Original citation:

Permanent WRAP url:
http://wrap.warwick.ac.uk/60817

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:
The version presented in WRAP is the published version or, version of record, and may be cited as it appears here. For more information, please contact the WRAP Team at: publications@warwick.ac.uk

http://wrap.warwick.ac.uk/
A semantic model for developing and justifying specifications of communicating real-time processes is proposed. The semantics is state-based and compositional. The basic semantic objects are timed-observations of values of program variables, where time is assumed to be in the domain of real numbers. The internal actions and communications of a command are treated in a uniform way to obtain a simple semantic domain. An ordering on this domain for information approximation is developed. The proposed semantics models termination, failure, divergence, deadlock, and starvation, and supports an arbitrary degree of parallelism.

† This work was supported by research grant GR/D 73881 from the Science and Engineering Research Council
1 Introduction

Real-time programs are the least well understood of concurrent programs: apart from being subject to all the usual problems of concurrency and communication, they must interact with a variety of agents at points in the execution that have a specified ordering relation with time. So, for example, the familiar partial order over the execution of the statements of a concurrent program must be made more elaborate to accommodate time-ordering and this requires a semantics that can account for programs whose execution may be constrained by limitations on the availability of resources. We illustrate this aspect of real-time programs with an example.

Consider the network $P :: P_1 || P_2 || P_3$ where $P_1$ and $P_2$ are processes

$$P_1 :: *[\ldots P_2!e \ldots]$$

$$P_2 :: *[P_3?x \rightarrow P_1?y; \text{delay } d; P_3?(f(x,y))]$$

and $P_3$ is a physical device with the behaviour

$$P_3 :: *[\text{true} \rightarrow P_2!a; \ldots; P_3?b]$$

where the 'delay d' command has the effect of suspending execution for exactly $d$ time units. Assume now that this program has been designed for a machine with two identical processors. Under what conditions can the correctness of the program be guaranteed if only one processor is available for its execution? With the existing techniques for analysis and verification, answering this question would require a complete re-examination of the program even though it is not the program but the execution environment that has changed.

In practice, real-time programs are far more complicated than this simple example and have many more processes. A variety of questions may then be raised: eg, what is the minimum number of processors for ensuring the correctness of a particular real-time program and what kind of process structure will ensure that a program is insensitive to bounded variations in the number of processors? Our concern here is to develop a model of real-time programs which addresses these questions. More generally, the introduction of time into the semantics can be viewed as modelling the execution of programs in a domain of limited resources of which the processor, which controls the execution speed of a program, is merely one.

We first consider a basic semantics for imperative real-time programs with explicit communication commands. The semantics is intended to support a real-time program specification and design method which is relatively language-independent. We therefore use a generic language L which represents a class of distributed programming languages whose primitive commands are left unspecified: thus the definition of L consists largely of definitions of combinators and their semantics. Where more concrete examples are needed, we shall use a slightly extended version of Hoare's communicating sequential processes [1].

To preserve expressive power, our specifications will be statements of a first-order logic and thus will not be executable. At the level of the semantics, there is

---

1 An agent may be another program, a person, or part of a physical system.
just one concrete event: assignment to a variable. This distinguishes the model from trace-based and algebraic models for concurrency which make use of an alphabet of abstract events or actions (e.g., [2,3]). Since we deal with semantics of real-time programs, a timed observation of an assignment is the basic semantic object and we shall make use of a time domain which is the set of non-negative real numbers.

2 The Languages L and XCSP

L is a generic language representing a wide class of distributed programming languages: its primitive commands are not specified. Although any particular version of L may have other combinators, we shall consider just three combinators and these are given below using variables \( C, C_1, \) and \( C_2 \) which range over the set of all commands of L.

- Sequential Composition: \( C_1; C_2 \)
- Parallel Composition: \( C_1 \parallel C_2 \)
- Program Constructor: \( \text{prog} \ C \ \text{end} \)

A command is independently executable only if it is of the form

\[
\text{prog} \ C \ \text{end}
\]

Such a command is called a program: an execution of this command is just an execution of the command \( C \). An execution of any other kind of command must be a part of the execution of some program command. The meaning of sequential composition is as usual: execution of \( C_1 \), and if it terminates, followed by an execution of \( C_2 \). The command \( C_1 \parallel \ldots \parallel C_n \) is called a network and the commands \( C_1, \ldots, C_n \) are the processes of the network. All the processes of a network start executing at the same time and the network terminates when all its processes have terminated. We assume that parallelism is explicit in L and that it is always specified by the combinator \( \parallel \).

XCSP

The language XCSP is a subset of CSP [1] augmented with asynchronous communication commands. The abstract syntax of XCSP and an informal description of its semantics are given below.

Let \( P \) range over programs, \( C, C_1, C_2, C_i \) over commands, \( IO \) over input-output commands, \( D \) over channel names, \( e \) over expressions, \( x \) over program variables, \( b \) over boolean expressions, and \( g, g_i \) over guards. The symbol \( n \) ranges over positive integers.

\[
P ::= \text{prog} \ C \ \text{end}
\]

\[
C ::= \text{skip} \mid x := e \mid \text{IO} \mid C_1; C_2 \mid \bigwedge_{i=1}^{n} g_i \rightarrow C_i \mid \bigwedge_{i=1}^{n} g_i \rightarrow C_i \mid C_1 \parallel C_2
\]

\[
IO ::= D?x \mid D?!x \mid D!e \mid D!le
\]

\[
g ::= b \mid \text{IO} \mid b; \text{IO}
\]
The introduction of asynchronous input $D??x$ and asynchronous output $D!!e$
leads to the following changes from CSP.

- A channel may be either synchronous or asynchronous. A process implicitly
  declares the type of its channels; the channel $D$ in the IO commands $D?x$ and
  $Dle$ is synchronous while in $D??x$ and $D!!e$ it is asynchronous.

- If the expression $e$ is evaluable, the action of $D!!e$ is to unconditionally place
  the value of $e$ on the asynchronous channel $D$.

- The effect of the asynchronous input command $D??x$ is the same as that of
  the synchronous input command $D?x$, but on an asynchronous channel.

- A guarded command whose guard contains an asynchronous output command
  to a channel $D$ can be selected for execution if
    a) the boolean part of the guard, if present, evaluates to $true$, and
    b) the process containing the corresponding input command for channel $D$ has
       neither terminated nor failed.

3 Semantics of $L$

3.1 Notation

We shall make extensive use of sequences and so we introduce some notation
for sequences. Finite sequences are represented by the symbols $s, s_1$, and $s_2$.

$< a_1, a_2, \ldots, a_n >$ a finite sequence of the elements $a_1, a_2, \ldots, a_n$, in that order;
$\lambda$ the empty sequence
$\#s$ the length of $s$ ($\#\lambda = 0$)
$s[i]$ the $i$th element of $s$
$s_1 \triangleleft s_2$ $s_1$ is an initial subsequence of $s_2$
$s_1 \triangleleft s_2$ $s_1$ is a proper initial subsequence of $s_2$ ($s_1 \triangleleft s_2$ and $s_1 \neq s_2$)
$s_1 \ast s_2$ the catenation of $s_1$ and $s_2$
$< 1, 3, 4 > \triangleleft < 2, 6 > = < 1, 3, 4, 2, 6 >$
first($s$) the first element of $s$
last($s$) the last element of $s$
frest($s$) the sequence obtained from $s$ by removing
      the first element
lrest($s$) the sequence obtained from $s$ by removing
      the last element
rest($k, s$) the sequence obtained from $s$ by removing the first $k$
      elements, if $k < \#s$, and $\lambda$ otherwise
range($s$) the range of the sequence $s$
      i.e., the set of the elements in the sequence $s$

An infinite sequence is written as $< a_1, a_2, \ldots >$, i.e., without a last element,
and the operators listed above can be extended to have infinite sequences as their
arguments. Some of the operators then become partial functions: the expressions
The symbol $\omega$ represents the set of all nonnegative integers; $[i, j]$ stands for the closed interval \{k $| i \leq k \leq j$\} of a partial order $\leq$. The symbol $T$ represents the set of nonnegative real numbers with a least element 0; the elements of this set represent points of time.

The semantics of $L$ is defined in terms of variables, observations and behaviours. Informally, a variable in the semantics ranges over the sequences of values a program variable might take during an execution; an observation is a snapshot of the variables (ie, the sequences denoting their values) at a particular point in the execution and a behaviour, which is a sequence of observations, is a mathematical representation of the execution. To avoid confusion, we shall refer to a value of a variable in the semantics as an 'observed value' of the variable.

Sequences are used to represent variables and behaviours. For clarity, when the functions first, last, frest, and lrest are applied to variables they will be written as firstOb, lastOb, frestOb, and lrestOb respectively, and when applied to behaviours as firstBv, lastBv, frestBv, and lrestBv respectively.

**Example 1** Execution of the command

$$x := x + 2; y := y + x$$

starting in the state $x = 3 \land y = 4$ is represented by the following behaviour:

<table>
<thead>
<tr>
<th>Observations</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$&lt; 3 &gt;$</td>
<td>$&lt; 3,5 &gt;$</td>
<td>$&lt; 3,5 &gt;$</td>
</tr>
<tr>
<td>$y$</td>
<td>$&lt; 4 &gt;$</td>
<td>$&lt; 4 &gt;$</td>
<td>$&lt; 4,9 &gt;$</td>
</tr>
</tbody>
</table>

where ellipsis indicates that the last observation is repeated indefinitely. A behaviour, such as the one above, whose observations eventually become constant may represent a terminated, failed, or deadlocked execution.

The variables of a command $C$ are partitioned into the set $LCL(C)$ of local variables and the set $CMN(C)$ of communication variables and, correspondingly, the actions of the command can be internal computations or communications. Communication between two components of a command is considered to be an internal computation of the command.

The semantics of a command $C$ in $L$ is described by a triple

$$< LCL(C), CMN(C), \Gamma(C) >$$

where $\Gamma(C)$ is the semantic function of $C$. $LCL(C)$, or $CMN(C)$, or both may be empty. $VAR(C)$ is the union of $LCL(C)$ and $CMN(C)$.
Example 2 If execution of the command of Example 1 is preceded by the receipt of the integer value 5 in the communication command $D?x$, ie

$$D?x; x := x + 1; y := y + x$$

then the following behaviour will be observed:

<table>
<thead>
<tr>
<th>Observations</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$&lt;3&gt;$</td>
<td>$&lt;3,5&gt;$</td>
<td>$&lt;3,5,6&gt;$</td>
<td>$&lt;3,5,6&gt;$</td>
</tr>
<tr>
<td>$y$</td>
<td>$&lt;4&gt;$</td>
<td>$&lt;4&gt;$</td>
<td>$&lt;4&gt;$</td>
<td>$&lt;4,10&gt;$</td>
</tr>
<tr>
<td>$?D$</td>
<td>$\lambda$</td>
<td>$&lt;5&gt;$</td>
<td>$&lt;5&gt;$</td>
<td>$&lt;5&gt;$</td>
</tr>
</tbody>
</table>

A program may contain a command that does not receive values in all of its local variables at the start of its execution and the initial observed value of each of these variables is undefined, represented by $\lambda$.

Example 3 Consider the command

$$D?x; x := x + 1; y := y + x$$

An execution of the command $D?x$ which receives the value 5 on channel $D$ and has initially undefined values in $x$ and $y$ is described by the following behaviour.

<table>
<thead>
<tr>
<th>Observations</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\lambda$</td>
<td>$&lt;5&gt;$</td>
<td>$&lt;5,6&gt;$</td>
<td>$&lt;5,6,\perp&gt;$</td>
</tr>
<tr>
<td>$y$</td>
<td>$\lambda$</td>
<td>$\lambda$</td>
<td>$\lambda$</td>
<td>$&lt;\perp&gt;$</td>
</tr>
<tr>
<td>$?D$</td>
<td>$\lambda$</td>
<td>$&lt;5&gt;$</td>
<td>$&lt;5&gt;$</td>
<td>$&lt;\perp&gt;$</td>
</tr>
</tbody>
</table>

The last command $y := y + x$ fails because the initial value of $y$ is undefined. In the semantics we shall assume that at this point failure is induced in $x$ and $?D$ as well.

A communication variable is called synchronous or asynchronous depending on the kind of channel on which it sends or receives values. A variable $v \in CMN(C)$ is called an input variable if it is connected to the output end of a communication channel, and an output variable if it is connected to the input end of a channel. The identifier forms $?v$, ??$v$, !$v$, and !!$v$ represent a synchronous input variable, an asynchronous input variable, a synchronous output variable, and an asynchronous output variable, respectively. The semantics of $L$ involves the following sets of variables:

- $VAR$ the set of all variables
- $SYN$ the set of variables of the form $?v$ or !$v$
- $ASYN$ the set of variables of the form ??$v$ or !!$v$
- $IN$ the set of variables of the form $?v$ or ??$v$
- $OUT$ the set of variables of the form !$v$ or !!$v$
- $CMP$ the set of variables which are not of the forms $?v$, ??$v$, !$v$, or !!$v$

\[\text{It is shown later that local variables may also have identifiers of these forms}\]
The following functions and predicates are defined over variables of the form ?v, ??v, !v, and !!v.

a) \text{Dual}(?v) = !v  
\text{b) Dual}(??v) = !!v  
\text{c) Dual}(!v) = ?v  
\text{d) Dual}(!!v) = ??v  
\text{e) IsIn}(u) \triangleq u = ?v \lor u = ??v  
\text{f) IsOut}(u) \triangleq u = !v \lor u = !!v  
\text{g) Syn}(u) \triangleq u = ?v \lor u = !v  
\text{h) Asyn}(u) \triangleq u = ??v \lor u = !!v  

Let X be a set of communication variables such that if \( v \in X \) then \text{Dual}(v) \notin X. The function \text{Dual} is extended to have all such sets in its domain:

\[
\text{Dual}(X) = \{ y \mid \exists x \in X : y = \text{Dual}(x) \}
\]

Obviously \( \text{Dual}(\emptyset) = \emptyset \).

3.3 Observations and behaviours

Let \( V \) be the value space of the L programs. We assume that \( V \) is countable and equipped with an equality (=) relation which is the minimum reflexive relation on \( V \). The distinguished values \( \bot \) and \( \delta \) are assigned respectively to the variables of a command which has failed or deadlocked. The conditions for deadlock or failure to occur are defined by the semantics of the particular programming language: for example, if the expression \( e \) is a partial function then the assignment command \texttt{x := e} fails when its execution starts in a state in which \( e \) is undefined. For our purposes, we assume that whenever a command fails, each of its variables is assigned the value \( \bot \) (\( \bot \in V \)). In the same way, the value \( \delta \) is assigned to each variable of a deadlocked command.

Let \( V^\star \) and \( V^+ \) be respectively the set of all finite sequences and the set of all (finite and infinite) sequences over \( V \), both including the empty sequence \( \lambda \). Define \[
V^\dagger = \{ s^* < d \mid s \in V^\star \land d \in \{ \bot, \delta \} \}
\]

Then the space \( W \) of 'observed values' is defined as \[
W = V^+ \cup V^\dagger
\]

The function \text{Data} is defined over \( W \):

\[
\text{Data}(v) = \begin{cases} 
\text{last}(v) & \text{if } v \in V^\dagger \\
v & \text{otherwise}
\end{cases}
\]

Thus \text{Data} returns the sequence of values in \( V \) held by a variable.
Definition 3.1

An observation $\theta$ over the set $VAR(C)$ of a command $C$ is a pair $(t, f)$ where $t \in T$ and $f$ is a function $f : VAR(C) \rightarrow W$ such that the following conditions hold:

a) $\forall v \in VAR(C) : \bot \in range(f(v)) \Rightarrow (\forall v \in VAR(C) : \bot \in range(f(v)) \wedge t = \sigma)$

b) $\forall v \in VAR(C) : \delta \in range(f(v)) \Rightarrow (\forall v \in VAR(C) : \delta \in range(f(v))$)

where $\sigma$ is an arbitrarily large number. It is formally described later.

If the domain of $f$ is empty, i.e., $VAR(C) = \emptyset$, then the function $f$ is represented by the symbol $\Theta$ and the observation $(t, \Theta)$ is called the empty observation at time $t$.

The components $t$ and $f$ of $\theta$ in this definition are called the time and the function components respectively, given by the expressions $Time(\theta)$ and $Function(\theta)$. Two observations $\theta$ and $\psi$ are equal, $\theta = \psi$, if they have the same time component as well as the same function component.

The expressions $VAR(\theta)$, $LCL(\theta)$, and $CMN(\theta)$ evaluate to the sets represented by $VAR(C)$, $LCL(C)$, and $CMN(C)$ respectively. We have

$\forall t \in T : VAR((t, \Theta)) = \emptyset$

The set of all observations of $C$ is denoted by $O(C)$.

The restriction of an observation $\theta \triangleq (t, f)$ to a subset $X$ of $VAR(\theta)$ is an observation $(t, f')$ where $f = \Theta$ if $X = \emptyset$; otherwise $f'$ is the restriction of the function $f$ to the set $X$.

The predicates Failed, Deadlocked, and Broken over observations are defined as

Failed($\theta$) $\triangleq (\forall v \in VAR(\theta) : lastOb(f(v)) = \bot)$
Deadlocked($\theta$) $\triangleq (\forall v \in VAR(\theta) : lastOb(f(v)) = \delta)$
Broken($\theta$) $\triangleq$ Failed($\theta$) $\vee$ Deadlocked($\theta$)

Together with Definition 3.1, this means that the failure of any component of a command induces the value $\bot$ in all the variables of the command; similarly the value $\delta$ is induced in the case of a deadlock.

A behaviour of a command $C$ is an formal representation of an execution of $C$ and is described by an infinite sequence of observations.

Initial Observation $\theta_0$: An execution of a command is assumed to start at time 0, and so the time component of $\theta_0$ is 0. The function component of $\theta_0$ gives the initial values of the variables of $C$. Since communication variables do not receive values by sequential composition, any such variable should have $\lambda$, $< \bot>$, or $< \delta>$ as its initial observed value.

Observation $\theta_k$, $k > 0$: If $C$ has either terminated or broken at time $t_{k-1}$ then $\theta_k = \theta_{k-1}$. Otherwise, $C$ executes at least one primitive command after time $t_{k-1}$. Then observation $\theta_k$ is made at the time $t_k > t_{k-1}$ of termination, failure, or deadlock of any primitive command of $C$, provided that a primitive communicating component of $C$ has not terminated, failed, or deadlocked at time $t$ such that $t_{k-1} < t < t_k$. Only those variables that belong to the primitive commands that terminate, fail, or
deadlock at time $t_k$ acquire new observed values at time $t_k$; the new observed value of such a variable is obtained by appending the current value of the variable to the preceding observed value of the variable.

We now give the formal definition of the behaviour of a command.

**Definition 3.2** Consider a command $C$. Let $O(C)$ be the set of all observations of $C$, and $U$ the set of all one-element sequences over the set $V \cup \{\delta\}$. A *behaviour* of $C$ is a function $F : \omega \rightarrow O(C)$, and is represented by an infinite sequence

$$< \theta_0, \theta_1, \theta_2, \ldots >$$

where $[\forall k \in \omega : F(k) = \theta_k]$. An element $\theta_k$ of this sequence is an observation $(t_k, f_k)$. The function $F$ has the following properties:

- a) $t_0 = 0$
- b) $\forall v \in LCL(C) : f_0(v) = \lambda \vee f_0(v) \in V$
- c) $\forall v \in CMN(C) : f_0(v) \in \{\lambda, \delta\}$
- d) $\forall k \in \omega : \forall v \in VAR(C) : \{3s \in U \cup \{\lambda\} : f_{k+1}(v) = f_k(v) \wedge t_k < t_{k+1}\}$
- e) $\forall k \in \omega : (f_k = f_{k+1} \vee \text{Broken}(\theta_k)) \Rightarrow [\forall j \in \omega : j > k \Rightarrow \theta_j = \theta_{k+1}]$
- f) $\forall k \in \omega : \theta_k \neq \theta_{k+1} \Rightarrow t_k < t_{k+1} \wedge f_k \neq f_{k+1}$

The first five properties have already been explained. Property (f) states that two successive observations in a behaviour either differ in both of their components or are equal. Together with properties (d) and (e), this shows that time increases from one observation to the next unless the observations become constant.

If there is some command $C$ such that $VAR(C) = \emptyset$, then any behaviour of $C$ is of the form

$$< (0, \emptyset), (t, \emptyset), (t, \emptyset), \ldots >$$

denoted by $\Omega_t$, where $t \in T - \{0\}$.

The notations $VAR(F)$, $LCL(F)$, and $CMN(F)$ stand for the sets $VAR(C)$, $LCL(C)$, and $CMN(C)$ respectively. We represent the $i$th element of $F$ by $F_i$ or by the pair $(t_i, f_i)$; if ambiguity does not arise we shall also use the form $(t_i, f_i)$. The set of all behaviours of a command $C$ will be denoted by $\beta(C)$. This set exhibits all possible initial assignments to the variables of $C$ and whenever an assignment to an input communication variable is performed, all possible values are considered for the assignment. In other words, $\beta(C)$ can be assumed to represent a forest; the roots of its trees correspond to the possible initial valuations of the variables of $C$, and the descendents of a node of a tree corresponds to the possible initial valuations of a communication variable.

A behaviour of a command $C$ will also be referred to as a behaviour over the set $VAR(C)$. The set of all possible behaviours over a subset $X$ of $VAR$ will be denoted by $\beta(X)$. Note that $\beta(\emptyset)$ is the set

$$\{\Omega_t \mid t \in T - \{0\}\}$$

The *restriction of a behaviour* $F$ to a subset $X$ of $VAR(F)$, written $F \uparrow X$, is an $\omega$-sequence whose $i$th observation, for all $i \in \omega$, is the restriction of $F_i$ to the set

8
It should be noted that a restriction of a behaviour is not necessarily another behaviour and that \( F \uparrow \emptyset \) is not necessarily the behaviour \( \Omega_\ell \). We show how the restrictions of behaviours can be converted into behaviours.

Consider an \( \omega \)-sequence \( S \) of observations over a set \( X \) of variables. \( S \) is said to be compatible with a behaviour \( F \), written \( S \text{ Compat } F \), if either \( S = F \) or the following conditions hold:

\[
\begin{align*}
\text{a)} & \quad \text{VAR}(F) = X \\
\text{b)} & \quad \forall i \in \omega : \text{Time}(S[i]) < \text{Time}(S[i+1]) \\
\text{c)} & \quad \forall i \in \omega : \exists j \in \omega : \exists k \in \omega : k > j > i \land F_i = S[j] \land F_{i+1} = S[k] \\
& \quad \land \forall l \in [j+1,k-1] : \text{Function}(S[l]) = \text{Function}(S[j])
\end{align*}
\]

Informally, an \( \omega \)-sequence \( S \) of observations compatible with a behaviour \( F \) is obtained by inserting an observation \( (t, f) \) into a sequence \( S' \) which is either \( F \) or an \( \omega \)-sequence compatible with \( F \), so that

\[
\text{Time}(F_i) < t < \text{Time}(F_{i+1}) \land \text{Function}(F_i) = f
\]

The relation \( \text{Compat} \) is partial order on the set of behaviours over a set of variables.

If \( F \) is a behaviour of a command \( C \), and \( X \) a non-empty subset of \( \text{VAR}(C) \), then we say that the behaviour \( H \), where

\[
F \uparrow X \text{ Compat } H
\]

is the corresponding behaviour of the activity of \( C \) in \( X \).

Consider a network

\[ P :: P_1 \| \ldots \| P_n \]

If \( P_i \), for some \( i \) in \([1,n]\), starves, then there is a behaviour \( F \) of \( P \), and a behaviour \( H \) of \( P_i \) such that

\[
(F \uparrow \text{VAR}(P_i) \text{ Compat } H) \land \text{Terminated}(H)
\]

However, starvation can be distinguished from termination only if the termination of a command \( C \) is indicated by appending a certain distinguished value to the observed values of all communication variables of \( C \) when \( C \) has terminated. Thus, our semantics would require only a minor extension to model starvation.

Let \( s \triangleq (t_0, f_0), \ldots, (t_k, f_k) \) > be an initial subsequence of a behaviour \( F \). We say that \( s \) is the executed part of \( F \), represented by \( \text{Exec}(F) \) if the following conditions hold:

\[
\begin{align*}
\text{a)} & \quad k > 0 \Rightarrow f_{k-1} \neq f_k \\
\text{b)} & \quad (\forall j \in \omega : j > k \Rightarrow f_j = f_k)
\end{align*}
\]

If there is no \( k \) satisfying the above conditions then \( \text{Exec}(F) = F \). The sequence \( \text{Exec}(F) \) contains all the information of the execution described by \( F \).

We now define a few predicates over behaviours.

\[
\begin{align*}
\text{Stopped}(F) & \triangleq \text{Exec}(F) \neq F \\
\text{Failed}(F) & \triangleq (\exists k \in \omega : \text{Failed}(F_k))
\end{align*}
\]
Deadlocked(F) \triangleq (\exists k \in \omega : \text{Deadlocked}(F_k))

Broken(F) \triangleq \text{Failed}(F) \lor \text{Deadlocked}(F)

Terminated(F) \triangleq \text{Stopped}(F) \land \neg \text{Broken}(F)

If \text{Stopped}(F) is true then \text{Exec}(F) is finite and the execution described by F stops (i.e., successfully terminates, or fails, or deadlocks). When the execution fails or deadlocks we have \text{Broken}(F) = \text{true}. Note that the predicates \text{Failed} and \text{Deadlocked} are mutually exclusive.

The function \text{Comptime} is defined for F if \text{Stopped}(F) is true and in that case it is the time component of the last observation of \text{Exec}(F). This function returns the computation time of a finite execution. \text{Timeset}(F) is the set of all time components of the observations of F.

The predicates \text{Internal} and \text{External} over the set \beta(C) \times \omega are defined as

\begin{align*}
a) \text{Internal}(F, k) & \triangleq k \geq 1 \land \exists v \in \text{LCL}(C) : f_k(v) \neq f_{k-1}(v) \\
b) \text{External}(F, k) & \triangleq k \geq 1 \land \exists v \in \text{CMN}(C) : f_k(v) \neq f_{k-1}(v)
\end{align*}

These two predicates need not be mutually exclusive. \text{Internal}(F, k) implies that the kth observation of F has recorded an assignment to a local variable. Similarly, \text{External}(F, k) implies that the value of a communication variable has changed in the kth observation of F.

We now give a formal description of the time \sigma. For all behaviours F,

\[ \forall i \in \omega : \text{Time}(F_i) \leq \sigma \]

This description of \sigma corresponds to the intuitive idea that a failed behaviour is one which can produce a value after an arbitrarily long time.

An abstraction function \Theta is a (partial) function from the set \beta(C) \times O(C) into \beta(C). Let \( F \in \beta(C) \) and \( \Theta \in O(C) \). Then \( F \Theta \) is defined if

\[ \exists k \in \omega : F_k = \Theta \land \text{Internal}(F, k) \]

and is a behaviour H of C which satisfies the following condition:

\[ \forall i \in \omega : (i < k \Rightarrow h_i = f_i) \]
\[ \land \forall v \in \text{LCL}(C) : f_{k+1}(v) = f_k(v) \Rightarrow [\forall i \geq k : h_i(v) = f_{i+1}(v)] \]
\[ \land f_{k+1}(v) \neq f_k(v) \Rightarrow [\forall i \geq k : h_i(v) = f_{k-1}(v) \ast \text{rest}(\#f_k(v), f_{i+1}(v))] \]

It can be shown that \Theta is associative, i.e., if \( \Theta \) and \( \Psi \) are behaviours then

\[ ((F \Theta \Theta) \Theta \Psi) = ((F \Theta \Psi) \Theta \Theta) \]

Let \( X \triangleq \{\theta_1, \ldots, \theta_n\} \) be a set of observations of F. Then \( F \Theta \) represents

\[ ((F \Theta \Theta) \Theta \{X - \{\theta\}\}) \]

for any \( \theta \) in \( X \), where \( F \Theta \Theta = F \). The abstraction function hides the observations of the internal actions from a behaviour.
Consider a behaviour $F$ such that

$$\neg \text{Stopped}(F) \land [\exists k \in \omega : (k = 0 \lor \text{External}(F, k)) \land \forall i > k : \neg \text{External}(F, i)]$$

where the behaviour $F$ diverges after the $k$th observation (i.e. engages indefinitely in internal actions). We say that the behaviour $F$ diverges. Since only the communicated values and the final result of a computation are of interest in a real-time system, divergence can alternatively be described as a failure. Such a representation of $F$ is given below.

Consider a behaviour $H$ such that

$$\text{VAR}(H) = \text{VAR}(F) \land [\forall i \in [0, k] : H_i = F_i]$$
$$\land H_k = (\varepsilon, \text{Function}(F_{k-1}))^* < \perp >$$

$H$ is said to be the failure representation of $F$.

Abstraction over the set of all behaviours is denoted by $\equiv$ and is recursively defined as follows:

Let $F$ and $H$ be behaviours. Then $H \leq F$ if $F$ diverges and $H$ is the failure representation of $F$ or if

$$H = F \lor [\exists k \in \omega : H = F \equiv k \lor H \equiv (F \equiv k)]$$

The behaviour $\Omega_i$ has no abstraction other than itself. It is easily seen that the relation $\leq$ is a partial order over behaviours.

**Definition 3.3** A behaviour $F$ is said to be observationally equivalent to another behaviour $G$, written $F \approx G$, if there exists a behaviour $H$ such that $H \equiv F \land H \equiv G$.

**Lemma 1** Let $H, H'$, and $G$ be behaviours. Then

$$H \leq G \land H' \leq G \Rightarrow H \approx H'$$

**Proof:** Case 1. $G$ does not diverge:

The hypothesis implies that there are (possibly empty) sets $X$ and $Y$ of observations such that

$$H = G \circ X \land H' = G \circ Y$$

Let $Z = X \cup Y$. Then, by the definition of $\leq$,

$$G \circ Z \leq G \circ X \land G \circ Z \leq G \circ Y$$

or, $G \circ Z \equiv H \land G \circ Z \equiv H'$. Thus $H \approx H'$ (by the definition of $\approx$).

Case 2. $G$ diverges:

We have $H = G' \circ X \land H' = G' \circ Y$ where $G' = G$ or $G'$ is the failure representation of $G$. Obviously, as in case 1, it follows that $H \approx H'$.

**Theorem 3.1** The relation $\approx$ is an equivalence relation on behaviours on a set of behaviours.
Proof: The reflexivity and symmetry of $\approx$ follow directly from its definition.

Transitivity: Let $F, G,$ and $H$ be behaviours such that $F \approx G$ and $G \approx H$. Then there exist behaviours $H'$ and $H''$ satisfying

$$H' \preceq F \land H' \preceq G \land H'' \preceq G \land H' \preceq H$$

But $H' \preceq G \land H'' \preceq G$ implies that there exists a behaviour $J$ such that

$$J \preceq H' \land J \preceq H''$$

Formulae 1 and 2, together with the transitivity of $\preceq$ give

$$J \preceq F \land J \preceq H$$

which, by the definition of $\approx$ implies $F \approx H$.

4 The semantic domain

We now define the basic semantic domain $\mathcal{D}$. Let

$$X \subseteq VAR \land \gamma \subseteq \beta(X) \land \gamma \neq \emptyset$$

Thus $\gamma$ is a set of behaviours over $X$.

The closure of $\gamma$ by abstraction is

$$\text{Cl}_A(\gamma) = \gamma \cup \{F \in \beta(X) \mid \exists G \in \gamma : F \preceq G\}$$

and $\gamma$ is said to be closed by abstraction, written $\text{Closed}_A(\gamma)$, if

$$\text{Cl}_A(\gamma) = \gamma$$

It can be observed that if $X = \emptyset$, then $\gamma$ is closed under abstraction.

The domain $\mathcal{D}$ is defined by:

$$\mathcal{D} = \{\gamma \mid \exists X \subseteq VAR : \gamma \subseteq \beta(X) \land \gamma \neq \emptyset \land \text{Closed}_A(\gamma)\}$$

Thus $\mathcal{D}$ consists of all sets of behaviours which are closed under abstraction.

An ordering on $\mathcal{D}$ for information approximation can be defined. We begin with a relation $\preceq$ on observations.

Let $\theta$ and $\psi$ be observations. Then $\theta \preceq \psi$ if

$$VAR(\theta) = VAR(\psi)$$

$$\land (\theta = \psi \lor (\text{Failed}(\theta) \land \neg\text{Failed}(\psi))$$

$$\land (\forall v \in VAR(\theta) : \text{Function}(\theta)(v) = \text{irestOb}(\text{Function}(\psi)(v))^\perp < \perp >))$$
Note that in the case $\theta \leq \psi \wedge \theta \neq \psi$, we have $\text{Time}(\theta) = \varepsilon$ and therefore we do not need an explicit temporal ordering of $\theta$ and $\psi$.

This relation on observations can be used to define a partial order $\sqsubseteq$ on behaviours.

Let $F$ and $G$ be behaviours over the same set of variables. Then $F \sqsubseteq G$ if

$$F_0 = G_0 \wedge (\text{Failed}(F_1) \lor [\forall i \in \omega : F_i \leq G_i])$$

**Theorem 4.1** The relation $\sqsubseteq$ on the set $\beta(X)$ of behaviours over variable set $X$ is a partial order$^3$.

**Proof:** Reflexivity of $\sqsubseteq$ follows directly from its definition.

**Antisymmetry:** Let $F$ and $G$ be in $\beta(X)$ and $F \sqsubseteq G \wedge G \sqsubseteq F$. We have $F_0 = G_0$ (definition of $\sqsubseteq$).

**Case 1.** $\text{Failed}(F_1)$:
Then, $\forall i \geq 2 : F_i = F_1$. If $\neg \text{Failed}(G_1)$ then $\forall i \in \omega : G_i \leq F_i$ (because $G \sqsubseteq F$).
But, by the definition of $\sqsubseteq$, $G_1 \leq F_1$ does not hold. It follows from this contradiction that $\text{Failed}(G_1)$. Then $G_1 = F_1$ (because $F_0 = G_0$). Also, $\forall i \geq 2 : G_i = G_1$.
Thus,

$$\forall i \in \omega : F_i = G_i \quad \text{i.e., } F = G.$$

**Case 2.** $\neg \text{Failed}(F_1)$:
By the same reasoning as above, we have $\neg \text{Failed}(G_1)$. Then,

$$\forall i \in \omega : F_i \leq G_i \wedge G_i \leq F_i$$

Let $i \in \omega$ such that $F_i \neq G_i$. By the definition of $\sqsubseteq$, then $\text{Failed}(F_i) \lor \text{Failed}(G_i)$ which implies that $F_i = G_i$ (because $F_i \leq G_i$ and $G_i \leq F_i$) — a contradiction. Thus

$$\forall i \in \omega : F_i = G_i \quad \text{i.e., } F = G.$$

**Transitivity:** Let $F, G,$ and $H$ be in $\beta(X)$ and $F \sqsubseteq G$ and $G \sqsubseteq H$. Then

$$F_0 = G_0 \wedge G_0 = H_0$$

Thus, $F_0 = H_0$. Transitivity immediately follows if $\text{Failed}(F_1)$. Suppose $\neg \text{Failed}(F_1)$.
Then,

$$[\forall i \in \omega : F_i \leq G_i] \land [\forall i \in \omega : G_i \leq H_i]$$

Let $i \in \omega$ such that $F_i \neq G_i$. Then

$$\text{Failed}(F_i) \land \neg \text{Failed}(G_i)$$ (definition of $\sqsubseteq$). But

$$\neg \text{Failed}(G_i) \land G_i \leq H_i \Rightarrow G_i = H_i$$

Thus $F_i \leq H_i$. On the other hand, if $F_i = G_i$, then obviously $F_i \leq H_i$. So, we have

$$\forall i \in \omega : F_i \leq H_i \quad \text{i.e., } F \sqsubseteq H$$

$^3$In fact, $(\beta(X), \sqsubseteq)$ can be shown to be a complete partial order; any chain of this order contains at most three elements.
Lemma 2 Let $G, H$, and $H'$ be behaviours such that
\[ G \sqsubseteq H \land H' \sqsubseteq H \]
Then there is a behaviour $G'$ such that $G' \sqsubseteq H'$.

**Proof:** Immediately follows from the definitions of $\sqsubseteq$ and $\sqsubseteq$.

Consider a set $X$ of variables. Let
\[
\beta_{\text{min}}(X) = \{ F \in \beta(X) \mid \text{Failed}(F) \}
\]
Obviously, $\beta_{\text{min}}(X)$ is the set of all minimal elements of $\beta(X)$ with respect to $\sqsubseteq$.

Let $\text{DivBv}(\gamma)$ be the set of all behaviours of $\gamma$ that diverge, and $\text{FailBv}(\gamma)$ the set of all behaviours $F$ of $\gamma$ such that $\text{Failed}(F)$.

Consider the subset $D_X$ of $D$ defined by
\[
D \triangleq \{ \gamma \mid \gamma \in D \land \gamma \in \beta(X) \}
\]
The relation $\sqsubseteq$ on behaviours induces a relation $\sqsubseteq$ on $D_X$ defined below.

Let $\gamma$ and $\nu$ be elements of $D$ and subsets of $\beta(X)$. Then $\gamma \sqsubseteq \nu$ if
\[
(\forall G \in \nu : \exists F \in \gamma : F \sqsubseteq G) \land (\gamma - \text{FailBv}(\gamma) - \text{DivBv}(\gamma)) \subseteq \nu
\]

Let $C \triangleq \gamma_0 \sqsubseteq \gamma_1 \sqsubseteq \gamma_2 \sqsubseteq \ldots \subseteq \gamma_i \sqsubseteq \ldots$ be an infinite chain\(^4\) of $(D_X, \sqsubseteq)$. Let
\[
\gamma' = \bigcup_{i=0}^{\infty} \gamma_i
\]
and
\[
\gamma = \{ G \mid G \in \gamma \land [\forall i \in \omega : \exists F \in \gamma_i : f \sqsubseteq G] \}
\]

**Lemma 3** $\gamma$ is the least upper bound (lub) of $C$.

**Proof:** We prove that $\gamma_i \sqsubseteq \gamma$ for any $i \in \omega$. Let $G \in \gamma$. Then,
\[
\exists F \in \gamma_i : F \sqsubseteq G
\]
So it only remains to prove that if $F \in \gamma_i$ and $\neg \text{Failed}(F) \land F \not\in \text{DivBv}(\gamma_i)$ then $F \in \gamma$. We have
\[
\forall j \in \omega : (j < i \Rightarrow \exists F' \in \gamma_j : F' \sqsubseteq F) \land (j > i \Rightarrow F \in \gamma_j)
\]
Then, by the definition of $\gamma$, $F \in \gamma$.

We now show that $\nu$ is an upper bound of $C$ then $\gamma \sqsubseteq \nu$. Let $G \in \nu$. We verify that
\[
\exists F \in \gamma : F \sqsubseteq G
\]
\(^4\)all elements in this representation of an $\omega$-chains are different

14
Let \( i \in \omega \) such that \( G \in \gamma_i \). If \( G \notin \text{gamma} \) then there is \( \gamma_j \in C \) such that
\[
\forall F \in \gamma_j : \neg(F \sqsubseteq g)
\]
Then, \( \neg(\gamma_j \sqsubseteq \nu) \) - a contradiction. Thus, \( G \in \gamma \). Suppose there is no \( i \in \omega \) such that \( G \in \gamma_i \). Consider \( \gamma_i \in C \). Since \( \gamma_i \sqsubseteq \nu \), we have
\[
\exists F \in \gamma_i : F \sqsubseteq G
\]
\( F \in \gamma \) then implies that
\[
F \in \gamma : F \sqsubseteq G
\]
Let \( f \notin \gamma \). Then there is \( j \notin \omega \) such that \( j \neq i \) and
\[
\neg(\exists F' \in \gamma_j : F' \sqsubseteq F)
\]
Obviously, \( F \notin \gamma_j \). But \( \gamma_j \sqsubseteq \nu \). So, there is \( \nu'' \in \gamma_j \) such that \( \nu'' \sqsubseteq G \). Proceeding in this way we find that either
\[
\exists F \in \gamma : F \sqsubseteq G
\]
or there is an infinite set of behaviours with \( G \) as its upper bound. By the definition of \( \sqsubseteq \), such a set cannot exist. Therefore we have an \( F \in \gamma \) such that \( F \sqsubseteq G \).

Now, let \( F \in \gamma \) such that \( \neg\text{Failed}(F) \) and \( F \notin \text{DivBv}(\gamma) \). Since \( F \subseteq \gamma_i \) for some \( \gamma_i \in C \), it follows that \( F \subseteq \nu \) (because \( \gamma_i \sqsubseteq \nu \)). \( \square \)

**Lemma 4** \( \gamma \) is closed under abstraction.

**Proof:** Let \( F \in \gamma \). Then there is \( \gamma_i \in C \) such that \( F \in \gamma_i \). If there is a behaviour \( H \) such that \( H \preceq F \) then \( H \in \gamma_i \). Consider any \( j \in \omega \) There is \( G \in \gamma_j \) such that \( G \subseteq F \). Then, by Lemma 2, there is a behaviour \( H' \) such that \( G \sqsubseteq H' \). Therefore, by the definition of \( \gamma \), \( H \in \gamma \). \( \square \)

**Theorem 4.2** \( (\mathcal{D}_X, \sqsubseteq) \) is a complete partial order (cpo).

**Proof:** Let \( \gamma, \nu \), and \( \eta \) be elements of \( \mathcal{D}_X \).

**Reflexivity:** Obvious.

**Antisymmetry:** Let \( \gamma \subseteq \nu \land \nu \subseteq \gamma \) and \( F \in \gamma \). Since \( \nu \subseteq \gamma \), it follows that there is \( H \in \nu \) such that \( H \sqsubseteq F \). If \( H \notin \text{FailBv}(\nu) \) and \( H \notin \text{DivBv}(\nu) \), then \( H = F \). If \( \text{Failed}(H) \) then either \( H = F \) or \( F \in \nu \) (because, if \( H \neq F \) then \( \text{Stopped}(F) \) and \( \neg\text{Failed}(F) \)). If \( H \in \text{DivBv}(\nu) \), then \( H = F \). Thus, \( F \in \gamma \Rightarrow F \in \nu \). By similar arguments (since \( \gamma \) and \( \nu \) can be interchanged), we obtain \( F \in \nu \Rightarrow F \in \gamma \). Thus, \( \gamma = \nu \).

**Transitivity:** Let \( \gamma \subseteq \nu \land \nu \subseteq \eta \). If \( H \in \eta \) then there is \( G \in \nu \) such that \( G \subseteq H \) (because \( \nu \subseteq \eta \)). Also, there exists \( F \in \gamma \) such that \( F \subseteq G \) (because \( \gamma \subseteq \nu \)). The transitivity of \( \sqsubseteq \) on behaviours (Theorem 3.1) then gives \( F \subseteq H \). Thus,
\[
H \in \eta \Rightarrow \exists F \in \gamma : F \subseteq H
\]

15
Now, let \( F \in \gamma \) such that \( F \not\in \text{FailBv}(\gamma) \) and \( F \not\in \text{DivBv}(\gamma) \). Then \( F \in \gamma \) (because \( \gamma \subseteq \nu \)). Also \( F \in \eta \) (because \( \nu \subseteq \eta \)). Thus

\[
(\gamma - \text{FailBv}(\gamma) - \text{DivBv}(\gamma)) \subseteq \eta
\]

and, therefore, \( \gamma \subseteq \eta \).

We have proved that \((D_X, \subseteq)\) is a partial order. Clearly, \( \beta_{\min}(X) \) is its least element. So it remains to prove that any infinite chain of this order has the least upper bound (lub). By Lemma 3 and Lemma 4, \( \gamma \) of Formula 3 is the lub of \( C \) and \( \gamma \) is closed under abstraction.

\[ \square \]

Let \( C \triangleq \gamma_0 \subseteq \gamma_1 \subseteq \gamma_2 \subseteq \ldots \subseteq \gamma_l \subseteq \ldots \)

be an infinite chain of \((D_X, \subseteq)\).

4.1 Environments and the semantics of a command

Execution of a command \( C \) takes place in an environment which may initiate an execution of \( C \), communicate with \( C \), or even share variables with \( C \). Informally, an environment of \( C \) is an object which affects the executions of \( C \). The formal notion of an environment is developed below.

If the programming language has \( n \) combinators then the environment of a behaviour of a command can be represented by an \( n \)-tuple \((\alpha_1, \ldots, \alpha_n)\). Each of the elements \( \alpha_1, \ldots, \alpha_n \) is a behaviour whose characteristics depend on its associated combinator. In this subsection we consider the cases of sequential and parallel composition; other combinators are discussed later.

An environment of a behaviour \( F \) corresponding to sequential composition is a sequential environment of \( F \) and is any behavior \( G \) such that

\[
\text{Stopped}(G) \land \text{Function}(\text{last}(\text{Exec}(G))) = f_0
\]

Obviously \( \text{VAR}(G) = \text{VAR}(F) \). The sequential environment of the behaviour \( \Omega_\epsilon \) is \( \Theta \). Although a sequential environment could simply be an observation, we treat all environments as behaviours to achieve orthogonality in concepts.

The notion of parallel environment is now developed. Consider the commands \( A \) and \( B \) executing in parallel. If there is any communication between \( A \) and \( B \) then a specific relation must exist between the observations of these commands. Such a relation serves as the semantics of the communication mechanisms of the language. We can therefore factor this relation into a number of conjuncts, each of which defines a particular communication mechanism. One of these is the predicate \( \text{Consistent}_a \) which models asynchronous communication.

The predicate \( \text{Consistent}_a \) is defined over pairs of observations having disjoint sets of variables. Let \( \theta \triangleq (t_1, f) \) and \( \psi \triangleq (t_2, g) \) be observations where

\[
\text{VAR}(\theta) \cap \text{VAR}(\psi) = \emptyset
\]
Since a command which does not have any variables cannot communicate with other commands it follows that

\[ f = \emptyset \lor g = \emptyset \Rightarrow \text{Consistent}_a(\theta, \psi) \]

Let \( v \in CMN(\emptyset) \). If \( Dual(v) \notin CMN(\emptyset) \), then the observed values of \( v \) are not constrained in any way by the observed values of the variables in \( VAR(\psi) \).

Now suppose \( u \in CMN(\psi) \land u = Dual(v) \). First we consider the case \( t_1 \leq t_2 \).
If \( v \) is an input variable (implying that \( u \) is an output variable) then we have

\[ Data(f(v)) \supseteq g(Dual(v)) \]

i.e., a value cannot be received by a variable until it has been sent by another variable. The initial subsequence relation also implies that communication preserves the order of data.

If \( v \) is an output variable and \( \text{Broken}(\emptyset) \) then only those data which are in \( f(v) \) can be in \( g(Dual(V)) \) at a time later than \( t_1 \). Thus

\[ Data(g(Dual(v))) \supseteq f(v) \]

However, if \( \emptyset \) is not broken, then either the above relation holds or we have

\[ t_1 < t_2 \land f(v) \supseteq g(Dual(v)) \]

The case of \( t_1 > t_2 \) is simple because the order of arguments of \( \text{Consistent}_a \) should be immaterial. Thus \( \text{Consistent}_a(\psi, \emptyset) \) is evaluated in this case using the above rules.

Putting these together we have

\[ Consistent_a(\theta, \psi) \triangleq \]

\[ \forall v \in CMN(\emptyset) : Dual(v) \in CMN(\psi) \Rightarrow \]

\[ ([t_1 \leq t_2 \land \text{IsIn}(v)) \Rightarrow Data(f(v)) \supseteq g(Dual(v))] \]

\[ \land ([t_1 \leq t_2 \land \text{IsOut}(v)) \Rightarrow Data(g(Dual(v))) \supseteq f(v)] \]

\[ \lor (t_1 < t_2 \land f(v) \notin V^1 \land f(v) \supseteq g(Dual(v))] \]

\[ \land [t_1 > t_2 \Rightarrow Consistent_a(\psi, \emptyset)] ] \]

In the same way, other predicates defining the consistency of the observations of parallel commands based on other communication mechanisms can be formulated. The conjunction of these predicates is written as 'Consistent'. The parallel environment of a behaviour \( F \) is a behaviour \( E \) which satisfies the following conditions:

(a) \( VAR(E) = Dual(CMN(F)) \)
(b) \( \forall i \in \omega : \forall j \in \omega : \text{Consistent}(E_i, F_j) \)
(c) \( \text{Deadlocked}(F) \Rightarrow [\text{Deadlocked}(E) \land \text{Comptime}(F) = \text{Comptime}(E)] \)
(c) \( \text{Failed}(F) \Rightarrow [\text{Failed}(E) \land \text{Comptime}(F) = \text{Comptime}(E)] \)

If \( CMN(F) = \emptyset \), then for any \( t \in T \), \( \Omega_t \) is a parallel environment of \( F \).

The expression \( Env(C) \) denotes the set of all environments of \( C \) and is the same as the set of all environments of the behaviours of \( C \).
5 Limited Processors Semantics

5.1 Partitions of a command

The semantic function $\Gamma(C)$ of a command $C$ maps an environment $\alpha$ into a set of functions which is in one-to-one correspondence with $\omega$. If $k \in \omega$, then $\Gamma(C)(\alpha)(k)$ is a set of pairs of the form $(\pi, \beta)$ where $\pi$ represents an 'allocation' of $k$ processors to the various component commands of $C$, and $\beta$ is a set of corresponding behaviours of $C$. This assumes that

- all the processors of the machine are available exclusively to the command $C$ during its execution,
- a sequential segment of $C$ does not change processors during its execution, and
- parallel processes are interleaved only if there are fewer processors than processes.

The first assumption is reasonable because the semantics of the command $C$ should be based on an examination of the behaviours of $C$ in isolation from all other commands. The second assumption precludes dynamic reallocation of processors to processes. This simplifies our discussion and, moreover, is not very restrictive as the computation time of a command does not depend on the processor allocation scheme provided that the time takes into account of all possible processor switching overheads\(^5\). We now develop a formal representation of processor allocation.

**Definition 5.1** Consider a command $C$. A maximal parallel partition of $C$ is a partition $\pi(C)$ of the set $VAR(C)$ into $m$ subsets such that the following conditions hold:

a) the set of variables of any component command of $C$ not containing a network is not split across the elements of $\pi(C)$,

b) if $C$ has components $C_1$ and $C_2$ which can execute in parallel, then their sets of variables should belong to different elements of $\pi(C)$.

It may be noted that at any time a command $C$ is capable of executing on at most $m$ processors. The integer $m$ is called the degree of parallelism of $C$ and is denoted by $Degree(C)$. A parallel $k$-partition of $C$ is a partition of the set $VAR(C)$ such that each of its $k$ elements is either an element of $\pi(C)$ or the union of two or more elements of $\pi(C)$. The set of all $k$-partitions of $C$ and the set of all parallel partitions of $C$ are denoted by $\Pi_k(C)$ and $\Pi(C)$ respectively. It is obvious that $\Pi_m(C)$ is the singleton $\{\pi(C)\}$.

\(^5\)This follows from our earlier assumption that parallelism in commands is explicit, and that implicit parallelism which can be obtained by several means such as data flow analysis, loop unfolding etc. is not exploited
\[ \Gamma(A, B, \alpha, k) = \bigcup \{ (\pi_1, \beta_1) \parallel (\pi_2, \beta_2) \mid \exists m, n : m + n \geq k \land \\
\exists \alpha_A \in \text{Env}(A) : \exists \alpha_B \in \text{Env}(B) : \\
[(\pi_1, \beta_1) \in \Gamma(A, \alpha_A, m) \land (\pi_2, \beta_2) \in \Gamma(B, \alpha_B, n) \land \\
[\forall G \in \beta_1 : \alpha[\parallel(\parallel(\parallel(\parallel(G \uparrow (CMN(G) \land \text{Dual}(CMN(H))))))) = \alpha_B] \land \\
[\forall H \in \beta_2 : \alpha[\parallel(\parallel(\parallel(\parallel(H \uparrow (CMN(H) \land \text{Dual}(CMN(G))))))) = \alpha_A] \}
\]

where \((\pi_1, \beta_1) \parallel (\pi_2, \beta_2) = \{ (\pi_3, \beta_3) \mid \pi_3 \in \pi_1 \oplus \pi_2 \land \\
[\forall F \in \beta_3 : \exists G \in \beta_1 : \exists H \in \beta_2 : F = G \uparrow H ] \}
\]

6 Semantics of XCSP

Any atomic command \(A\) is executed on a single processor. Thus, for all environments \(\alpha\) and all \(k > 1\),

\[ \Gamma(A, \alpha, k) = \Gamma(A, \alpha, 1) \]

We assume that the computation time of \(A\) executing in environment \(\alpha\) can be any element of a real interval denoted by \(\text{Interval}_\alpha(A)\). This interval takes into account of any discrepancy between the (conceptual) local clock of \(A\) and the clocks of other commands, and also of any computation time overhead (for processor allocation etc.) which is not derivable from the text of \(A\).

6.1 An informal semantics of expressions

Let \(e\) be an expression. The set of all free variables of \(e\) is represented by \(\text{var}_e\). Let \(X\) be a nonempty subset of \(\text{VAR}\) and \(f\) a function \(f : X \rightarrow W\). Then the meta-expression \(e.f\) is \(\perp\) if

\[ \exists v \in \text{var}_e : v \notin X \lor f(v) = \lambda \]

Otherwise \(e.f\) is the value to which \(e\) evaluates when each \(v \in \text{var}_e\) is assigned the value \(\text{last}(f(v))\). The precise meaning of the evaluation of an expression can be defined in a data algebra; we do not consider this aspect of the semantics in this paper.

Let \(\alpha\) be an environment in some \(X \subseteq \text{VAR}\). Let

\[ \text{EvalFun}(\alpha) = \text{Function}(\text{last}((\text{Exec} ; (\alpha)E ))) = f \text{(say)} \]

The function \(f\) is the function part of the final observation of the sequential composition component of \(\alpha\). So the value of the expression \(e\) evaluated in \(\alpha\) is \(e.f\).

The set of evaluation times of \(e\) in \(\alpha\) is given by the interval \(\text{Interval}_\alpha(e)\).

6.2 The semantics of atomic commands

We will describe \(LCL(A), CMN(A), \) and \(\Gamma(A, \alpha, 1)\) for each atomic command \(A\) of XCSP and each environment \(\alpha\). If \(F\) is a behaviour of \(A\) such that

\[ \neg \text{Failed}(F_0) \land \neg \text{Deadlocked}(F_0) \]
then \(#\text{Exec}(F) = 2\) because any execution of an atomic command stops after the second observation is made.

**skip**

This command does not operate on any program variable. So, it appears that any behaviour of \(\text{skip}\) is \(\Omega_t\) for some \(t \in T - \{0\}\). However, with this semantics it is not possible to derive the nonterminating behaviours of certain commands such as

\[**[true \rightarrow \text{skip}]\]

Therefore we make a simplifying assumption: any command \(C\) has a distinguished variable \(\text{skip}\) which is assigned the distinguished value \(p\) only when a skip command is executed as part of the execution of \(C\). With this assumption the semantics of \(\text{skip}\) can be given as follows.

\[
\begin{align*}
\text{a) } & LCL(\text{skip}) = \{\text{skip}\} \\
\text{b) } & CMN(\text{skip}) = \emptyset \\
\text{c) } & \Gamma(\text{skip}, \alpha, 1) = \{(\{\{\text{skip}\}\}, \cup\{\{F\}\}) | \alpha \in Env(F) \land 1 \leq \#\text{Exec}(F) \leq 2 \\
& \land \#\text{Exec}(F) = 2 \Rightarrow \ (f_1(\text{skip}) = f_0(\text{skip}) < p) \\
& \land t_1 \in \text{Interval}_\alpha(\text{skip})\} \\
\end{align*}
\]

**assignment**

Define the operation "\(\prec\)" on (sequence,value) pairs by

\[s \prec d = \begin{cases} 
  s \prec \bot & \text{if } d = \bot \\
  s & \text{otherwise}
\end{cases}\]

The sets of local and external communication variables of the assignment command "\(x := e\)" are

\[
\begin{align*}
\text{a) } & LCL(x := e) = \text{var}_e \cup \{x\} \\
\text{b) } & CMN(x := e) = \emptyset
\end{align*}
\]

The following semantic function of "\(x := e\)" states that if the assignment is not performed in a failed or deadlocked environment, then the second observation records the assignment of the value of \(e\) to \(x\). The other variables remain unchanged unless the evaluation of \(e\) fails in which case these variables receive the value \(\bot\).

\[
\Gamma(x := e, \alpha, 1) = \{(\text{var}_e \cup \{x\}, \cup\{\{F\}\}) | \\
\alpha \in Env(F) \land 1 \leq \#\text{Exec}(F) \leq 2 \\
\land [\#\text{Exec}(F) = 2 \Rightarrow \ (f_1(x) = f_0(x) < e.f_0) \\
\land [\forall v \in \text{var}_e : v \neq z \Rightarrow \ f_1(v) = f_0(v) < e.f_0] \\
\land t_1 \in \text{Interval}_\alpha(x := e))\}\]

---

\(\text{we assume an IO command cannot have nonterminating execution; if it does not (successfully) terminate, then it either fails or deadlocks}\)

\(\text{thus any observed value of } \text{skip} \text{ during an execution of the command } "x := 1" \text{ is } \lambda\)
synchronous input

We assume that \( \text{Interval}_{\alpha}(D?x) \) is a final segment of \( T \) with least element \( \tau \).

a) \( LCL(D?x) = \{x\} \)
b) \( CMN(D?x) = \{?D\} \)

The semantic function of \( D?x \) is given by

\[
\Gamma(D?x, \alpha, 1) = \left\{ \varphi_{\alpha} \cup \{?D\}, \cup\{\{F\}\} \mid \begin{array}{c}
\alpha \in Env(F) \land 1 \leq \#\text{Exec}(F) \leq 2 \\
\land [\#\text{Exec}(F) = 2 \Rightarrow (f_1(x) = f_0(x) \land t_1 \geq \tau)]
\end{array} \right\}
\]

asynchronous input

The semantics of the asynchronous input command \( "D??x" \) is defined exactly as above (with \( ?D \) replaced by \( ??D \)).

synchronous output

We assume that \( \text{Interval}_{\alpha}(D!e) \) is a final segment of \( T \) with least element \( \tau \) which is greater than the least element of \( \text{Interval}_{\alpha}(e) \).

The semantics of \( D!e \) is that of the (meta)assignment command \( "!D := e" \) with

\[
\text{Interval}_{\alpha}(!D := e) = \text{Interval}_{\alpha}(D!e)
\]

asynchronous output

The semantics of the asynchronous output command \( "D!!x" \) is defined exactly as above (with \( !D \) replaced by \( !!D \)).

Note: The set constructor predicate \( \alpha \in Env(F) \) in the definitions of \( \Gamma(C, \alpha, 1) \) makes the semantic description of the IO commands simple and also allows us to use the same description for synchronous and asynchronous commands.

6.3 The semantics of command combinators

The sequential and parallel composition operators have already been defined. The semantics of the alternative and the repetitive commands are described below.

alternation

Let \( Alt \) denote the alternative command

\[
[\square^{n}_{i=1} g_i \rightarrow C_i]
\]
The boolean part of $g_i$ is represented by $b_i$. Let

$$BB = \land_{i=1}^n b_i$$

and $GLOC$ and $GCOM$ the set of local and the set of external communication variables appearing in the guards of this command. Also $GCOM_i$ is the singleton of the communication variable of $g_i$ if it has any, and $eset$ otherwise.

We assume that the time between the start of execution of any alternative command and the selection of an alternative (if possible) or abortion of the command lies in the interval $Interval(ALT)$.

The sets of variables of $Alt_n$ are

a) $LCL(Alt_n) = GLOC \cup \bigcup_{i=1}^n VAR(C_i)$

b) $CMN(Alt_n) = GCOM \cup \bigcup_{i=1}^n CMN(C_i)$

The following two functions are used in the description of the semantic function of $Alt_n$.

Consider a command $C$ and a parallel partition $\pi$ of $VAR(C)$. Let $X$ be a set of variables which do not belong to any process within $C$. Then $Include(X, \pi)$ is a partition $\pi'$ of $VAR(C) \cup X$ satisfying

$$Nonproc(\pi') = X \cup Nonproc(\pi)$$

Let $F$ be a behaviour and $t \in T$. Then $Shift(F, t)$ is a behaviour $G$ defined by

$$\forall i \in \omega - \{0\} : t_{gi} = t_{fi} + t$$

Thus $Shift(F, t)$ is a behaviour obtained by adding time $t$ to the time components of all observations, except the first (initial), of $F$. If $\beta$ is set of behaviours then $Shift(\beta, t)$ is the set of behaviours

$$\{F' : \exists F \in \beta : F' = Shift(F, t)\}$$

The expression $\Gamma(Alt_n, \alpha, k)$ is defined below.

The set of $k$-partitions of $Alt_n$ is

$$\Pi_k(Alt_n) = \{Include(GLOC \cup GCOM, \pi) \mid \forall i \in [1, n] : \pi \in \Pi\}$$

where $\Pi$ is the set

$$\{\pi_1 \uplus \pi_2 \uplus \ldots \uplus \pi_n \mid \forall i \in [1, n] : \pi \in \Pi(C_i)\}$$

Let $GCOM_i \neq \emptyset \Rightarrow g_i = b_i; IO_i$ and $Alt'_n$ the alternative command

$$[\Box_{i=1}^n g'_i \rightarrow C'_i]$$

\[^8\text{for simplicity we assume that this interval does not depend on the number or the texts of the guards or on the environment of the command.}\]
where \( \forall i \in [1, n] : g'_i = b_i \)
\( \land GCO_M = \emptyset \Rightarrow C'_i = C_i \)
\( \land GCO_M \neq \emptyset \Rightarrow C'_i = IO_i; C_i \)

**Case 1.** \( [\exists i \in [1, n] : b_i = true \land \neg External([l](x) \uparrow Dual(GCO_M), 1)] \)

The predicate \( \neg Broken([l](x)) \) ensures that no variable in \( Dual(GCO_M) \) is assigned the value \( \bot \) or \( \delta \).

\[
\Gamma(Alt_n, \alpha, k) = \{(\pi, \beta) \mid \pi \in \Pi_k(Alt_n)
\land [\exists F \in \beta : \alpha \in Env(F)]
\land [\exists i \in [1, n] : b_i \land EvalFun(\alpha) = true
\land [\exists (\pi', \beta') \in \Gamma(C, \alpha, k) : \exists t \in Interval(Alt) :
\pi' = \pi \circ VAR(C_i)
\land \beta \uparrow VAR(C_i) = Shift(\beta', t)
\land \forall F \in \beta :
\#Exec(F \uparrow (VAR(Alt_n) - VAR(C_i))) = 1]})
\]

**Case 2.** \( BB \land EvalFun(\alpha) = false \)

The alternative command fails in this case.

\[
\Gamma(Alt_n, \alpha, k) = \{(\pi, \beta) \mid \pi \in \Pi_k(Alt_n)
\land [\forall F \in \beta : \alpha \in Env(F) \land Failed(F_i)]
\]

**Case 3.** \( BB \land EvalFun(\alpha) = true \)

\[
\land (\forall i \in [1, n] : b_i = false \lor \neg External([l](x) \uparrow Dual(GCO_M), 1)]
\lor \neg Broken([l](x)))
\]

In this case the command \( Alt_n \) behaves exactly like the command \( Alt'_n \), that is
\[
\Gamma(Alt_n, \alpha, k) = \Gamma(Alt'_n, \alpha, k)
\]

**repetition**

Let \( Rep_n \) denote the repetitive command
\[
*[\emptyset]_i g'_i \rightarrow C_i
\]

\( Rep_n \) can also be written as \( *Alt_n \). The sets of local and external communication variables are exactly the corresponding ones of \( Alt_n \). The definition of the expression \( \Gamma(Rep_n, \alpha, k) \) has the following two cases (remember that XCSP does not use a distributed termination rule).

**Case 1.** \( BB \land EvalFun(\alpha) = true \)

\[
\Gamma(Rep_n, \alpha, k) = \Gamma(Alt_n; Rep_n, \alpha, k)
\]

**Case 1.** \( BB \land EvalFun(\alpha) = false \)

In this case the repetitive command terminates immediately.
\[ \Gamma(Re_p, \alpha, k) = \{ (\pi, \beta) \mid \forall F \in \beta : \alpha \in Env(F) \land \#Exec(F) = 1 \} \]

**program constructor**

Let \( C' \) represent the command \texttt{prog} \( C \) \texttt{end} . Then

a) \( LCL(C') = LCL(C) \)

b) \( CMN(C') = CMN(C) \)

**Definition 6.1** An environment is said to be **empty** if each of its components is a behaviour whose variables are all \( \lambda \). The empty environment of a command \( C \) is represented by \( ProgEnv(C) \).

The semantic function of \( C' \) is given by

\[
\Gamma(C', \alpha, k) = \begin{cases} 
\Gamma(C', \alpha, k) & \text{if } \alpha = ProgEnv(C) \\
\{ (\pi, 0) \mid \pi \in \Pi_k(C) \} & \text{otherwise}
\end{cases}
\]

**7 Discussion**

The basic semantic objects of our approach are the values of program variables and the times at which they are observed. The semantic model has been constructed for developing and justifying logic-based specification and proof systems.

The existing models of real-time processes can broadly be classified as algebraic models and state-based models. The algebraic models [4,5] lead to transformational systems for program development. However, a program specification in such a system is itself a program and we believe that intuitive justification of a real-time program should always be avoided. Although the theories underlying the algebraic models can be used to develop logic-based specification systems dealing with "timed-actions", such a system would require an algebra of 'events' or 'actions' to support itself. A specification system of this kind is yet to be seen. Our state-based model, on the other hand, requires data algebras, most of which are well-known and conceptually simple.

A prime prerequisite for a simple specification language is a simple semantic domain. The semantic domains of the present state-based models of real-time processes [6,7,8,9] quite complicated. The source of the complexity lies in describing communication in a way very different from the one in which internal actions are described. We treat internal actions in a uniform way and thereby have obtained a reasonably simple semantic domain.

The real-time models in [6,4,9] use explicit references to the 'waiting' times of commands: a command waits either for a value communication or for a signal from a time-out mechanism. Where possible, we avoid reference to waiting periods when specifying real-time systems, because these can adequately be subsumed by a temporal ordering of starting and termination times of actions if waiting is considered as part of execution. This again leads to a simple domain description.
The proposed semantics is compositional and models termination, failure, divergence, deadlock, and starvation. Although untimed-models of concurrency can (and should) ignore assumptions about the physical environment, a real-time model should take all possible environments into consideration. Existing models based on 'maximal parallelism' (e.g. [6,7,8,9]) are not suited for many applications. The model presented in this paper considers all possible degrees of parallelism, and so it can be used for arbitrary processor organizations.

Our current work based on this semantic model of real-time processes involves the development of a specification language and proof system. This semantic model will also be used to justify an algebraic model of real-time processes which is being developed.

References


