ for some $st(h_1) \leq i < end(h_1)$. Since $h_1^S = h_2^S$, we also have $\phi(h_2)$ and $b \in h_2^C(i)$, so $h_2^C(i) \in T$. This implies that $g_{ab} \in h_2^F(i)$, and because $MERGE(\bar{h_1}, h_2, h)$ holds, $g_{ab} \in h^F(i)$.

For the converse, suppose $\psi(h)$, that is, $g_{ab} \in h^F(i)$. This implies that $g_{ab} \in \bar{h_1}^F(i)$ or $g_{ab} \in h_2^F(i)$. Because $g_{ab}$ is hidden in $X$, So $h_2^C(i) \in T$.

## Lemma 5

$F_{tu}$ is a filter for

$\phi_{tu} : in(h) = t \wedge out(h) = u \wedge st(h) = m \wedge end(h) = n$ by
$\psi_{tu} : g_{st} \in h^F(m) \wedge g_{end} \in h^F(n+1) \wedge end(h) = n+1$
Furthermore, $h \in [\![F_{tu}[P]]\!] \wedge g_{st} \in h^F(m)$ implies $st(h) = 0$.

*Proof*
Assume that $g_{st}$ and $g_{end}$ do not occur in $f$. Let $f \in [\![P]\!]$ and $f = (m, *, \bar{f}, u, n)$ with $m > 0$. Then there is $g_1 \in [\![U_2]\!]$ with

$$g_2 = (m, t_2, \bar{f}, u, n)$$

Clearly, there exists $g_3 \in [\![U_1]\!]$ with $out(g_3) = t_1$, $end(g_3) = m$, $st(g_3) = 0$, and $g_{st} \in g_3^F(m)$. Furthermore, there is $h \in 2^G$ with $g_3^C(i) = h(i)$ and $en(U_1) \in h(0)$ and $en(U_1) \notin h(i)$ for $i > 0$, so $h \in T_E(tm(en(U_1), m))$.

Hence, there is

$$g_4 \in [\![\mathbf{Conc}(U_1, t_1, t_2, \mathbf{Stat}(U_2, \mathbf{Hide}(\{g_{st}, g_{end}\}, P)))]\!]$$

with $\bar{g_4} = \bar{g_2} || \bar{g_3}$, $st(g_4) = 0$, $out(g_4) = *$, $end(g_4) = n$.

Next, define history $g_5$ with $in(g_5) = t_4$, $st(g_5) = n$, $end(g_5) = n+1$, $g_{end} \in g_5^F(n+1)$, and $g_5^S(i) = g_4^S(i)$ for $i \leq n$. Clearly, $g_5 \in [\![U_3]\!]$ and, hence, there is

$$h \in [\![\mathbf{Conc}(U_1, t_1, t_2, \mathbf{Conc}(\mathbf{Stat}(U_2, \mathbf{Hide}(\{g_{st}, g_{end}\}, P), u, t_3, U_3)))]\!]$$

with $\bar{h} = \bar{g_4} || \bar{g_5}$, $st(h) = 0$, so $g_{end} \in h^F(n+1)$ and $g_{st} \in h^F(m)$. □

## Lemma 6

Let the integer $k$, the events $a, b$, and the transitions $t, u$ be given, and let $c, d$, and $e$ be arbitrary events not equal to $g_x$, $g_{xy}$, $g_{st}$, or $g_{end}$. Then

$F_a$ is transformationally transparent for $st(h) = k$, $\bar{\psi_{tu}}$, $\bar{\phi_{cd}}$, $\bar{\phi e}$, $\bar{\psi_e}$, and $\chi_F$ for any events $c$ and $d$ and any event $c \neq a$.

$F_{ab}$ is transformationally transparent for $st(h) = k$, $\bar{\psi_{tu}}$, $\bar{\phi_{cd}}$, $\bar{\phi e}$, $\bar{\psi_e}$, and $\chi_F$ for any events $c$ and any two events $(d, e) \neq (a, b)$.

$F_{tu}$ is transformationally transparent for $c \in h^C(i)$, $c \notin h^C(i)$, $(c,d) \in h^{\leq}(i)$, $(c,d) \notin h^{\leq}(i)$, and $\chi_F$ for any two events $c$ and $d$ and any integer $i$.

*Proof*
Clear from the proofs of the respective filters. The history transformations are such that they do not change the properties stated in the lemma. □

# Chapter 4

# On the semantics of reactive systems

# On the Semantics of Reactive Systems *

C. Huizing R. Gerth

Eindhoven University of Technology †
January 14, 1991

**Abstract**

We explain that real-time reactive systems pose specific problems in defining languages to specify and program them. Three criteria are formulated, responsiveness, modularity, and causality, that are important to have for a high-level specification language for these systems. As new results, we prove that these properties can not be combined in one semantics. Since these properties are mandatory for a structured development of real-time reactive systems, we introduce a two-levelled semantics in which the three properties hold on different levels the semantics: global events are treated more abstractly with respect to time than local events.

## 1 Introduction

There is a fundamental dichotomy in the analysis of computing systems. This dichotomy crosses all borderlines between sequential and parallel systems, central and distributed systems, and between functional and imperative systems. This is the dichotomy between *transformational* and *reactive systems* [HP85]. Transformational systems are well described by a relation between input and output value. They read some input value, then produce, perhaps non-deterministically, an output value and terminate. A reactive system, however, maintains a continuous interaction with its environment. Typically, the environment reacts upon the output of the system and in many cases the system is not expected to terminate.

Reactive systems can be found anywhere: they include digital watches, interactive software systems, integrated circuits, real-time embedded systems. Design, programming and verification of reactive systems is an important challenge, since existing techniques for transformational systems are not satisfactory for this purpose [HP85].

Recently, several formalism for the development of reactive systems have been proposed. We mention Esterel [BG88, BC85], Lustre [BCH85b], and Statecharts [Har87]. In the development of these formalisms, serious problems have been encountered. Apparently, it is not so simple to design a high-level language for reactive systems. The central problem is that all these languages try to combine the following three properties, or criteria, in one formalism. These properties are for the first time formally defined in this paper.

The first property is *responsiveness*, meaning that a system's output comes simultaneously with the input that causes it. This requires an abstract notion of time, since there is always

*This research is partially supported by ESPRIT projects 937 (DESCARTES) and 3096.

†Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O.Box 513, 5600 MB Eindhoven, The Netherlands. Electronic mail: wsinkees@win.tue.nl or wsdckeesh@heitue5.bitnet.

some physical time needed to compute a reaction, ultimately. This property is important for high-level specification where one does not want to bother -yet- with implementation details on the one hand, but on the other hand does want to specify in an accurate, non-fuzzy way. Furthermore, it allows for step-wise refinement, without having to redo the timing over and over again.

The second property, *modularity*, means that all parts of the system should be treated symmetrically. The interface between the environment and the system should be the same as the interface between the parts of the system itself. Furthermore, every part of the system should have the same view of the events occurring in the total system *at any moment*. Consequently, in all the formalisms mentioned above, the communication mechanism between the subsystems is the immediate, asynchronous *broadcast*[1].

The third property, *causality*, means that for any event generated at a particular moment there must be a causal chain of events leading to the action that generated this event. No causal loops may occur and no events may be generated "spontaneously", i.e., without an input event that directly or indirectly caused it. This allows for an intuitive, operational understanding of the system's behaviour.

Unfortunately, these three properties can not be united in one semantics, as we prove in the paper. To prepare the way for this result we classify the semantics of reactive systems currently available — for Esterel [BC85, BG88], for Lustre [BCH85b], for Signal [LBBG85], for Statecharts [HPPSS87a, *i-Logix Inc*89, PS88, HGdR88] — in basically 5 types of semantics, each one trying to improve upon the others, but no one succeeding in a semantics which is satisfactory from the point of view of structured program development. To this end a bare-bones language for reactive systems is introduced, which can be identified with subsets of any of these languages for the purpose of our criteria. To define and enable comparison of our various semantics for this language, a simple formalism is introduced for transition systems with edges labelled by event/action pairs. As already stated, we prove that our criteria cannot be met by any uniform semantics[2].

We know a way out, however. Although very useful, the properties of modularity and causality are applied at different levels of development. Modularity is useful at a relative high level, where too much detailed knowledge of the execution of the subsystems would obstruct a good overview of the system. Causality, however, is useful at the level of operational reasoning, where a local part of the system can be completely understood. Therefore, we introduce the concept of *modules* into the language. A module is a relatively independent part of the system, in essence a reactive system in its own right. *Between* modules, the principle of modularity holds, thus helping to keep a global understanding of the system. *Within* modules, however, the principle of causality holds, making it possible to develop a smaller subsystem in an intuitive, operational way. This leads to a hybrid semantics: local events, which are used only inside a module, interact in a way satisfying the causality principle, whereas global events, which are used between modules, possibly in the whole system, are treated in a modular fashion.

[1]If the travelling time of signals is too high, e.g. widely distributed systems, one has to introduce an explicit delay between the moment that an an event can be generated and the moment it will actually be sensed by the other components. This can be done in the current framework.

[2]This is serious, and it is worth to reflect a moment on its implications, for it concerns a veritable principle of (reactive) distributed computing. After all, structured program development, together with mechanization, promises the only hope for improving future software quality; and the important producers of critical software, s.a Boeing, McDonald-Douglas, Hughes etc., all use specification systems to which our result applies.

The paper is structured as follows. In section 2, we define our formal framework and the language we are going to use to specify reactive systems. Section 3 defines and discusses the definitions of the properties we want a semantics for reactive systems to have. Section 4 gives five proposals of a semantics, illustrating the problems encountered in the attempt to find a semantics that has all the desirable properties. In section 5 we introduce the concept of modules and give a semantics that is modular on one level and causal on the other, using this concept.

## 2 Framework

Let some alphabet of primitive events, $\Pi$, be given. Primitive events can be generated by the environment as an input to the system, or by the processes in the system either as an output to the environment or for interaction with other processes of the system. These are instantaneous signals that cannot be interrupted or undone.

We model the behaviour of a reactive system as an infinite sequence of pairs $(I, O)$, where $I, O \subseteq \Pi$. $I$ is the set of input events, possibly containing timing information such as clock ticks. $O$ is the set of output events, containing all events generated by the system. $I$ and $O$ are not necessarily disjunct. The events in $I$ and $O$ are considered to occur at the same time. Hence, the output events are timed by the events. The amount of detail that is desired can be achieved by providing an regular input event of which the timing is known (e.g., the tick of an external clock). The semantics of a reactive system is the set of all possible behaviours.

We use infinite sequences, since many reactive systems are not expected to terminate. Termination can be modelled by emitting a special output event and after that producing only empty output sets.

If $S$ is a reactive system and $S$ produces output $O$ at the moment that input $I$ is provided, we write

$$S \xRightarrow[I]{O} S'$$

In general, the state of the system has changed after the transition; the system with its new state is denoted by $S'$.

In case of singletons, we sometimes omit the curly braces: $S \xRightarrow[f]{e} S'$

Leaving out something in this notation means existential quantification, e.g., $S \xRightarrow[I]{}$ means $\exists O, S' : S \xRightarrow[I]{O} S'$.

We define the semantics $\mathcal{O}(P_0)$ of a reactive system $P_0$ as the set of all behaviours of the form $(I_1, G_1)(I_2, G_2) \ldots (I_n, G_n) \ldots$ such that there exist systems $P_1, P_2, \ldots, P_n, \ldots$ and for all $i > 0$

$$P_{i-1} \xRightarrow[I_i]{G_i} P_i$$

holds.

Here, we give the language in which we express reactive systems and various proposals for the semantics as they have been circulating.

### 2.1 The language

We consider a very simple derivative of Statecharts in which every reactive system is a composition of flat statecharts, i.e. statecharts without hierarchy of states. Such a flat statechart can be viewed as a transition system where labels associated to transitions have the form "event-expression/action". When the machine is in the source state of such a transition and the event-expression is enabled, the transition will be taken (and if there is no non-determinism present enabling another transition it must be taken) and the action will be performed. This action is the generation of some primitive events. The event-expression is a propositional combination of primitive events, e.g.,

$$(\neg a \wedge b) \vee c\ .$$

A transition labelled with this event-expression becomes enabled either when primitive event $c$ occurs, or when $b$ occurs and $a$ does not occur, or both. Exactly when the action is performed, and exactly what "event $a$ does not occur" means, is different in the various versions of the semantics. A composition of transition systems is executed synchronously: all machines that can perform a transition at some moment, do this simultaneously.

**Definition 1** *A Transition Machine is a triple $(S, s, T)$, where $S$ is a set of states, $s \in S$ is the initial state, $T \subseteq S \times L \times S$ is the set of transitions.*

A *Transition Machine*, or simply machine in the context of this paper, differs from the classical Finite State Machine in as much that it has no final state and that its purpose is not to accept words over its event-alphabet, but to serve as part of a formalism to describe reactive systems, since general reactive systems are combinations of synchronously executing transition machines.

**Definition 2** *A* machine expression *has the syntax*

$$\langle \mathit{machine\ exp} \rangle \quad \Rightarrow \quad [\ \langle \mathit{label} \rangle;\ ]\ [\ \langle \mathit{state} \rangle\ ]\ \langle \mathit{machine} \rangle$$

*where $\langle machine \rangle$ is an Enhanced Finite State Machine, as defined above, and $\langle state \rangle$ is one of its states. $sM$ stands for $M$ with initial state $s$, so $s'(S, s, T) = (S, s', T)$. $l; M$ stands for $M$ prefixed with a transition labelled $l$, so $l; (S, s, T) = (S \cup \{s'\}, s', T \cup \{(s', l, s)\})$, where $s' \notin S$.*

*A* reactive system *is the composition of one or more machine expressions:*

$$\langle \mathit{reactive\ system} \rangle \quad \Rightarrow \quad \langle \mathit{machine\ exp} \rangle \,\|\, \ldots \,\|\, \langle \mathit{machine\ exp} \rangle$$

*If two reactive systems $S_1$ and $S_2$ make a step together we write, of course, $S_1 \,\|\, S_2 \xRightarrow[I]{O} S_1' \,\|\, S_2$. If this step is the combination of two local steps $S_1 \xRightarrow[I_1]{O_1} S_1'$ and $S_2 \xRightarrow[I_1]{O_1} S_2'$, we say that these local steps* combine *into the global step.*

## 3 Criteria

### 3.1 Responsiveness

We can now define the criteria on which we judge the semantics.

As we have seen, in a reactive system it is important to know how much time elapses between an input and the resulting output.

One approach is to specify for each situation a concrete amount of time. This is cumbersome and not in accordance to the level of abstraction we are aiming at. It forces us to quantify time right from the beginning. At this stage one is in most cases only interested in the relative order and the coincidence of events.

Another approach is fixing the reaction time to, say, one time unit (assume we have a discrete time domain). This is simpler, but still not abstract, since specifications using this principle are difficult to refine without changing their high level meaning.

This approach has another disadvantage. In practice, a fixed amount of reaction time will be some kind of upperbound upon the execution times of different statements in different situations in the actual implementation. So the implementation will have to be artificially delayed in order to meet its specification. In many cases, however, we want the reaction to be as quick as possible. The delay of 1 time unit was only introduced for uniformity and the implementation is slower than necessary.

A third approach is to leave things open: only say that execution of a reaction takes some positive amount of time and see at a later stage (closer to the actual implementation) how much time things did take. This is also awkward, however, since it introduces a lot of non-determinism, which will make it difficult or even impossible to prove interesting things at an early stage of the development.

In our framework it is possible to specify that a certain reaction to an input comes simultaneously with that input. Although there is always some physical time needed to do the computing, in many cases this time is much shorter than the rate of the incoming events. In other words, the time scale of the computations is much shorter than the time scale of the environment. Therefore, it makes sense to adopt the *abstraction* that the reactions are immediate on the time scale of the environment. Naturally, it depends on the application and the implementation whether this abstraction is reasonable or not.

This abstraction gives us several important advantages:

(i). The reaction time is accurately known (i.e., 0), even at early stages of the development. This is important, since the relative timing of input and output events is important in reactive systems, as we have seen.

(ii). The reaction time does not depend on the actual implementation.

(iii). The reaction time is as short as possible (namely, 0). No artificial delays have to be introduced at an early stage of development to enforce synchronisation. Later on, when the implementation is better known, these delays could turn out unnecessary.

(iv). The timing behaviour is abstract, allowing further refinement without having to redo the timing. E.g., if a certain reaction is refined to include several sub-reactions, the timing behaviour is not changed, since $0 + 0 = 0$.

We call a semantics in which it is possible to perform instantaneous reactions *responsive*.

In our framework this notion can be formalised as follows.

**Definition 3** *A system $S$ is* **responsive** *of there exist input sets $I_1$ and $I_2$ and output set $O$ such that*

$$S \overset{O}{\underset{I_1}{\Longrightarrow}} \text{ and } S \overset{O}{\underset{I_2}{\not\Longrightarrow}}$$

Responsiveness of a system does not tell you much. It only says that there exist two input sets to which the system reacts differently in the current moment, *i.e. immediately.* A more general property is the following.

**Notation:**
If a reactive system consists of subsystems $S_1, \ldots, S_n$, we denote this by $S_1 \| \ldots \| S_n$.

**Definition 4** *A semantics is* **responsive** *if for any two distinct input sets $I_1$ and $I_2$ and non-empty output set $O$ with $O \cap (I_1 \cup I_2) = \emptyset$, there exists a system $S$ such that*

$$S \xRightarrow[I_1]{O} \text{ and } S \not\xRightarrow[I_2]{O}$$

In other words, any two different inputs can be distinguished and *immediately so.*

## 3.2 Modularity

The first aspect of modularity is the symmetry of interface. The way the system interacts with, e.g., a human being as part of the environment, should be not different from the way it interacts with another reactive system. In other words, if we put two reactive systems together to form a new, bigger one, they see each other's behaviours as sequences of pairs $(I, O)$, exactly as the environment sees them. In particular, the composition of two reactive systems is defined on basis of their observable behaviours: no inner details of the execution can be seen by the other system.

The second aspect of modularity is the uniformity of the view every subsystem has of what is going on. When an event is generated, it is broadcast all around the system and it is immediately available to everyone. Hence, every part of the system has the same view *at any moment.* This simplifies analysis and design considerably. Of course, this is not realistic for widely distributed systems in which it takes a considerable time for a signal to travel between parts of the system. In this case, one has to introduce an explicit delay between the moment that an event can be generated and the moment it will actually be sensed by the other components. We stress, however, that our framework is designed for tightly coupled systems in which a synchronous execution is realistic.

**Definition 5** *A semantics is* **modular** *if for any two systems $S_1$ and $S_2$ the following two statements are equivalent:*

*(i).* $S_1 \| S_2 \xRightarrow[I]{O} S_1' \| S_2'$

*(ii).* $S_i \xRightarrow[I \cup O_{3-i}]{O_i} S_i'$ *for* $i = 1, 2$

*where $O = O_1 \cup O_2$ and the first step is the combination of the last two.*

In this definition we can see the two aspects of modularity. First, the interface between the subsystems is the same as the interface between a system and its environment, i.e., only the sets $I$ and $O$ are taken into account. Second, the output of one system is immediately available as input to the other one.

## 3.3 Causality

The combination of the principles modularity and responsiveness is what Gérard Berry calls the *synchrony hypothesis* [BG88]. This combination leads to several semantic problems. First, one can specify systems for which it is unclear what the semantics of their combination should be. The modular combination of these system can create cycles of reactions in which the reaction nullifies the action that caused it In Esterel [BC85],[BG88],[Gon88] and Lustre [BCH85a] combinations like these are simply forbidden. A check at compile time is performed to rule out programs that (might) give problems. In Statecharts [HPPSS87b] the principle of modularity is sacrificed to achieve a semantics in which every program combination is legal. In the appendix we discuss all these semantic versions.

Although not modular, these versions have the advantage that they are *causal*: for every event that is generated there is a causal chain of events that leads to this event. In the modular-responsive semantics, however, events can occur "out of the blue"[3].

**Definition 6** *We call a semantics* **causal** *if we can add to every step $S \underset{I}{\overset{O}{\Longrightarrow}} S'$ a partial order $\leq$ on $I \cup O$, such that:*

*(i). if $S \underset{I}{\overset{O}{\Longrightarrow}}$ and $S \underset{I'}{\overset{O}{\not\Longrightarrow}}$, and $I, O \neq \emptyset$, then there is at least one dependency between $I$ and $O$, i.e., $\exists a \in I, b \in O$ with $a \leq b$*

*(ii). this ordering respects the composition of systems, i.e., if $S_1 \parallel S_2 \underset{I}{\overset{O}{\Longrightarrow}} S_1' \parallel S_2'$ with causal order $\leq$, then there should exist a partioning into processes $T_1, \ldots, T_n$ and causal orders $\leq_1, \ldots, \leq_n$ such that $\leq \restriction (I_i \cup O_i) = \leq_i$ and $T_1 \parallel \ldots \parallel T_n = S_1 \parallel S_2$, $n \geq 2$, and for each $i$, $T_i \underset{I_i}{\overset{O_i}{\Longrightarrow}} T_i'$, these steps combine into the step of $S_1 \parallel S_2$.*

Here, $\restriction$ denotes the restriction of a relation.

Note that in *(iii)* we do not demand equality of the restricted relation and the local relation, since $I_1$ and $I_2$ may overlap and two unrelated events in one set may be related in the other one.

**Theorem 1** *No semantics of reactive systems can be responsive, modular and causal at the same time.*

*Proof:*
Suppose the contrary. Take events $a, b$ with $a \neq b$. Then, by responsiveness, there must be $S_1$ and $S_2$ with

$S_1 \underset{a}{\overset{b}{\Longrightarrow}} S_1'$ and $S_1 \underset{\emptyset}{\overset{b}{\not\Longrightarrow}}$ and

$S_2 \underset{b}{\overset{a}{\Longrightarrow}} S_2'$ and $S_2 \underset{\emptyset}{\overset{a}{\not\Longrightarrow}}$.

If causality holds, there must be partial orders $\leq_1$ and $\leq_2$, such that $a \leq_1 b$ and $b \leq_2 a$, since there is a dependency between $\{a\}$ and $\{b\}$, resp. $\{b\}$ and $\{a\}$.

[3]Unless programs in which this may occur are ruled out, as is done in Esterel and Lustre.

By modularity, we must have

$$S_1 \| S_2 \xrightarrow[\emptyset]{\{a,b\}} S_1' \| S_2'$$

and for this step no causal order exists that respects $\leq_1$ and $\leq_2$, since $a \neq b$. □

## 4 Semantics

We now describe five versions of semantics for reactive systems in the framework described above.

**A** The events generated as a reaction to some input can only be sensed in the step following the input. The main drawback of this solution is that it is not possible to express simultaneity of action and reaction. Specifying a chain of reactions and independently the reaction time becomes cumbersome, because every element in the chain adds one step to the reaction time. This semantics is not *responsive*.

**B** In order to make the semantics responsive, the notion of *micro-step* was introduced. Every observable step is divided into an arbitrary number of micro-steps. Action and reaction strictly follow each other in micro-steps, but observably take place simultaneously. A detailed treatment of this semantics, applied to full Statecharts, can be found in [HPPSS87b] (operational model) and [HG89] (fully abstract model). The problem with this semantics is that it introduces a lot of non-determinism: if you take the micro-steps in a different order, you may get a different observable result. This semantics turned out to be too subtle and non-deterministic to be of practical use.

**C** Semantics C overcomes this problem by demanding global consistency of every micro-step. This means that a reaction of the system should not only be enabled by the events generated in previous micro-steps, but also be enabled by the set of events generated in the full macro-step. A full description of this semantics can be found in [PS88]; [HR88] gives an axiomatisation of Statecharts based on this semantics. Semantics C does not fully solve the problem of *modularity*, i.e., the behaviour of a process cannot be explained only in terms of macro-steps. This implies that a modular development of the system is cumbersome, since every developer has to know the detailed micro-behaviour of the other processes.

**D** In semantics D all events that are generated during some macro-step are considered as if they were present right from the start of the step, no matter at which particular micro-step they were generated. As a consequence, the macro-behaviour of a process suffices to describe its interaction with other processes. The advantage of semantics C over D, however, is that the first respects *causality*: each reaction can be traced back to the input from the environment via chain of reactions each causing the next one. In semantics D, however, it is possible that reactions trigger themselves.

**E** The current implementation of Statecharts models this fifth version of the semantics. This is an "acceleration" of semantics A. Events are generated at the next step, but before the reaction of the system has completely died out, no input from the environment is possible. This semantics is heavily non-modular, since one macro-step may contain several steps of the type of semantics A. Events remain active only for the duration of such a step, hence, in one macro-step an event can be activated and de-activated several times, thus leading to a much more complex interface between subsystems than between the system and the environment.

The following picture shows how each version of the semantics is an attempt to improve on another one.

$$\begin{tikzcd}
A \arrow[r, "\textit{responsiveness}"] \arrow[d, "\textit{responsiveness}"] & B \arrow[r] & C \arrow[r, "\textit{modularity}"] & D \arrow[l, bend left=60, "\textit{causality}"] \\
E \arrow[u, bend left=60, "\textit{modularity}"] & & &
\end{tikzcd}$$

## 4.1 Micro-semantics

The second transition relation is a labelled transition relation in the style of Plotkin [Plo81] reflecting the transformation of a configuration in one micro-step. A configuration is a pair $(P, v)$, where

$P$ is the system in its current state

$v$ is the set of machines (processes) that have already finished the current macro-step

We denote that machine $M$ has finished its macro-step by $\bar{M}$. E.g., we may write $M_1 \,||\, \bar{M}_2 \,||\, M_3$ instead of $M_1 \,||\, M_2 \,||\, M_3, \{M_2\}$ and $M_1 \,||\, M_2 \,||\, M_3$ instead of $M_1 \,||\, M_2 \,||\, M_3, \emptyset$ . We need this information, since in general a machine can only perform a limited amount of computation in one macro step; in many cases this is exactly one transition. Hence, we have to know whether a machine is still allowed to perform transitions, or whether it has completed its current macro-step.

## 4.2 Further definitions

Event-expressions are propositional formulae with primitive events as atomic propositions. The following definition tells us when a set of primitive events enables an event-expression.

**Definition 7** . *Let* $e \in \Pi$ *and* $I \subseteq \Pi$. *Then*

$I \models e$ *iff* $e \in I$

$I \models e_1 \wedge e_2$ *iff* $I \models e_1$ *and* $I \models e_2$

$I \models e_1 \vee e_2$ *iff* $I \models e_1$ *or* $I \models e_2$

$I \models \neg e$ *iff* $I \not\models e$.

We define $gen(a)$ as the set of events generated by the action $a$. So

$gen(e) = \{e\}$ iff $e \in \Pi$

$gen(e1, e2) = gen(e1) \cup gen(e2)$.

The *null machine* N is the Transition Machine with only one node and no transitions. $P \not\rightarrow$ means: there exist no $P'$, $I$ and $O$ such that $P \xrightarrow[I]{O} P'$.

## 4.3 Structure of the step relation

The definition of the step relation[4] has the following structure.

(i). There are one or more **transition axioms**, that define how a transition in one of the machines is taken.

(ii). There is one universal rule for parallel composition of machines. This is the following rule:

$$\textbf{PAR}\quad \frac{C_1 \xrightarrow[I]{O} C_1'}{C_1 \,\|\, C_2 \xrightarrow[I]{O} C_1' \,\|\, C_2 \quad C_2 \,\|\, C_1 \xrightarrow[I]{O} C_2 \,\|\, C_1'}$$

(iii). There is a **macro-step rule** with which one derives an observable step from a sequence of micro-steps.

If a global step uses the same micro-steps as several local steps, then these local steps combine into the global step.

## 4.4 Semantics A

The transition relations for A are defined by the following axioms and rules.

**A1.1** $sM \xrightarrow[I]{\emptyset} /a; s'\bar{M}$ if $(s, e/a, s')$ is a transition in $M$ and $I \models e$;

**A1.2** $/a; M \xrightarrow[I]{O} M$ where $O = gen(a)$;

$$\textbf{A2}\quad \frac{S_0, \emptyset \xrightarrow[I \cup O]{O_1} S_1, v_1 \xrightarrow[I \cup O]{O_2} \cdots \xrightarrow[I \cup O]{O_n} S_n, v_n \not\xrightarrow[I \cup O]{}}{S_0 \overset{O}{\underset{I}{\Longrightarrow}} S_n}, \text{ where } O = O_1 \cup \ldots \cup O_n.$$

In semantics A, the events generated as a result of taking a transition become available only in the next macro-step. This means that after execution of a transition label *event/action*, control is left just before the */action*-part, which remains for the next macro-step. Axiom 1.2 deals with performing this action. After applying this axiom the machine has not completed its macro-step yet: it may still take a transition.

Rule 2 gives the completion of a macro-step and relates the two transition relations. A macro-step can be made of any maximal sequence of micro-steps.

[4]Although, traditionally, this relation is called a *transition relation*, we want to reserve this term for the relation defining the computation steps in the Finite State Machines.

Although the micro-step relation seems to be not efficitive, since the output of later steps is used as input set for a step, this is not the case. The sequence of micro-steps can be rearranged in such a way that all output generating steps are at the beginning. Since these steps are not dependent on the input, no "lookahead" is needed and the computation is effective. After these steps, no output is generated and the set $I \cup O$ is fixed.

**Lemma 1** *Semantics A is order-independent, i.e., if*

$$S_0, \emptyset \xrightarrow[I]{O_1} \ldots \xrightarrow[I]{O_n} S_n, v_n \not\xrightarrow[I]{}$$

*and $\pi$ is a permutation of $[1, \ldots, n]$, then there are $S'_1, v'_1, \ldots, S'_{n-1}, v'_{n-1}$ with*

$$S_0, \emptyset \xrightarrow[I]{O_{\pi(1)}} \ldots \xrightarrow[I]{O_{\pi(n)}} S'_n, v'_n \not\xrightarrow[I]{}$$

*Proof.* Let $i$ range over $1 \ldots m$ and let

$$\Vert_i M_i, v \xrightarrow[I]{O'} \Vert_i M'_i, v' \xrightarrow[I]{O''} \Vert_i M''_i, v''$$

be two consecutive steps in the derivation of $S_0$. Then, by the nature of the transition system, there must be machines $M_j$ and $M_k$ that did the actual step in the two steps above: $M_i \neq M'_i$ for $i \neq j$ and $M_j \xrightarrow[I]{O'} M'_j$ and for $M_k$ likewise: $M'_i = M''_i$ for $i \neq k$ and $M'_k \xrightarrow[I]{O''} M''_k$. These steps must have been derived either from axiom A1.1 or from A1.2. Now, by repetitive application of rule **PAR**, we can derive that (assume $j < k$)

$$\Vert_i M_i, v \xrightarrow[I]{O''} M'_1 \Vert \ldots \Vert M'_{k-1} \Vert M''_k \Vert M'_{k+1} \Vert \ldots \Vert M'_n, v'' \setminus (v' \setminus v)$$

and

$$M'_1 \Vert \ldots \Vert M'_{k-1} \Vert M''_k \Vert M'_{k+1} \Vert \ldots \Vert M'_n, v'' \setminus (v' \setminus v) \xrightarrow[I]{O'} \Vert_i M''_i, v''$$

Hence, by repetitively exchanging neighbours in the micro-step sequence, we can achieve the desired permutation. □

A consequence of this lemma is that the output of a macro-step does not depend on the input, since all output generating steps, which are input independent, can be put at the beginning of the sequence of micro-steps.

**Corollary 1** *Semantics A is not responsive.*

**Lemma 2** *Semantics A is modular.*

*Proof*
*(ii) ⇒ (i)*
Let $S \xRightarrow[I \cup O_2]{O_1} S'$ and $S \xRightarrow[I \cup O_2]{O_1} S'$. Since rule A2 must have been applied to achieve these relations, there exist an $n$ and $S_1, \ldots, S_{n-1}, v_1, \ldots, v_n, O_1^1, \ldots, O_1^{n-1}$ such that

$$S, \emptyset \xrightarrow[I \cup O_2 \cup O_1]{O_1^1} \ldots \xrightarrow[I \cup O_2 \cup O_1]{O_1^{n-1}} S', v_n \not\xrightarrow[I \cup O_2 \cup O_1]{}$$

Likewise for $T$ there exist an $m$ and $T_1, \dots, T_{m-1}, v_1, \dots, v_m, O_2^1, \dots, O_2^{m-1}$ such that

$$T, \emptyset \xrightarrow[I \cup O_1 \cup O_2]{O_2^1} \cdots \xrightarrow[I \cup O_1 \cup O_2]{O_2^{m-1}} T', v_m \not\xrightarrow[I \cup O_1 \cup O_2]{}$$

By repetitive application of rule **PAR**, these two sequences can be merged:

$$S \,\|\, T, \emptyset \xrightarrow[I \cup O_2 \cup O_1]{O_1^1} \cdots \xrightarrow[I \cup O_2 \cup O_1]{O_1^{n-1}} S' \,\|\, T, v \xrightarrow[I \cup O_1 \cup O_2]{O_2^1} \cdots \xrightarrow[I \cup O_1 \cup O_2]{O_2^{m-1}} S' \,\|\, T', v_m \not\xrightarrow[I \cup O_1 \cup O_2]{}$$

and hence, by rule A2,

$$S \,\|\, T \xrightarrow[I]{O_1 \cup O_2} S' \,\|\, T'$$

*(i)* ⇒ *(ii)*

Suppose $S \,\|\, T \overset{O}{\underset{I}{\Longrightarrow}} S' \,\|\, T'$. Then there exist $n$ and $S_0, \dots, S_n, v_1, \dots, v_n, T_0, \dots, T_n$, $O^1, \dots, O^n$ with $S_0 = S, T_0 = T, S_n = S', T_n = T', v_0 = \emptyset, O^1 \cup \cup O^n = O$, and

$$S_0 \,\|\, T_0, v_0 \xrightarrow[I \cup O_1 \cup O_2]{O^1} \xrightarrow[I \cup O_1 \cup O_2]{O^n} S_n \,\|\, T_n, v_n \not\xrightarrow[I \cup O_1 \cup O_2]{}$$

By the nature of the transition relation, we know that in all these micro-steps either $S_i$ or $T_i$ took the real step, i.e., appeared in the premise of rule **PAR**. So there is a partition of $\{1, \dots, n\}$ into $J_1$ and $J_2$ such that the steps of $S$ have index in $J_1$ and the steps of $T$ have index in $J_2$. So,

$$i \in J_1 \Rightarrow S_{i-1}, v'_{i-1} \xrightarrow[I \cup O_1 \cup O_2]{O^i} S_i, v'_i \wedge T_{i-1} = T_i$$

(where $v'_j$ is the restrivtion of $v_j$ to the processes of $S$).

Furthermore, $S_{\max(J_1)} = S_{\max(J_1)+1} = \dots = S_n$, since $S$ makes no move after the one indexed with the last element in $J_1$. And since $S_n \,\|\, T_n, v_n \not\xrightarrow[I \cup O_1 \cup O_2]{}$, we also have $S_{\max(J_1)}, v_{\max(J_1)} \not\xrightarrow[I \cup O_1 \cup O_2]{}$ and thus we have established the premise of rule A2 and we can conclude

$$S \overset{O_1}{\Longrightarrow} S'$$

where $O_1 = \bigcup_{i \in J_1} O^i$ and likewise for $T$ with $O_2 = \bigcup_{i \in J_2} O^i$. Hence, $O_1 \cup O_2 = O$. □

One can easily see that semantics A is causal: since the output generating steps are not dependent on the input, one can use the identity relation as the causality relation for any step.

## 4.5 Semantics B

In semantics B, events are sensed in the same macro-step in which the transition takes place that generates them, but only from the next micro-step onwards.

**B1** $sM \xrightarrow[I]{O} s'\bar{M}$ if $(s, e/a, s')$ is a transition in $M$ and $O = gen(a)$.

**B2** $$\frac{S_0, \xrightarrow[I]{O_1} S_1, v_1 \xrightarrow[I \cup \bar{O}_1]{O_2} \cdots \xrightarrow[I \cup \bar{O}_{n-1}]{O_n} S_n, v_n \not\xrightarrow[I \cup \bar{O}_n]{}}{S_0 \overset{\bar{O}_n}{\underset{I}{\Longrightarrow}} S_n}$$, where $\bar{O}_i = O_1 \cup \ldots \cup O_i$.

Note that the sequence of input sets is an ascending chain: $I \cup \bar{O}_i \subseteq I \cup \bar{O}_{i+1}$ for any $i < n$.

It is easy to see that semantics B is responsive. Let $I_1$ and $I_2$ be given and $I_1 \neq I_2$. Suppose $e \in I_1 \setminus I_2$ (assume without loss of generality that $I_1 \setminus I_2 \neq \emptyset$), and let $a \notin I_1 \cup I_2$. Then $e/aM \xrightarrow[I_1]{a} \bar{M}$, whereas $e/aM \not\xrightarrow[I_2]{}$. Consequently, $e/aM \overset{a}{\underset{I_1}{\Longrightarrow}} M$ and $e/aM \overset{\emptyset}{\underset{I_2}{\Longrightarrow}} M$.

It is easy to find a counterexample for the modularity of semantics B. Take $a/bM_1$ and $b/aM_2$. Then $a/bM_1 \overset{b}{\underset{a}{\Longrightarrow}} M_1$ and $b/aM_2 \overset{a}{\underset{b}{\Longrightarrow}} M_2$. In contradiction to modualrity, however, $a/bM_1 \| b/aM_2 \overset{\{a,b\}}{\underset{\emptyset}{\Longrightarrow}}$ does not hold. One can only derive $a/bM_1 \| b/aM_2 \overset{\{a,b\}}{\underset{a}{\Longrightarrow}}$ or $a/bM_1 \| b/aM_2 \overset{\{a,b\}}{\underset{a}{\Longrightarrow}}$.

**Lemma 3** *Semantics B satisfies causality.*

*Proof:* If $S_0 \overset{O}{\underset{I}{\Longrightarrow}} S_n'$ is a valid step, and $S_0, \emptyset \xrightarrow[I]{O_1} \cdots \xrightarrow[I \cup \bar{O}_{n-1}]{O_n} S_n, v_n$ is the sequence of micro-steps from the premise of rule 2, define the causal order for this step as follows:

$$a \preceq b \text{ if there is a micro-step } i \text{ in which } a \in I \cup \bar{O}_{i-1} \text{ and } b \in O_i \setminus (I \cup \bar{O}_{i-1}).$$

and take the reflexive closure. We now show that this relation is a partial order. Let in the following $I_i$ be the input set of micro-step $i$, i.e., $I \cup \bar{O}_{i-1}$.

(i). $\preceq$ is transitive. Suppose $a \preceq b$ and $b \preceq c$. Then there are micro-steps $i$ and $j$ with

$$\begin{array}{lcl} a \in I_i & \text{and} & b \in O_i \setminus I_i \\ b \in I_j & \text{and} & c \in O_j \setminus I_j \end{array}$$

Now, $i \leq j$, since otherwise $I_j \subseteq I_{i-1} \subseteq I_i$ and hence, $b \in I_i$, which is not the case. Hence, $I_i \subseteq I_j$ and $a \in I_j$, which implies $a \preceq c$.

(ii). $\preceq$ is anti-symmetric. Suppose $a \preceq b$ and $b \preceq a$. Assume $a \neq b$, then there must be micro-steps $i$ and $j$ with

$$\begin{array}{lcl} a \in I_i & \text{and} & b \in O_i \setminus I_i \\ b \in I_j & \text{and} & a \in O_j \setminus I_j \end{array}$$

Now, if $i \leq j$, then $I_i \subseteq I_j$ and $a \in I_j$, so $a \notin O_j \setminus I_j$, which is not the case. A symmetric argument for $b$ applies if $j \leq i$. In both cases we derived a contradiction, hence $a = b$.

We now have to check that $\preceq$ satisfies the properties of a causal ordering relation.

(i). Suppose $S \overset{O}{\underset{I_1}{\Longrightarrow}} S'$ and $S \not\overset{}{\underset{I_2}{\Longrightarrow}}$ and $I_1, O \neq \emptyset$. Then there must exist a micro-step $S_{i-1}, v_{i-1} \xrightarrow[I \cup \bar{O}_{i-1}]{O_i} S_i, v_i$ with $b \in O_i$. Hence, $a \preceq b$.

(ii). Suppose $S_1 \| S_2 \xrightarrow[I]{O} S_1' \| S_2'$ with causal order $\preceq$. Choose the maximal decomposition, i.e., $M_1 \| \ldots \| M_n$ with associated causal orderings $\preceq_1, \ldots, \preceq_n$. By the nature of the micro-step relation, there can be at most one micro-step in the sequence that led to the macro-step in which $M_i$ makes a step.

Assume that $M_i$ made a step at $j$. So, $M_i \xrightarrow[I\cup\bar{O}_{j-1}]{O_j} \bar{M}_i'$ and hence, $M_i \xRightarrow[I\cup\bar{O}_{j-1}]{O_j} M_i'$.
Now, if $a \preceq b$ on behalf of this step, we also have $a \preceq_i b$ and vice versa.

If $M_i$ did not make a step, we have $M_i \not\xrightarrow[I\cup O]{}$ and hence, $M_i \not\xRightarrow[I\cup O]{}$ and no oredering relations are induced from this.

## 4.6 Semantics C

Like in semantics B, events are available in the same macro-step as the transition, but an additional consistency constraint is made: every transition must be enabled by the complete set of events that is available after all output has been generated; i.e., consistency must be maintained.

**C1** = **B1**

**C2** $$\frac{\begin{array}{c} S_0, \xrightarrow[I]{O_1} S_1, v_1 \xrightarrow[I\cup\bar{O}_1]{O_2} \dots \xrightarrow[I\cup\bar{O}_{n-1}]{O_n} S_n, v_n \not\xrightarrow[I\cup\bar{O}_n]{} \\ S_0, \emptyset \xrightarrow[I\cup\bar{O}_n]{O_1} S_1, v_1 \xrightarrow[I\cup\bar{O}_n]{O_2} \dots \xrightarrow[I\cup\bar{O}_n]{O_n} S_n, v_n \not\xrightarrow[I\cup\bar{O}_n]{} \end{array}}{S_0 \xRightarrow[I]{\bar{O}_n} S_n}, \text{ where } \bar{O}_i = O_1 \cup \ldots \cup O_i.$$

Semantics C is responsive. The same construction as is used for semantics B can be applied here.

Semantics C is causal, since in the derivation of an arbitrary macro-step, the premise of rule C2 implies the premise of rule B2 and the same argument can be used here.

By theorem 1, semantics C is not modular, since it is responsive and causal.

## 4.7 Semantics D

Like in semantics B and C, events are generated in the same step as the transition, but, here, all transitions are triggered by the same set of events, viz. the complete set of events after the generation of all ouput.

**D1** = **B1**

**D2** $$\frac{S_0, \emptyset \xrightarrow[I\cup O]{O_1} S_1, v_1 \xrightarrow[I\cup O]{O_2} \dots \xrightarrow[I\cup O]{O_n} S_n, v_n \not\xrightarrow[I\cup O]{}}{S_0 \xRightarrow[I]{O} S_n}, \text{ where } O = O_1 \cup \ldots \cup O_n.$$

The difference between C and D is that in semantics D, transitions can cause their own trigger. For example,

$$a/aM \xRightarrow[\emptyset]{\{a\}} M \quad \text{or} \quad a/bM_1 \,\|\, b/aN_2 \xRightarrow[\emptyset]{\{a,b\}} M_1 \,\|\, N_2$$

In semantics C, this is not possible. Although triggers are evaluated in a global set of events, there is an additional constraint that the events generated so far must also enable the transition. This leads to two premises in rule 2: both the complete set, $I \cup \bar{O}_n$, and the events that are currently available, $I \cup \bar{O}_i$, must be capable of triggering step $i$.

Semantics D is responsive. Use the same construction as for semantics B.

Semantics D is modular. The proof of modularity of semantics A uses only rule 2 and rule **PAR**. Since these rules are the same for semantics A and D, the same proof can be used here.

Because semantics D is already responsive and causal, it cannot satisfy modularity, due to Theorem 1.

In both semantics C and D, it is not always the case that there exists a macro-step for a given input. Take for instance $\neg a/aM$. Then

$$\neg a/aM \xrightarrow[I]{a} \bar{M} \text{ for any } I \not\ni a$$
$$\neg a/aM \not\xrightarrow[I]{} \text{ for any } I \ni a$$

Consequently, a premise of rule C2 and D2 that would yield the step $\neg a/aM \xRightarrow[O]{I} S$ for some $O$ and $S$ does not exist.

Another example is $\neg a/bM_1 \,\|\, b/aM_2$. This example also shows that two well-behaved systems can lead to problems when put together.

## 4.8 Semantics E

This semantics is basically semantics A, but now a sequence of macro-steps that cannot be extended without new input from the environment is squeezed into one macro-step. The correspondence to semantics B is that events are generated in the micro-step following the step in which the transition is taken, but they remain only for the sequence of micro-steps that are immediately caused by the transition.

**E1.1** = **A1.1**

**E1.2** = **A1.2**

$$\textbf{E2}\quad \frac{P_0,\emptyset \xrightarrow[I]{O_1}{}^{*} P_1, v_1 \not\xrightarrow[I]{} \;\ldots\; P_{n-1},\emptyset \xrightarrow[I\cup O_{n-1}]{O_n}{}^{*} P_n, v_n \not\xrightarrow[I\cup O_{n-1}]{}}{P_0 \xRightarrow[I]{O_1\cup\ldots\cup O_n} P_n}$$

Here, $P, v \xrightarrow[I]{O}{}^{*} P', v'$ abbreviates

$$\exists P_1 \ldots P_{n-1}\, O_1 \ldots O_n : P, v \xrightarrow[I]{O_1} P_1, v_1 \xrightarrow[I]{O_1} \ldots \xrightarrow[I]{O_{n-1}} P', v' \text{ and } O = O_1 \cup \ldots \cup O_n$$

With the same construction as used for the other semantics, we can prove that semantics E is responsive. Here, we take the null machine, prefixed with a transition, to avoid unwanted transitions. First we have $e/a\mathrm{N} \xrightarrow[I_1]{a} /a\bar{\mathrm{N}}$. Because the step is finished, we have $/a\bar{\mathrm{N}} \not\xrightarrow[I_1]{}$. Then we have $/a\mathrm{N} \xrightarrow[I_1]{a} \mathrm{N}$. Since N is the null machine, we can not move any furhter: $\mathrm{N} \not\xrightarrow[I_1]{}$ and applying rule E2 with $n = 2$ we get $e/a\mathrm{N} \xRightarrow[I_1]{a} \mathrm{N}$. On the other hand, $e/a\mathrm{N} \not\xrightarrow[I_2]{}$ and so $e/a\mathrm{N} \not\xrightarrow[I_2]{}$ (apply rule E2 with $n = 1$).

Semantics E is not causal, since it is possible to have an event that is several times "on" and "off" during one macro-step. E.g., $M_1 \| M_2$ where

$$\begin{aligned} M_1 &= a/bM_1' \\ M_1' &= c/aM_1'' \\ M_2 &= b/cM_2' \end{aligned}$$

Then

$$\begin{array}{l} M_1 \| M_2 \xrightarrow[a]{\emptyset} /b\bar{M}_1' \| M_2 \not\xrightarrow[a]{} \\ /bM_1' \| M_2 \xrightarrow[\emptyset]{b} M_1' \| /cM_2' \xrightarrow[b]{\emptyset} \\ M_1' \| /cM_2' \not\xrightarrow[b]{} \\ M_1' \| /cM_2' \xrightarrow[\emptyset]{c} M_1' \| M_2' \xrightarrow[c]{\emptyset} \\ /a\bar{M}_1'' \| M_2' \not\xrightarrow[b]{} \\ /aM_1'' \| M_2' \xrightarrow[b]{a} M_1'' \| M_2' \not\xrightarrow[a]{} \end{array}$$

Like in semantics C and D, there are systems in semantics E that can not react properly to some inputs. Take for instance

$$M = (\{s\}, s, \{(s, a/a, s)\})$$

Then

$$\begin{array}{c} M \xrightarrow[a]{\emptyset} /a\bar{M} \not\xrightarrow[a]{} \\ /aM \xrightarrow[a]{a} M \xrightarrow[a]{\emptyset} /a\bar{M} \not\xrightarrow[a]{} \\ \vdots \end{array}$$

**Theorem 2** *The following table gives an overview of the properties of the various semantics.*

| semantics | responsiveness | modularity | causality |
|---|---|---|---|
| A | no | yes | yes |
| B | yes | no | yes |
| C | yes | no | yes |
| D | yes | yes | no |
| E | yes | no | no |

## 5 Hybrid semantics

From theorem 2 we see that none of the five semantics satisfy all the criteria. Therefore, we propose a new version of the semantics in which modularity and causality are applied at different levels. We introduce the notion of *modules* and local events into the language. Modules are clusters of one or more subsystems that are closely connected. The idea is that they can easily be overviewed and hence the criterium of causality is applicable on the events that are local to a module. The events that are visible *between* modules, however, are treated in a modular fashion, since the interface between modules should be simple and transparent.

We change the syntax as follows.

$$\begin{aligned} \langle \mathit{reactive\ system} \rangle &\Longrightarrow \langle \mathit{module} \rangle\ [\ \| \langle \mathit{reactive\ system} \rangle\ ] \\ \langle \mathit{module} \rangle &\Longrightarrow \mathbf{mod}\ \langle \mathit{locals} \rangle\ \mathbf{in}\ \langle \mathit{machine\ exp} \rangle \,\|\ldots\|\, \langle \mathit{machine\ exp} \rangle \end{aligned}$$

where $\langle \mathit{locals} \rangle$ denotes a subset of $\Pi$.

We have to change the micro-semantics in such a way that the label now records the output of the micro-step. This does not change the relation $\longrightarrow$ as restricted to configurations, since the label $E$ is not used in the definition of semantics B. This leads to the following definitions.

**M1** $s\bar{M} \xrightarrow[I]{O} s'M$ if $(s, e/a, s')$ is a transition in $M$; $I \models e$ and $O = gen(a)$.

**M2** $$\frac{P_0; \emptyset \xrightarrow[I \cup O^M]{O_1} P_1, v_1 \xrightarrow[I \cup O^M \cup \bar{O}_1]{O_2} \ldots \xrightarrow[I \cup O^M \cup \bar{O}_{n-1}]{O_n} P_n, v_n \not\xrightarrow[I \cup O^M \cup \bar{O}_n]{}}{P_0 \overset{O^M}{\underset{I}{\Longrightarrow}} P_n},$$

where $O^M = (O_1 \cup \ldots \cup O_n) \setminus H$.

The events that are local to the module (denoted by the set $H$), are removed from the input of the micro-steps and from the output of the macro-step. Any event that is left must be global and must be treated as if it was available right from the start of the macro-step, in order to satisfy modularity. Therefore, the output of the micro-steps is added to the initial input.

## References

[BC85] B. Berry and L. Cosserat. The synchronous programming language Esterel and its mathematical semantics. In *Proceedings CMU Seminar on Concurrency*, pages 389–449. LNCS 197, Springer-Verlag, 1985.

[BCH85a] J.-L. Bergerand, P. Caspi, and N. Halbwachs. Outline of a real-time data flow language. In *Proceedings IEEE Real-Time Systems Symposium*, 1985.

[BCH85b] J.-L. Bergerand, P. Caspi, and N. Halbwachs. Outline of a real-time dataflow language. In *Proc. IEEE-CS Real-Time systems Symposium, San Diego*, 1985.

[BG88] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. Technical report, Ecole Nationale Supérieure des Mines de Paris, 1988. Technical Report.

[Gon88] G. Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones; Application à ESTEREL.* PhD thesis, University of Orsay, 1988.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HG89] C. Huizing and R. Gerth. On the semantics of reactive systems. Technical report, Eindhoven University of Technology, 1989.

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

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. NATO, ASI-13, Springer-Verlag, 1985.

[HPPSS87a] D. Harel, A. Pnueli, J. Pruzan-Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proceedings Symposium on Logic in Computer Science*, pages 54–64, 1987.

[HPPSS87b] D. Harel, A. Pnueli, J. Pruzan-Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proceedings Symposium on Logic in Computer Science*, pages 54–64, 1987.

[HR88] J. Hooman and S. Ramesh. Statecharts assertional framework. Computing Science Note CSN 88/14, Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, May 1988.

[*i-Logix Inc*89] *i-Logix Inc. The Semantics of* STATECHARTS, 1989. *in* Documentation for the STATEMATE System.

[LBBG85] Guernic P. Le, A. Benveniste, P. Bournai, and T. Gonthier. Signal: A data flow oriented language for signal processing. Technical Report IRISA Report 246, IRISA, Rennes, France, 1985.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical report, 1981. Lecture Notes.

[PS88] A. Pnueli and M. Shalev. What is in a step. Technical report, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 1988. Draft.

# Chapter 5

# Formalisms related to Statecharts

# Formalisms related to Statecharts

C. Huizing

August, 1990

Since this thesis discusses mainly Statecharts as a language for reactive systems, we discuss several related formalisms now briefly.

## 1 Argos

An interesting dialect of Statecharts is *Argos* [Mar87] [Mar89]. It is derived from Statecharts, but makes some different decisions and is more restricted in many ways.

The most interesting restriction is that so-called *inter-level* transitions are not allowed. These are transitions that cross borderlines of states, i.e., transitions between states that are not immediate descendants of the same parent state (see fig. 1). In Statecharts, one can avoid this kind of transitions by using the in(...) predicate in the condition of the label of the transition (see fig. 2). Argos, however, has the philosophy that parts of the specification should only communicate by means of events, just like other reactive systems. States are fully internal objects and should not be referred to by other components.

Instead, Argos has another mechanism to transport information up the hierarchy. In contrast to Statecharts, the computation that takes place inside a state is not preempted when the state is left, but the current time step is completed. Hence, in fig. 3 transitions $t_1$ and $t_2$ are not in conflict, but can be taken at the same moment, e.g., when events $a$ and $b$ are present and, consequently, $c$ will be generated. In Statecharts, however, only one of $t_1$ and $t_2$ can be taken at the same time, and since $t_2$ has priority, this one will be taken and event $c$ will not be generated.

With this so-called *non-preemptive interrupt*, one can transport information from substates to their parent state or other ancestor states by means of events. A very common example of this is when an error exit has to be performed *on behalf of* an inner component. Figure 4 shows how this is expressed in Argos: When transition $t_1$ is taken, the event **error** is generated and *at the same time* transition $t_2$ is triggered. As a result, the system goes directly from state S to Error handler. figures 5 to 8 show several ways to express the same behaviour in Statecharts. Note that figure 8 depends on a special version of the semantics in which generated events are not available until the next step (version E in "On the semantics of reactive systems".

The important advantage of the Argos specification is that it uses the graphical notion of refinement better. The high-level specification, without the inner components of state $A$, is a correct specification on its own, without any reference to objects that are to be implemented yet.

Another important difference between Statecharts and Argos is that Argos, like Esterel, does not allow any non-determinism. All programs that could lead to causal paradoxes are illegal, also the programs of the type of figure 9. The compiler checks for this behaviour and this is implemented in such a way that all non-deterministic programs are also rejected. We do not want to argue that this is an undesirable side-effect or a design decision, we just want to mention it here. It is also interesting to mention that in Argos this check is performed at the composition of sub-charts, not in one global check as the Esterel compiler does it, but at every application of the operators orthogonal composition and refinement (the Argos version of statification, an operator that puts one or more sub-charts inside a super-state).

## 2 ATP

Not only programming languages have been designed for reactive systems, also process algebras are being defined for this purpose. We mention the timed failure model [GB87], which is partly based on upon DNP-R, the language used in Chapter 3 of this thesis. It has the same restriction that execution of a statement always takes some positive amount of time. The same holds for ACP$\rho$ [BB90]. As we point out in Chapter 2, reactive systems need a more liberal treatment of timing, which is provided by ATP (Algebra of Timed Processes) [NRSV90]. Any amount of asynchronous computation can be performed between the synchronous ticks of time. This provides a very general, but primitive, framework for discrete time. Concepts such as maximal parallelism, synchronous communication, etc. can not be expressed, unless by adding new axioms.

It is interesting to remark that the way ATP treats time is very similar to how the current implementation of Statecharts does it. No causal paradoxes can arise, because there is no notion of synchrony: a reaction can never influence its triggering action.

## References

[BB90] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. Technical Report P8916b, Programming Research Group, University of Amsterdam, March 1990.

[GB87] R. Gerth and A. Boucher. A timed failures model for extended communicating processes. In *Proc. 14th Colloquium Automata, Languages and Programming ICALP*, LNCS 267, pages 95–114, 1987.

[Mar87] F. Maraninchi. *Statecharts: sémantique et application à la spécification de systèmes.* PhD thesis, INP Grenoble, 1987.

[Mar89] F. Maraninchi. Argonaute: Graphical description, semantics and verification of reactive systems by using a process algebra. In *Workshop on Automatic Verification methods for Finite State Systems, Grenoble 12-14 June 1989.* Springer-Verlag, 1989.

[NRSV90] X. Nicollin, J.-L. Richier, J. Sifakis, and J. Voiron. ATP: an Algebra for Timed Processes. In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, Sea of Gallilee, Israel, 1990.

Figure 1

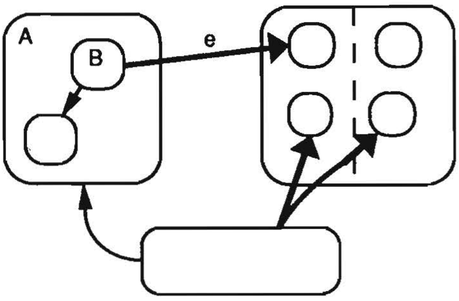


Figure 2

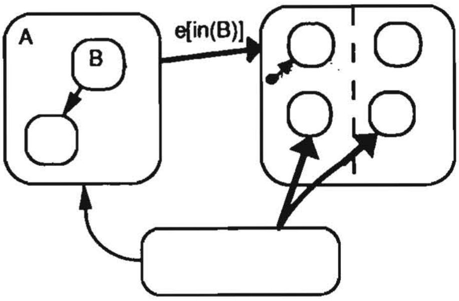


Figure 3

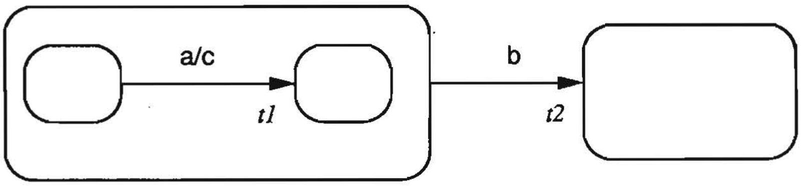


Figure 4

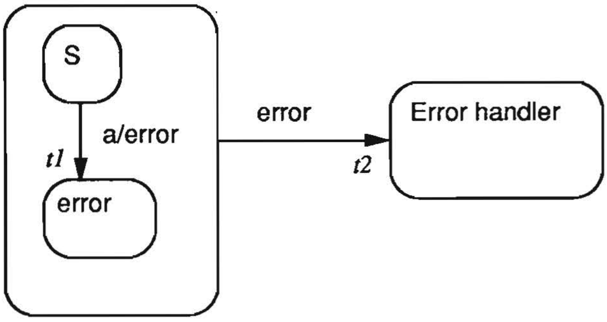


Figure 5

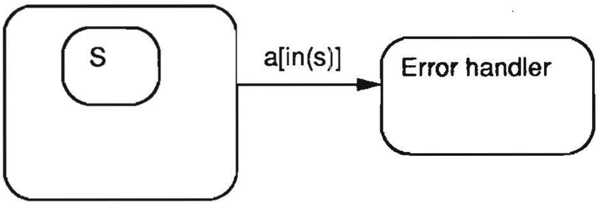


**Figure 6**

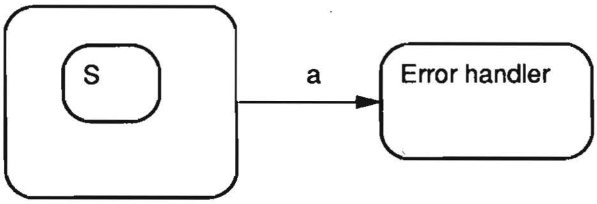


**Figure 7**

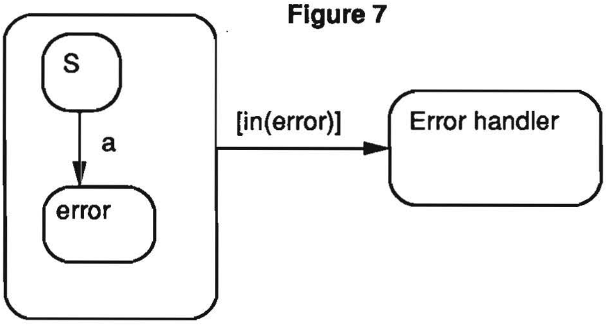


**Figure 8**

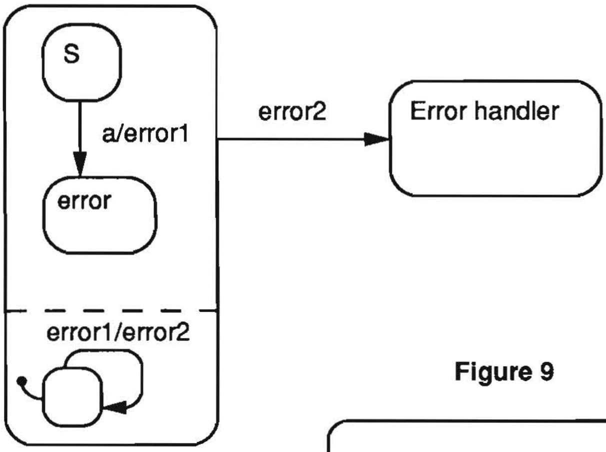
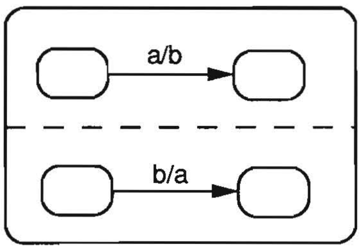


**Figure 9**



## Bibliography of Statecharts

Most of the articles and books below are referenced somewhere in this thesis, but it makes sense to present a reasonably complete list of references to Statecharts.

## References

[1] D. Drusinsky and D. Harel. On the Power of Bounded Concurrency I: The Finite Automata Level. Submitted to *J. Assoc. Comput. Mach.*

[2] D. Drusinsky and D. Harel. On the Power of Cooperative Concurrency. In *Proceedings of Concurrency 88*, pages 74–103. Springer-Verlag, 1988.

[3] D. Drusinsky and D. Harel. Using Statecharts for Hardware Description and Synthesis. *IEEE Transactions on Computer-Aided Design*, 8(7):798–807, July 1989.

[4] D. Harel. Biting the Silver Bullet. Submitted to *IEEE Computer.*

[5] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[6] D. Harel. On visual formalisms. *Communications of the ACM*, 31:514 - 530, 1988.

[7] D. Harel. A Thesis for Bounded Concurrency. In *Proc. 14th Symp. on Math. Found. of Comput. Sci.*, LNCS 379, pages 35–48, New York, 1989. Springer-Verlag.

[8] D. Harel and C.-A. Kahana. On statecharts with overlapping. Submitted, 1990.

[9] 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 Transactions on Software Engineering*, 16(4):403–414, April 1990.

[10] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. NATO, ASI-13, Springer-Verlag, 1985.

[11] D. Harel, A. Pnueli, J. Pruzan-Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proceedings Symposium on Logic in Computer Science*, pages 54–64, 1987.

[12] D. Harel and S. Rolph. Modelling and Analyzing Complex Reactive Systems. In *Proc. AIAA Computers in Aerospace VII Conf.*, Monterey, CA, Oct. 1989.

[13] D. Harel, R. Rosner, and M. Vardi. On the Power of Bounded Concurrency III: Reasoning about Programs. In *Symp. on Logic in Computer Science*, pages 479–488. Springer-Verlag, 1990.

[14] T. Hirst and D. Harel. On the Power of Bounded Concurrency II: The Pushdown Automata Level. In *Proc. CAAP '90, Trees in Algebra and Programming*, pages 1–17. Springer-Verlag, 1990.

[15] J. Hooman, S. Ramesh, and W.P. de Roever. A Compositional Axiomatisation of Safety and Liveness Properties for Statecharts. In *Proc. of the international BCS-FACS Workshop Semantics for Concurrency*, pages 242–261. Springer-Verlag, 1990.

[16] C. Huizing and W.P. de Roever. Introduction to design choices in the semantics of Statecharts. To be published in *Information Processing Letters*, 1990.

[17] C. Huizing and R. Gerth. On the semantics of reactive systems. Technical report, Eindhoven University of Technology, The Netherlands, 1989.

[18] C. Huizing, R. Gerth, and W.P. de Roever. Modelling statecharts behaviour in a fully abstract way. In *Proc. 13th CAAP*, LNCS 299, pages 271–294, 1988.

[19] C. Huizing, R. Gerth, and W.P. de Roever. A compositional semantics for statecharts. Computing Science Note CSN 87/15, Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, 1987.

[20] *i-Logix Inc*, Burlington, Mass. *The Languages of* STATEMATE, 1987, revised version 1990. *In* Documentation for the STATEMATE System.

[21] *i-Logix Inc*, Burlington, Mass. *The Semantics of* STATECHARTS, 1989. *In* Documentation for the STATEMATE System.

[22] F. Maraninchi. *Statecharts: sémantique et application à la spécification de systèmes.* PhD thesis, INP Grenoble, 1990.

[23] A. Pnueli and M. Shalev. What is in a step. Technical report, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 1988. Draft.

[24] S. Rolph and T. Alfano. Statemate by Example: Specifying a Control System. To be published, 1990.

[25] S.L. Smith and S.L. Gerhart. Statemate and cruise control: A case$^2$ study. In *Proc. COMPAC,'88, 12th Int. IEEE Comput. Software and Applicat. Conf.*, pages 49–56, New York, 1988. IEEE Press.

[26] M. Trachtman. *Elements of Statemate Style: A Framework for Embedded System Specifications.* 1990. To be published.

## Promotiereglement artikel 15.3b

The EUT "promotiereglement" requires that if a thesis contains co-authored papers it should be indicated which parts are based on active contributions of the author of the thesis.

Chapter 1, which is co-authored by W.P. de Roever is a typical joint article that is difficult to entangle. The article found its form by the joint preparation of a lecture to be given by W.P. de Roever. Although most of the parts were written by the author of this thesis, they found its roots in joint discussions.

For chapter 2, the co-authors W.P. de Roever and R. Gerth provided the idea of making the CSP-R semantics fully abstract. In fact, they put me on this subject. Furthermore, they contributed a lot to the set-up of the article and the actual formulation of the text. The author of the thesis has formulated the operational semantics of CSP-R, and has found the way to make the denotational semantics fully abstract, as well as the proof.

In the case of chapter 3, the idea came forth from the ESPRIT project DESCARTES. The co-authors, R. Gerth and W.P. de Roever "debugged" the ideas of the author of this thesis, but the technical work was all his. Gerth and Huizing worked together on the proof of full abstraction and found the idea of modularisation of the proof. The actual modularisation was Huizing's work.

For chapter 5 the idea of the article came forth form unhappiness with the existing semantics of statecharts and found its root in many discussions between the authors and with, a.o., Amir Pnueli and Willem-Paul de Roever. The author of the thesis made the semantic framework, and Gerth suggested to formulate criteria to evaluate them. Most of the work of finding which criteria and how to formulate them, was done by the author, although again it found its root in the discussions with the people mentioned above. I think Amir Pnueli is responsible for the idea of the criterium of causality.

# Samenvatting

Reactieve systemen zijn computersystemen die gekenmerkt worden door een voortdurende interactie met de buitenwereld. Die buitenwereld kan een heel computersysteem op zich zijn, bijvoorbeeld de boordcomputer van een vliegtuig, maar ook een menselijke gebruiker, een meet- en regelsysteem, of een combinatie hiervan. Er kan veel afhangen van een snel en accuraat functioneren van deze systemen; de toepassingen hebben bovendien de neiging een steeds groter beroep te doen op de computers en steeds minder op de mensen die haar bedienen. Men denke aan het controlesysteem van een kerncentrale, de automatische piloot van een verkeersvliegtuig of het antiblokkeersysteem in de remmen van een auto. Juist daarom is het bedenkelijk dat de theoretische fundering hiervan nog niet zover gevorderd is als die van conventionele computerprogramma's, de zogenaamde transformationele systemen.

Dit proefschrift bestudeert een belangrijk aspect van die theoretische fundering: de semantiek. Een semantiek beschrijft het gedrag van een programma of specificatie in een wiskundig model en kan dienen als basis voor analyse- en ontwerpmethoden.

Dit proefschrift bestaat uit vijf hoofdstukken.

In hoofdstuk 1 wordt uitgelegd wat het fundamentele verschil is tussen reactieve en transformationele systemen: de laatste zijn bevredigend te beschrijven als een functie van input naar output; bij reactieve systemen lukt dit niet omdat er feedback kan optreden van de reacties van het systeem via de buitenwereld terug naar het systeem. Daarom is het van belang te weten wanneer precies een output gegenereerd wordt ten opzichte van de inputs en is een functie die rijen inputs op rijen ouputs afbeeldt onvoldoende om het hele gedrag te beschrijven. De consequenties hiervan worden aan de hand van een grafische taal voor reactieve systemen, Statecharts, bestudeerd. Er wordt beargumenteerd waarom het noodzakelijk is dat de semantiek van zo'n taal voldoet aan de *synchroniciteitsaanname* (synchrony hypothesis), wat wil zeggen dat de reactie van het systeem gelijktijdig is met de stimulus die haar veroorzaakte. Dit is alleen conceptueel, omdat er altijd een zekere tijd nodig is om de reactie te berekenen, maar het is in veel gevallen houdbaar omdat de reactietijd van het systeem veel korter is dan de intervallen tussen de stimuli.

Toch brengt deze benadering fundamentele problemen met zich mee, omdat er causale paradoxen kunnen ontstaan, bijvoorbeeld als een reactie haar eigen stimulus ongedaan maakt (vergelijk kortsluiting, rondzingen, etc.).

In hoofdstuk 4 worden deze problemen uitvoerig bestudeerd aan de hand van één semantisch raamwerk waarin verscheidene oplossingen tegen elkaar afgewogen kunnen worden. Dit gebeurt in het licht van drie criteria, die alle op zich wenselijke eigenschappen van een semantiek voor reactieve systemen zijn. Het blijkt dat deze criteria niet tegelijk vervuld kunnen worden.

Een belangrijke eigenschap van een semantiek is compositionaliteit. Een semantiek is compositioneel als het gedrag van de combinatie van twee programmadelen beschreven kan worden in termen van de semantiek van de componenten. Dit betekent dat de semantiek in het algemeen ingewikkelder wordt: men moet immers a priori rekening houden met alle mogelijke samenstellingen. Het voordeel is echter dat de semantiek van een component maar één keer berekend hoeft te worden en niet steeds bij iedere verandering van het totale programma. Bovendien verschaft een compositionele semantiek de basis voor analyse- en ontwerpmethoden volgens een verdeel-en-heers-principe.

Een logisch vervolg op compositionaliteit is full abstraction. Dit houdt in dat alle extra informatie die vastgelegd moet worden om compositionaliteit te berekenen ook noodzakelijk is.

Bijgevolg onderscheidt een semantiek die aan full abstraction voldoet niet méér programma's dan noodzakelijk. Abstractie, dat wil zeggen het niet onderscheiden van gelijkwaardige programma's, is een essentiële eigenschap van een semantiek.

In hoofdstuk 2 wordt een bestaande compositionele semantiek voor een real-time taal zo aangepast dat ze aan full abstraction voldoet. In deze taal kan het gedrag van een systeem met betrekking tot het tijdsverloop nauwkeurig gespecificeerd worden. Dit is in principe geschikt om het actie-reactie-gedrag van reactieve systemen te beschrijven, maar het moet op een vrij laag niveau plaatsvinden.

In hoofdstuk 3 wordt Statecharts behandeld, een taal die reactieve systemen op een hoger niveau kan beschrijven. Er wordt een compositionele semantiek gedefinieerd, wat een nieuwe definitie van samenstelling van componenten vereist, omdat Statecharts een grafische taal is, dat wil zeggen uit diagrammen in plaats van tekst bestaat. Full abstraction wordt bereikt en het bewijs hiervan is modulair opgebouwd. Dat alle elementen van de semantiek noodzakelijk zijn wordt aangetoond door een zogenaamde context te construeren. Dit is een Statecharts-programma dat tegelijk met een willekeurig ander programma geëxecuteerd kan worden. Een eventueel verschil in semantiek tussen twee programma's wordt dan zichtbaar in een daadwerkelijk verschillend gedrag in die context. Dit contextprogramma nu bestaat uit een aantal modulen, voor ieder element van de semantiek één. Bij een uitbreiding van de taal kan het bewijs eenvoudig aangepast worden door een module toe te voegen die de overeenkomstige uitbreiding van de semantiek zichtbaar maakt.

Hoofdstuk 5 ten slotte geeft verbanden aan met talen die verwant zijn aan Statecharts.

## CURRICULUM VITAE

Ik ben geboren op 12 augustus 1961 in Bergen op Zoom.

In 1979 deed ik eindexamen VWO aan de Christelijke Scholengemeenschap Blaise Pascal en ging wiskunde studeren aan de Rijksuniversiteit te Utrecht. Na een kortstondige excursie in de filosofie koos ik het bijvak informatica.

In 1982 deed ik kandidaatsexamen en kwam ik in contact met Willem-Paul de Roever die er in slaagde mijn belangstelling voor de logica om te buigen naar de theoretische informatica.

In 1985 studeerde ik af met groot bijvak informatica en didactische aantekening. De scriptie ging over een fully abstract semantiek voor CSP-R.

In hetzelfde jaar kwam ik bij Willem-Paul de Roever aan de toenmalige Technische Hogeschool Eindhoven te werken als toegevoegd onderzoeker in het ESPRIT-project DESCARTES.

Sinds 1989 ben ik als universitair docent aan de Technische Universiteit Eindhoven werkzaam.

# Stellingen

behorende bij het proefschrift

Semantics of reactive systems: comparison and full abstraction

van

C. Huizing

1. High-level real-time talen zoals Esterel en Statecharts zijn niet *real*-time in de zin dat er een direct verband is te leggen tussen de executie van een statement en de werkelijke tijd waarop dat plaats vindt, zoals dat wel in Real-Time CSP en OCCAM kan.

   G. Berry and G. Gonthier, The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. ENSMP-INRIA, Sophia-Antipolis, 1988.

   J. Hooman and J. Widom, A temporal-logic based compositional proof system for real-time message passing. In *Parallel Architectures and Languages Europe*, volume II, pages 424-441. LNCS 366, Springer-Verlag, 1989.

2. Vanuit het oogpunt van bruikbaarheid zijn de criteria *reponsiveness* (reacties kunnen ogenblikkelijk plaatsvinden), *modularity* (interface tussen deelsystemen is dezelfde als tussen een systeem en de buitenwereld) en *causality* (gebeurtenissen hangen causaal samen) ieder op zich zeer wenselijke eigenschappen van een specificatietaal voor reactieve systemen. Ze zijn echter niet verenigbaar in één semantiek.

   Zie hoofdstuk 4, paragraaf 3 van dit proefschrift (stelling 1).

3. In de huidige implementatie van STATEMATE is de semantiek m.b.t. de tijd van *"on entering do ..."* in state $S$ niet gelijk aan die van *"on enter(S) do ..."*.

   Voor een ontwikkelingsomgeving van complexe tijdcritische systemen is een goed doordachte semantiek essentieel, omdat niemand zulke systemen geheel kan overzien en uitputtende mechanische verificatie onmogelijk is.

   Dat STATEMATE in dit opzicht tekort schiet, blijkt uit hoofdstuk 4 van dit proefschrift en uit inconsistenties als de bovenstaande. In de toepassingen waar STATEMATE voor bedoeld is kan dit tot letterlijk levensgevaarlijke situaties leiden.

   Zie: STATEMATE version 3.0. i-Logix Inc., Burlington, Mass.

4. Een semantiek heet compositioneel als de betekenis van een samengesteld programma gedefinieerd is in termen van de semantiek van de samenstellende delen. In het algemeen moet een compositionele semantiek meer informatie vastleggen dan alleen het observeerbaar gedrag en zij onderscheidt daardoor ook meer programma's.

   *Full abstraction* is de eigenschap van een semantiek dat zij niet méér programma's onderscheidt dan noodzakelijk is om compositioneel te zijn, gegeven een bepaalde notie van observeerbaar gedrag. In het bewijs van full abstraction wordt doorgaans gebruik gemaakt van een contextprogramma dat semantische verschillen tussen twee programma's zichtbaar maakt door executie van de programma's in die context.

   Doordat er in een real-time semantiek zoveel meer observeerbaar is aan een programma en doordat de taal zoveel meer controle geeft over het construeren van de context, is het full-abstraction-bewijs voor een CSP-achtige taal in het real-time geval in dit opzicht eenvoudiger dan in het niet-real-time geval.

   Zie hoofdstuk 2, paragraaf 7 van dit proefschrift.

5. Het construeren van de context voor het full-abstraction-bewijs van een specificatie- of programmeertaal is een goede test voor het gemak waarmee men in die taal een gedrag nauwkeurig kan specificeren.

   Zo zou het ontbreken van het *empty statement*, een skip statement dat geen executietijd kost, in het oorspronkelijke DNP-R en CSP-R een full-abstraction-bewijs nodeloos ingewikkeld maken, zonder dat er een inherente reden is dat het ontbreekt.

   Zie hoofdstuk 2, paragraaf 7 van dit proefschrift.

6. Manna en Pnueli introduceren een manier om het gedrag van een programma $P$ te beschrijven met een temporeel-logische formule, zeg $T(P)$. In de ene variant, het ongeankerde systeem, geldt *niet* dat het programma $P$ voldoet aan zijn eigen formule $T(P)$ (m.a.w. $P$ **sat** $T(P)$ is niet waar). In de andere variant, het geankerde systeem, geldt dit wel. Deze tegenintuïtieve eigenschap van de ongeankerde variant wordt onvoldoende genoemd in de motivatie van de geankerde variant.

   Zie: Z. Manna and A. Pnueli, How to Cook a Temporal Proof System for your Pet Language. In *Proc. of the ACM Symposium on Principles of Programming Languages*, Austin, Texas, **10** (January 1983) pp. 101-154.

   en: A. Pnueli, Applications of temporal logic to the specification and verification of reactvie systems: a survey of current trends. In *Proc. ESPRIT/LPC Advanced School on Current Trends in Concurrency*, LNCS 224, pp. 510-584, 1985.

7. Het is een wezenlijke beperking om het tijdsdomein en het domein waarin de berekeningsrijtjes gemodelleerd zijn te laten samenvallen (zoals in [K89]). Alle vier de combinaties van een dicht resp. discreet tijdsdomein met een dicht resp. discreet berekeningsdomein zijn zinvol en komen voor.

| | *discreet tijdsdomein* | *dicht tijdsdomein* |
|---|---|---|
| *discreet berekeningsdomein* | [K89] | [PH88] |
| *dicht berekeningsdomein* | [NRSV90] | [K98] |

[K89] R. Koymans, *Specifying Message Passing and Time-Critical Sytems with Temporal Logic.* Proefschrift Technische Universiteit Eindhoven, 1989.

[NRSV90] X. Nicollin, J.-L. Richier, J. Sifakis, and J. Voiron. ATP: an Algebra for Timed Processes. In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, Sea of Gallilee, Israel, 1990.

[PH88] A. Pnueli, E. Harel, Applications of Temporal Logic to the Specification of Real-time Systems, *in Proc. Formal Techniques in Real-Time and Fault-Tolerant Sytems.* LNCS 331, pp. 84-98, 1988.

8. Het slecht gemotiveerde besluit van het Rekencentrum van de Technische Universiteit Eindhoven, en van vrijwel alle faculteiten in zijn kielzog, om de Apple Macintosh computer niet te ondersteunen doet vrezen dat het begrip ergonomie nog niet tot deze instelling is doorgedrongen. Het kan dan ook geen toeval zijn dat juist een van de belangrijkste onderzoeksinstituten in Nederland op het gebied van de ergonomie, het Instituut voor Perceptie Onderzoek, een van de weinige instellingen op het universiteitsterrein is die deze computers wèl ondersteunt.

9. De keuze tussen dynamische en statische binding, die in een programmeertaal doorgaans door de ontwerper van de taal gemaakt wordt, kan in een natuurlijke taal als het Nederlands door voegwoorden en zinsbouw aangegeven worden. Bij dynamische binding is de waarde van variabelen in een procedure afhankelijk van de context waarin die procedure wordt aangeroepen. Bij statische binding is de waarde van die variabelen afhankelijk van de context waarin de procedure is gedeclareerd.

   Voorbeeld: "Gisteren zei hij: 'Ik kom morgen' " (statische binding van *morgen*) tegenover "Gisteren zei hij, dat hij vandaag zou komen (dynamische binding van *vandaag*)".

10. In de paradox van het onverwachte proefwerk (Leraar tot klas: 'Jullie krijgen komende week een proefwerk, maar je zult het de avond tevoren niet weten') komen beide partijen tot een juiste gevolgtrekking. De leerlingen beredeneren dat het proefwerk de laatste dag van de week niet kan worden gegeven, omdat ze het dan de vorige avond zouden weten; evenzo kan het niet de op een na laatste dag zijn, enzovoorts. Zij concluderen dat ze geen proefwerk kunnen krijgen. De leraar echter deelt op dinsdag de blaadjes uit en meent dat hij in overeenstemming met zijn uitspraak gehandeld heeft: de klas verwachtte het immers niet! Dat beide gevolgtrekkingen tegelijk juist

kunnen zijn komt doordat de aankondiging van de leraar ongerijmd (*falsum*) is. Dit volgt uit de redenering van de leerlingen. Deze paradox is een instructief voorbeeld van de wet uit de logica dat uit het ongerijmde alles volgt.

Zie: T.H. O'Beirne, *Puzzles and paradoxes*, Oxford University Press, 1965.

11. De gedachte dat positieve discriminatie nut heeft, berust op een verwarring van het probleem, ongelijke behandeling in gelijke omstandigheden, en de indicatie van het bestaan van het probleem, de statistiek van de verdeling.

12. De neiging van verzekeringsmaatschappijen om zich te presenteren als zouden zij bescherming en veiligheid bieden, in plaats van alleen een financiële vergoeding, is in de meeste gevallen onjuist, misleidend en moreel verwerpelijk. Deze neiging komt voort uit een gebrek aan reële verkoopargumenten.