Research Report 121

A SEMANTIC MODEL FOR THE SPECIFICATION OF REAL-TIME PROCESSES

Asis Goswami, Mathai Joseph[†]

(RR121)

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

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

April 1988

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[1] 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

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

and P_3 is a physical device with the behaviour

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

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: $C_1 \| C_2$
- Program Constructor: **prog** C **end**

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

prog C **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 $C_1 \| \ldots \| C_n$ is called a *network* and the commands $C_1, \ldots, C_n$ are the *processes* of the network. All the proceses 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 '||'.

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.

$$
\begin{array}{lll}
P & ::= & \textbf{prog } C \textbf{ end} \\
C & ::= & \textbf{skip} \mid x := e \mid IO \mid C_1; C_2 \mid \\
 & & [\Box_{i=1}^{n} g_i \rightarrow C_i] \mid {}^{*}[\Box_{i=1}^{n} g_i \rightarrow C_i] \mid C_1 \| C_2 \\
IO & ::= & D?x \mid D??x \mid D!e \mid D!!e \\
g & ::= & b \mid IO \mid b; IO
\end{array}
$$

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 $D!e$ 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; a_1 and a_n are the first and last element of the sequence
λ	the empty sequence
$\#s$	the length of s $(\#\lambda = 0)$
$s[i]$	the ith element of s
$s_1 \underline{\propto} s_2$	s_1 is an initial subsequence of s_2
$s_1 \propto s_2$	s_1 is a proper initial subsequence of s_2 ($s_1 \underline{\propto} s_2$ and $s_1 \neq s_2$)
$s_1 \hat{} s_2$	the catenation of s_1 and s_2 $< 1,3,4 > \hat{} < 2,6 > = < 1,3,4,2,6 >$
$first(s)$	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 λ 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

$\#s, last(s), lrest(s)$ are undefined if s is infinite and the expression $s_1 \hat{} s_2$, is undefined if s_1 is infinite.

3.2 Basic ideas

The symbol ω represents the set of all nonnegative integers; $[i, j]$ stands for the closed interval $\{k \mid i \leq k \leq j\}$ 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 $x = 3 \wedge y = 4$ is represented by the following behaviour:

	Observations			
	1	2	3	
x :	$<3>$	$<3,5>$	$<3,5>$	
y :	$<4>$	$<4>$	$<4,9>$	$\cdots$

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:

	Observations				
	1	2	3	4	
x :	$<3>$	$<3,5>$	$<3,5,6>$	$<3,5,6>$	
y :	$<4>$	$<4>$	$<4>$	$<4,10>$	$\cdots$
?D :	λ	$<5>$	$<5>$	$<5>$	

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 λ.

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.

	Observations				
	1	2	3	4	
x :	λ	$<5>$	$<5,6>$	$<5,6,\perp>$	
y :	λ	λ	λ	$<\perp>$	$\cdots$
?D :	λ	$<5>$	$<5>$	$<\perp>$	

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[2]. 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 $!!v$

[2]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) $Dual(?v) = !v$
b) $Dual(??v) = !!v$
c) $Dual(!v) = ?v$
d) $Dual(!!v) = ??v$
e) $IsIn(u) \triangleq u = ?v \vee u = ??v$
f) $IsOut(u) \triangleq u = !v \vee u = !!v$
g) $Syn(u) \triangleq u = ?v \vee u = !v$
h) $Asyn(u) \triangleq u = ??v \vee u = !!v$

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

$$Dual(X) = \{y \mid \exists x \in X : \ y = Dual(x)\}$$

Obviously $Dual(\emptyset) = \emptyset$.

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 $\perp$ and δ 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 "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 $\perp$ ($\perp \notin$ V). In the same way, the value δ is assigned to each variable of a deadlocked command.

Let V^* 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 λ. Define

$$V^\dagger = \{s^\frown < d > \mid s \in V^* \wedge d \in \{\perp, \delta\}\}$$

Then the space W of 'observed values' is defined as

$$W = V^+ \cup V^\dagger$$

The function $Data$ is defined over W:

$$Data(v) = \begin{cases} lrest(v) & \text{if } v \in V^\dagger \\ v & \text{otherwise} \end{cases}$$

Thus $Data$ returns the sequence of values in V held by a variable.

Definition 3.1 An *observation* θ 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) $[\exists v \in VAR(C) : \perp \in range(f(v))] \Rightarrow ([\forall v \in VAR(C) : \perp \in range(f(v))] \wedge t = \sigma)$

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

where σ is an arbitrarily large number. It is formally described later.

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

The components t and f of θ in this definition are called the *time* and the *function* components respectively, given by the expressions $Time(\theta)$ and $Function(\theta)$. Two observations θ and ψ 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

$Failed(\theta) \triangleq [\forall v \in VAR(\theta) : \ lastOb(f(v)) = \perp]$

$Deadlocked(\theta) \triangleq [\forall v \in VAR(\theta) : \ lastOb(f(v)) = \delta]$

$Broken(\theta) \triangleq Failed(\theta) \vee Deadlocked(\theta)$

Together with Definition 3.1, this means that the failure of any component of a command induces the value $\perp$ in all the variables of the command; similarly the value δ 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 θ_0 : An execution of a command is assumed to start at time 0, and so the time component of θ_0 is 0. The function component of θ_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 λ, $< \perp >$, 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 θ_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 θ_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) $\forall k \in \omega : \ \forall v \in VAR(C) : \ [\exists s \in U \cup \{\lambda\} : \ f_{k+1}(v) = f_k(v)\hat{}s] \wedge t_k < t_{k+1}$
e) $\forall k \in \omega : \ (f_k = f_{k+1} \vee Broken(\theta_k)) \Rightarrow [\forall j \in \omega : \ j > k \Rightarrow \theta_j = \theta_{k+1}]$
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

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

denoted by Ω_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 ith element of F by F_i or by the pair (t_{f_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 exibits 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 ω-sequence whose ith observation, for all $i \in \omega$, is the restriction of F_i to the set

X. 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 Ω_t. We show how the restrictions of behaviours can be converted into behaviours.

Consider an ω-sequence S of observations over a set X of variables. S is said to be *compatible* with a behaviour F, written S *Compat* F, if either $S = F$ or the following conditions hold:

a) $VAR(F) = X$
b) $\forall i \in \omega : Time(S[i]) < Time(S[i+1])$
c) $\forall i \in \omega : \exists j \in \omega : \exists k \in \omega : k > j > i \wedge F_i = S[j] \wedge F_{i+1} = S[k]$
$\wedge [\forall l \in [j+1, k-1] : Function(S[l]) = Function(S[j])]$

Informally, an ω-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 ω-sequence compatible with F, so that

$$Time(F_i) < t < Time(F_{i+1}) \wedge Function(F_i) = f$$

The relation *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 $VAR(C)$, then we say that the behaviour H, where

$$F \uparrow X \ Compat \ H$$

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

$$(F \uparrow VAR(P_i) \ Compat \ H) \ \wedge \ Terminated(H)$$

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 $Exec(F)$ if the following conditions hold:

a) $k > 0 \Rightarrow f_{k-1} \neq f_k$
b) $(\forall j \in \omega : \ j > k \Rightarrow f_j = f_k)$

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

We now define a few predicates over behaviours.

$Stopped(F) \triangleq Exec(F) \neq F$
$Failed(F) \triangleq (\exists k \in \omega : \ Failed(F_k))$

$Deadlocked(F) \triangleq (\exists k \in \omega : \ Deadlocked(F_k))$
$Broken(F) \triangleq Failed(F) \vee Deadlocked(F)$
$Terminated(F) \triangleq Stopped(F) \wedge \neg Broken(F)$

If $Stopped(F)$ is true then $Exec(F)$ 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 $Broken(F) = \text{true}$. Note that the predicates $Failed$ and $Deadlocked$ are mutually exclusive.

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

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

a) $Internal(F,k) \triangleq k \geq 1 \wedge [\exists v \in LCL(C) : \ f_k(v) \neq f_{k-1}(v)]$
b) $External(F,k) \triangleq k \geq 1 \wedge [\exists v \in CMN(C) : \ f_k(v) \neq f_{k-1}(v)]$

These two predicates need not be mutually exclusive. $Internal(F,k)$ implies that the kth observation of F has recorded an assignment to a local variable. Similarly, $External(F,k)$ 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 σ. For all behaviours F,

$$\forall i \in \omega : \ Time(F_i) \leq \sigma$$

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

An *abstraction function* $\ominus$ 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 $F \ominus \theta$ is defined if

$$\exists k \in \omega : \ F_k = \theta \wedge Internal(F,k)$$

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

$$\begin{aligned}[\forall i \in \omega : (i < k \Rightarrow h_i = f_i)] \\ \wedge[\forall v \in LCL(C) : \ f_{k+1}(v) = f_k(v) \Rightarrow [\forall i \geq k : \ h_i(v) = f_{i+1}(v)] \\ \wedge f_{k+1}(v) \neq f_k(v) \Rightarrow [\forall i \geq k : \ h_i(v) = f_{k-1}(v)^\frown rest(\#f_k(v), f_{i+1}(v))]]\end{aligned}$$

It can be shown that θ is associative, i.e., if θ and ψ are behaviours then

$$((F \ominus \theta) \ominus \psi) = ((F \ominus \psi) \ominus \theta)$$

Let $X \triangleq \{\theta_1, \ldots, \theta_n\}$ be a set of observations of F. Then $F\ominus$ represents

$$((F \ominus \theta) \ominus \{X - \{\theta\}\})$$

for any θ in X, where $F \ominus \emptyset = F$. The abstraction function hides the observations of the internal actions from a behaviour.

Consider a behaviour F such that

$$\neg Stopped(F) \wedge [\exists k \in \omega : (k = 0 \vee External(F,k)) \wedge \forall i > k : \neg External(F,i)]$$

where the behaviour F diverges after the kth 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

$$VAR(H) = VAR(F) \wedge [\forall i \in [0,k] : H_i = F_i]$$
$$\wedge H_k = (\epsilon, Function(F_{k-1})\hat{} < \perp >)$$

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

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

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

$$H = F \vee [\exists k \in \omega : \ H = F \ominus k \vee H \preceq (F \ominus k)]$$

The behaviour Ω_t has no abstraction other than itself. It is easily seen that the relation $\preceq$ 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 $H \preceq F \wedge H \preceq G$.

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

$$H \preceq G \wedge H' \preceq G \Rightarrow H \approx H'$$

Proof: Case 1. G does not diverge:
The hypothesis implies that there are (possibly empty) sets X and Y of observations such that

$$H = G \ominus X \wedge H' = G \ominus Y$$

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

$$G \ominus Z \preceq G \ominus X \wedge G \ominus Z \preceq G \ominus Y$$

or, $G \ominus Z \preceq H \wedge G \ominus Z \preceq H'$. Thus $H \approx H'$ (by the definition of $\approx$).

Case 2. G diverges:
We have $H = G' \ominus X \wedge H' = G' \ominus Y$ 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

$$H' \preceq F \wedge H' \preceq G \wedge H'' \preceq G \wedge H' \preceq H \tag{1}$$

But $H' \preceq G \wedge H'' \preceq G \Rightarrow H' \approx H''$ (Lemma 1). By the definition of $\approx$, $H \approx H''$ implies that there exists a behaviour J such that

$$J \preceq H' \wedge J \preceq H'' \tag{2}$$

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

$$J \preceq F \wedge J \preceq H$$

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

□

4 The semantic domain

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

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

Thus γ is a set of behaviours over X.

The *closure of γ by abstraction* is

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

and γ is said to be closed by abstraction, written $Closed_A(\gamma)$, if

$$Cl_A(\gamma) = \gamma$$

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

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

$$\mathcal{D} = \{\gamma \mid \exists X \subseteq VAR : \gamma \subseteq \beta(X) \wedge \gamma \neq \emptyset \wedge Closed_A(\gamma)\}$$

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 $\trianglelefteq$ on observations.

Let θ and ψ be observations. Then $\theta \trianglelefteq \psi$ if

$$\begin{array}{l} VAR(\theta) = VAR(\psi) \\ \wedge(\theta = \psi \vee (Failed(\theta) \wedge \neg Failed(\psi)) \\ \wedge[\forall v \in VAR(\theta) : Function(\theta)(v) = lrestOb(Function(\psi)(v))^\frown < \perp >])) \end{array}$$

Note that in the case $\theta \trianglelefteq \psi \wedge \theta \neq \psi$, we have $Time(\theta) = \epsilon$ and therefore we do not need an explicit temporal ordering of θ and ψ.

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

$$F_0 = G_0 \wedge (Failed(F_1) \vee [\forall i \in \omega : \ F_i \trianglelefteq G_i])$$

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

$\forall i \in \omega : \ F_i = G_i$ i.e., $F = G$.

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

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

Let $i \in \omega$ such that $F_i \neq G_i$. By the definition of $\trianglelefteq$, then $Failed(F_i) \vee Failed(G_i)$ which implies that $F_i = G_i$ (because $F_i \trianglelefteq G_i$ and $G_i \trianglelefteq F_i$) – a contradiction. Thus

$\forall i \in \omega : \ F_i = G_i$ i.e., $F = G$.
Transitivity: Let F, G, and H be in $\beta(X)$ and $F \sqsubseteq G \wedge G \sqsubseteq H$. Then

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

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

$$[\forall i \in \omega : \ F_i \trianglelefteq G_i] \wedge [\forall i \in \omega : \ G_i \trianglelefteq H_i]$$

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

$Failed(F_1) \wedge \neg Failed(G_1)$ (definition of $\trianglelefteq$). But

$$\neg Failed(G_i) \wedge G_i \trianglelefteq H_i \Rightarrow G_i = H_i$$

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

$\forall i \in \omega : \ F_i \trianglelefteq H_i$ 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*

$$G \sqsubseteq H \wedge H' \preceq H$$

Then there is a behaviour G' such that $G' \sqsubseteq H'$.

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

Consider a set X of variables. Let

$$\beta_{min}(X) = \{F \in \beta(X) \mid Failed(F_1)\}$$

Obviously, $\beta_{min}(X)$ is the set of all minimal elements of $\beta(X)$ with respect to $\sqsubseteq$. Let $DivBv(\gamma)$ be the set of all behaviours of γ that diverge, and $FailBv(\gamma)$ the set of all behaviours F of γ such that $Failed(F)$.

Consider the subset $\mathcal{D}_X$ of $\mathcal{D}$ defined by

$$\mathcal{D} \triangleq \{\gamma \mid \gamma \in \mathcal{D} \wedge \gamma \in \beta(X)\}$$

The relation $\sqsubseteq$ on behaviours induces a relation $\sqsubseteq$ on $\mathcal{D}_X$ defined below.

Let γ and ν be elements of $\mathcal{D}$ and subsets of $\beta(X)$. Then $\gamma \sqsubseteq \nu$ if

$$(\forall G \in \nu : \exists F \in \gamma : F \sqsubseteq G) \wedge (\gamma - FailBv(\gamma) - DivBv(\gamma)) \subseteq \nu$$

Let $C \triangleq \gamma_0 \sqsubseteq \gamma_1 \sqsubseteq \gamma_2 \sqsubseteq \ldots \sqsubseteq \gamma_i \sqsubseteq \ldots$ be an infinite chain[4] of $(\mathcal{D}_X, \sqsubseteq)$. Let

$$\gamma' = \bigcup_{i=0}^{\infty} \gamma_i$$

and

$$\gamma = \{G \mid G \in \gamma \wedge [\forall i \in \omega : \exists F \in \gamma_i : f \sqsubseteq G]\} \tag{3}$$

Lemma 3 *γ 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 $\neg Failed(F) \wedge F \notin DivBv(\gamma_i)$ then $F \in \gamma$. We have

$$\forall j \in \omega : (j < i \Rightarrow \exists F' \in \gamma_j : F' \sqsubseteq F) \wedge (j > i \Rightarrow F \in \gamma_j)$$

Then, by the definition of γ, $F \in \gamma$.

We now show that i ν 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 ω-chains are different

Let $i \in \omega$ such that $G \in \gamma_i$. If $G \notin gamma$ 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 $F'' \in \gamma_j$ such that $F'' \sqsubseteq G$. 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 $\neg Failed(F)$ and $F \notin DivBv(\gamma)$. Since $F \subset \gamma_i$ for some $\gamma_i \in C$, it follows that $F \in \nu$ (because $\gamma_i \sqsubseteq \nu$.

□

Lemma 4 *γ 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 $G \sqsubseteq F$. Then, by Lemma 2, there is a behaviour H' such that $G \sqsubseteq H$. Therefore, by the definition of γ, $H \in \gamma$.

□

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

Proof: Let γ, ν, and η be elements of $\mathcal{D}_X$.
Reflexivity: Obvious.
Antisymmetry: Let $\gamma \sqsubseteq \nu \wedge \nu \sqsubseteq \gamma$ and $F \in \gamma$. Since $\nu \sqsubseteq \gamma$, it follows that there is $H \in \nu$ such that $H \sqsubseteq F$. If $H \notin FailBv(\nu)$ and $H \notin DivBv(\nu)$, then $H = F$. If $Failed(H)$ then either $H = F$ or $F \in \nu$ (because, if $H \neq F$ then $Stopped(F)$ and $\neg Failed(F)$). If $H \in DivBv(\nu)$, then $H = F$. Thus $F \in \gamma \Rightarrow F \in \nu$. By similar arguments (since γ and ν can be interchanged), we obtain $F \in \nu \Rightarrow F \in \gamma$. Thus,$\gamma = \nu$.
Transitivity: Let $\gamma \sqsubseteq \nu \wedge \nu \sqsubseteq \eta$. If $H \in \eta$ then there is $G \in \nu$ such that $G \sqsubseteq H$ (because $\nu \sqsubseteq \eta$). Also, there exists $F \in \gamma$ such that $F \sqsubseteq G$ (because $\gamma \sqsubseteq \nu$). The transitivity of $\sqsubseteq$ on behaviours (Theorem 3.1) then gives $F \sqsubseteq H$. Thus,

$$H \in \eta \Rightarrow \exists F \in \gamma : F \sqsubseteq H$$

Now, let $F \in \gamma$ such that $F \notin FailBv(\gamma)$ and $F \notin DivBV(\gamma)$. Then $F \in \gamma$ (because $\gamma \sqsubseteq \nu$). Also $F \in \eta$ (because $\nu \sqsubseteq \eta$). Thus

$$(\gamma - FailBv(\gamma) - DivBv(\gamma)) \subseteq \eta$$

and, therefore, $\gamma \sqsubseteq \eta$.

We have proved that $(\mathcal{D}_{\mathcal{X}}, \sqsubseteq)$ 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, γ of Formula 3 is the lub of C and γ is closed under abstraction.

□

Let $C \triangleq \gamma_0 \sqsubseteq \gamma_1 \sqsubseteq \gamma_2 \sqsubseteq \ldots \sqsubseteq \gamma_i \sqsubseteq \ldots$
be an infinite chain of $(\mathcal{D}_{\mathcal{X}}, \sqsubseteq)$.

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

$$Stopped(G) \wedge Function(last(Exec(G))) = f_0$$

Obviously $VAR(G) = VAR(F)$. The sequential environment of the behaviour Ω_t is Θ. 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 $Consistent_a$ which models asynchronous communication.

The predicate $Consistent_a$ 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

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

Since a command which does not have any variables cannot communicate with other commands it follows that

$$f = \Theta \vee g = \Theta \Rightarrow Consistent_a(\theta, \psi)$$

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

Now suppose $u \in CMN(\psi) \wedge u = Dual(v)$. 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

$$Data(f(v)) \underline{\propto} g(Dual(v))$$

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 $Broken(\theta)$ then only those data which are in $f(v)$ can be in $g(Dual(V))$ at a time later than t_1. Thus

$$Data(g(Dual(v))) \underline{\propto} f(v)$$

However, if θ is not broken, then either the above relation holds or we have

$$t_1 < t_2 \wedge f(v) \underline{\propto} g(Dual(v))$$

The case of $t_1 > t_2$ is simple because the order of arguments of $Consistent_a$ should be immaterial. Thus $Consistent_a(\psi, \theta)$ is evaluated in this case using the above rules.

Putting these together we have

$Consistent_a(\theta, \psi) \triangleq$

$$\begin{aligned}
[\forall v \in CMN(\theta) : \; & Dual(v) \in CMN(\psi) \Rightarrow \\
& [(t_1 \leq t_2 \wedge IsIn(v)) \Rightarrow Data(f(v)) \underline{\propto} g(Dual(v))] \\
& \wedge [(t_1 \leq t_2 \wedge IsOut\ (v)) \Rightarrow Data(g(Dual(v))) \underline{\propto} f(v) \\
& \qquad \vee (t_1 < t_2 \wedge f(v) \notin V^{\dagger} \wedge f(v) \underline{\propto} g(Dual(v)))] \\
& \wedge [t_1 > t_2 \Rightarrow Consistent_a(\psi, \theta)]]
\end{aligned}$$

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) $[\forall i \in \omega : \forall j \in \omega : Consistent(E_i, F_j)]$
(c) $Deadlocked(F) \Rightarrow [Deadlocked(E) \wedge Comptime(F) = Comptime(E)$
(c) $Failed(F) \Rightarrow [Failed(E) \wedge Comptime(F) = Comptime(E)$

If $CMN(F) = \emptyset$, then for any $t \in T$, Ω_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 α into a set of functions which is in one-to-one correspondence with ω. If $k \in \omega$, then $\Gamma(C)(\alpha)(k)$ is a set of pairs of the form (π, β) where π represents an 'allocation' of k processors to the various component commands of C, and β 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 twoor 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

$$\begin{aligned}\Gamma(A\|B,\alpha,k) = \cup\{(\pi_1,\beta_1)\sharp(\pi_2,\beta_2) \mid \exists m,n:\ & m+n \geq k \wedge \\ & \exists \alpha_A \in Env(A):\ \exists \alpha_B \in Env(B): \\ & [(\pi_1,\beta_1) \in \Gamma(A,\alpha_A,m) \wedge (\pi_2,\beta_2) \in \Gamma(B,\alpha_B,n) \wedge \\ & [\forall G \in \beta_1:\ \alpha[\|/\|((\alpha) \Updownarrow G \uparrow (CMN(G) \cap Dual(CMN(H))))] = \alpha_B] \wedge \\ & [\forall H \in \beta_2:\ \alpha[\|/\|((\alpha) \Updownarrow H \uparrow (CMN(H) \cap Dual(CMN(G))))] = \alpha_A]]\}\end{aligned}$$

where $(\pi_1,\beta_1)\sharp(\pi_2,\beta_2)) = \{(\pi_3,\beta_3) \mid \pi_3 \in \pi_1 \oplus_k \pi_2 \wedge [\forall F \in \beta_3:\ \exists G \in \beta_1:\ \exists H \in \beta_2:\ F = G \Updownarrow H]\}$

6 Semantics of XCSP

Any atomic command A is executed on a single processor. Thus, for all environments α and all $k > 1$,

$$\Gamma(A,\alpha,k) = \Gamma(A,\alpha,1)$$

We assume that the computation time of A executing in environment α can be any element of a real interval denoted by $Interval_\alpha(A)$. 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 var_e. Let X be a nonempty subset of VAR and f a function $f : X \to W$. Then the meta-expression $e.f$ is $\perp$ if

$$\exists v \in var_e : v \notin X \vee f(v) = \lambda$$

Otherwise $e.f$ is the value to which e evaluates when each $v \in var_e$ is assigned the value $last(f(v))$. 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 α be an environment in some $X \subseteq VAR$. Let

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

The function f is the function part of the final observation of the sequential composition component of α. So the value of the expression e evaluated in α is $e.f$.

The set of evaluation times of e in α is given by the interval $Interval_\alpha(e)$.

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 α. If F is a behaviour of A such that

$$\neg Failed(F_0) \wedge \neg Deadlocked(F_0)$$

then $\#Exec(F) = 2$ because any execution of an atomic command stops[6] after the second observation is made.

skip

This command does not operate on any program variable. So, it appears that any behaviour of **skip** is Ω_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

$$*[\textbf{true} \rightarrow \textbf{skip}]$$

Therefore we make a simplifying assumption: any command C has a distinguished variable *skip* which is assigned the distinguished value ρ only when a **skip** command is executed as part of the execution of C[7]. With this assumption the semantics of **skip** can be given as follows.

a) $LCL(\textbf{skip}) = \{skip\}$
b) $CMN(\textbf{skip}) = \emptyset$
c) $\Gamma(\textbf{skip}, \alpha, 1) = \{(\{\{skip\}\}, \cup\{\{F\}\}) \mid \alpha \in Env(F) \wedge 1 \leq \#Exec(F) \leq 2$
$\wedge \#Exec(F) = 2 \Rightarrow$
$(f_1(skip) = f_0(skip)^\frown < \rho >$
$\wedge t_1 \in Interval_\alpha(\textbf{skip}))\}$

assignment

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

$$s \lhd d = \begin{cases} s^\frown < \perp > & \text{if } d = \perp \\ s & \text{otherwise} \end{cases}$$

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

a) $LCL(x := e) = var_e \cup \{x\}$
b) $CMN(x := e) = \emptyset$

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 $\perp$.

$$\begin{aligned} \Gamma(x := e, \alpha, 1) = \{&(var_e \cup \{x\}, \cup\{\{F\}\}) \mid \\ &\alpha \in Env(F) \wedge 1 \leq \#Exec(F) \leq 2 \\ &\wedge [\#Exec(F) = 2 \Rightarrow \\ &\quad (f_1(x) = f_0(x)^\frown < e.f_0 > \\ &\quad \wedge [\forall v \in var_e : \ v \neq x \Rightarrow \\ &\quad\quad f_1(v) = f_0(v) \lhd e.f_0] \\ &\quad \wedge t_1 \in Interval_\alpha(x := e))]\} \end{aligned}$$

[6] we assume an IO command cannot have nonterminating execution; if it does not (successfully) terminate, then it either fails or deadlocks

[7] thus any observed value of *skip* during an execution of the command "$x := 1$" is λ

synchronous input

We assume that $Interval_\alpha(D?x)$ is a final segment of T with least element τ.

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

The semantic function of $D?x$ is given by

$$\begin{aligned}\Gamma(D?x,\alpha,1) = \{&(var_e \cup \{\{?D\}\}, \cup\{\{F\}\}) \mid \\ &\alpha \in Env(F) \wedge 1 \leq \#Exec(F) \leq 2 \\ &\wedge[\#Exec(F) = 2 \Rightarrow \\ &\qquad (f_1(x) = f_0(x)^\frown < lastOb(f_1(?D) > \\ &\qquad \wedge\, t_1 \geq \tau)]\}\end{aligned}$$

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 $Interval_\alpha(D!e)$ is a final segment of T with least element τ which is greater than the least element of $Interval_\alpha(e)$.

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

$$Interval_\alpha(!D := e) = Interval_\alpha(D!e)$$

asynchronous outputput

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_n denote the alternative command

$$[\Box_{i=1}^{n} g_i \rightarrow C_i]$$

The boolean part of g_i is reprented by b_i. Let

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

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 $Interval(ALT)$[8].

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 π 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 π' 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

$$\forall i \in \omega - \{0\} : \; t_{g_i} = t_{f_i} + t$$

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 β is set of behaviours then $Shift(\beta, t)$ is the set of behaviours

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

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 Π is the set

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

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]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 $\forall i \in [1,n] : g'_i = b_i$
$\wedge GCOM_i = \emptyset \Rightarrow C'_i = C_i$
$\wedge GCOM_i =\neq \emptyset \Rightarrow C'_i = IO_i; C_i$

Case 1. $[\exists i \in [1,n] : b_i = \textbf{true} \wedge External(\|(\alpha) \uparrow Dual(GCOM_i), 1)]$
$\wedge \neg Broken(\|(\alpha)_1)$

The predicate $\wedge \neg Broken(\|(\alpha)_1)$ ensures that no variable in $Dual(GCOM_i)$ is assigned the value $\perp$ or δ.

$$\begin{aligned}\Gamma(Alt_n, \alpha, k) \;=\; \{(\pi,\beta) \;| \; & \pi \in \Pi_k(Alt_n) \\ & \wedge[\forall F \in \beta : \alpha \in Env(F)] \\ & \wedge[\exists\, i \in [1,n] :\; b_i \bullet EvalFun(\alpha) = \textbf{true} \\ & \quad \wedge[\exists\, (\pi',\beta') \in \Gamma(C_i, \alpha, k) :\; \exists t \in Interval(Alt) : \\ & \qquad \pi' = \pi \oslash VAR(C_i) \\ & \qquad \wedge \beta \uparrow VAR(C_i) = Shift(\beta', t) \\ & \qquad \wedge \forall F \in \beta : \\ & \qquad\quad \#Exec(F \uparrow (VAR(Alt_n) - VAR(C_i))) = 1]]\}\end{aligned}$$

Case 2. $BB \bullet EvalFun(\alpha) = \textbf{false}$
The alternative command fails in this case.

$$\begin{aligned}\Gamma(Alt_n, \alpha, k) \;=\; \{(\pi,\beta) \;| \; & \pi \in \Pi_k(Alt_n) \\ & \wedge \forall F \in \beta :\; \alpha \in Env(F) \;\wedge\; Failed(F_1)\}\end{aligned}$$

Case 3. $BB \bullet EvalFun(\alpha) = \textbf{true}$
$\wedge(\; [\forall i \in [1,n] : b_i = \textbf{false} \vee \neg External(\|(\alpha) \uparrow Dual(GCOM_i), 1)]$
$\vee Broken(\|(\alpha)_1))$

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

$$*[\Box_{i=1}^{n} g_i \rightarrow C_i]$$

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. $BB \bullet EvalFun(\alpha) = \textbf{true}$

$$\Gamma(Rep_n, \alpha, k) \;=\; \Gamma(Alt_n; Rep_n, \alpha, k)$$

Case 1. $BB \bullet EvalFun(\alpha) = \textbf{false}$
In this case the repetitive command terminates immediately.

$$\Gamma(Rep_n, \alpha, k) \;=\; \{(\pi, \beta) \mid \forall F \in \beta :\; \alpha \in Env(F) \;\wedge\; \#Exec(F) = 1\}$$

program constructor

Let C' represent the command **prog** C **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 λ. The empty environment of a command C is represented by $ProgEnv(C)$.

The semantic function of C' is given by

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

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 specifcation 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

[1] C.A.R. Hoare, "Communicating Sequential Processes", *Comm. ACM*, vol. 21, no. 8, 1978, pp. 666-677.

[2] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International U.K., 1985.

[3] R. Milner, "Calculi for Synchrony and Asynchrony", *Theor. Comput. Sci.*, vol. 25, 1983, pp. 267-310.

[4] G.M. Reed and A.W. Roscoe, "A Timed Model for Communicating Sequential Processes", in: *Lecture Notes in Comp. Sci.* 226, Springer-Verlag, Heidelberg, 1986.

[5] A. Zwarico and I. Lee, "A Syntax and Semantics for Deterministic Real-Time Computing", Tech. Rep., Dept. Computer and Information Science, Univ of Pennsylvania, Philadelphia, 1987.

[6] J. Hooman, "A Compositional Proof Theory for Real-Time Distributed Message Passing", Tech. Rep., Dept. of Mathematics and Computing Science, Eindhoven Univ of Technology, Eindhoven, 1987.

[7] R. Gerth and A. Boucher, "A Timed Failures Model for Extended Communicating Processes", Tech. Rep. TR.4-4(1), Dept. of Mathematics and Computing Science, Eindhoven Univ. of Technology, Eindhoven, 1987.

[8] C. Huizing, R. Gerth, and W.-P. de Roever, "Full Abstraction of a Real-Time Denotational Semantics for an OCCAM-like Language", in: *Proc. 14th ACM Symp. POPL*, 1987, pp. 223-238.

[9] R. Koymans, R.K. Shyamasundar, W.-P. de Roever, R. Gerth, and S. Arun-Kumar, "Compositional Semantics for Real-Time Distributed Computing", in: *Lecture Notes in Comp. Sci.* 193, Springer-Verlag, Heidelberg, 1985, pp. 167-190.