Full Abstraction of a Denotational Semantics for Real-time Concurrency

by

C. Huizing
P. Gerth
Full Abstraction of a
Denotational Semantics for
Real-time Concurrency

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

November 1986
COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science of Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
All rights reserved
editor: F.A.J. van Neerven
European Strategic Programme of Research and Development in Information Technology

Project 937: Debugging and Specification of Ada Real-Time Embedded Systems
Package 4: Formal Semantics and Proof Systems for Real-Time Languages

Mail to:
Doc. No. : PE.01
Type : PE
Title : Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language
Author : C. Huizing, R. Gerth, W.P. de Roever
Date : 11-11-86 Version : 0

Document Status: Submitted
Confidentiality Level: public-domain

GSI-TECSI
SYSTEAM KG
FOXBORO Netherlands NV
ELECTRONIQUE SERGE DASSAULT
EINDHOVEN UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF STIRLING
ADCAD Ltd

Copyright 1986 by the DESCARTES consortium formed by the companies and universities listed above.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, and that the DESCARTES copyright notice and the title of this document and date appear.
Full Abstraction of a
Real-Time Denotational Semantics for an
OCCAM-like Language *

C. Huizing 1,2
R. Gerth 1
W.P. de Roever 1

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, commu­
nication along channels, and the time communication takes place, into observ­
ables. 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 Fran­
cez, 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 paral­
lelism model of Salwicki and M{"u}ldner.

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;

* This paper is based on C. Huizing's M.Sc. Thesis [HGR85].
1 The authors are working in and partially supported by Esprit Project 937: Debugging and Specification of Real-Time
Embedded Systems (DESCARTEs).
2 E-mail address: mcvax1eurec3!winktees.UUCP.
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 real-time. However, this very fact causes such problems when defining its semantics that no undisputed results on the meaning of the language exist. 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. However, in case of concurrency, a semantics must record more than just 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 [KSRGA86], 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 (e.g., the messages sent and received) and some non-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 a 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 its semantics are given.

2. The language DNP-R

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

2.1. Syntax

Definition 1.

- \( \text{Var} \) is the set of program variables, ranged over by \( x \).
- \( \text{Chan} \) is the set of channels, ranged over by \( \alpha; A \subseteq \text{Chan} \)
- \( e \) 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:

\[
S := x := e \parallel g \parallel S_1 ; S_2 \parallel IOC \mid *IOC \parallel S_1 \parallel_A S_2 \parallel [S]_A
\]

\[
g := \alpha!e \mid \alpha?x \mid \text{wait} \, n \mid -
\]

\[
IOC := [\square \; b_i ; g_i \rightarrow S_i ]
\]

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

- \( \text{ichan}(S) \) denotes the set of internal channels of \( S \), which is defined as the union of all sets \( A \) occurring in any substatement \( S_1 \parallel_A S_2 \) of \( S \).
- \( \text{chin}(S) \) 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 \( \text{ichan}(S) \).
- chout(S) denotes the (external) output-channels of S, defined likewise.

- hid(S) denotes the hidden-channels of S, defined as the union of all sets A that occur in a construct [S]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 ∈ Stat, then chin(S) ∩ chout(S) = Ø

(ii) if S ⊦₂ S₂ ∈ Stat, then S₁ ∈ Stat and S₂ ∈ Stat and

(ii.1) var(S₁) ∩ var(S₂) = Ø

(ii.2) (chout(S₁) ∩ chout(S₂)) ∪ (chout(S₁) ∩ chout(S₂)) = A

(ii.3) chin(S₁) ∩ chin(S₂) = chout(S₁) ∩ chout(S₂) = Ø

(iii) if [S]A ∈ Stat, then S ∈ Stat and A = ichan(S)

(iv) if S ∈ Stat and S ⊦₂ S₂ is a substatement of S, then none of the channels in A occur anywhere outside S₁ ⊦₂ S₂ in S.

Examples. The following statements are excluded by Definition 3.

ad (i). α?x; α!0

A process cannot send values to itself.

ad (ii.1). x:=0∥x:=1

There are no shared variables.

ad (ii.2). α?x∥_β α!0

The index set should at least contain α.

ad (ii.3). α?x∥₂ α?y and α!1∥₂ α!2

A channel connects exactly two processes: one sender and one receiver.

ad (iii). [α?x∥_α α!0]₀

The index set of the hiding operator should be {α}.

ad (iv). [α?x∥_α α!1]_α;[α?y∥_α α!1]_α

Although hidden, the name α 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 $e$ 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 $S_1 || A S_2$ 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 $* \text{IO} e$ stands for repeated execution of the I/O guarded conditional $\text{IO} e$ (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 $[\square b_i; g_i \rightarrow S_i]$ 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 boolean expressions yields true, the conditional does not fail, but is skipped. There is no priority of local actions over communications and vice versa.

A conditional may also contain wait-guards, $b_i; \text{wait} n$. 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

\[ P;\alpha!3\parallel_{(\alpha)}[\alpha?x \rightarrow \neg\Box\beta?x \rightarrow \neg] \parallel_{(\beta)} Q;\beta!4. \]

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\(^1\). 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 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: \( O[P]_\sigma \). This is formalized by the operational semantics in Section 6. A behaviour is a pair

\( ^{1} \) We take the view that evaluating an expression takes time. Hence, \( \text{wait } n \) even if \( n \) evaluates to 0, takes 1 time unit.
\(<t, h\rangle\), where \(t\) is the end state if \(P\) terminates - otherwise \(t = \infty\) - 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 than 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 \(t\)
- \(h_1 h_2\) 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 \(h[i] = \emptyset\)
- \(h\upharpoonright A\) denotes the restriction of sequence \(h\) to the set of channels \(A: (h\upharpoonright A)[i] = \{\alpha_v \in h[i] \mid \alpha \in A\}\)
- \(h_1 < h\) (" \(h_1\) is a prefix of \(h\)") iff there exists a sequence \(h_2\) such that \(h_1 h_2 = h\). \[\]

5. Denotational Semantics

Our denotational semantics, \(D\), 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 (\(\bot\)) 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 \(<\bot, \lambda>\) as least element. All denotations, \(D\), will be prefix-closed. This means that for any \(<\sigma, h> \in D\) and \(h' < h\) we have \(<\sigma, h'> \in D\). This cpo structure also allows us to define the semantics of the iterative construct as a least fixed point. For more information on cpo's 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"\(^1\) to the sets in the histories.

\(^{1}\) 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.
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>, ..., $ whereas the denotation of $Q=\text{wait } 1; \alpha ? x$ includes $<\sigma', \alpha_0>, <\sigma', R_\alpha \alpha_0>, ..., $ 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 $y \neq x$).

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 $\varnothing R_\alpha \alpha_0$ are not consistent. Thus maximality is enforced.

(ii) agree on the communications on the joint channels. So, e.g., $\alpha_0$ and $\varnothing \alpha_0$ or $R_\alpha \alpha_0$ and $\varnothing \alpha_1$ are not consistent. Thus synchronization of communications is expressed.

To be more specific, the semantics of the parallel composition is as follows:

$$D \llbracket P \| A \| P_2 \rrbracket \sigma = \text{Cl} \{<\sigma_1, \sigma_2, h_1 || A \| h_2> ||$$

$$<\sigma_1, h_1> \in D \llbracket P_1 \rrbracket \sigma,$$

maximal $(h_1, h_2)$, synchronous $(h_1, h_2, A)$,

$$\text{comparable} (\sigma_1, h_1, \sigma_2, h_2).$$

This definition uses the following operators.

$\text{Cl}$ 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

$$\def\dfrac{\sigma_1(x)}{\sigma_2(x)} $$

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

(hence, $\sigma_1 ||_\sigma \bot = \bot ||_\sigma \sigma_2 = \bot$).

This definition is unambiguous, because $P_1$ and $P_2$ cannot both change $x$ (there are no shared variables).
$h_1 \parallel h_2$ is defined by

$$\langle h_1 \parallel h_2 \rangle[j] = (h_1[j] \cup h_2[j]) \setminus \{ R_\alpha \parallel \alpha \in A \}$$

(remember the convention that $h[j] = \emptyset$ if $j > 1$).

This is the pointwise union, except that readies on channels in $A$ are not preserved; they are not needed anymore.

\[
\text{maximal}(h_1, h_2) \iff \forall j, \alpha : R_\alpha \not\in h_1[j] \cap h_2[j]
\]

embodies the maximality constraint of (i) above,

\[
\text{synchronous}(h_1, h_2, A) \iff \forall j, \alpha, \nu : \alpha \in A \rightarrow (\alpha \in h_1[j] \leftrightarrow \alpha \in h_2[j])
\]

expresses synchrony as in (ii) above,

\[
\text{comparable}(\sigma_1, h_1, \sigma_2, h_2) \iff \forall i \in \{1,2\} : \sigma_i = \bot \rightarrow |h_{3-i}| \leq |h_i|
\]

guarantees that no incomplete history will be merged with a longer one.

To understand the necessity of this constraint, consider the program $P = \alpha!0\parallel\alpha?x$

Then, e.g., $\langle \bot, R_\alpha R_\alpha \rangle \in D[\alpha?x] \sigma$

and $\langle \bot, x \rangle \in D[\alpha!0] \sigma$

Without the comparability check we would have

$\langle \bot, \emptyset \rangle = \langle \bot, R_\alpha R_\alpha \parallel \parallel (\alpha) \rangle \in D[\alpha?x \parallel \parallel (\alpha)] \sigma$

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 $D$ can be found in the appendix.

6. Operational Semantics

We give an operational semantics $O$, 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, $O$ captures - indeed defines - exactly the observable behaviour in a way that is independent from the denotational semantics.
6.1. The Labeled Transition System

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

\[ L(P, \sigma) \rightarrow (P', \sigma') \]

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:**

\[ (x:=e, \sigma) \xrightarrow{(1, \sigma)} (-, \sigma(e)/x) \]

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:**

\[ (\alpha!e, \sigma) \xrightarrow{(0, \sigma)} (-, \sigma) \]

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 [Plot83], 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:

\[
L_1 \uparrow_A = L_2 \uparrow_A, (P_1, \sigma) \rightarrow (P'_1, \sigma'_1), (P_2, \sigma) \rightarrow (P'_2, \sigma'_2)
\]

\[
\text{where } L = (L_1^n + L_2^n, L_1^c \cup L_2^c)
\]

\[\sigma' \text{ is defined by } \sigma'(x) = \begin{cases} 
\sigma'_1(x) & \text{if } x \in \text{var}(P_1) \\
\sigma'_2(x) & \text{if } x \in \text{var}(P_2) \\
\sigma(x) & \text{otherwise}
\end{cases} \]

The condition \(L_1 \uparrow_A = L_2 \uparrow_A\) 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

\[P = \alpha?x || \alpha!3\]

admits both the transitions

\[a) (P, \sigma) \xrightarrow{(\alpha,x)} (-\|_1, \sigma[3/x]) \text{ and } \]

\[b) (P, \sigma) \xrightarrow{(0,\sigma)} (P, \sigma). \]

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:

\[(n_1, c_1) \leq_I (n_2, c_2) \iff n_1 \leq n_2 \land c_1 \uparrow I \subseteq c_2 \uparrow I \land c_1 \uparrow \bar{I} = c_2 \uparrow \bar{I} \]

\(\bar{I}\) 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 \((0, \emptyset) \leq_{\{a\}} (0, \{a_3\})\). This leads to the operational semantics \(O\).

Definition 7.
Let \(P_0\) be a DNP-R program and \(\sigma_0\) a state.
\[
O\left[\left[\begin{array}{c}
P_0 \\
\sigma_0
\end{array}\right]\right] =
\left\{\begin{array}{c}
\langle \sigma, h \rangle | P \xrightarrow{i, \sigma_i} L, \forall 1 \leq i \leq h |
\end{array}\right\}
\wedge (|h| < \infty \rightarrow \sigma = \sigma_{\lfloor h \rfloor} \wedge terminated(P_{\lfloor h \rfloor}))
\wedge (|h| = \infty \rightarrow \sigma = \infty)
\}
\]
where \(vis(P) = Chan_{\text{hid}}(P)\), 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 \(\_|A\).

The operational and denotational semantics, \(O\) and \(D\), are related by the following.

Theorem 1. \(O = \beta \circ D\), where \(\beta\) is an abstraction function.

Here, \(\beta\) deletes all non-observable information, viz. readies, and smooths away the differences between the two domains; e.g., infinite chains of finite histories are replaced by their limits.

The tedious and very lengthy proof of this theorem is only available in manuscript from the first author. 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 \(D\) is fully abstract w.r.t. a semantics \(O\) iff for all programs \(P\) and \(Q\):
\[
D \left[\begin{array}{c}
P \\
\end{array}\right] = D \left[\begin{array}{c}
Q \\
\end{array}\right] \iff \forall \text{ contexts } C: O \left[\begin{array}{c}
C(P) \\
\end{array}\right] = O \left[\begin{array}{c}
C(Q) \\
\end{array}\right].
\]
This definition can be found in the literature [HP79]. Its motivation lies in the following
(Folk?) Theorem 2: $D$ is fully abstract w.r.t. $O$ iff

(i) $D$ is syntax-directed,

(ii) $D$ is the coarsest semantics that distinguishes at least as much as $O$ 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. $P,Q \in \text{Stat}$ are syntactically comparable iff for any context $C$ holds

$$C(P) \in \text{Stat} \iff C(Q) \in \text{Stat}$$

In effect, this boils down to $P$ and $Q$ having the same sets $\text{chin}$, $\text{chout}$, $\text{ichan}$, and $\text{var}$.

We redefine full abstraction as follows:

Definition 10. A semantics $D$ is fully abstract w.r.t. a semantics $O$ iff for all syntactically comparable programs $P$ and $Q$:

$$D[P] = D[Q] \iff \forall \text{contexts } C: O[C(P)] = O[C(Q)].$$

Relative to this modified notion, $D$ is not fully abstract with respect to $O$ 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

$$P = [\text{true} \rightarrow \alpha!0 \square \text{true} \rightarrow \beta!0 \square \text{true} \rightarrow [\alpha!0 \rightarrow \square \beta!0 \rightarrow \text{false}]$$

$$Q = [\text{true} \rightarrow \alpha!0 \square \text{true} \rightarrow \beta!0]$$

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 $<\sigma, h_1, \mathcal{R}, h_2> , <\sigma, h_1, \mathcal{R}, h_2'>$ are in $D$ then all pairs $<\sigma, h_1, \mathcal{R}, h_2'>$ with $R_1 \subseteq R \subseteq R_1 \cup R_2$ are added to $D$.

Although this turns the readiness semantics into a fully abstract one, this does not suffice in our case. Consider for instance the two programs

$$P = [\text{true} \rightarrow \text{wait} \ 1; \alpha!0 \square \text{true} \rightarrow [\alpha!0 \rightarrow \square \text{false} \ 1 \rightarrow \square \text{true} \rightarrow \alpha!0]$$

$$Q = [\text{true} \rightarrow \text{wait} \ 1; \alpha!0 \square \text{true} \rightarrow \alpha!0 \rightarrow \square \text{false} \ 1 \rightarrow \square].$$

These two programs have different denotations, since e.g., the pairs $<\sigma, \partial \mathcal{R}_\alpha \alpha_o>$, $<\sigma, \partial \mathcal{R}_\alpha \alpha_o>$,... occur in the denotation of $P$, but do not occur in the denotation of $Q$. However, there is no context $C$ that can separate these programs: $O[C(P)] = O[C(Q)]$ 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 $h' \sqsubseteq_R h$, iff $|h'| = |h|$ and for any $i \leq |h|$

$$h'[i] \setminus h[i] \subseteq \text{Readies},$$
where \( \text{Readies} = \{ R_{\alpha} \mid \alpha \in \text{Chan} \} \).

Note that a history which is consistent with \( h' \) is also consistent with \( h \), if \( h' \sqsupset_R h \).

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.**

\[
D_a [P] \sigma = \{ < \alpha', h > \mid \exists h': < \alpha', h' > \in D [P] \sigma \text{ and } h \sqsupset_R h' \}^1
\]

**Theorem 3:** \( D_a \) is fully abstract with respect to \( \sigma \):

\[
D_a [P] = D_a [Q] \iff \forall C \left[ C(P) = C(Q) \right].
\]

We prove this theorem by two lemmas.

**Lemma 1.** \( D_a [P] = D_a [Q] \Rightarrow \forall C : [C(P)] = [C(Q)] \)

**Lemma 2.** \( D_a [P] \neq D_a [Q] \Rightarrow \exists C : [C(P)] \neq [C(Q)] \)

From these two lemmas, we immediately infer Theorem 2.

**Proof of Lemma 1:** Suppose \( D_a [P] \sigma = D_a [Q] \sigma \) and let \( C \) be given.

Because \( D_a \) is defined using induction on the structure of \( \text{DNP-R} \), we have

\[
D_a [C(P)] \sigma = D_a [C(Q)] \sigma \text{ and hence } \beta(D_a [C(P)] \sigma) = \beta(D_a [C(Q)] \sigma).
\]

From \( \sigma = \beta \circ D \) we can easily infer \( \sigma = \beta \circ D_a \) and hence \( [C(P)] \sigma = [C(Q)] \sigma \).

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

Assume \( < \alpha, k > \in D_a [P] \sigma \setminus D_a [Q] \sigma \). On the basis of this history \( k \), we shall construct a program \( L \) that produces the "ready-complement" \( l \) of \( k \) (and some state \( u \)). E.g., if \( k = (R_{\alpha} R_{\beta}) \sigma R_{\alpha} R_{\beta} \), and the only channels occurring in \( P \) and \( Q \) are \( \alpha \) and \( \beta \), then \( l = \varnothing (R_{\alpha} R_{\beta}) R_{\beta} R_{\alpha} \). (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).

---

1. This semantics is also syntax-directed, and, by the nature of \( \beta \), also \( \sigma = \beta \circ D_a \) 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 \( k \preceq_R k' \) (i.e. any other key that turns in the lock will fit "more loosely" than \( k \)). This follows from the construction of \( l \) as the tightest lock fitting around \( k \). Hence, \( \langle \tau, k \rangle \in D_a[[Q]] \sigma \), since otherwise \( \langle \tau, k \rangle \in D_a[[Q]] \sigma \), contradicting our initial assumption by definition of \( D_a \). So, \( \langle \tau \|_a \nu, k \| l \rangle \in D_a[[P \parallel L]] \sigma \), but \( \langle \tau \|_a \nu, k \| l \rangle \notin D_a[[Q \parallel L]] \sigma \) 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 00 is able to merge with \( l \). Although 00 and \( k \) are observably different (there is a difference in termination time), 00/11 is observably the same as 00/11. 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 \((\langle \text{hole} >; \gamma ! 0) \| \gamma ? x \rangle \| L \), 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 \((P \text{ or } Q) \). This makes the length of a shorter history that merges with \( l \) visible at the time at 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 \( h^c[i] = \{ \alpha_r | \alpha_r \in h[i] \} \)
- If \( D \) is a denotation, then its observable reduct \( D^c \) is defined by \( \{ \alpha \|_c h^c > | \langle \alpha, h^c > \in D \} \]

**Lemma 3.** If there is a \( \langle \tau, h \rangle \) such that \( \langle \tau, h \rangle \in (D_a[[P_1]] \sigma)^c \) and \( \langle \tau, h \rangle \in (D_a[[P_2]] \sigma)^c \) then \( O[[P_1]] \sigma \neq O[[P_2]] \sigma \).

**Proof:** Suppose \( \langle \tau, h \rangle \) is given as in the lemma. There are two cases:
(a) If $\tau \neq 1$, it is clear that $<\tau, h> \in \beta(D_a[[P_1]]\sigma) \backslash \beta(D_a[[P_2]]\sigma)$ and hence $O[[P_1]]\sigma \neq O[[P_2]]\sigma$.

(b) If $\tau = 1$ there must be

(i) some $<\tau', h'> \in D_a[[P_1]]\sigma$ with $\tau' \neq 1$ and $h \leq h'$, or

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

This fact is an immediate consequence of the definition of $D_a$.

If case (i) applies, we see that $<\tau', h'^c> \notin (D_a[[P_2]]\sigma)^c$, (otherwise $<\tau, h> \in (D_a[[P_2]]\sigma)^c$ by prefix-closure), and we can apply the lemma, since $\tau' \neq 1$ and this case has been proven already.

If case (ii) applies, we know that $<\infty, h^c> \in \beta(D_a[[P_1]]\sigma)$ where $h''_n = \lim_{n \to \infty} h_n^c$.

Because, for some $n$, $<\perp, h_n> \notin D_a[[P_1]]\sigma$, $<\infty, h^c> \notin \beta(D_a[[P_2]]\sigma)$ and hence $O[[P_1]]\sigma \neq O[[P_2]]\sigma$.

The following is the key-lemma in the proof.

**Lemma 4.** Let a history $k$, a state $\sigma \neq 1$, a set of input channels $I$, and a set of output channels $O$ be given. Assume that $k$ has the property that whenever $\alpha_i \in k[i]$ and $\alpha_\omega \in k[i]$ then $v = \omega$.

Then there exist a program $L$ and a state-history pair $<v, I>$ with the following properties:

(i) $<v, I> \in D_a[[L]]\sigma$

(ii) $v \neq 1$ and $|I| = |k|$

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

(iv) for all $1 \leq i \leq |I|$ and all $\alpha \in I \cup O$:

\[ \alpha_i \in k[i] \iff \alpha_\omega \in k[i] \]

\[ R_\alpha \in k[i] \iff R_\omega \notin k[i] \]

(v) for all $<v', I'> \in D_a[[L]]\sigma$:

\[ v' \neq 1 \rightarrow |I'| = |I| \text{ and } l^c = l' \rightarrow l = L \]

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

1) All histories generated by a program have this property.
where $L^{(i)}_a = \begin{cases} 
[\alpha!v \rightarrow \square \text{wait } 1 \rightarrow -] & \text{if } \alpha \in k[i] \\
\text{wait } 1 & \text{if } \alpha \in k[i] \text{ and } R_a \in k[i]. \end{cases}$ 

if $\alpha \in O$.

If $\alpha \in I$ we take $\alpha?x$ 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.

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 $L\upharpoonright P$ and $L\upharpoonright Q$ are syntactically correct programs.

**Proof of Lemma 2:** Suppose $D_a \left[ P \right] \neq D_a \left[ Q \right]$. Assume without loss of generality that there are $\tau,k$ and $\sigma \neq \perp$, with $<\tau,k> \in D_a \left[ P \right] \sigma D_a \left[ Q \right] \sigma$. It suffices to prove that there are $C,h$ and $\rho$ such that $<p,h> \in D_a \left[ (C,P) \right] \sigma \cap D_a \left[ (C,Q) \right] \sigma \cap$. (This follows from Lemma 1.)

Let $l = \text{chout}(P)$, $O = \text{chin}(P)$. Applying Lemma 4 to $k,\sigma,l$ and $O$ gives us a program $L$, history $h$ and state $v$ with the properties (i) to (v) as stated in the lemma. Define $C =<\text{chol} ; \gamma(0)_{L} \gamma y > x$ where $A = I \cup O$ 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 $\rho = \left( \text{ct}_c \nu \right)(0/x)$ and $h = (k \gamma_1 l)^c$ and let $n = 1/k l$. It is clear from properties (i) and (ii) that

$<p,h> = \left( \text{ct}_c \nu, (k \gamma_1 l)^c \right) \in D_a \left[ ((P;\gamma_1 0)_{L} \gamma y x) \sigma \right] \sigma^c$.

Now suppose $<p,h> \in \left( D_a \left[ ((Q;\gamma_1 0)_{L} \gamma y x) \sigma \right] \sigma \right)^c$.

By definition of $D_a$ and $\cdot$, there must be $<\tau',k'> \in D_a \left[ Q \right] \sigma$, $<\nu',l'> \in D_a \left[ L \right] \sigma$, $<\phi_1,\gamma_1> \in D_a \left[ \gamma_1 0 \right] \sigma$, and $<\phi_2,\gamma_2> \in D_a \left[ \gamma y x \right] \sigma$, such that $\text{consistent}(\phi_1, k' g_1, \nu', l', A)$, $\text{consistent}(\phi_2, g_1 l, \phi_2, \gamma_2, \nu)$, $\left( k' g_1 l, \gamma_2 \right)^c = h$ and $\left( \phi_1, \nu' \right) \phi_2 = \rho$.

Here $k'$ is chosen such that it only contains ready-s in $I \cup O$. (all other reads appear only on behalf of ready-closure). Straightforward application of the definitions gives us that $g_1 = \gamma_0$, $\phi_1 = \nu'$, $\phi_2 = \nu' (0/x)$, $g_2 = R_1 \gamma_0$. Hence,
(k' \parallel A L') \parallel g_2 = k' \parallel A L' and \( \rho = (\tau' \parallel v') \{0/x\} \).

Because \((k' \parallel A L') \parallel g_2 = (k' \parallel A L') = h\),

we see that \( 1k'1 = 1k1 \) and from properties (ii) and (v) also \( 1k1 = 1l1 = 1l'1 \).

Claim 1: \( k'^c = k^c \) and \( l'^c = l^c \).

Proof of claim: suppose \( \alpha, \in k'[i] \), so \( \alpha = \gamma \), then \( \alpha, \in (k' \parallel A L[i]) \), by (1), so \( \alpha, \in k[i] \) or \( \alpha, \in l[i] \). If \( \alpha \in A \), then \( \alpha, \in k[i] \cap l[i] \) by consistency. If \( \alpha \notin A \), then \( \alpha, \in l[i] \), because \( \alpha \) cannot be in the channels of \( L \). So, in both cases \( \alpha, \in k[i] \). All other cases are symmetric.

Claim 2: \( \tau' = \tau \)

Proof of claim: we know that \( (\tau' \parallel v') \{0/x\} = (\tau \parallel v') \{0/x\} \) by (3). Let \( y \in \text{var} \). If \( y \in \text{var}(P) = \text{var}(Q) \), then \( v'(y) = v(y) \) and hence \( \tau'(y) = (\tau' \parallel v') \{0/x\}(y) = (\tau \parallel v') \{0/x\}(y) = \tau(y) \). A similar argument applies if \( y \in \text{var}(L) \) or \( y \notin \text{var}(P) \cup \text{var}(L) \).

From Claim 1 and property (v) we infer \( l' \subseteq_R l' \). (3)

Now we prove: \( k' \subseteq_R k \). Let \( R, \in k'[i] \). Then \( R, \notin l'[i] \) by consistency of \( k' \) and \( l' \). Hence, by (2), \( R, \notin l[i] \) and by (iv): \( R, \in k[i] \). But now we have a contradiction, because, by ready closure and claim 2: \( \langle \tau, k \rangle \in D_a[Q] \sigma \) and hence \( \langle \rho, h \rangle \in (D_a[C(\Sigma)] \sigma)^c \).

Case II \( \tau = \bot \).

Choose \( \rho = \bot \) and \( h = (k \parallel l)^c \). Again we prove that \( \langle \tau, k \rangle \in D_a[Q] \sigma \), which leads to a contradiction. If \( \langle \rho, h \rangle \in D_a[((Q ; \gamma) \parallel A L)] \gamma \{x\} \sigma \), then there must be state-history pairs \( \langle \tau', k' \rangle \in D_a[Q] \sigma \) and \( \langle \nu', l' \rangle \in D_a[L] \sigma \) with consistent \( \langle \tau', k', \nu', l', A \rangle \) \( (k' \parallel l')c = (k \parallel l)^c \) and \( \tau' = \bot \).

By consistency, in particular comparability, we know that \( 1k'1 \geq 1ll'1 \). By definition of \( h \parallel A h_2 \) we have in general:

\( 1(k_1 \parallel A h_2)1 = \max(1h_1, 1h_2) \), so \( 1k'1 = \max(1k'1, 1l'1) = 1(k \parallel l')1 = 1(k \parallel l)1 = 1k1 \).

We also have \( ll'1 = ll1 \). If \( \nu' = \bot \), this follows from the same argument as above and if \( \nu' \neq \bot \) 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 \( \langle \tau, k \rangle \in D_a[Q] \sigma \) - contradiction. \( \square \)
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 \(L^i_{a+1}\) for any empty statement in \(L^i_a\). This may leave us with an empty statement in \(L^a\).

Now, if \(\alpha_\in k[n]\), then we can replace \(L^a\) by \(\alpha!v\) or \(\alpha?x_\alpha\). If \(k^c[n]=\emptyset\), we may replace \(L^a\) by \(\text{wait} 1\). 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 \(\sigma \neq \bot\) and \(\exists \alpha: R_\alpha \in t \land R_\alpha \not\in t'\). 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, Lehman and Pnueli’s method of linear history semantics had to be modified, 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).

Acknowledgements

The authors thank Mrs. E. van Thiel for her speedy and expert typing of this abstract, Mark N. Wegman for making us understand what it means to be a POPL referee [W86], and Gérard Berry, one of our referees, for bringing [SW79] to our attention.
References


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

Syntax of the language

\[
S ::= x := e \mid S_1 ; S_2 \mid GC \mid \nu S_1 \parallel A S_2 \mid [S]_A \downarrow - \\
\overline{g} ::= \alpha! e \mid \alpha? x \mid \text{wait } n \mid - \\
GC ::= [\bigwedge_{i=1}^{n} b_i ; \overline{g}_i \rightarrow S_i]
\]

\(\alpha\) 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

\(D = P (\text{State} \times \text{His})\)

\(\text{State} : \) states, valuating the program variables, or \(\perp\).

\(\text{His} : \) sequences of sets of records, consisting of communication records \(\alpha_v\) (value from channel \(\alpha\)) and readies \(R_\alpha\).

\(D [S] : \text{State} \rightarrow D\)

\(G [g] : P(g) \rightarrow \text{State} \rightarrow D\)

Define some auxiliary functions

\(\cdot^+ : (\text{State} \rightarrow D) \rightarrow (D \rightarrow D)\) by

\(\phi^+ (U) = \{ \langle \alpha, h_1, h_2 \rangle \mid \exists \sigma' \in \text{State} : \langle \alpha', h_1 \rangle \in U \wedge \langle \alpha, h_2 \rangle \in \phi(\sigma') \}\)

\(R((b_1 ; \overline{g}_1, \ldots, b_n ; \overline{g}_n) ; \sigma) = \{R_\alpha \mid \exists i : \sigma(b_i) = n \wedge (\overline{g}_i = \alpha! e \vee \overline{g}_i = \alpha? x)\},\)

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

\(\text{minwait} (G, \sigma) = \min(\{\text{waitvalue} (g, \sigma) \mid g \in G\}).\)

A set \(U\) is prefix-closed

\(\text{iff } \langle \alpha, h \rangle \in U \wedge h = h_1 h_2 \rightarrow \perp, h_1 \rangle \in U.\)

\(\text{Cl}(U)\) is the smallest prefix-closed set that contains \(U\).

\(G [\text{wait } n] G, \sigma = \text{Cl} (\langle \sigma, R (G, \sigma)^t \rangle \downarrow t = \text{minwait} (G, \sigma) \wedge t = \min(\sigma(d), 1))\)
\[
G[\alpha \varepsilon \sigma] = \text{Cl} \{<\sigma, R(G, \sigma)^t \{\alpha(\sigma)\}>|0 < t < \text{minwait}(G, \sigma) \vee t = 0\}
\]
\[
G[x \varepsilon \sigma] = \text{Cl} \{<\sigma, R(G, \sigma)^t \{\alpha_{\sigma} > |0 < t < \text{minwait}(G, \sigma) \vee t = 0, \alpha_{\sigma} \in \text{Values}\}
\]
\[
G[- \varepsilon \sigma] = \text{Cl} \{<\sigma, \emptyset >\}
\]
\[
G[b ; \emptyset] \sigma = \text{if } \sigma(b) = \text{true } \text{then } G[\emptyset] \sigma \text{ else } (<\emptyset, \emptyset ) \text{ fi}
\]
\[
\llbracket S \rrbracket \lambda = (<\emptyset, \emptyset >) \text{ for any } S \ (\lambda \text{ is the empty sequence})
\]
\[
\llbracket x := e \rrbracket \sigma = \text{Cl} \{\sigma(\sigma(e)/x), \emptyset >\}
\]
\[
\llbracket g \rrbracket \sigma = G[\emptyset] \{g \text{ if } g \neq -\}
\]
\[
\llbracket - \rrbracket \sigma = \text{Cl} \{<\sigma, \emptyset >\}
\]
\[
\llbracket S_1, S_2 \rrbracket = (\llbracket S_2 \rrbracket) \text{}(\llbracket S_1 \rrbracket)\sigma
\]
\[
\llbracket \{b_i : g_i \rightarrow S_i \} \rrbracket \sigma = \bigcup_{i=1}^{n}(\llbracket S_i \rrbracket)^\text{c}(G[\emptyset] \{g_i \}) \text{ if } \exists i: \sigma(b_i) = \text{true}
\]
\[
\quad = \text{Cl} \{<\sigma, \emptyset >\}
\quad \text{ otherwise}
\]
\[
D[\text{GC}] \sigma = \mu \phi \cdot \lambda \sigma. \text{ if } \exists i: \sigma(b_i) = \text{true } \text{then } \phi^\text{c}(D[\text{GC}] \sigma) \text{ else } \text{Cl} \{<\sigma, \emptyset >\} \text{ fi}
\]
\[
D[P_1 \llbracket A P_2] \sigma = \text{Cl} \{<\sigma_1, \sigma_2, h_1 \llbracket A h_2 >]\}
\quad \{\sigma_i, h_i > \in D[P_i] \sigma,
\quad \text{maximal}(h_1, h_2), \text{ synchronous } (h_1, h_2, A),
\quad \text{ comparable } (\sigma_1, h_1, \sigma_2, h_2)\}.
\]

where

\[
\sigma_1 \llbracket A \sigma_2 \text{ is a strict function defined by}
\]
\[
(\sigma_1 \llbracket A \sigma_2)(x) = \begin{cases} 
\sigma_1(x) & \text{if } \sigma_1(x) \neq \sigma(x) \\
\sigma_2(x) & \text{if } \sigma_2(x) \neq \sigma(x) \\
\sigma(x) & \text{otherwise}
\end{cases}
\]

\[
(h_1 \llbracket A h_2)[j] = (h_1[j] \cup h_2[j]) \setminus \{R_\alpha | \alpha \in A\}
\]

\[
\text{maximal}(h_1, h_2) \iff \forall j. \alpha: R_\alpha \not\in h_1[j] \cap h_2[j]
\]

\[
\text{synchronous}(h_1, h_2, A) \iff \forall j, \alpha, \gamma: \alpha \in A \rightarrow (\alpha, \gamma \in h_1[j] \leftrightarrow \alpha, \gamma \in h_2[j])
\]

\[
\text{comparable}(\sigma_1, h_1, \sigma_2, h_2) \iff \forall i \in \{1, 2\}: \sigma_i = \downarrow \rightarrow |h_{3-i} | \leq |h_i|
\]

\[
D[\llbracket S \rrbracket] \sigma = \text{Cl} \{<\sigma, h >\} \exists h': <\sigma, h'> \in D[\llbracket S \rrbracket] \sigma \wedge h \upharpoonright A = h', \text{ where } h' \upharpoonright A \text{ is the history that results after deleting all communications and readies on channels in } A \text{ from } h'.
\]
Define \textit{terminated} as the least predicate on \textit{stat} satisfying:
(i) \( \text{terminated} (-) \)
(ii) if \( \text{terminated}(S_1) \) and \( \text{terminated}(S_2) \) then \( \text{terminated}(S_1 \parallel S_2) \)
(iii) if \( \text{terminated}(S) \) then \( \text{terminated}([S]_A) \)

\textbf{Operational Semantics}

We do not bother to formally define the transition system but concentrate on the transition relation.

\( \rightarrow \subseteq (stat \times States) \times (N \times P(Chan \times Val)) \times (Stat \times States) \) is defined as the least relation satisfying the following set of axioms and rules:

(Notation: instead of \((P,a,n,c,P',\sigma') \in \rightarrow \) we write \((P,a) \rightarrow (P',\sigma').)\)

1a) \( (\text{wait } d,a) \xrightarrow{0,\sigma} (-,\sigma) \)

1b) \( (\text{wait } d,a) \xrightarrow{0,\sigma} (\text{wait } d',\sigma) \) where \( \sigma(d') = \sigma(d) - 1 \)

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

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

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

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

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

5a) \( (P,\sigma) \xrightarrow{L} (P',\sigma') \)

\( (P;Q,\sigma) \xrightarrow{L} (P';Q,\sigma') \)

5b) \( (Q,\sigma) \xrightarrow{L} (Q',\sigma') \) if \( \text{terminated}(P) \)

\( (P;Q,\sigma) \xrightarrow{L} (Q',\sigma') \)
6a) \( ([\Box b_i ; g_i \rightarrow P_i], \sigma) \xrightarrow{\frac{1,\sigma}{L}} (P_i, \sigma) \) if \( \sigma(b_i) = \text{true} \) and \( g_i = 0 \)

6b) \( ([\Box b_i ; g_i \rightarrow P_i], \sigma) \xrightarrow{\frac{L}{L}} (P_i, \sigma) \) if \( \sigma(b_i) = \text{true} \).

for all \( i : \sigma(b_i) = \text{true} \Rightarrow (g_i, \sigma) \xrightarrow{0,\sigma} (g'_i, \sigma) \)

6c) \( ([\Box b_i ; g_i \rightarrow P_i], \sigma) \xrightarrow{\frac{0,\sigma}{L}} (P_i, \sigma) \)

where \( g'_i = g_i \) if \( \sigma(b_i) = \text{false} \)

7a) \( (GS, \sigma) \xrightarrow{\frac{L}{L}} (P, \sigma') \) if \( \sigma(b_i) = \text{true for some i} \)

7b) \( (xGS, \sigma) \xrightarrow{\frac{1,\sigma}{L}} (P, \sigma') \) if \( \sigma(b_i) = \text{false for all i} \).

8a) \[
\begin{align*}
L \uparrow_A = L_1 \uparrow_A \rightarrow (P_1, \sigma_1), (P_2, \sigma_2) \xrightarrow{L_2} (P_1, \sigma_2) \rightarrow (P_2, \sigma_2) \\
\frac{L}{L} \quad \frac{L}{L} \\
\frac{(P_1, \sigma_1) \rightarrow (P_1, \sigma_2), L \uparrow A = \emptyset \text{ terminated } (P_2)}{\frac{(P_1 \uparrow A, P_2, \sigma) \rightarrow (P_1 \uparrow A, P_2, \sigma')}{(P_1 \uparrow A, P_2, \sigma) \rightarrow (P_1 \uparrow A, P_2, \sigma')}}
\end{align*}
\]

8b) \[
\begin{align*}
(P_1, \sigma) \xrightarrow{L} (P_1, \sigma'), L \uparrow A = \emptyset \text{ terminated } (P_2) \\
\frac{(P_1 \uparrow A, P_2, \sigma) \rightarrow (P_1 \uparrow A, P_2, \sigma')}{\frac{(P_1 \uparrow A, P_2, \sigma) \rightarrow (P_1 \uparrow A, P_2, \sigma')}{(P_1 \uparrow A, P_2, \sigma) \rightarrow (P_1 \uparrow A, P_2, \sigma')}}
\end{align*}
\]

9) \( (P, \sigma) \xrightarrow{\frac{L}{L}} (P', \sigma') \) where \( A' = \text{ichan}(P) \)

\( ([P]_A, \sigma) \rightarrow ([P]_A, \sigma') \)
<table>
<thead>
<tr>
<th>No.</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>85/01</td>
<td>R.H. Mak</td>
<td>The formal specification and derivation of CMOS-circuits</td>
</tr>
<tr>
<td>85/02</td>
<td>W.M.C.J. van Overveld</td>
<td>On arithmetic operations with M-out-of-N-codes</td>
</tr>
<tr>
<td>85/03</td>
<td>W.J.M. Lemmens</td>
<td>Use of a computer for evaluation of flow films</td>
</tr>
<tr>
<td>85/04</td>
<td>T. Verhoeoff, H.M.J.L. Schols</td>
<td>Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate</td>
</tr>
<tr>
<td>86/01</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems</td>
</tr>
<tr>
<td>86/02</td>
<td>G.A. Bussing, K.M. van He, M. Voorhoeve</td>
<td>ELISA, A language for formal specifications of information systems</td>
</tr>
<tr>
<td>86/03</td>
<td>Rob Hoogerwoord</td>
<td>Some reflections on the implementation of trace structures</td>
</tr>
<tr>
<td>86/04</td>
<td>G.J. Houben, J. Paredaens, K.M. van He</td>
<td>The partition of an information system in several parallel systems</td>
</tr>
<tr>
<td>86/05</td>
<td>Jan L.G. Dietz, Kees M. van He</td>
<td>A framework for the conceptual modeling of discrete dynamic systems</td>
</tr>
<tr>
<td>86/06</td>
<td>Tom Verhoeoff</td>
<td>Nondeterminism and divergence created by concealment in CSP</td>
</tr>
<tr>
<td>86/07</td>
<td>R. Gerth, L. Shira</td>
<td>On proving communication closedness of distributed layers</td>
</tr>
</tbody>
</table>
86/08  R. Koymans
     R.K. Shyamasundar
     W.P. de Roever
     R. Gerth
     S. Arun Kumar

Compositional semantics for real-time distributed computing (Inf.&Control 1987)

86/09  C. Huizing
     R. Gerth
     W.P. de Roever

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

86/10  J. Hooman

A compositional proof theory for real-time distributed message passing

86/11  W.P. de Roever

Questions to Robin Milner - A responder's commentary (IFIP86)

86/12  A. Boucher
     R. Gerth

A timed failure semantics for communicating processes

86/13  R. Gerth
     W.P. de Roever

Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)

86/14  R. Koymans

Specifying passing systems requires extending temporal logic

87/01  R. Gerth

On the existence of sound and complete axiomatizations of the monitor concept

87/02  Simon J. Klaver
     Chris F.M. Verberne

Federatieve Databases

87/03  G.J. Houben
     J. Paredaens

A formal approach distributed information systems

87/04  T. Verhoeff

Delay-insensitive codes - An overview
Available Reports from the Theoretical Computing Science Group

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Title</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIR83.1  R. Koymans, J. Vytopil, W.P. de Roever</td>
<td>Real-Time Programming and Synchronous Message passing (2nd ACM PODC)</td>
<td>EUT DESCARTES</td>
</tr>
<tr>
<td>TIR84.1  R. Gerth, W.P. de Roever</td>
<td>A Proof System for Concurrent Ada Programs (SCP4)</td>
<td>EUT DESCARTES</td>
</tr>
<tr>
<td>TIR84.2  R. Gerth</td>
<td>Transition Logic - how to reason about temporal properties in a compositional way (16th ACM FOCS)</td>
<td>EUT DESCARTES</td>
</tr>
<tr>
<td>TIR85.1  W.P. de Roever</td>
<td>The Quest for Compositionality - a survey of assertion-based proof systems for concurrent programs, Part I: Concurrency based on shared variables (IFIP85)</td>
<td>EUT DESCARTES</td>
</tr>
<tr>
<td>TIR85.2  O. Grünberg, N. Francez, J. Makowsky, W.P. de Roever</td>
<td>A proof-rule for fair termination of guarded commands (Inf. &amp; Control 1986)</td>
<td>EUT DESCARTES</td>
</tr>
<tr>
<td>TIR85.3  F.A. Stomp, W.P. de Roever, R. Gerth</td>
<td>The μ-calculus as an assertion language for fairness arguments (Inf. &amp; Control 1987)</td>
<td>EUT DESCARTES</td>
</tr>
<tr>
<td>TIR85.4  R. Koymans, W.P. de Roever</td>
<td>Examples of a Real-Time Temporal Logic Specification (LNCS207)</td>
<td>EUT DESCARTES</td>
</tr>
<tr>
<td>TIR86.1  R. Koymans, W.P. de Roever</td>
<td>Specifying Message Passing and Real-Time Systems (extended abstract)</td>
<td>CSN86/01</td>
</tr>
<tr>
<td>TIR86.2  J. Hooman, W.P. de Roever</td>
<td>The Quest goes on: A Survey of Proof Systems for Partial Correctness of CSP (LNCS227)</td>
<td>EUT-Report 86-WSK-01</td>
</tr>
</tbody>
</table>
TIR86.3  R. Gerth, L. Shira  On Proving Communication Closedness of Distributed Layers (LNCS236)  CSN86/07


TIR86.5  C. Huizing, R. Gerth, W.P. de Roever  Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language  CSN86/09 PE.01

TIR86.6  J. Hooman  A Compositional Proof Theory for Real-Time Distributed Message Passing  CSN86/10 TR.4-1-1(1)

TIR86.7  W.P. de Roever  Questions to Robin Milner - A Responder’s Commentary (IFIP86)  CSN86/11

TIR86.8  A. Boucher, R. Gerth  A Timed Failure Semantics for Communicating Processes  CSN86/12 TR.4-4(1)

TIR86.9  R. Gerth, W.P. de Roever  Proving Monitors Revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)  CSN86/13

TIR86.10  R.Koymans  Specifying Message Passing Systems Requires Extending Temporal Logic  CSN86/14 PE.02

TIR87.1  R. Gerth  On the existence of sound and complete axiomatizations of the monitor concept  CSN87/01