Research report 123

WHAT'S 'REAL' ABOUT REAL-TIME SYSTEMS?

Mathai Joseph, Asis Goswami

(RR123)

A real-time system is typically a concurrent (or distributed) system whose computations and actions must satisfy some real-time constraints. Guaranteeing that such a system will in fact meet its constraints can thus be viewed either in terms of some extended model of program correctness or as a problem of scheduling, eg one of establishing a feasible schedule. However, the crucial distinction between real-time and other concurrent programs is not merely that of time, but that the former must execute on a system with limited resources. In this paper, we give an informal account of a semantic model of programs which may execute with limited resources: thus, the model can serve both as the basis for a formal specification system for real-time programs and to characterise real-time scheduling problems.

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

May 1988

What's 'Real' about Real-time Systems ?

Mathai Joseph, Asis Goswami

University of Warwick[1]

Abstract

A real-time system is typically a concurrent (or distributed) system whose computations and actions must satisfy some real-time constraints. Guaranteeing that such a system will in fact meet its constraints can thus be viewed either in terms of some extended model of program correctness or as a problem of scheduling, eg one of establishing a feasible schedule. However, the crucial distinction between real-time and other concurrent programs is not merely that of time, but that the former must execute on a system with limited resources. In this paper, we give an informal account of a semantic model of programs which may execute with limited resources: thus, the model can serve both as the basis for a formal specification system for real-time programs and to characterise real-time scheduling problems.

[1]Address for correspondence: Department of Computer Science, University of Warwick, Coventry CV4 7AL, U.K. This work was supported by research grant GR/D 73881 from the Science and Engineering Research Council.

1 Introduction

Viewed simply, a real-time system has two main parts: an external environment with a number of devices such as sensors and actuators, and a programmed system which registers events from the sensors and responds by producing actions to drive the actuators. It is called a 'real-time' system because the incoming events are distributed in time and induce timing constraints on the sequencing of the outgoing events.

Different aspects of such systems have been studied: specification (eg, [13], [9], [8]), languages (eg, [11], [2]), models and semantics(eg, [15], [6]), and scheduling (eg, [12], [16]). It is clear that in many of these respects a real-time system has a great deal in common with any other concurrent system and it is sometimes argued that there are no significant differences at all between them. Equally, it is well-accepted that real-time systems do have many specific scheduling problems, especially those relating to feasibility for hard-real-time, that are not of concern in general concurrent systems. So the 'real-time' problem is often seen with two views: as one of representing time in the execution of a concurrent program or as one of scheduling the tasks in a real-time program to meet particular timing constraints. But there is some common ground, in that there are scheduling results that should clearly be related in some way to results from other studies of concurrency (a good example is synchronization, where Mok [12] has shown that the decision problem about scheduling certain classes of processes which use semaphores to ensure mutual exclusion is NP-hard).

In this paper, we argue that a real-time program is a concurrent program which is executed on a system with *limited resources*. If processors were infinitely fast, memory unlimited and channels had no restrictions of capacity, for example, a real-time system would be like any other concurrent system and there would be no 'real-time' problem at all. Since resource limitations are central to the execution of a real-time program, they must be represented in semantic models for real-time languages.

This paper gives an informal description of a semantic model for concurrent programs which execute with limited resources. We show how the model can represent limitations in any general resource and how the problem of computing these limitations can be related to problems of scheduling in hard-real-time systems.

2 A Semantic Model

Consider a programming language in which a program has a number of variables and there is an assignment operation to place a value in a variable (it is easy to generalize this operation to permit multiple assignment [4]). Assume each variable is represented by a *sequence* of elements which are the values assigned to the variable during an execution of the program; let this sequence be the *observed value* of the variable. Initially, the observed value of each variable is the empty sequence λ and each assignment to the variable adds a value to a variable. So, for the program with

the variables x and y and the sequence of operations

$$x := 3; y := x + 2 \tag{1}$$

the observed values of x and y before and after the execution of these assignments are

	Observations		
	1	2	3
x	λ	$<3>$	$<3>$
y	λ	λ	$<5>$

Define an *observation* of a program as the set of observed values of its variables before or after the execution of an assignment operation. Then a *behaviour* of a program can be represented by a sequence of observations. In some cases, eg with non-deterministic commands, a program may have many possible behaviours and the *computation* of a program is the set of all its possible behaviours.

In general, a behaviour of a program is an infinite sequence (or an ω-sequence) of observations. Each observation is made at a discrete point in a time domain of real numbers and the number of an observation is an index into this time domain. More formally, an observation θ over the set $VAR(C)$ of variables of a command C is a pair $(Time(\theta), Function(\theta))$. Two observations θ_1 and θ_2 are equal *iff*

$$Time(\theta_1) = Time(\theta_2) \wedge Function(\theta_1) = Function(\theta_2) \tag{2}$$

The Function component of an observation θ of a command C gives the observed values of the variables of C (ie, maps the variables of C into their observed values) at $Time(\theta)$. Each element in an observed value is either taken from the (countable) value space of C or is a distinguished value representing a particular result of executing C. For example, the sequence of commands

$$x := 3; x := x + y \tag{3}$$

will result in x taking the value of the expression $x + y$ only when that is defined. But if before executing the commands the observed values of x and y are empty sequences, the new value of x is not in the value space of C and the command fails. Failure of a sequential command induces the value $\perp$ in all its variables.

	Observations		
	1	2	3
x	λ	$<3>$	$<3, \perp>$
y	λ	λ	$<\perp>$

Let $\theta_0, \theta_1, \cdots$ be an infinite sequence of observations of an execution of a command C. The command is assumed to start execution at time 0, so $Time(\theta_0) = 0$. The observation $\theta_{i+1}, i > 0$, is made after the execution of at least one primitive command of C following the observation θ_i (provided that C has not stopped, normally or abnormally) so we have

$$Time(\theta_{i+1}) > Time(\theta_i) \wedge Function(\theta_{i+1}) \neq Function(\theta_i) \tag{4}$$

Obviously, the observed value of at least one variable of C is extended by one element from observation θ_i to θ_{i+1}. Also, no observed value is extended by more than one element. If at some observation $\theta_k, k > 0$, the command has failed or normally terminated then all subsequent observations of the command remain constant. Clearly, if (t, f) and (t, g) are two observations in a behaviour, then $f = g$.

The executed part of F, $Exec(F)$, is the maximum initial subsequence of a behaviour F in which no two successive observations are equal; for a non-terminating execution, $Exec(F) = F$. The set of all possible behaviours of a command C is represented by $\beta(C)$; this is the set of all behaviours with all possible initial observed values in $VAR(C)$.

3 Sequential and Parallel Composition

We have already seen, informally, how the execution of sequential commands is modelled in the semantics. In this section, we consider more general sequential and parallel composition of commands.

Sequential Composition

Let C represent the command $C_1; C_2$ formed by the sequential composition of the commands C_1 and C_2.

The use of sequential composition widens the space of possible initial *observed* values of variables. For example, if C_2 can execute then each of its variables which is also in $VAR(C_1)$ has exactly one element in its initial observed value, and that element is the last value assigned to it in the execution of C_1. However, the observed values of the other variables of C_2 are all λ.

The predicate $Consistent_{(;)}$ tests whether the behaviours $F_1 \in \beta(C_1)$ and $F_2 \in \beta(C_2)$ are consistent with respect to sequential composition, i.e., whether they combine to form behaviours of the command C. Informally, F_1 and F_2 are consistent with respect to sequential composition if either F_1 does not terminate or F_1 terminates and the final state of C_1 in F_1 is the same as the initial state of C_2 in F_2. The condition required to hold when F_1 terminates is now more formally described.

Let $last(s)$ be the last (or, rightmost) element in a sequence s. Since F_1 terminates, $Exec(F_1)$ must be finite; let

$$last(Exec(F_1)) \;=\; (t, g)$$

and $(0, h)$ be the first observation in F_2. Then F_1 and F_2 are consistent with respect to sequential composition, written as $Consistent_{(;)}(F_1, F_2)$, if

$$\forall v \in VAR(C_1) \cap VAR(C_2) :\; h(v) \;= < last(g(v)) >$$

Let $Combine_{(;)}$ be a function which, for sequential composition, can be used to combine the behaviours of two commands to produce the behaviour of the sequential composition of those commands. Thus, $Combine_{(;)}(F_1, F_2)$ can be used to combine

the behaviours F_1 and F_2 to produce a behaviour of $C_1; C_2$. If $\neg Consistent_{(;)}(F_1, F_2)$ then $Combine_{(;)}(F_1, F_2)$ is undefined, and if F_1 does not terminate then

$$Combine_{(;)}(F_1, F_2) \;=\; F_1$$

Suppose F_1 terminates; let F_1 and F_2 be consistent and let their observation sequences be represented by $\theta_0, \theta_1, \cdots$ and $\psi_0, \psi_1, \cdots$ respectively. Let $last(Exec(F_1)) = (t, g)$ and $\psi_0 = (0, h)$. Then

$$Combine_{(;)}(F_1, F_2) \;=\; F$$

where F is a behaviour

$$(t_0, f_0), (t_1, f_1), (t_2, f_2), \cdots$$

with $t_0 = 0$ and t_i for $i > 0$, and f_i for $i \geq 0$ defined as follows.

The domain of each f_i is $VAR(C_1) \cup VAR(C_2)$ because F is a behaviour of $C_1; C_2$. Let the length of $Exec(F_i)$ be k. All the observed values in F_1 must be in F. Thus

$$\forall v \in VAR(C_1) : \; \forall i \in [1, k-1] : \; f_i(v) \;=\; Function(\theta_i)(v)$$

The observed value of each variable of $VAR(C_2) - VAR(C_1)$ in the first k observations of F must be the same as its first observed value in F_2. So,

$$\forall v \in VAR(C_2) - VAR(C_1) : \; \forall i \in [1, k-1] : \; f_i(v) \;=\; h(v)$$

From the kth observation onwards, each variable in $VAR(C_1) - VAR(C_2)$ has the same observed value as in the $(k-1)$th observation. Thus,

$$\forall v \in VAR(C_1) - VAR(C_2) : \; \forall i \geq k : \; f_i(v) \;=\; g(v)$$

Let $v \in VAR(C_1) \cap VAR(C_2)$. Consider some $i \geq k$. Since F_1 and F_2 are consistent, the last value in $g(v)$ is the same as the first value in $Function(\psi_{k-i+1})(v)$. The rest of the sequence $Function(\psi_{k-i+1})(v)$ should be appended to (the right of) $g(v)$ to obtain $f_i(v)$. The time components of the observations of F are defined by

$$\forall i \in [1, k-1] : \; t_i \;=\; Time(\theta_i) \;\wedge \forall i \geq k : \; t_i \;=\; Time(\psi_{i-k+1})$$

The set of all possible behaviours of the command C is

$$\beta(C) \;=\; \{Combine_{(;)}(F_1, F_2) \mid F_1 \in \beta(C_1) \;\wedge\; F_2 \in \beta(C_2) \;\wedge\; Consistent_{(;)}(F_1, F_2)\}$$

Example 1: Consider two commands C_1 "$x := x + 2; y := y + x$" and C_2 "$y := y + 1; z := y$" with the associated behaviours F_1 and F_2:

	Behaviour F_1 of C_1				
Observations	1	2	3	4	
Time	0	3.2	8.4	8.4	
x	$< 3 >$	$< 3, 5 >$	$< 3, 5 >$	$< 3, 5 >$	
y	$< 4 >$	$< 4 >$	$< 4, 9 >$	$< 4, 9 >$	$\cdots$

	Behaviour F_2 of C_2				
Observations	1	2	3	4	
Time	0	2.3	3.6	1.3	
y	$<9>$	$<9,10>$	$<9,10>$	$<9,10>$	
z	$<2>$	$<2>$	$<2,10>$	$<2,10>$	$\cdots$

Here, F_1 terminates and F_1 and F_2 are consistent so $Combine_{(;)}(F_1, F_2) = F$ is the behaviour represented by

Observations	1	2	3	4	5	6	
Time	0	3.2	8.4	10.7	12.0	12.0	
x	$<3>$	$<3,5>$	$<3,5>$	$<3,5>$	$<3,5>$	$<3,5>$	
y	$<4>$	$<4>$	$<4,9>$	$<4,9>$	$<4,9,10>$	$<4,9,10>$	$\cdots$
z	$<2>$	$<2>$	$<2>$	$<2>$	$<2,10>$	$<2,10>$	

Parallel Composition

Let $C \triangleq C_1 \| \cdots \| C_n$ represent the parallel composition of the commands $C_1, \ldots, C_n$, $n \geq 2$. Assume the processes communicate with each other using some communication mechanism $\mathcal{M}$.

The predicate $Consistent_{(\mathcal{M})}$ tests whether the behaviours $F_i \in \beta(C_i)$, $i \in [1, n]$ are consistent with respect to $\mathcal{M}$; in other words, this predicate serves as a definition of the mechanism $\mathcal{M}$. The conjunction of $Consistent_{(\mathcal{M})}$, taken over all the communication mechanisms of the language under consideration, is represented by $Consistent_{(\|)}$.

Communication among the components of a parallel composition is modelled as a pair of primitive actions – *output* and *input*. Like any other primitive action, these are also represented by assignments to variables: output is recorded by appending the communicated value to the observed value of an *output variable*, and input is recorded in an *input variable*. If v is an output variable, $Complement(v)$ is the set of input variables which receive values from v. Similarly, for an input variable w, the set $Complement(w)$ includes all the output variables which sends values to w. Usually, the complement of an input variable is a singleton.

The introduction of combinators such as parallel composition needs further elaboration of the way in which observations are made. Suppose an observation of a command A records a communication of a value to A. In this case, all possible values are taken into consideration (including values representing communication failure). In general, all possible interactions with the environment of A are considered; the interactions depend on the combinators of the language. For example, the use of sequential composition requires that λ or any any single-element sequence over the value space of the language be taken as a possible initial observed value of a variable of a command.

The expressions $Input_{\mathcal{M}}(C)$ and $Output_{\mathcal{M}}(C)$ define the set of input and the set of output variables which record communication across the boundaries of the component commands $C_1, \ldots, C_n$ of C. The communication mechanism $\mathcal{M}$ generates a subset of $\beta(C_1) \times \cdots \times \beta(C_n)$, characterized by $Consistent_{(\mathcal{M})}$.

Consider the specific example of *asynchronous* communication defined by the predicate $Consistent_A$. The function *Complement* will have singleton values and the sets $Input_{\mathcal{M}}(C)$ and $Output_{\mathcal{M}}(C)$ will be disjoint.

For uniformity, let $Consistent_A$ have pairs of observations as its arguments. Let $\theta \triangleq (t_1, f)$ and $\psi \triangleq (t_2, g)$ be observations of F_i and F_j respectively. Let

$$v \in (Input_A(C) \cup Output_A(C)) \cap VAR(C_i)$$

ie v is an input or output variable of C_i for asynchronous communication with some other process. If

$$Complement(v) \cap VAR(C_j) = \emptyset$$

then the observed values of v in θ are not constrained in any way by the observed values of the variables in $VAR(C_j)$.

The observed value of a variable v in $Input_A(C) \cup Output_A(C)$ is of the form $s\hat{}q$, where "$\hat{}$" represents sequence catenation, and $q \in \{\lambda, \delta\}$. δ is the failure value representing deadlock in parallel composition, in the same way that $\perp$ is the failure value for sequential composition. Failure of a parallel composition induces the value δ in all its variables. Define $Data(v) = v$ if $q = \lambda$ and $Data(v) = s$ otherwise.

Let $u \in VAR(C_j)$ and $u \in Complement(v)$.

Case 1. $t_1 \leq t_2$

Let v be an input variable (implying that u is an output variable). Then we have

$$Data(f(v)) \preceq g(u)$$

ie 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 transmission of data.

If v is an output variable and the observed value of v has $\perp$ or δ as the last element, then only that data which is in $f(v)$ can be in $u(V)$ at a time later than t_1. Thus

$$Data(u) \preceq f(v)$$

However if $Data(v) = v$, then either the above relation holds or we have

$$t_1 < t_2 \wedge f(v) \preceq g(u)$$

Case 2. $t_1 > t_2$

This case 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.

Two behaviours $< F \triangleq F_0, F_1, \ldots >$ and $< G \triangleq G_0, G_1, \ldots >$ are consistent with respect to asynchronous communication if

$$\forall i \in \omega : \forall j \in \omega : Consistent_A(F_i, G_j)$$

The set of all possible behaviours of C is

$$\beta(C) = \{Combine(F_1, \ldots, F_n) \mid [\forall i \in [1,n] : F_i \in \beta(C_i) \wedge Consistent_{(\|)}(F_1, \ldots, F_n)\}$$

The function $Combine_{(\|)}$ first transforms each behaviour F_i into another behaviour F_i' over $VAR(C_i)$; it is possible that $F_i = F_i'$ but it is not necessary that $F_i' \in \beta(C_i)$.

Example 2: Consider the parallel combination of three processes P_1, P_2 and P_3 whose actions can be represented by the following commands:

$P :: P_1 \| P_2 \| P_3$
$P_1 :: \; !!b;\; < a := a + b;\; !!b >;\; < a := a + 2;\; !!b >$
$P_2 :: \; y := y - 1;\; ??x;\; !!y;\; ??x$
$P_3 :: z := 1$

where ?? and !! represent asynchronous input and output respectively. Let the individual behaviours F_1, F_2 and F_3 be

Behaviour F_1 of P_1

Observations	1	2	3	4	5	
Time	0	2	3	5	5	
a	$< 2 >$	$< 2 >$	$< 2,5 >$	$< 2,5,7 >$	$< 2,5,7 >$	
b	λ	$< 3 >$	$< 3,4 >$	$< 3,4,6 >$	$< 3,4,6 >$	$\cdots$

Behaviour F_2 of P_2

Observations	1	2	3	4	5	
Time	0	1	4	5	5	
x	λ	λ	$< 3 >$	$< 3,\delta >$	$< 3,\delta >$	
y	$< 3 >$	$< 3,2 >$	$< 3,2 >$	$< 3,2,\delta >$	$< 3,2,\delta >$	$\cdots$

Behaviour F_3 of P_3

Observations	1	2	3	4	5	
Time	0	1	1	1	1	
z	λ	$< 1 >$	$< 1 >$	$< 1 >$	$< 1 >$	$\cdots$

Thus, since all communication is asynchronous, $Input_A(P) = \{x\}$ and $Output_A(P) = \{b,\ y\}$, with $Complement(x) = \{b\}$. Now F_1 and F_2 are consistent with respect to parallel composition so they can be combined to produce a behaviour of P. However, F_2 and F_3 show that a deadlock is detected in P_2 at time 5, whereas P_1 terminates normally. Since deadlock is an abnormal termination, the composite command must also deadlock at time 5. So F_1 is transformed into the following behaviour F_1'.

Behaviour F_1' of P_1

Observations	1	2	3	4	5	
Time	0	2	3	5	5	
a	$< 2 >$	$< 2 >$	$< 2,5 >$	$< 2,5,\delta >$	$< 2,5,\delta >$	
b	λ	$< 3 >$	$< 3,4 >$	$< 3,4,\delta >$	$< 3,4,\delta >$	$\cdots$

The behaviour F_2 remains unchanged, ie $F_2' = F_2$. The behaviour of F_1 would become F_1' even if the time of the 4th observation of F_1 was greater than 5, or if P_1 and P_2 did not communicate with each other.

□

It should be clear from this example how the behaviours are transformed by $Combine_{(\|)}$ so we omit the formalisms. Exactly the same approach is taken to transform the behaviours when failure (represented by $\perp$) occurs.

We now come back to the general case of the command $C \triangleq C_1 \| \cdots \| C_n$.

The expression $Combine_{(\|)}(F_1, \ldots, F_n)$ evaluates to a behaviour F (of C) which, if *restricted* to $VAR(C_i)$, for any $i \in [1, n]$, produces a sequence S of observations which can be shown to have a particular relation with the behaviour F_i'. Informally, S is either the behaviour F_i' or an ω-sequence of observations of C_i obtained from F_i' by iteratively introducing observations such that time still increases monotonically with the observation index (until it becomes constant) and there are one or more pairs of successive observations, (t_1, f) and (t_2, g), in S such that $f = g$ but $t_1 \neq t_2$. We write $S \rightsquigarrow F$ (read as "S is compatible with F"). The relation $\rightsquigarrow$ is now formally developed.

Let F be an ω-sequence of observations. Then $Timeseq(F)$ is a function from ω into the set of non-negative real numbers such that $Timeseq(F)(k)$ is the time component of the kth observation of F. If F is a behaviour, then $Timeseq(F)$ is an ω-sequence of times $< t_0, t_1, \ldots >$ having the following properties:

P1. $t_0 = 0$
P2. $[\exists i \in \omega : t_i = t_{i+1}] \Rightarrow [\forall k > i : t_k = t_i]$

Let W be the set of all ω-sequences of observations of a command C such that for all $F \in W$, $Time(F)$ satisfies the properties P1 and P2 above. Then there exists a relation $\rightsquigarrow$ (read as "compatible with") from W to $\beta(C)$ defined as follows: if $F \in W$ and $G \in \beta(C)$ then $F \rightsquigarrow G$ if any observation in G is also an observation in F, and for any observation (t, f) in F, there is an observation (t', f) in G such that t' is the largest observation time in G which is less than or equal to t.

The behaviour

$$F \triangleq < \theta_0, \theta_1, \ldots >$$

of a command C when *restricted* to a nonempty subset X of $VAR(C)$, $F \uparrow X$, is an ω-sequence of observations $< S_0, S_1, \ldots >$, where, for all $i \in \omega$, $Time(S_i) = Time(\theta_i)$ and $Function(S_i)$ is the restriction $Function(F_i) \uparrow X$ of the function $Function(\theta_i)$ to the set X.

Let F be an ω-sequence of observations. The expression $Timeset(F)$ is the set of the elements in $Timeseq(F)$. If

$$C \triangleq C_1 \| \cdots \| C_n$$

and $F_i \in \beta(C_i)$, $1 \leq i \leq n$, then $Combine_{(\|)}(F_1, \ldots, F_n)$ is an ω-sequence F of observations of C such that $Timeseq(F)$ satisfies the properties P1 and P2 above, and

a) $Timeset(F) = \bigcup_{i=1}^{n} Timeset(F_i')$
b) $\forall i \in [1, n] : (F \uparrow VAR(C_i)) \rightsquigarrow F_i'$

Condition (b) states that $Combine_{(\|)}$ merges several behaviours into one in which the temporal ordering of the observations is preserved.

Example 3: Given the program $P :: P_1 \| P_2 \| P_3$ of the previous example, the following behaviour F of P is produced by $Combine_{(\|)}(F_1, F_2)$.

Behaviour F of P

	1	2	3	4	5	6	7	
Time	0	1	2	3	4	5	5	
a	$<2>$	$<2>$	$<2>$	$<2,5>$	$<2,5>$	$<2,5,\delta>$	$<2,5,\delta>$	
b	λ	λ	$<3>$	$<3,4>$	$<3,4>$	$<3,4,\delta>$	$<3,4,\delta>$	...
x	λ	λ	λ	λ	$<3>$	$<3,\delta>$	$<3,\delta>$	
y	3	$<3,2>$	$<3,2>$	$<3,2>$	$<3,2>$	$<3,2,\delta>$	$<3,2,\delta>$	
z	λ	$<1>$	$<1>$	$<1>$	$<1>$	$<1,\delta>$	$<1,\delta>$	

Note that $(F \uparrow \{a, b\})$ is a sequence which can be obtained from ths behaviour by removing the three rows corresponding to x, y, and z. Clearly, $(F \uparrow \{a, b\})$ is not a behaviour because the 1st and the 4th observations are repeated, while the observations do not become constant before the 5th observation. This justifies the use of the relation "$\rightsquigarrow$" in the definition of $Combine_{(\|)}$.

The soundness of the function $Combine_{(\|)}$ is easily established. When it is defined, the expression $Combine_{(\|)}(F_1, \ldots, F_n)$, evaluates to a unique behaviour of C. Suppose $Combine_{(\|)}(F_1, \ldots, F_n)$ evaluates to an ω-sequence F of observations of C. Let F be the sequence $< \theta_0, \theta_1, \ldots >$. First, we show that F is a behaviour of C.

Let (t, f) be any observation in F. For F to be a behaviour of C

- the observations in F, except the first, must be made only when at least one primitive command of C terminates, and
- for all $v \in VAR(C)$, the observed value of v at time t must be given by $f(v)$

Let $t > 0$. Since

$$t \in \bigcup_{i=1}^{n} Timeset(F_i)$$

there is some $i \in [1, n]$ such that $t \in Timeset(F_i)$. Obviously, one or more primitive command of C_i, and hence of C, terminates at time t.

Consider some $v \in VAR(C)$. There is some C_i such that $v \in VAR(C_i)$. Now, $(t, f \uparrow VAR(C_i))$ is an observation in $F \uparrow VAR(C_i)$. We have

$$F \uparrow VAR(C_i) \rightsquigarrow F_i$$

If $t \in Timeset(C_i)$, then $(t, f \uparrow VAR(C_i))$ is an observation in F_i. Then $(f \uparrow VAR(C_i))(v)$, and hence $f(v)$, is the observed value of v at time t. If $t \notin Timeset(C_i)$, then $(t', f \uparrow VAR(C_i))$ is an observation in F_i, where t' is the maximum of all times in $Timeset(F_i)$ which are less than t. Since one or more primitive commands of C terminates at time t, and no primitive command of C_i terminates in the interval

$(t', t]$, the observed value of the variable v at time t is unchanged from time t', and is $(f \uparrow VAR(C_i))(v)$, ie, $f(v)$.

So F is a behaviour of C. Suppose $Combine_{(\|)}(F_1, \ldots, F_n)$ can evaluate to some other behaviour F' of C. Since $Timeset(F) = Timeset(F')$, from the discussion above we know that both F and F' give the observed values of the variables of C at each point in the same time-space. Thus, $F = F'$.

4 Commands and Combinators

In general, a program is formed by composing commands with combinators of which the sequential and parallel combinators are two examples. Commands may be nested, ie a command may contain other commands and combinators. A command is *primitive* with respect to a combinator if it contains no further occurrences of that combinator; for example, a single or a multiple assignment statement is primitive with respect to the sequential combinator ";" and a program consisting of sequential commands only is primitive with respect to the parallel combinator "$\|$" .

Combinators such as ";" and "$\|$" are *syntactic* combinators and they appear in a program with a precedence which determines the structure of the program. The semantics of a particular programming language is defined in terms of the semantics of its syntactic combinators and the commands that are primitive with respect to these combinators.

Let $\diamond \triangleq \{\diamond_1, \diamond_2, \cdots, \diamond_n\}$ be the finite set of syntactic combinators of a language. Then for each $\diamond_i$, define the set of distinguished values assigned to the variables at the end of all possible finite executions of any command which is primitive with respect to $\diamond_i$ as

$$Stop_{\diamond_i} \triangleq \{terminate\} \cup fail_{\diamond_i} \tag{5}$$

where *terminate* denotes normal termination and $fail_{\diamond_i}$ is the set representing all possible abnormal terminations of commands. For example, for the parallel combinator "$\|$"

$$Stop_{\|}(C) = \{terminate\} \cup \{deadlock\} \tag{6}$$

Let $Fail \triangleq \bigcup_{i=1}^{n} fail_{\diamond_i}$.

In terms of observations and behaviours, a finite execution of a command always produces a constant observation: for normal termination, the values in this observation are from the value space of the language while for abnormal termination the last value in each of the observed values of $VAR(C)$ is a distinguished value (in this case δ for deadlock).

Note that we are not using the value *terminate* to denote normal termination because this is represented by constant observation without any failure value assigned to the variables. However, description of livelock properties of parallel programs would need normal termination to be represented by *terminate*. For example, consider the parallel command

$$P :: P_1 \| \ldots \| P_n$$

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

$$F \uparrow VAR(P_i) \rightsquigarrow H$$

and observation in H becomes eventually constant. But the observed values in H have elements only from the value space of the language. Thus H represents a normally terminating execution whereas it should represent starvation. So, starvation can be distinguished from termination only if the termination of a command C is indicated by appending the distinguished value *terminate* to the observed values of all communication variables of C when C has terminated.

Let V be the value space of programs. Assume that V is countable and equipped with an equality $(=)$ relation which is the minimum reflexive relation on V. 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

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

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

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

The function component of an observation θ, ie $Function(\theta)$ maps $VAR(C)$ into W such that

$$[\exists v \in VAR(C) : \ (Function(\theta))(v) \in V^\dagger] \\ \Rightarrow [\forall v \in VAR(C) : \ (Function(\theta))(v) \in V^\dagger]$$

Thus, whenever C fails, the corresponding failure element is induced in all variables of the command. If θ is the corresponding observation, the predicate $Failed(\theta)$ is true.

Let U be the set of all one-element sequences over the set $V \cup Fail$. A *behaviour* of C is an ω-sequence F given by

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

An element θ_k of this sequence is an observation (t_k, f_k), where $t_k = Time(\theta_k)$ and $f_k = Function(\theta_k)$. The sequence F has the following properties:

a) $t_0 = 0$
b) $\forall v \in VAR(C) : \ f_0(v) = \lambda \vee f_0(v) \in U$
c) $\forall k \in \omega : \ \forall v \in VAR(C) : \ [\exists s \in U \cup \{\lambda\} : \ f_{k+1}(v) = f_k(v)^\frown s] \wedge t_k < t_{k+1}$
d) $\forall k \in \omega : \ (f_k = f_{k+1} \vee Failed(\theta_k)) \Rightarrow [\forall j \in \omega : \ j > k \Rightarrow \theta_j = \theta_{k+1}]$
e) $\forall k \in \omega : \ \theta_k \neq \theta_{k+1} \Rightarrow t_k < t_{k+1} \wedge f_k \neq f_{k+1}$

The semantics of a combinator $\diamond_i$ defines how the behaviour of the composite command it forms can be obtained from the behaviours of its arguments. The examples of sequential and parallel composition in the previous section show how this can be done with a uniform approach for any combinator. Let $\diamond_i$ be an n-ary combinator, and $C \triangleq \diamond_i(C_1, \ldots, C_n)$. Then the semantics of $\diamond_i$ is given by the formula

$$\beta(C) = \{Combine_{\diamond_i}(F_1,\ldots,F_n) \mid [\forall i \in [1,n] : F_i \in \beta(C_i)] \wedge Consistent_{\diamond_i}(F_1,\ldots,F_n)\}$$

The predicate $Consistent_{\diamond_i}$ tests whether the behaviours F_i describe only those observed values of the variables of C_i that are consistent with respect to the computation mechanism of $\diamond_i$. If so, then these behaviours can be combined to produce a behaviour of C.

It not necessary that $Consistent_{\diamond_i}$ should be a relation on all the variables of C. Associated with $\diamond_i$ is a function $VAR_{\diamond_i}$ such that only variables of $VAR_{\diamond_i}(C)$, which is a nonempty subset of $VAR(C)$, participate in the operation of $\diamond_i$. In the case of parallel composition, for example, $VAR_{\|}(C_1\| \cdots \|C_n)$ is the union of the sets produced by the *Input* and *Output* functions corresponding to the communication mechanisms.

The function $Combine_{\diamond_i}$ essentially reorders the observed values of its argument commands into a new total order of time values which is obtained from the observation times and certain properties of the observed values (eg, deadlock in the case of parallel composition) in the arguments. This is explained in more detail in the next section.

Certain behaviours can be derived from a given behaviour of command C in two ways – by hiding variables and by hiding observations. For example, in the program

$$P :: Q_1; (P_1\| \cdots \|P_n); Q_2$$

the variables used only for communication between the parallel programs P_i should not be visible when P is considered as a sequence of three commands. Thus, in general, if

$$C \triangleq \diamond(C_1,\ldots,C_N)$$

then C can be transformed into another command C' such that

$$VAR(C') = VAR(C) - VAR_{\diamond}(C)$$

and the behaviours of C' can be obtained from those of C by hiding variables as appropriate. This hiding operation can easily be defined with the help of restriction. If the subset Y of the variables of C is hidden, a behaviour F of C gives a behaviour F' of the resulting command such that

$$F \uparrow (VAR(C) - Y) \rightsquigarrow F'$$

Variable hiding is meaningful only when some useful equivalence between the original and the resulting commands can be defined.

Hiding an observation from a behaviour can be used to avoid the description of unnecessary detail in the actions of a command. If a certain observation in a behaviour of a command C records assignment only to a proper subset Y of $VAR(C)$, then hiding the variables in Y obviously hides that observation. However, the command C is transformed by variable hiding. Observation hiding without command transformation is achieved by deleting an observation θ_i from a subsequence "$\theta_{i-1}, \theta_i, \theta i + 1$" of a behaviour such that

$$Time(\theta_{i-1}) < Time(\theta_i) < Time(\theta_{i+1})$$

This condition is justified because hiding the first or a constant observation is not meaningful. The deletion of θ_i will require readjustment of all subsequent times and deletion of the last elements of the observed values in θ_i from the observed values in all subsequent observations.

We have assumed that it is not necessary while recording a behaviour of a command C, to observe all instances of assignments to the variables of C. Therefore, the set $\beta(C)$ should be *closed* under the operation of observation hiding. A more detailed examination of this operation and the domains $\beta(C)$ based on formal definitions can be found in [7].

It is usual practice to define the semantics of a program in terms of the semantics of its commands but this carries with it the assumption that the computer system on which the program may be executed has unlimited resources. But if for example a program is too large to be stored in the computer system, it may need to be divided into units ('working sets') which each require less memory than the whole program. Thus in resource- limited cases a program may need to be divided into units separated by *non-syntactic* combinators. Note that, in general, these combinators have no fixed precedence relation with syntactic combinators and many of the difficulties of scheduling a program on a system with limited resources come from attempting to do this in terms of arbitrary non-syntactic combinators when the demands upon the resources are made by the program at times determined by its syntactic combinators (see eg Belady [1]). We shall return to this point later when discussing the semantics of limited resource execution.

5 Representation of Time

In a sequential program, the value of the time component of successive observations increases monotonically until the observation becomes constant. So we may say that time imposes a total order on these observations. In a parallel program, observations on individual sequential processes have a total order and the predicate $Consistent_{(||)}$ specifies the time relation between events in the different processes; for example, synchronous communication takes place when the values of the sending and receiving variables change simultaneously (ie in the same observation). With independent observation of time in the diferent processes, parallel composition requires tranformation of the times of each process into times in a new total order so that the observations in different processes can be related and combined into an observation of the parallel command (as in [10]). Clearly, this is possible (in general for any combinator) if the meaning of any command C is given by a pair $(I, \beta(C))$, where I is the interpretation of 'time' used in the behaviours in $\beta(C)$. More formally, I is a well-order with the least element represented by 0_I. Thus, 'time' in a command is strictly a local property.

The notion of simultaneity, which cannot easily be expressed in trace logics, is not restricted to synchronous communication and can apply to any two events in a parallel program which are observed with appropriately transformed local times. The relative ordering of the observations of two processes during a parallel composition brings with it the operational possibility of one process 'waiting' for another.

This has often been modelled in semantics as the execution of a *wait* t command, where t denotes a time interval. It is difficult to see how, in a distributed system, t can be effectively computed if process executions are truly independent. Sometimes, however, a *wait* t instruction is explicity used in a process as a timeout (eg, as in Ada's timed entry call and delay alternative): in our model, such use of a wait command would be equivalent to synchronization by parallel composition with a clock process, the value t being interpreted in the new order produced by this combination.

Thus, with these semantics, the values of time in a parallel program are relative, rather than absolute. However, this time can always be related to the time of some reference, such as a clock process, or even to a Standard Time which is itself defined in relation to the more stable reference of an atomic clock. This essentially relative view of time stands in contrast with the concept of an absolute global clock which has been used elsewhere to relate time with program executions.

6 Limited Resources and Real-time Scheduling

A program C (or, in general, any command) can be partitioned in a number of ways depending on the combinators used in its description. Most often, the behaviour of a real-time program is determined by partitioning it with respect to one of its syntactic combinators, such as the parallel combinator $\|$.

Without paying too much attention to the specific syntax of the language used, assume that C is partitioned into m units or processes $C_1 \ldots C_m$ which are primitive with respect to $\|$ so that $\forall i, j \in 1 \cdots m, VAR(C_i) \cap VAR(C_j) = \phi$. Then, m is the *degree of maximum parallelism* of C; if C is a sequential command, clearly $m = 1$. The program C has a maximally parallel *implementation* if it is executed on a system with m processors and the semantics of this execution are the same as that for any system with v processors, $v > m$.

As for many practical real-time systems, assume the processes $C_1 \cdots C_m$ are *periodic* and have infinite, non-divergent executions, for example of the form $C_i :: *[c_{i1}; c_{i2}; c_{i3} \cdots c_{in_1}]$ where each c_{ip} is a terminating sequential or communicating command. The periodicity p_i of a process C_i can be established by letting one communicating command, say c_{i_k}, complete its execution at a fixed time after the completion of its previous execution, all times being relative to some external clock. Let there be a *deadline* d_i for each execution of process C_i and let Sl_i be the *minimum slack*, which the smallest difference between the time of completion of any execution of C_{i_k} and the deadline. For the program to meet its real-time deadlines, ie for some implementation to be 'feasible', the minimum slack for each process must be positive for the maximally parallel execution.

Let the m processors be identical and assume that they have an execution speed such that the minimum slack for at least one process is zero. This gives the minimum processor speed for the maximally parallel execution to be feasible. Let S be the subset of processes with non-zero slack times; if $S = \phi$ the implementation is optimal (but note that in general the problem of finding such an implementation is NP-hard). The more interesting case is when $S \neq \phi$; each C_i in S has some slack time and it

is possible to consider *limited* processor executions of C, ie executions of C in which fewer than m processors are used.

For simplicity, assume that each process executes only on the processor to which it is assigned (if this is not the case, a time overhead is incurred each time a process changes processors, and this must be represented by adding a 'switch' command to the process). Then a limited processor execution of C is one in which it is executed on r processors, $r < m$, so that each of $m - r$ processes shares a processor with one or more other processes. For example, if $r = m - 1$, the following problem must be solved if the $m - 1$ limited processor execution is to be feasible:

- to choose two of the m processes such that their interleaved execution on one processor remains feasible.

An interleaved execution of processes c_i and c_j can be represented by a behaviour in which the primitive commands of c_i and c_j are executed in some sequential order so that the partial order of their observations in $\beta(C)$ is preserved. In general, for any positive integer k, it is possible to find the subset $\beta_k(C)$ of $\beta(C)$ which contains all behaviours of C executed on k processors [7]. If m is the degree of parallelism of C, then $\forall k \geq m : \beta_k(C) = \beta_m(C)$. If $k < m$, then $\beta_k(C)$ gives all behaviours of C with interleaving. For parallel composition of $C \triangleq C_1, \ldots, C_n$, we obtain $\beta_k(C)$ by combining behaviours from the sets $\beta_{k_i}(C_i)$, for $i \in [1, n]$, such that

$$k_1 + k_2 + \cdots + k_n \geq m$$

where m is the degree of parallelism of C. This condition is required so that no processor can remain unused if a command needs it. For each syntactic primitive command C, the degree of parallelism is unity, and consequently, only $\beta_1(C)$ suffices as the limited processor semantics of C.

Let $\beta_{||}(C_{m-1})$ be the set of all behaviours of the parallel partition of C, ie of the processes $c_1 \cdots c_n$, for which any two processes have interleaved execution. The problem is then to choose the set of all behaviours in $\beta_{||}(C_{m-1})$ for which the execution of all the processes remains feasible. If this set is empty, there is no feasible $m - 1$ limited processor execution of C. In general, we must consider the set of all behaviours in $\beta_{||}(C_r)$ for which process executions are feasible.

For the general case, even given the behaviours of all the processes, finding a feasible interleaving of a subset of the processes is an NP-complete problem (sequencing within intervals [5]) as would be the 'optimal' problem of finding the smallest number r of processors for which a feasible interleaving is possible. Moreover, finding all possible behaviours for a program is an exponential problem even if the times for each primitive command are known *a priori*.

Another way of approaching the problem is to consider the speed of execution of the (identical) processors. If Sp is the minimum speed, in some suitable units, of the processors for which the maximally parallel implementation with m processors is feasible, then feasible solutions can be found for processors of speed $Sp \times k$, where k is a positive integer, provided some k processes are interleaved on each processor. In fact, it is easily shown that *any* k processes can be interleaved for a feasible m/k

limited processor implementation, and the difficult problem of process allocation is avoided. *Sp* is related to the longest execution path of the processes in a program: finding such a path is in general of exponential complexity but simplifying program structure, eg by considering only deterministic programs or programs with simple communication mechanisms, can reduce the effective cost of the computation.

It is easier to take scheduling decisions based on the syntactic units of a program (eg, its processes) than on non-syntactic units as the latter have no fixed precedence relation with program structure. But it would be possible for, say, a compiler to divide the instruction space of a program into units of some chosen size, or for the run-time system to do the same for the data space. Neither of these possibilities is an attractive means of executing a large program in a small amount of memory: compromises adopted in previous systems include the segmentation schemes used in Multics [3] and some Burroughs systems [14], where a new syntactic boundary (the segment) was devised to provide a higher-level unit of memory allocation than either the syntactic unit of a program variable or the non-syntactic unit of the page. In terms of our semantics, non-syntactic units would need to be delimited by special combinators and then the model would be as applicable as for syntactic combinators.

7 Conclusions

A real-time system has to execute its actions in some relation to processes of the external world and it has to do so with a limited set of resources. It is the real limitations of these resources that make the essential difference between real-time systems and other systems, not merely the introduction of time. The semantics described here show how it is possible to model the execution of a program in terms of different combinators, under a regime of limited resources, and it seems clear that any semantics for real-time systems will need to take resource limitations into account. Given such a semantics, it is possible to consider how other real-time techniques, such as priority-based scheduling, can be modelled but that would require a separate presentation.

References

[1] L.A. Belady, R.A. Nelson, G.S. Schedler, " An Anomaly in Space-time Characteristics of Certain Programs Running in a Paging Machine", *Comm. ACM, 12*, 6, 1969, pp349-353.

[2] G. Berry, L. Cosserat, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics", LNCS 197, Springer-Verlag, 1985, pp389-449.

[3] J.B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems", *J. ACM, 12*, 4, 1965, pp589-602.

[4] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, 1976.

[5] M.R. Garey, D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman, 1979.

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

[7] A. Goswami and M. Joseph, "A Semantic Model for the Specification of Real-Time Processes", Res. Rep. 121, Dept of Computer Science, Univ of Warwick, Coventry, 1988.

[8] 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.

[9] R. Koymans, W.-P. de Roever, "Examples of a Real-time Temporal Logic Specification", LNCS 207, Springer-Verlag, 1985, pp232-252.

[10] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Comm. ACM*, *21*, 7, 1978, pp558-565.

[11] I. Lee, V. Gehlot, "Language Constructs for Distributed Real-time Programming", Proc. IEEE RTS Symposium, 1985, pp57-66.

[12] A.K. Mok, *Fundamental Design Problems of Distributed Systems for the Hard Real-time Environment*, Ph.D. Thesis, M.I.T., 1983.

[13] A.K. Mok, "SARTOR - a Design Environment for Real-time Systems", Proc. COMPSAC 85, 1985, pp174-181.

[14] E.I. Organick, *Computer Systems Organization: The B5700/B6700 Series*, Academic Press, New York, 1973.

[15] G.M. Reed and A.W. Roscoe, "A Timed Model for Communicating Sequential Processes", LNCS 226, Springer-Verlag, 1986.

[16] J.A. Stankovic, A.K. Ramamritham, S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-time Systems", *IEEE Trans. Comp.*, *34*, 12, 1985, pp1130-1143.