Advanced Design Tools
for Aircraft Systems and Airborne Software
(SafeAir)

Component performance evaluation
Issue 1.0

Aérospatiale-Matra/Airbus (AMB)
DaimlerChrysler Aerospace Airbus (DA)
Israeli Aircraft Industrie (IAI)
I-Logix (ILX)
Institut National de Recherche en Informatique et en Automatique (INRIA)
Oldenburger Forschungs- u. Entwicklungsinstitut f. Informatik-Werkzeuge u. –Systeme (OFFIS)
Siemens (SIE)
Snecma Moteurs (SNM)
Techniques Nouvelles d'Informatique (TNI)
Verilog Telelogic (VER)
Weizmann Institute of Science (WZ)

Issued by: INRIA

Abstract: This paper describes principles of timing prediction implementation for applications developed within SafeAir. These principles are presented using the example of the SIGNAL synchronous language, which is the (hidden) basis of ModelBuild.

Contents

Page

1 Preface 3
1.1 Table of revisions 3
1.2 Table of references and applicable documents/standards 3
1.3 Table of abbreviations 3
1.4 Table of terms and definitions 3
2 Scope of this document 4
3 Motivation and general approach 4
4 Temporal interpretation of SIGNAL processes 5
4.1 Signal availability dates 6
4.2 Non-functional interpretations 7
4.3 Temporal interpretation renaming conventions 8
5 The date computation model 9
5.1 The ideal parallel case 10
5.1.1 Pure dataflow 11
Obtaining Results 12
5.1.2 Accounting for the control-flow 13
5.2 Modeling of sequential execution 17
5.2.1 Modeling dynamic scheduling 19
5.2.2 Example: dynamic scheduling based on input availability 19
5.2.3 Obtaining results in the presence of scheduling 20
5.2.4 Minimizing the scheduling modeling complexity 20
6 Temporal interpretation of the HCDG 23
6.1 I/O nodes 23
6.2 Operation nodes 24
6.3 Down-sampling nodes 25
6.4 Multiplexing nodes 26
6.5 Memory nodes 27
7 Summary 29

1 Preface

1.1 Table of revisions

Issue	Date	Description & Reason for the Modification	Affected Sections
0.1	07/Feb/01	Creation	

1.2 Table of references and applicable documents/standards

Reference	Title and editorial information	Author or Editor	Year
[KL96]	Profiling of SIGNAL Programs and its application in the timing evaluation of design implementations, Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems, IEE, pages 6/1-6/9, HP Labs, Bristol, UK	A. Kountouris, P. Le Guernic	1996
[GL99]	Code generation in the SACRES project. Towards System Safety, Proceedings of the Safety-critical Systems Symposium, SSS'99, Springer, Huntingdon, UK	T. Gautier, P. Le Guernic	1999
[MB]	ModelBuild V0.9 manual, SafeAir document	F. Dupont	2000

1.3 Table of abbreviations

Abbreviation	Full description
HCDG	Hierarchical Conditional Dependency Graph

1.4 Table of terms and definitions

Term	Definition

2 Scope of this document

This document describes the principals of a framework for reasoning about the execution time of programs developed in the SafeAir project. This description is based on the SIGNAL language, which is the basis of ModelBuild. However, the timing evaluation framework implemented by TNI in ModelBuild will not necessarily be a direct application of the principals presented here. The timing evaluation framework implemented in ModelBuild is described in the ModelBuild manual [MB], which will be updated following the evolutions. The framework described in this document will be made available during the SafeAir project in the INRIA POLYCHRONY environment, based on the SIGNAL language. POLYCHRONY can handle ModelBuild models via the ModelBuild export of textual SIGNAL programs. Both approaches, that directly accessible in ModelBuild, and that provided in POLYCHRONY, will be complementary.

New versions of the present document might be provided, following the evolutions of the project.

Part of the results presented here were obtained previously by Apostolos Kountouris [KL96]. However, they have not been implemented in the SIGNAL environment. This will be done in SafeAir. In addition, previously obtained results are further developed.

Given an implementation Q of a program and a model of time consumption for each of the atomic actions in Q, we propose to automatically generate a program $T(Q)$ homomorphic to Q; $T(Q)$ is the parallel composition of the images $T(Q_i)$ of the components Q_i (including communications) of Q. $T(Q_i)$ are given by the user as SIGNAL components whose interfaces are composed of integer signals $T(ak)$ instead of the original ak signals. $T(ak)$ represents the sequence of the availability dates for the occurrences of the original ak signal.

$T(Q)$ is thus a model of real time consumption of the application (functional specification and architectural support), that can be simulated. Some real time properties to be satisfied can also be described as predicates in SIGNAL. Then some of these properties may be checked by using verification tools.

3 Motivation and general approach

During the *Development Lifecycle* of an application, at some point of the design process, implementation details have to be introduced in the specification in order to transform it into an executable one. Introducing these details at an early phase has the advantage of refining the specification without a great cost in effort and time. Consequently a means of evaluating different implementation alternatives is needed as a facility in the design space exploration which by the way is one of the principal preoccupations in Hardware/Software co-design.
We consider here the development of such a facility in the context of the SIGNAL language and its POLYCHRONY environment. Speaking of a facility we mean the tool that implements the mechanism of design evaluation and the methodology that implements the policy of using the tool to perform such an evaluation.

To address the temporal validation issues, a particular interpretation of synchronous SIGNAL specifications is defined. It is called the *temporal interpretation* because it aims in extracting the temporal dimension implicit in a functional specification when implementation related design choices are known. First, it is described how a generic interpretation scheme can be used to derive the temporal interpretation. This interpretation consists of a specific renaming for the specification entities (i.e., signals, operators, processes) and a specific computation model that extracts the desired information. This is the date computation model that computes output dates from input dates, accounting for data dependencies and control execution flow. This model is described in detail and it is shown how it can easily account for a variety of *system-level* implementation related decisions, like for instance partitioning and scheduling. A parameterization scheme is used to account for lower-level implementation decisions like specific component choice (i.e., processors and interconnects).
To evaluate the system performance against the timing constraints, performance evaluation platforms can be automatically built using the temporal interpretations. Composing the temporal interpretation with the initial specifications and some additional processes, special simulators can be derived. These simulators co-simulate the functional and temporal behaviors of a system. The additional processes needed are one that models the scheduling policy (if one is defined) and one that validates the timing requirements. For the first one, a brief summary of how scheduling decisions can be represented in the HCDG (Hierarchical Conditional Dependency Graph, which is the internal representation of a SIGNAL program), is given. For the second, a representation of timing constraints as constraint edges between HCDG nodes, is used. These edges can be used to produce the date equations between the corresponding dates in the temporal interpretation, evaluating whether the timing constraint is respected.

4 Temporal interpretation of SIGNAL processes

An *interpretation* of a SIGNAL specification is a SIGNAL process that exposes a different view of the initial SIGNAL specification. The structure of the interpretation process is essentially the same but its computations expose another aspect of its behavior. The *temporal interpretation* exposes the time aspect and permits to see how an implementation of a specified functionality will behave over time.

A SIGNAL process specifies a piece of functionality as a system of equations. This functionality can be of arbitrary complexity ranging from primitive operations to complex tasks. Parallely composing kernel processes (operators) results in more complex processes which in turn can be further composed to build even more complex ones. In this way a system specification can be viewed as a composition of simpler processes. Even more, an initial, implementation independent functional specification in SIGNAL, can be transformed in a functionally equivalent specification that models various system-level design choices (e.g., distribution, scheduling). Consequently, as a design evolves from higher to lower levels of abstraction, where implementation related design decisions and choices are known, a model in SIGNAL can be derived.

For each SIGNAL process independently of its complexity level, another SIGNAL process can be automatically derived to model its temporal behavior on a given implementation. These processes are called *temporal interpretations*. For a SIGNAL

process *P* its *temporal interpretation* for an implementation *I* will be denoted by T(P_I), where P_I is the SIGNAL process that models implementation *I* of *P*. Thus, if a system specified by a SIGNAL process *P* has a variety of possible implementations I_1, to I_k, then each implementation can be modeled by:

$$P_{I(i)} \qquad i \in [1,k]$$

and for each $P_{I(i)}$ a temporal interpretation T($P_{I(i)}$) can be derived.

In this way a comparative performance evaluation of the different implementations can be performed and the design space of possible implementations can be effectively explored before committing the design to a particular one. Such an approach permits to concentrate the design effort to a set of candidate implementations.

4.1 Signal availability dates

For each signal in the initial SIGNAL specification a date signal is defined in its temporal interpretation:

$$x \in P \rightarrow \mathrm{T}(x) \in \mathrm{T}(P)$$

For any signal *x* in *P* we have a *date_x* in T(*P*) with *x* synchronous to *date_x*.

$$P \rightarrow \mathrm{T}(P)$$
$$x \rightarrow \mathrm{T}(x) = date\_x$$
$$\text{x ^= date\_x}$$

These date signals are a kind of time-stamps providing the availability times for the values of the corresponding signals in the functional specification, in respect to a global time reference. Depending on the implementation context, time can be measured using either *physical time* units or full *clock cycles*. In the first case the date signals are positive real numbers and in the second positive integers. From a cycle count integer measurement we can pass to physical time measurement by multiplying the cycle count to the cycle period.

Throughout this presentation, at each node a delay is associated. This delay is represented by the same data type as the data type used to represent dates and is a function of several parameters. The actual node delay is obtained by giving values to these parameters. The delay depends on parameters like: operation performed by the node, data types involved, chosen implementation, etc. Furthermore, a delay can be represented by a pair of numbers corresponding to the worst and best case delays. Having delays represented by intervals results in dates represented as intervals too. Computing these dates takes into account the processing delays. How these dates are actually computed is the topic of a subsequent paragraph, what is important to underline is that this date mechanism permits us to pass from logical to physical time. This passage is necessary in order to evaluate whether a particular implementation respects the *synchrony* hypothesis.

4.2 Non-functional interpretations

The *temporal interpretation* of a SIGNAL specification is just a special case of a general *non-functional interpretation.* Let us briefly discuss some aspects of such non-functional interpretations. The *non-functional interpretations* are SIGNAL processes and as such they can be decomposed into a *control* and a *data* part. The control computations are identical to those in the initial processes from which the interpretations are derived. What changes are the data computations since they extract the information related to the particular interpretation.
For a SIGNAL process P we know that $P = C_P \mid D_P$, with C_P the control part of the process P and D_P its data part. Similarly an interpretation of P, denoted by $TR(P)$, decomposes into a control and a data part:

$$TR(P) = C_{TR(P)} \mid D_{TR(P)}$$

Since the interpretation of a complex process can be defined as the recursive composition of the interpretations of the constituent processes for $TR(P)$ we have:

$$TR(P) = TR(C_P) \mid TR(D_P)$$

with:

$$TR(C_P) = C_{TR(C_P)} \mid D_{TR(C_P)} \text{ and } TR(D_P) = C_{TR(D_P)} \mid D_{TR(D_P)}$$

For the control part we have:

$$C_{TR(P)} = C_P$$

The process of obtaining an interpretation $TR(P)$ of a process P is graphically depicted in Figure 1. The data part (D_P) of the process P computes output values (O_D) from input values (I). The computations are conditioned by activation events (H) computed in the control part (C_P). To compute the activation conditions H, C_P uses Boolean input signals (I_b) and intermediate Boolean signals B computed by D_P. Finally, certain outputs are output events (H_O) computed by C_P. The control parts of the initial process and its interpretation are identical, but the data computations differ. The data computations in $TR(P)$ extract the information of interest, implicit in the initial specification P.

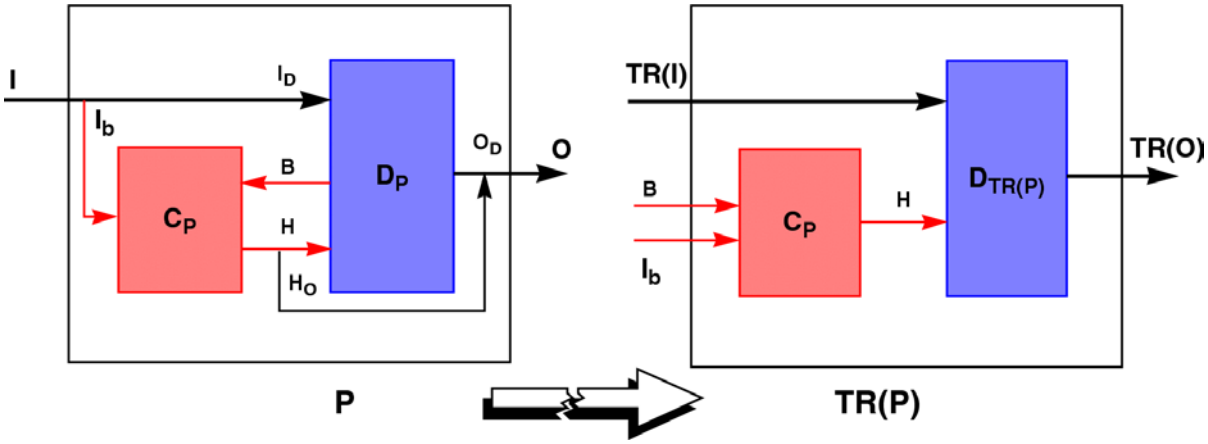

Figure 1. The generic interpretation of process P

The SIGNAL kernel operators are the simplest processes that can be composed to build more complex ones. Similarly, the *interpretation* of a process can be viewed as the composition of the interpretations of the primitive processes making up the initial process.
Each interpretation defines a renaming scheme for the entities contained in the initial specification. These specification entities consist of signals and processes (primitive and higher-level). For each specific interpretation a set of renaming conventions has to be defined. Signal renaming consists of adding a prefix to the signal names, i.e. "*pref_*". The renamed signals have also a type depending on the particular interpretation. Process renaming consists of adding a prefix to the process name, i.e. "*pref_*", to build the name of the kernel interpretation process that corresponds to the kernel process in the initial program.
The interpretations of the kernel processes perform the appropriate computations relating to a particular interpretation. These compute the value of the renamed output from the values of the renamed inputs. The computations may be functions of the interpretation parameter values passed to the interpretation process. The interpretations of the kernel processes are organized in a collection which is the interpretation library *pref*_LIB. This library for each interpreted process is augmented with the interpretations of external function calls and other separately compiled processes, used in the initial process.

For instance, the interpretation of the addition monochronous operator **C := A + B** is:

*pref*_C := *pref*_ADD {interpretation parameters} (*pref*_A, *pref*_B)

The participating signal A, B, C can be traced in the resulting interpretation as *pref*_A, *pref*_B, *pref*_C respectively. Similarly the addition operator to the *pref*_ADD process that computes the value for *pref*_C from the values of *pref*_A, *pref*_B. The interpretation parameters are optional and depend on the particular interpretation.

Another example of kernel operator illustrates the addition of control inputs to the resulting interpretation process in order to compute the activation events.
C := A when B yields:

*pref*_C := *pref*_WHEN {interpretation parameters} (*pref*_A, *pref*_B, B)

As it can be seen the Boolean signal B is part of the interpretation interface since the interpretation results depend on its value.

4.3 Temporal interpretation renaming conventions

In the previous paragraph it was shown how a generic interpretation can be obtained for SIGNAL processes of arbitrary complexity. To obtain the temporal interpretation, two things need to be defined:

- A set of renaming conventions of SIGNAL specification entities. Specification entities are: signals, kernel processes, higher order processes and functions.
- The interpretation computations for each SIGNAL kernel process.

The interpretation processes have their interfaces derived from the initial processes. These interfaces consist of signals used to define activation events inside the process bodies and renamed signals participating in the interpretation computations. The interface derivation for the temporal interpretation T(*P*) of process *P* is shown in Figure 2. Process *P* can be viewed as the composition of its control part (C_P) and its data part (D_P). Similarly, the temporal interpretation process can be decomposed in a control ($C_{T(P)}$) and a data part ($D_{T(P)}$). $C_{T(P)}$ should be identical to C_P.

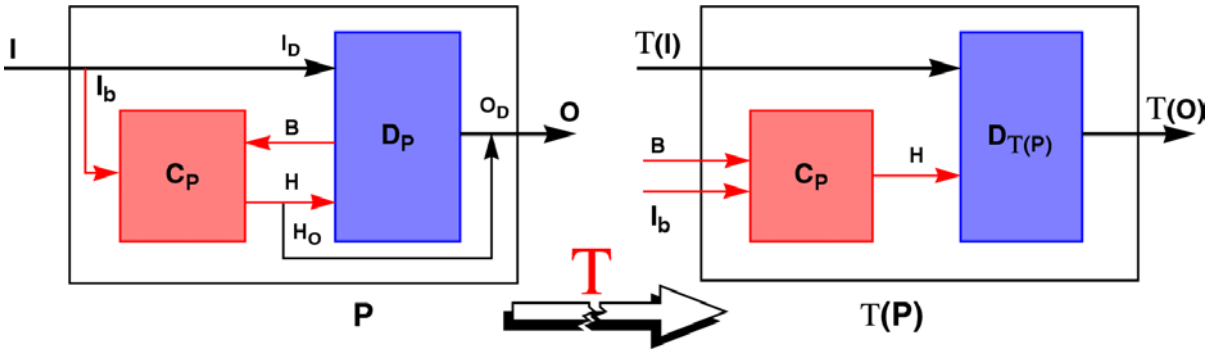

Figure 2. The temporal interpretation process interface

The interface of T(*P*) is derived as follows:

- The input signals T(*I*) are obtained by renaming the input signals *I* of *P*. Renaming consists in prefixing the names of signals in *I* by "*date_*" of data type appropriate to express time measurements. This can be *integer* if time is measured in full clock cycles or *floating point* if it is measured in seconds.
- I_b are obtained by extracting from *I* those signals contributing in the definition of activation events belonging to *H*. I_b consist of signals of Boolean or event types.
- *B* are signals of Boolean type computed by D_P. Since the computations of D_P disappear in $D_{T(P)}$, they have to become inputs of T(*P*).
- The output signals T(*O*) are obtained by renaming the output signals *O* of *P*. Renaming follows the same conventions as for T(*I*).

Finally, the process name of T(*P*) is obtained by prefixing the name of *P* with "*T_*". The above interface derivation and renaming schemes apply to any process *P* of arbitrary complexity, and consequently to the SIGNAL kernel processes.

5 The date computation model

What a SIGNAL program defines is how the availability (presence/absence) and values of the inputs influences the availability and values of the outputs. Having assigned at each input an availability date then we can exactly calculate the output availability dates if we account for all those things that affect the program execution flow and the various operation delays.
To present the date computation model, in Figure 3 the basic translation rules that when applied to a SIGNAL process yield its temporal interpretation, are given. These rules are:

- Substitute each signal by its corresponding date
- Substitute each operator by an appropriate SIGNAL process
- Provide parameter values for the parameters determining the exact operation delay

In this example for an addition operation C := A + B the date of the addition result (*date_C*) is a function of the addition argument dates (*date_A*, *date_B*). The delay of the addition operator is taken into account inside the “T_ADD” process which models a desired implementation by means of a set of parameters (*add_pars*).

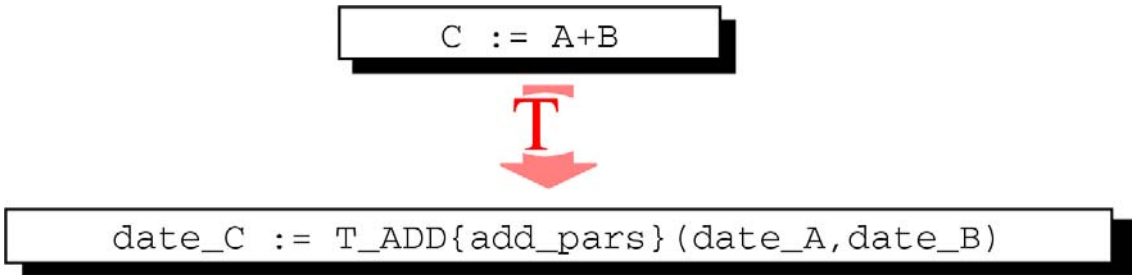

Figure 3. Simplified translation rules for temporal interpretation

The temporal interpretation process parameters reflect information that is either found in the HCDG or is provided by the user (it is the topic of a subsequent section). For instance floating point addition might be costlier than integer addition, in this case the addition type becomes a parameter. For now it is sufficient to know that this information can be extracted and subsequently used in order to obtain an operation delay.

It is clear that reasoning about the temporal behavior of a program is a complex task that necessitates finding solutions to different sub-problems. In order to better present our approach to temporal property extraction we proceed in a step-by-step fashion going from the simpler towards the more complex case. We start by making a set of simplifying assumptions that correspond to an ideal execution. At each step we remove one by one the simplifying assumptions by introducing a new aspect so that in the end we treat a realistic execution scheme that covers all the details concerning the problems of temporal interpretation generation and temporal property extraction.

5.1 The ideal parallel case

In this case an ideal execution platform is considered. The basic assumptions are: an unlimited number of resources (processors and communication links); zero communication delays; constant and deterministic operation delays. Under these assumptions at every instant of program execution all the potential *functional* parallelism can be exploited. For an ideal parallel execution we assume zero communication delays and finally to make things even simpler we also assume constant operation delays. This last assumption allows us to neglect any implementation details pertaining to specific components that the operation executes and the possible variations of its execution, and concentrate on the presentation of the basic ideas. For this case we examine two types of programs starting with pure data ones where the SIGNAL HCDG can be considered as a dataflow graph, on which the basic principles of date calculation are

presented. Examining next the case of programs that also contain control-flow we extend the model for the more general case of HCDG's. Finally, we present how results can be obtained by using the temporal interpretation and how these results can be exploited always staying within the SIGNAL context.

5.1.1 Pure dataflow

In a pure data program there is no control so its HCDG can be viewed like a synchronous dataflow graph. In this case the data dependencies remain unchanged in every iteration of the system represented by the graph.

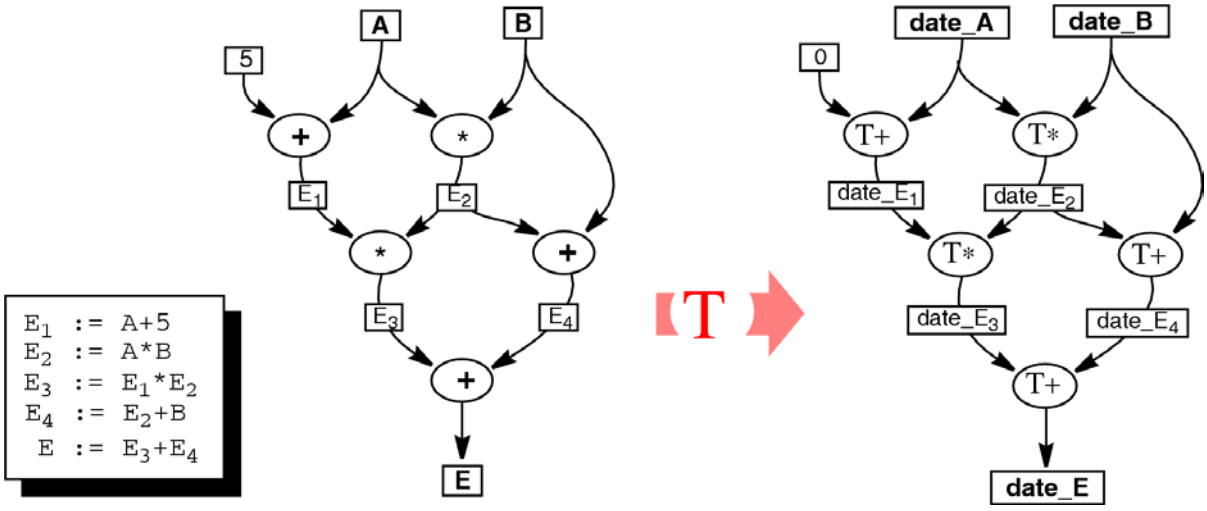

Figure 4. Date computations for pure data programs

Every node corresponds to an operation and has a set of incoming edges representing the operation arguments and a set of outgoing edges representing the use of the operation result in other operations. An example is shown in Figure 4 where the graph on the left corresponds to the program shown in the shaded box. Input/Output and intermediate signals are shown in rectangles. Ovals correspond to operation nodes. On the right of Figure 4 the graph corresponding to the temporal interpretation of the program, is given. This graph is produced by applying the translation rules. The graph of the corresponding temporal interpretation has exactly the same structure but the edges now represent dates and the nodes the computation of the result date as a function of the argument dates.

For each node, we give the Δ_{op} which is the delay of the operation *op* in the initial graph. The date computation of T_{op} consists in finding the maximum of incoming dates and adding to it the delay Δ_{op} corresponding to the operation in the original graph. This is graphically depicted in Figure 5. In this way starting with the system input dates and by traversing the graph from inputs to outputs, we obtain the system output dates.

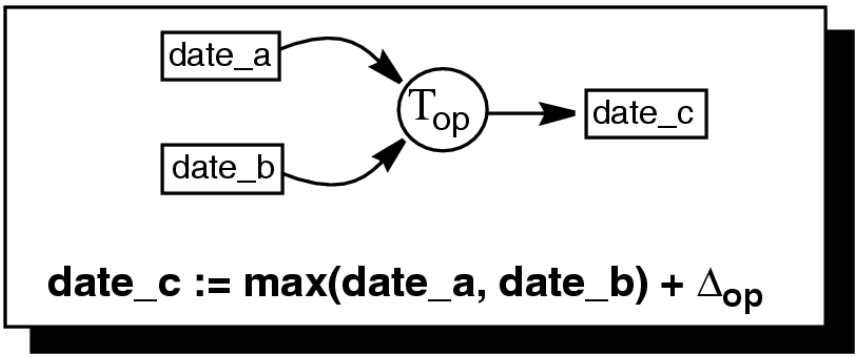

Figure 5. Date computations in the interpretation of a monochronous operation node

Obtaining Results

Even though this execution scenario seems to be very unrealistic its results can give the designer an upper limit of system capabilities. Intuitively, any possible implementation will have a lower performance and thus knowing this upper limit may prompt for algorithmic optimizations at the functional specification level.

An information that can be obtained is the *optimal latency* of the system which is the time it takes to produce the outputs once the inputs are available. This latency is optimal in the sense that the execution corresponds to an ideal execution scheme. By setting the input dates to *zero* we obtain each output date. This date corresponds to the time it takes to execute the path in the graph that leads from the inputs to this output. The *maximum* of the output dates corresponds to the *critical* (longest delay) path in the graph.

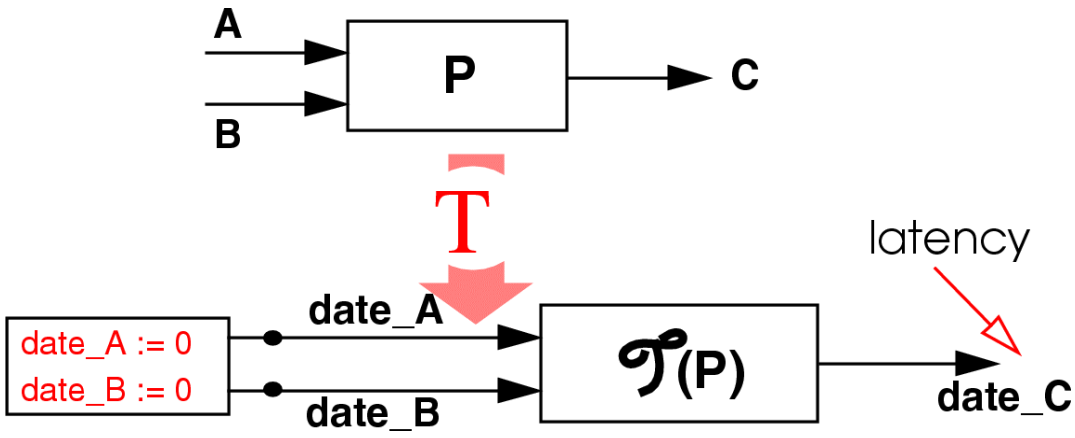

Figure 6. Finding the optimal latency of a system

In Figure 6 the necessary configuration in order to obtain the optimal latency, is shown. In this example for the output *C* the optimal latency is given by:

$$latency\_C = max\ ((date\_C - date\_A), (date\_C - date\_B))$$

if $date\_A = date\_B = 0$ then:

$$latency\_C = max\ (date\_C, date\_C) = date\_C$$

5.1.2 Accounting for the control-flow

Staying always in the ideal parallel case we consider now programs that contain control and thus the execution paths from inputs to outputs may differ, over time, depending on the control conditions involved. Control branching in a SIGNAL program is introduced by means of the two control expressions shown in Figure 7 which are part of the SIGNAL language kernel.

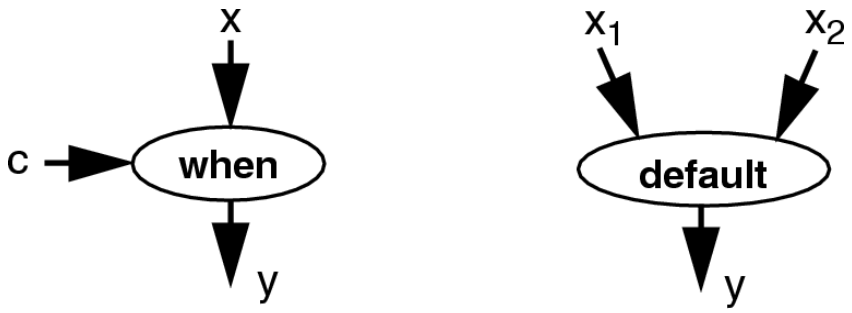

Figure 7. SIGNAL kernel constructs influencing the control execution flow

The first one is the *sub-sampling* and is represented in SIGNAL as:

```
y := x when c
```

y, x are of any valid type and c is a Boolean. The semantics of "when" can be informally defined as:

y gets the value of x if x is present and c is present with value `true`.

The second is referred to as *priority multiplexing* and is represented in SIGNAL as:

```
y := x1 default x2
```

Informally:

y gets the value of x_1 if it is present or of x_2 if x_2 is present and x_1 is absent. If both x_1 and x_2 are absent y is absent as well.

In assigning a value to y, x_1 is considered having higher priority over x_2 meaning that in the case that both are present y gets its value from x_1.

The temporal interpretations of these processes are shown in Figure 8. For "`y := x when c`" its temporal interpretation is:

`date_y` gets the maximum value of `date_x` and `date_c` if `date_x`, `date_c` are present and `c` is present `true`. To it the delay assigned to the `when` operator is added.

For "`y := x1 default x2`" its temporal interpretation is:

`date_y` gets the value of `date_x1` if it is present or `date_x2` if it is present and `date_x1` is absent. To it the delay assigned to the `default` operator is added. If both `date_x1` and `date_x2` are absent `date_y` is also absent.

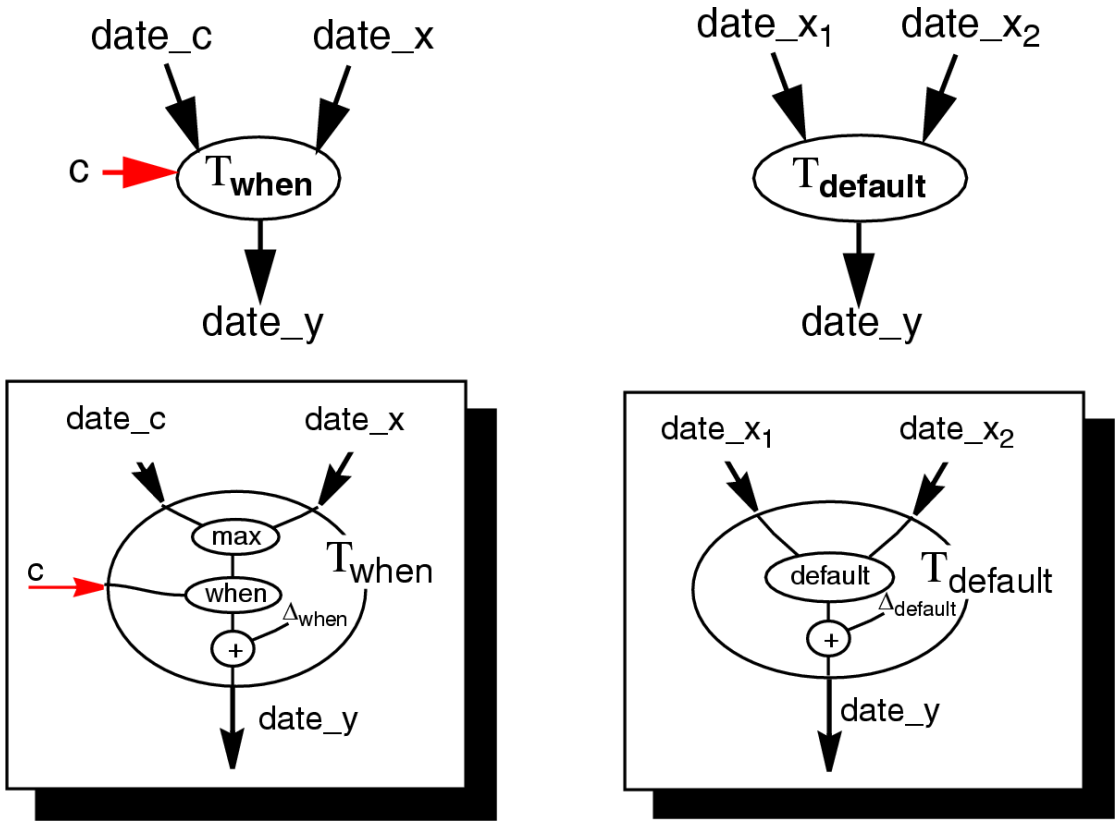

Figure 8. Temporal interpretations of the kernel control processes

An example of temporal interpretation of control data programs is shown in Figure 9.

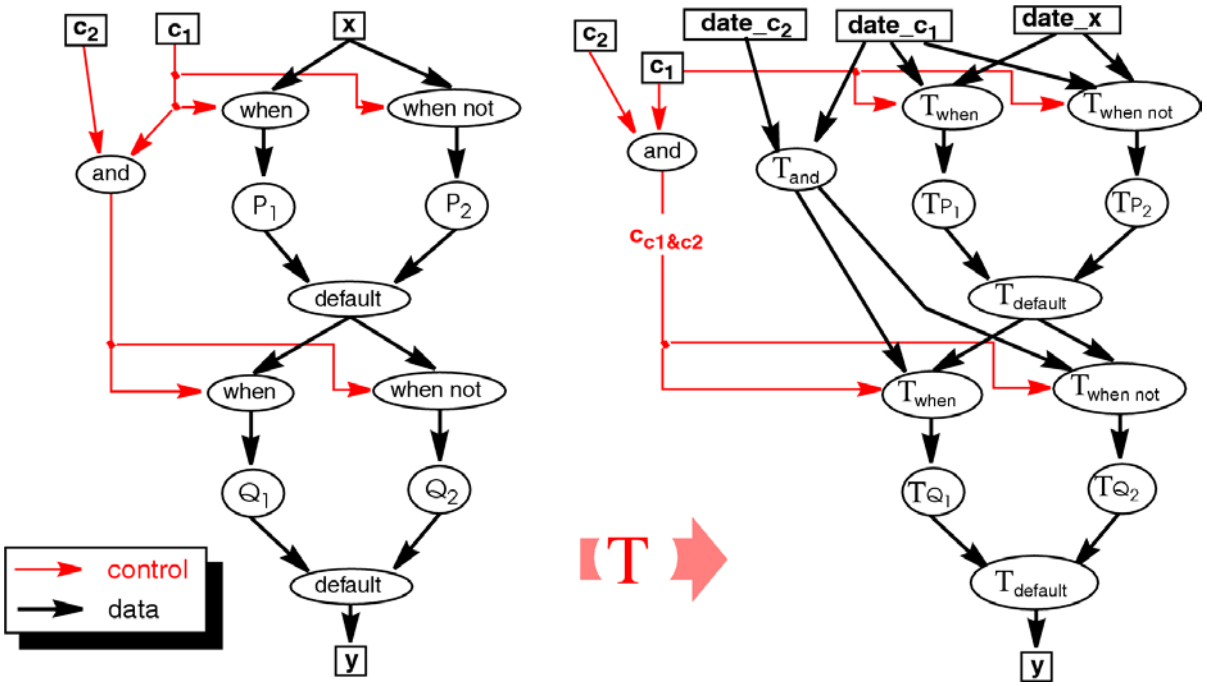

Figure 9. Finding longest paths in the SIGNAL graph

The preservation of Boolean inputs in the initial program contributing in the definition of control conditions (events) has to be considered. In the example, Boolean inputs c_1 and c_2. In the generic interpretation scheme it was shown that such Booleans have to become inputs in the interpretation process so that the activation events in the temporal interpretation process can be computed. In the example of Figure 9 c_1 is a Boolean input in the initial program and $c_{c1\&c2}$ is internally computed by "c_1 and c_2".

In addition, many control branches depend on Boolean signals that are either internally computed by the evaluation of relational (equality/inequality) operations on signal data values, or represent Boolean state variables. Since in the temporal interpretation process no such computations contained in the original program take place (only Boolean control and date computations take place), these clocks cannot be internally computed by the temporal interpretation process. In order for date calculations to effectively consider the influences of conditional behavior, we have to provide the Booleans that define such clocks as extra inputs to the temporal interpretation program. In the example of Figure 10, c_1 is a Boolean input defining a control condition and so it has to be provided as extra input to the temporal interpretation process. In addition the arithmetic relation "$x_1 > 0$" is also a control condition that results in the extra Boolean input $c_>$. The *true* instants of a Boolean c_2 define a clock; in order to update the dates correctly c_2 has to be provided as an input to the temporal interpretation to compute node dates only as often as the corresponding computations occur in the initial program and under the same conditions.

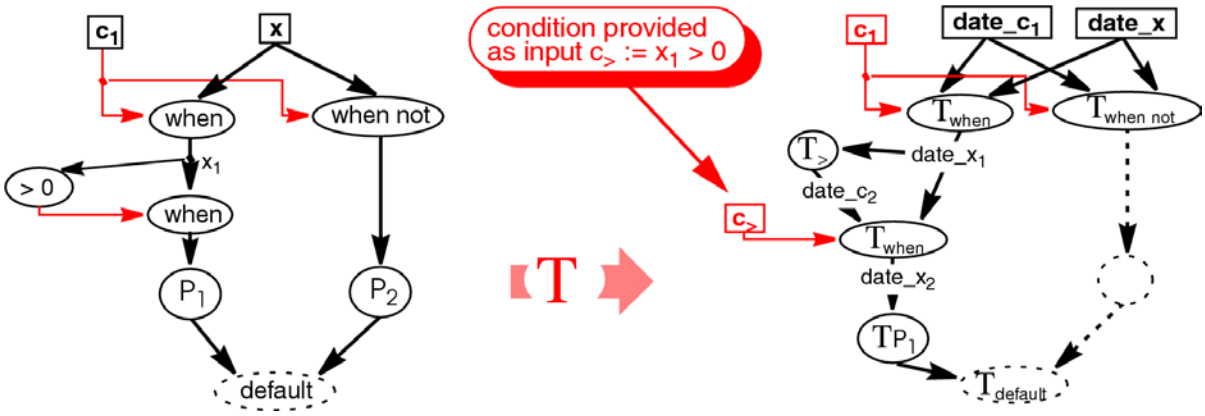

Figure 10. Date computations in control/data programs

Thus, a process *P* that contains conditional behavior is interpreted to a temporal interpretation process T(*P*) with a number of extra Boolean inputs represented as a vector denoted by CV_P (Condition Vector of *P*): $CV_P = \{ c_i \mid c_i \in \{true, false, \perp\}\}$ with $\perp$ denoting absence.

Obtaining Results

Since the elements in the SIGNAL HCDG are dynamic in nature, it is easy to account for only the active paths in the graph during successive system iterations and even more, exclude the processing paths that can never exist (*false paths*). Such paths contain dependencies whose clocks are mutually exclusive. This information is discovered during clock calculus. These advantages of the HCDG in respect to traditional graphs are illustrated in the example given in Figure 9. In this example thick lines correspond to program data and thin ones to control data that affects the execution flow. It is easy to see that P_2 and Q_1 are mutually exclusive so the path containing both should not be considered in time consumption counting. Thanks to clock calculus the relationships between the various clocks in a program become explicit during compilation and thus at each logical instant only the active paths may be considered in the date calculations.

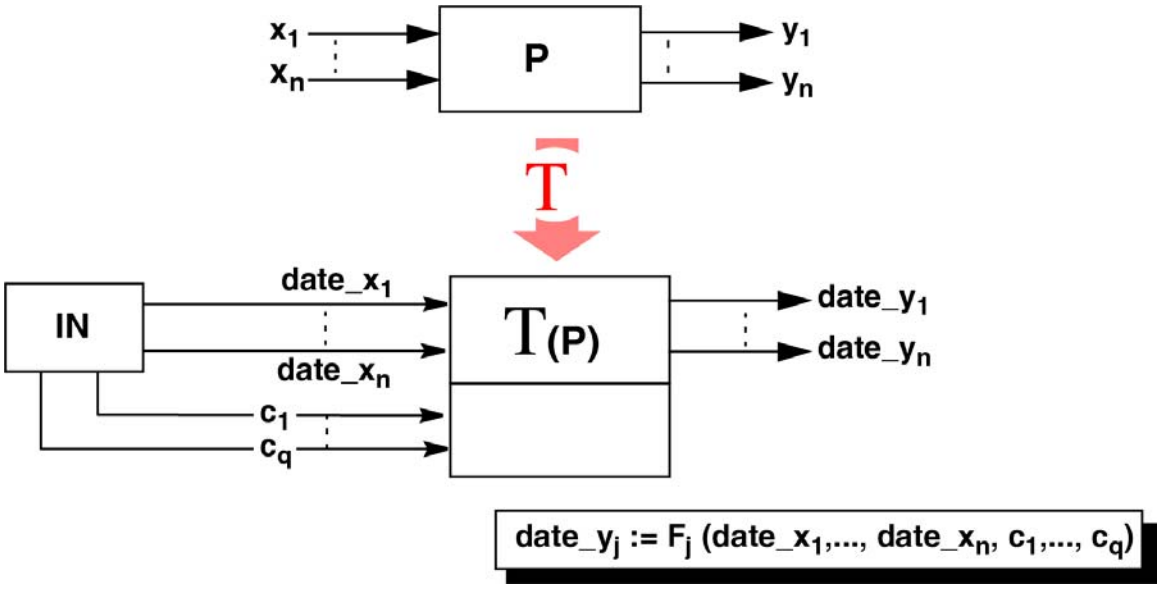

Figure 11. Temporal behavior simulation

In order to simulate the temporal response of a system in respect to defined operation delays, we generate its temporal interpretation which is the main element of such a simulator. The temporal simulator's SIGNAL specification is shown in Figure 11. At each iteration the output dates depend on the input dates and the *control configuration* represented by a valuation of the condition vector $[c_1,...c_q]$. More specifically each output date depends on a subset of input dates and on a control configuration sub-vector. For a simulation we have to provide test vectors $[c_1,...c_q]$ which are the clock defining Booleans. In a straightforward approach we can provide a set of vectors that covers all the possible combinations for the control flow. Such an approach has two major drawbacks. The first is the exponential growth on the number of test vectors. The second consists on the inclusion of infeasible control configurations by ignoring the inclusion/exclusion relationships of the clocks defined by these Booleans. In this cases clock calculus can be used to provide information on the inclusion/exclusion relationships between the Booleans and thus lower the number of test vectors, while excluding infeasible control configurations. Another possibility is to gather Boolean vectors during the functional simulation of the original program and use this smaller set of test vectors.

Considering clock equivalence/inclusion/exclusiveness relationships may significantly reduce the number of control configurations for an exhaustive temporal simulation of processes containing conditional behavior.

5.2 Modeling of sequential execution

Having considered the case of unconstrained parallel execution, the model can be augmented to account for sequential execution as well. Since the graph corresponding

to a program is a partial order, to achieve sequential execution we have to enforce it with additional dependencies between potentially parallel nodes (operations). The main preoccupation is to preserve the same model for date calculations as before. In Figure 12 an example of operation scheduling is given. For this simple case there are no dependencies between operations P_2 and P_3 in the graph Figure 12a, meaning that there are two possibilities for sequential execution: P_1, P_2, P_3, P_4 or P_1, P_3, P_2, P_4, that is either P_2 before P_3 or vice versa, as shown in Figure 12b.

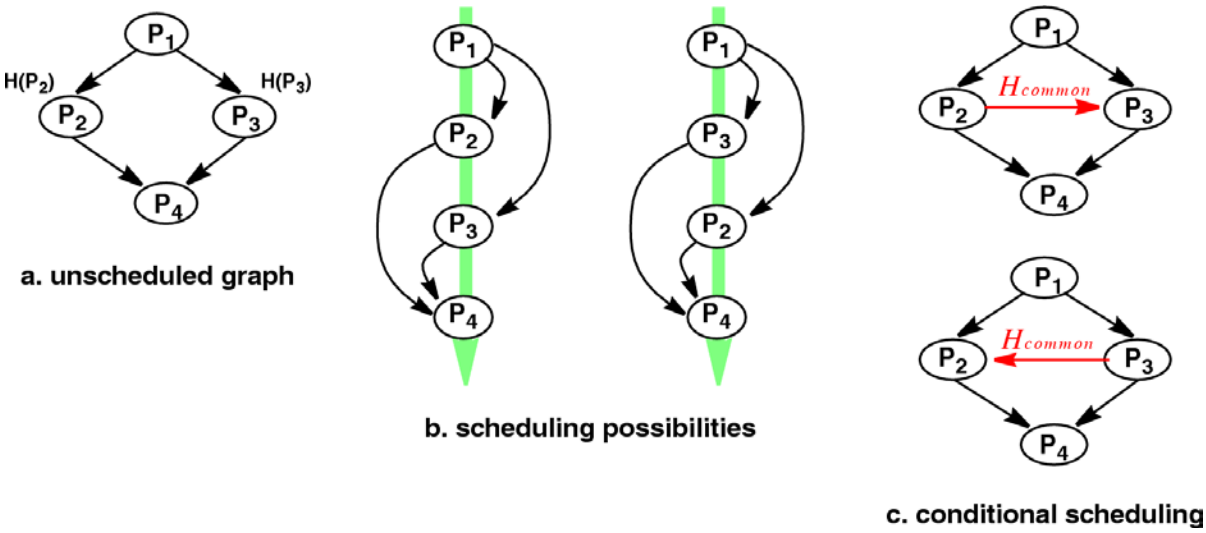

Figure 12. Operation scheduling for sequential execution

A first step in the operation scheduling is to add a dependency between P_2 and P_3. This dependency should be conditional in order to be active only at the instants that both operations can be executed at the same time. The clock of this *extra* dependency is the intersection of the clocks of the two operations P_2 and P_3 (H_{common}):

$$H_{common} = H(P_2) \cap H(P_3)$$

This way when both operations may execute simultaneously this clock is present thus the dependency of P_2 to P_3 is effective meaning that P_2 will execute before P_3. This is graphically depicted in Figure 12c where a control dependency (shown in grey) conditioned by H_{common} is added. Its direction depends on the execution order statically decided by a scheduler. In this way the scheduling transformation can be effectively represented in the HCDG representing the system.

The scheduled HCDG, denoted by $HCDG_S$, can now be used for the temporal interpretation that will also account for the scheduling decisions that influence the execution latencies of individual parts or whole systems. The date computations will take into account the scheduling dependencies in the same way data dependencies. For instance in the temporal interpretation of an operation node the operation delay will be added the maximum of the incoming dates that represent data, control and scheduling dependencies, to produce the date of the node's result.

5.2.1 Modeling dynamic scheduling

In order to make our scheduling scheme more flexible we may add an inverse conditional dependency from P_3 to P_2. When a scheduling dependency is effective, in one direction, the inverse dependency should be ineffective. To enable such a scheme to work we define a Boolean signal *s*. The clock of *s* is:

$$H(s) = H(P_2) \cap H(P_3) = H_{common}$$

When s is *true* the dependency in one direction is effective and when it is *false* the dependency in the inverse direction is effective. This is shown in Figure 12c where the scheduling dependencies are labelled by [*s*] in one direction (P_2 to P_3) and by [$\neg s$] in the inverse direction (P_3 to P_2). In this way we can manipulate the orientation of the scheduling edges by choosing the appropriate values for the *scheduling* Boolean signals.
In the presence of scheduling dependencies, the date computation model remains the same by assuming that the scheduling dependency carries the date value of its source node. If for example we assume that the clocks of P_i's, in Figure 12, are equal and that the scheduling Boolean *s* is *true*, the production date of P_4 is calculated as follows:

$$\begin{aligned} \text{date\_}P_4 &= max\,(\text{date\_}P_2, \text{date\_}P_3) + \Delta P_4 \\ &= max\,((\text{date\_}P_1 + \Delta P_2), (max\,(\text{date\_}P_1, \text{date\_}P_2) + \Delta P_3) + \Delta P_4 \\ &= max\,((\text{date\_}P_1 + \Delta P_2), (max\,(\text{date\_}P_1, (\text{date\_}P_1 + \Delta P_2)) + \Delta P_3) + \Delta P_4 \\ &= \text{date\_}P_1 + \Delta P_2 + \Delta P_3 + \Delta P_4 \end{aligned}$$

which corresponds to the chosen sequential execution path. The introduction of conditional scheduling dependencies permits the modeling of both static and dynamic scheduling decisions. Static scheduling corresponds to the addition of scheduling dependencies without an associated scheduling Boolean.

5.2.2 Example: dynamic scheduling based on input availability

To demonstrate how a dynamic scheduling policy can be modeled in respect to the date computation model, a simple example, shown in Figure 13, will be used. In this example the system functionality is partitioned into two separate concurrent tasks modeled by SIGNAL processes P_1 and P_2. Assuming that the two tasks are assigned for execution on the same processing element, an external scheduler (process S in Figure 13a) will decide the task execution order depending on the task input availability by providing the appropriate *true/false* value for the scheduling Boolean s_1. For instance, at the logical instants that the two tasks can simultaneously execute, if the inputs of P_1 are available before those of P_2 then P_1 will execute before P_2 and vice versa. Depending on the chosen execution order the task output dates will vary. In terms of input dates this means that the task with smaller input dates will execute first. Both ordering possibilities are modeled by the scheduling dependencies between P_1 and P_2 conditioned by Boolean conditions [s_1] and [not s_1] respectively. Consequently the *true/false* value of s_1 depends on the dates of the task inputs. The value of s_1 for the temporal interpretation is given by the following equation computed by process S_T in Figure 13b that models the scheduling policy:

$$s_1 := (date\_inP_1 \text{ when } h_{common}) \leq (date\_inP_2 \text{ when } h_{common})$$

where: h_{common} := $H(P_1)$ when $H(P_2)$ represents the logical instants at which both P_1 and P_2 can execute simultaneously. The ordering dependency is effective only at these logical instants.

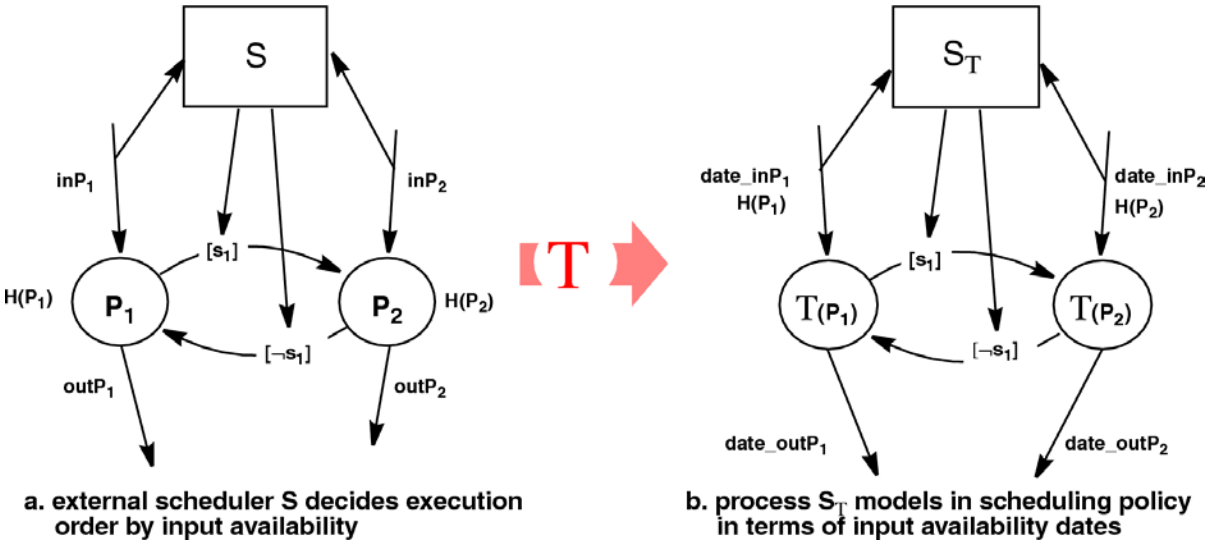

Figure 13. Calculation of scheduling conditions to model ASAP on input availability

5.2.3 Obtaining results in the presence of scheduling

When scheduling information (static or dynamic) is incorporated in the system's specification by means of the extra ordering dependencies, the temporal interpretation has a set of extra Boolean inputs (s_1, ..., s_p) consisting of the Boolean variables conditioning the direction of the dynamic scheduling dependencies. To obtain results we use the simulation configuration of Figure 14, which is essentially the same as the one in Figure 11 so the points made in that case apply in this one too.
In this scheme the temporal interpretation is slightly different since now we also have to provide the scheduling Booleans s_i as inputs. These Boolean inputs can obtain their values by means of a SIGNAL process (*Sched*) that models the scheduling policy by means of the involved signal dates. These dates can be input, output or intermediate signal dates and have to be provided to "*Sched*". A final word concerns the capability of the conditional scheduling dependency scheme to model static scheduling policies as well. This can be achieved by providing at each logical instant the same values for the scheduling Booleans (s_1, ..., s_p).

5.2.4 Minimizing the scheduling modeling complexity

Considering dynamic scheduling at a fine granularity (operations) is of little practical value. Instead existing code scheduling (for software implementations) and high-level synthesis scheduling techniques can be considered mature and adequate enough to provide efficient solutions. Nevertheless, fine grain scheduling at the operation level influences the performance at the system-level and thus the need to be able to include such information in our internal design representation. Assuming that an adequate

interface to the scheduling mechanism exists, fine grain scheduling can be represented by adding extra conditional dependencies of fixed direction between HCDG operation nodes. These dependencies will be taken into account by the proposed date computation model.

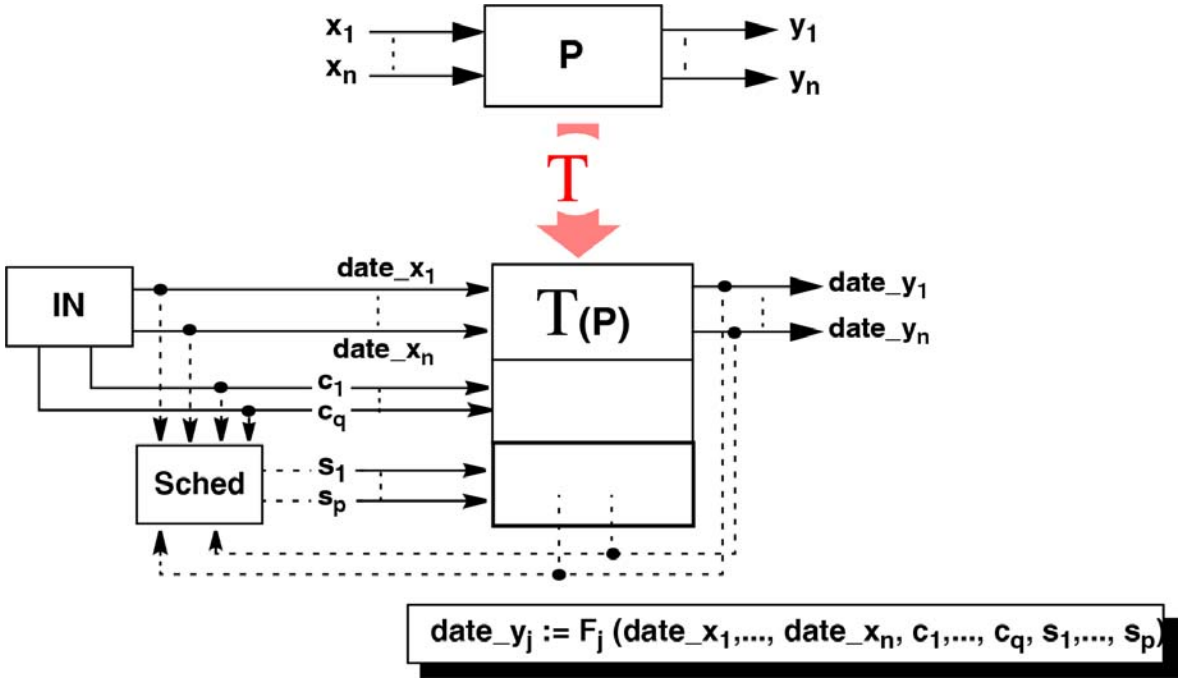

Figure 14. Temporal behavior simulator for sequential execution

Dynamic scheduling needs to be considered at the task level, when a complex system can be viewed as a collection of co-operating tasks. The advantage of considering the addition of conditional dependencies whose direction is controlled by scheduling Boolean signals, at the task level, is that the number of these Boolean signals is kept relatively small. This is graphically depicted in Figure 15. At each logical instant every *present* node of task T_1 *precedes* every *present* node of task T_2 at the *true* instants of s_1 and the *inverse* at the *false* instants. For each task its activation clock is defined as the union of the clocks of the nodes belonging to the task. In the example the clocks of T_1, T_2 are:

$$H(T_1) = \cup_i h_{1i} \text{ and } H(T_2) = \cup_j h_{2j}$$

respectively, where h_{1i} and h_{2j} are the clocks of the nodes of T_1 and T_2 respectively. Considering each task as HCDG nodes dynamic scheduling dependencies are added at this level with the scheduling Booleans having as clock the intersection of the task clocks. In the example:

$$H(s_1) = H(T_1) \cap H(T_2)$$

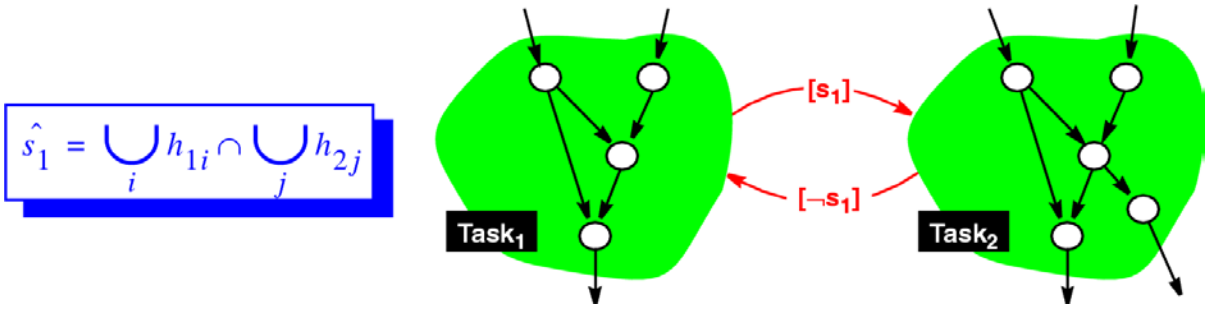

Figure 15. Scheduling dependencies on a coarser granularity level

The methodology of task partitioning as SIGNAL processes and dynamic task scheduling representation has been elaborated in the SACRES project [GL99]. Code distribution is viewed as the partitioning of an initial synchronous SIGNAL specification into a set of individual co- operating processes and their assignment for execution onto a set of interconnected processing elements.
The first step in the process is the decomposition of the initial specification *P*, into a set of communicating tasks (threads) P_1 - P_n. This is achieved by partitioning the initial HCDG into a set of sub-graphs. Each sub-graph can be considered as an HCDG node in the sense that it has an interface given by the incoming and outgoing data/control dependencies, and it is labelled by a clock that corresponds to its activation condition. Whenever this clock is present some activity will occur inside the task node. At this stage it is guaranteed by construction that:

$$P = P_1 \mid P_2 \mid \ldots \mid P_n$$

Next comes the assignment of tasks onto processing elements that corresponds into grouping together task processes assigned to the same processing element. This can be represented by:

$$P = P_1 \mid P_2 \mid \ldots \mid P_n = P_{P1} \mid \ldots \mid P_{Pk}$$

where *k* is the number of processing elements.
For instance could be $P_{P1} = P_1 \mid P_4 \mid P_7$.

The next step is to represent the communications between the various tasks inside each processor and between each partition assigned to each processor. At this level *intra*- and *inter*- processor communications can be modeled, by means of *read*/*write* (*send*/*receive*) nodes inserted at the interface points of each task (partition) node. These nodes model the transfer of control/data information across task (partition) boundaries and are labelled by clocks indicating when the communications are to take place.
Depending on the chosen communication schemes and the eventual physical implementation the communication nodes will obtain the appropriate implementation. A point of interest is that clock information can be used in order to find the best suited communication clocks that will minimize the communication frequency and possibly allow the sharing of communication links consequently minimizing the bandwidth requirements and the resource contention.

Finally, scheduling has to be performed inside each task (fine grain) and among tasks sharing the same processor (coarse grain). These tasks have to be scheduled either statically or dynamically. Similarly communications sharing the same link have to be scheduled as well. Scheduling can also be modeled by a scheduler process that observes both the task data/control dependencies and the synchronization constraints for consuming/producing values. In this methodology the implementation will be deadlock-free by scheduling in such a way that no cycles are induced on the initially *acyclic* graph representing the system.

6 Temporal interpretation of the HCDG

In this section the temporal interpretation of the most important SIGNAL kernel processes as represented internally by an HCDG, will be described. These temporal interpretation have also an internal HCDG representation.
In the HCDG representation of a SIGNAL process the following types of nodes can be found: i/o nodes, operation nodes (monochronous), down-sampling nodes, multiplexing nodes, memory nodes ($, cell). Each type of node has a temporal interpretation that produces the result date for a node as a function of the node input dates and the delay associated with the node.

6.1 I/O nodes

Input nodes are nodes that get input values from the external environment and make them available to the system. Similarly, output nodes make values internally computed in the system available to the external environment. Interfacing, to the external environment, may incur processing delays depending on the implementation. Consequently, our model should be able to account for them. Assume that an input (read) node has an associated delay which may be a function of a certain number of parameters, denoted by: Δ_{RD}(*read_pars*).

In Figure 16 for an input node (?*a*) shown on the left, its temporal interpretation (*date_*?*a*) is shown on the right. The input value *a* may be physically available before the system is ready to observe it. When the system is ready is indicated by the event h_a which is the clock of *a*. Following the renaming conventions previously defined the temporal interpretation of the input node "?*a*" is:

$$T(?a) \ \text{☞}date\_?a := T\_RD\{read\_pars\}(date\_h_a, date\_a, h_a)$$

The signal process T_RD computes the node availability date (*date_*?*a*) and expands to the following expression:

$$date\_?a := \max((date\_h_a \ when \ h_a), date\_a) + \Delta_{RD}(read\_pars)$$

The SIGNAL expression (*date_*h_a when h_a) gives the date of the occurrence of event h_a or in other words the date that the Boolean variable associated to clock h_a evaluates to *true*. The above expression simply states that:

the input date (*date_?a*) is equal to the sum of the delay it takes to get its value (Δ_{RD}) and the *maximum* of the date the input value is physically available (*date_a*) and the date at which the system can observe it (*date_*h_a when h_a).

Figure 16. Temporal interpretation of HCDG input nodes

Similarly for output (write) nodes (e.g. !*o*) following the renaming conventions previously defined their temporal interpretation is:

T(!o) ☞*date_!o* := T_WR{*write_pars*}(*date_*h_o, *date_o*, h_o)

The signal process T_WR computes the node availability date (*date_*!*o*) and expands to the following expression:

date_!o := max((*date_*h_o *when* h_o), *date_o*) + Δ_{WR}(*write_pars*)

The SIGNAL expression (*date_*h_o when h_o) gives the date of the occurrence of event h_o or in other words the date at which the Boolean variable associated with clock h_o evaluates to *true*. The above expression simply states that:

> the output date (*date_!o*) is equal to the sum of the delay it takes to make its value available to the environment (Δ_{WR}) and the *maximum* of the date the output value is internally computed (*date_o*) and the date at which the system can make it available to the environment (*date_*h_o when h_o).

6.2 Operation nodes

Operation nodes in the HCDG correspond to the monochronous operators of arithmetic (+, -, *, /), Boolean (and, or, not), and relational type (<, <=, >, >=, =, /=). They have an inbound *control* dependency from their clock signifying that the result can be computed when the clock is present (or in other words when the Boolean variable associated with the clock evaluates to *true*). The node and its inbound data dependencies are labelled by the node's clock.

In Figure 17 the HCDG of an operation node is shown on the left. Its temporal interpretation following the defined renaming conventions is shown on the right. In SIGNAL the temporal interpretation can be represented by:

T(c) ☞*date_c* := T_OP{*op_pars*}(*date_a, date_b, date_h*$_c$, *h*$_c$)

The signal process T_OP computes the availability date (*date_c*) for the value of node *c* and expands to the following expression:

date_c := max ((*date_a* when *h*$_c$), (*date_b* when *h*$_c$), (*date_h*$_c$ when *h*$_c$)) + Δ_{OP}(*op_pars*)

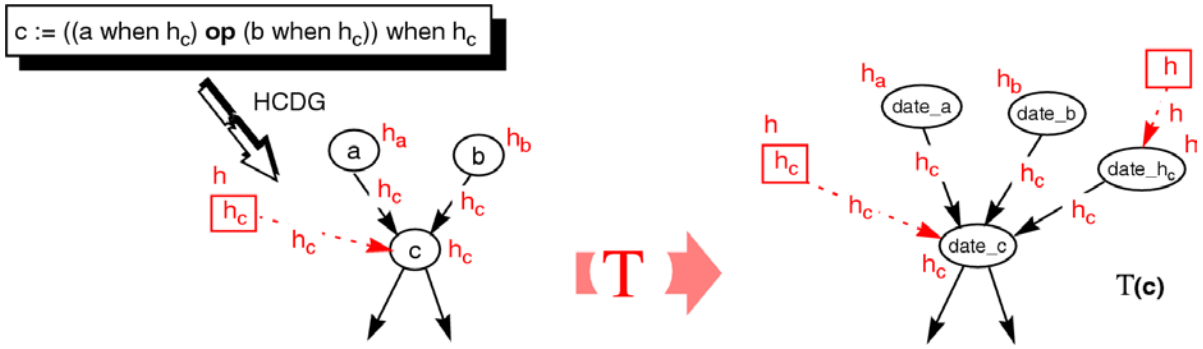

Figure 17. Temporal interpretation of HCDG operation nodes

The SIGNAL expression (*date_h*$_c$ when *h*$_c$) gives the date of the occurrence of event *h*$_c$ or in other words the date at which the Boolean variable associated with clock *h*$_a$ evaluates to *true*. The above expression simply states that:

> the node result date (*date_c*) is equal to the sum of the delay it takes to perform the operation that computes its value (Δ_{OP}) and the *maximum* of the dates the operation argument values are available (*date_a* when *h*$_c$), (*date_b* when *h*$_c$) and the date at which the system can perform the operation (*date_h*$_c$ when *h*$_c$).

6.3 Down-sampling nodes

Down-sampling nodes in the HCDG correspond to the down-sampling *when* SIGNAL operator. They have an inbound *control* dependency from their clock signifying that the result can be computed when the clock is present (or in other words when the Boolean variable associated with the clock evaluates to *true*). The node and its inbound data dependencies are labelled by the node's clock.
In Figure 18 the HCDG of a down-sampling node is shown on the left. Its temporal interpretation following the defined renaming conventions is shown on the right. In SIGNAL the temporal interpretation can be represented by:

T(c) ☞*date_c* := T_WHEN{*when_pars*}(*date_a, date_h*$_c$, *h*$_c$)

The signal process T_WHEN computes the availability date (*date_c*) for the value of node *c* and expands to the following expression:

$$date\_c := \max((date\_a \text{ when } h_c), (date\_h_c \text{ when } h_c)) + \Delta_{WHEN}(when\_pars)$$

The SIGNAL expression (*date_h*$_c$ when h_c) gives the date of the occurrence of event h_c or in other words the date at which the Boolean variable associated with clock h_a evaluates to *true*. The above expression simply states that:

> the node result date (*date_c*) is equal to the sum of the delay it takes to perform the down-sampling of the input value (Δ_{WHEN}) and the *maximum* of the date at which the input value is available (*date_a* when h_c) and the date at which the system can perform the down-sampling operation (*date_h*$_c$ when h_c).

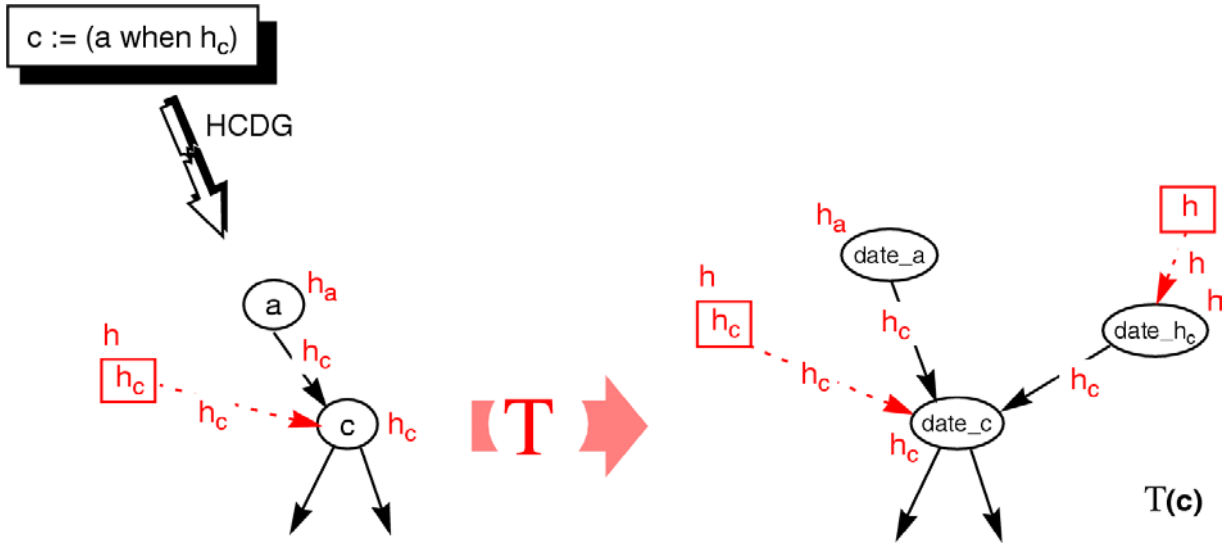

Figure 18. Temporal interpretation of HCDG down-sampling nodes

6.4 Multiplexing nodes

Multiplexing nodes in the HCDG correspond to the *default* SIGNAL operator. They have an inbound *control* dependency from their clock signifying that the result can be computed when the clock is present (or in other words when the Boolean variable associated with the clock evaluates to *true*). The inbound data dependencies are labeled by mutually exclusive clocks. The node has control dependencies from these clocks as well. In Figure 19 the HCDG of a multiplexing node is shown on the left.

Its temporal interpretation following the defined renaming conventions is shown on the right of Figure . In SIGNAL the temporal interpretation can be represented by:

$$T(c) \Rightarrow date\_c := T\_DEFAULT\{default\_pars\}(date\_a, date\_b, date\_h_c, h_1, h_2, h_c)$$

The signal process “T_DEFAULT” computes the availability date (*date_c*) for the value of node *c* and expands to the following expression:

$$date\_c := \max((date\_h_c \text{ when } h_c), ((date\_a \text{ when } h_1) \text{ default } (date\_b \text{ when } h_2))) + \\ + \Delta_{\text{DEFAULT}}(default\_pars)$$

The SIGNAL expression (*date_h*$_c$ when h_c) gives the date of the occurrence of event h_c or in other words the date at which the Boolean variable associated with clock h_c evaluates to *true*. The SIGNAL expression ((*date_a* when h_1) default (*date_b* when h_2)) gives the date of the availability of whichever value (*a* or *b*) defines the value of *c*.
The above expression simply states that:

> the node result date (*date_c*) is equal to the sum of the delay it takes to perform the multiplexing of the input values (Δ_{DEFAULT}) and the *maximum* of the date at which the input value is available ((*date_a* when h_1) default (*date_b* when h_2)) and the date at which the system can perform the multiplexing operation (*date_h*$_c$ when h_c).

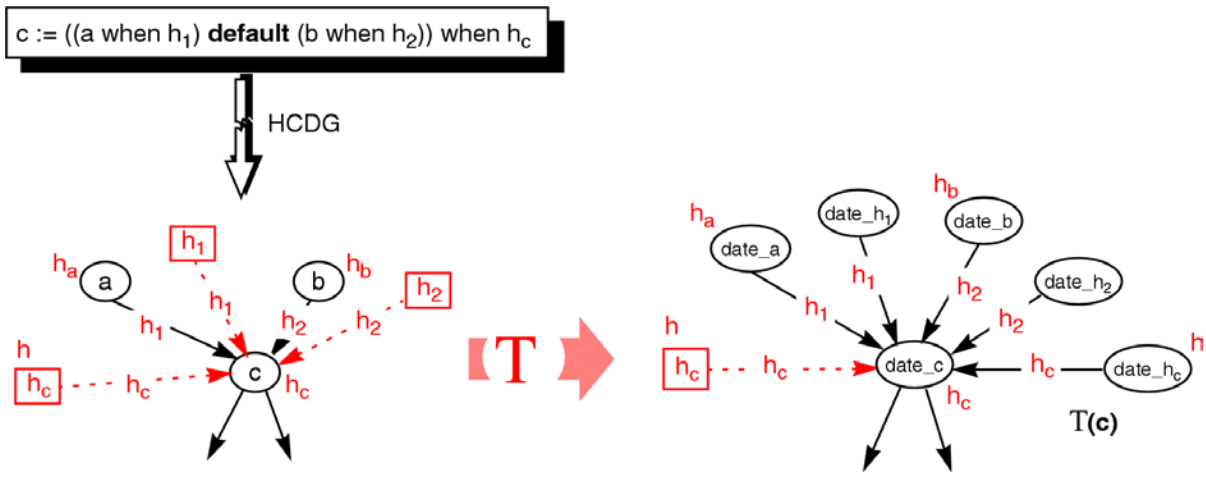

Figure 19. Temporal interpretation of HCDG multiplexing nodes

6.5 Memory nodes

Two types of memory nodes can be distinguished. The first type corresponds to memory nodes that behave as *synchronous registers* in the sense that their new value is available at the next logical instant at which the node is active (their clock is present). These nodes are represented by the SIGNAL expression:

$$za := (a \ \$1) \text{ when } h_a \mid h_a \text{ ^= } a \text{ ^= } za$$

In the left of Figure 20 the HCDG graphical representation of the above expression is given. The memory node is represented by the polygonal node. The fact that *a*, *za* are synchronous is represented by the control dependencies they have from their clock (h_a).

The second type corresponds to memory nodes that behave as transparent latches, in the sense that whenever the node is active and a new value is computed, at that logical

instant, this value becomes available at that instant; otherwise the value of the node at the previous logical instant is used. These nodes are represented by the SIGNAL expression:

$$c := (a \text{ when } h_1) \text{ cell } h_c \mid h_c \text{ ^= } c$$

h_1 is the clock indicating the availability of a new value for c and h_1 is included in h_c. This type of memory node, even though it can be derived by using a multiplexing node combined with a register memory node, is so frequently used in practice that it is represented as particular HCDG node. Its graphical representation in the HCDG is given in the left of Figure 21.

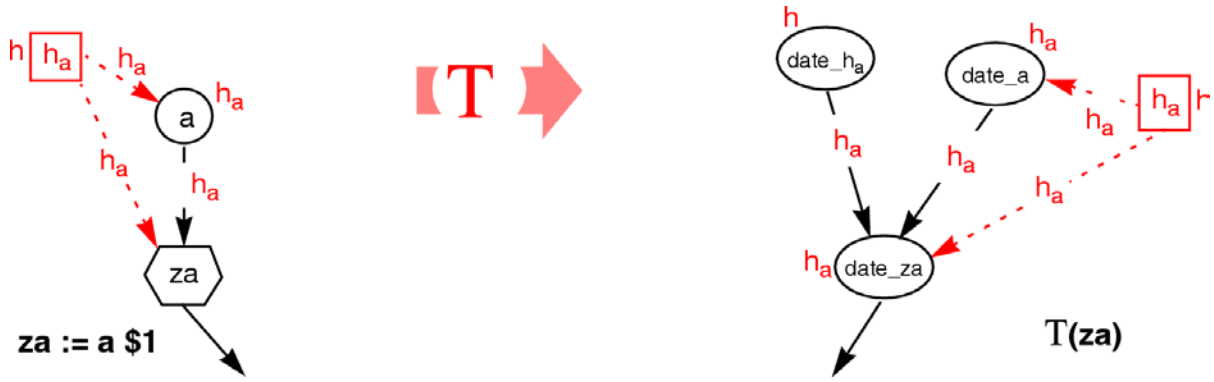

Figure 20. Temporal interpretation of HCDG memory nodes of register behavior

Memory nodes have two delays associated with them. The first corresponds in reading the stored value (Δ_{rd_mem}) and the second in storing a value (Δ_{wr_mem}).

In SIGNAL the temporal interpretation of the first type of delay nodes can be represented by:

$$T(c) \text{ ☞} date\_za := \text{T\_DELAY}\{delay\_pars\}(date\_a, date\_h_a, h_a)$$

The HCDG representation of the temporal interpretation is given in the right part of Figure 20. The signal process “T_DELAY” computes the availability date (*date_za*) for the value of node *za* and expands to the following expression:

$$date\_za := \max((date\_h_a \text{ when } h_a), ((date\_a \$1) + \Delta_{wr\_mem}(delay\_pars))) + \Delta_{rd\_mem}(delay\_pars)$$

The SIGNAL expression (*date_ha* when *ha*) gives the date of the occurrence of event *ha* or in other words the date at which the Boolean variable associated with clock *ha* evaluates to *true*. The SIGNAL expression ((*date_a* $1) + Δ_{wr_mem}(*delay_pars*))) gives the date at which the value of *a* computed in the *previous* logical instant is effectively stored in the memory corresponding to *za*. The above expression simply states that:

the memorized value date (*date_za*) is equal to the sum of the delay it takes to read the value from memory (Δ_{rd_mem}) and the *maximum* of the date at which the value is actually stored ((*date_a* $1) + Δ_{wr_mem}(*delay_pars*))) and the date at which the system is ready to read the memorized value (*date_h*$_a$ when *h*$_a$).

In SIGNAL the temporal interpretation of the second type of delay nodes is represented by:

$$T(c) \ \text{☞}date\_c := T\_CELL\{cell\_pars\}(date\_a, date\_h_1, date\_h_c, h_1, h_c)$$

The HCDG representation of the temporal interpretation is given in the right part of Figure 21. The signal process "T_CELL" computes the availability date (*date_c*) for the value of node *c* and expands to the following expression:

$$date\_c := \max((date\_h_c \text{ when } h_c), (date\_h_1 \text{ when } h_1) \text{ default } (\max((date\_h_1 \text{ when } h_1), ((date\_c \$1) + \Delta_{wr\_mem}(delay\_pars))) + \Delta_{rd\_mem}(delay\_pars))) + \Delta_{DEFAULT}(default\_pars)$$

What the above expression actually means can be easily derived by considering the explanations of the temporal interpretation of multiplexing and register memory nodes.

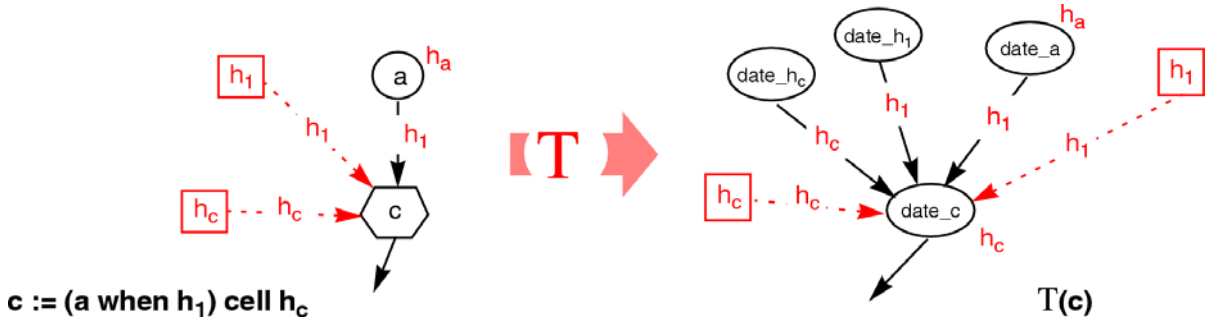

Figure 21. Temporal interpretation of HCDG memory nodes of transparent latch behavior

7 Summary

A generic interpretation model is used to define the temporal interpretation of a SIGNAL specification mapped to an implementation. In this interpretation from a SIGNAL process modeling the functional behavior of a system, we can obtain another SIGNAL process modeling the consumption of time by the specified functionality. If in the initial specification implementation refinements (i.e. partitioning-scheduling, component choice, etc.) are included then the temporal interpretation may account for them as well.

The date computation model was presented in detail in a step-by-step fashion, considering initially an ideal execution platform and moving progressively to more realistic execution contexts. The influence of the control-flow introduced by signal down-

sampling or multiplexing can be easily accounted for. The same goes for partitioning and scheduling. Partitioning is modeled by decomposing the initial process into sub-processes, one for each partition, and introducing interfacing operations for the communications across partitions. Scheduling, is modeled by extra dependencies. These extra dependencies are taken into account by the date computation model and thus computed dates effectively account for the impact of scheduling on the system's performance.

Finally, the choice of specific components, can be accounted for via a set of parameters used to get the appropriate delay for operations/macro-operations/tasks, depending on the granularity level. These delays are used in the date computations and thus the date computation model effectively accounts for the choice of the components making up the implementation architecture.