A Real-Time HW/SW Co-Design Approach Based on the SIGNAL Language and its Environment

Apostolos Kountouris, Christophe Wolinski

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

A real-time hw/sw co-design approach based on the SIGNAL language and its environment

A. Kountouris, C. Wolinski

N° 3046

Octobre 1996

THEME 2

Calcul symbolique, programmation et génie logiciel

ISSN 0249-6399

A real-time hw/sw co-design approach based on the SIGNAL language and its environment

A. Kountouris* et C. Wolinski**

Thème 2: Calcul symbolique, programmation et génie logiciel

Projet EP-ATR

Rapport de recherche n°3046- Octobre 1996

30 pages

Abstract: In the present document the idea of building around the SIGNAL language and its environment, a methodology for the evaluation of implementation alternatives, design space exploration and implementation of R/T systems, is investigated. The driving force behind this effort is the desire to exploit the advantage SIGNAL in the domain of functional specification, as well as the numerous tools built around it. A Real-Time co-design methodology is proposed and the basic tools that will assist the user in the development process, are described. Such a methodology intends to bridge the existing gap in the SIGNAL environment between specification and actual implementation.

Key-words: specification, SIGNAL, methodology, Real-Time, hw/sw co-design

(Résumé : tsvp)

* EP-ATR, kountour@irisa.fr
** EP-ATR, wolinski@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : (33) 99 84 71 00 - Télécopie : (33) 99 38 38 32

Une approche pour la conception conjointe matériel-logiciel temps réel, basée sur le langage SIGNAL et son environnement.

Résumé : Dans ce document nous examinons l'idée de la construction, autour du langage SIGNAL et de son environnement, d'une méthodologie d'évaluation des différentes possibilités pour l'implémentation des systèmes temps réel. Par cette approche nous désirons exploiter les avantages du langage SIGNAL, dans le domaine de la spécification fonctionnelle, ainsi que les nombreux outils disponibles dans son environnement. Nous décrivons une méthodologie temps-réel pour la conception conjointe matériel-logiciel et les outils nécessaires pour qu'elle soit réalisée. Une telle méthodologie peut éventuellement servir de liaison entre la spécification et l'implémentation finale.

Mots-clé : spécification, SIGNAL, méthodologie, temps réel, conception conjointe matériel-logiciel.

Hardware/Software co-design is a research area that for sometime now has been receiving considerable attention. Its appeal lies mainly to its concrete as well as important goals. To mention only a few of them: implement feasible and cost-effective systems, reduce design and time-to-market time, remove ambiguity and unsafety of ad-hoc techniques, etc. referring the interested reader to [11], [15], [21] for more details. In this paper we present the work towards the definition and implementation of a co-design methodology that introduces a new aspect in the domain. The novelty is the use of the SIGNAL synchronous dataflow language [1], [2] as a means of specification. SIGNAL offers some excellent and desirable possibilities like functionally correct specifications formal verification capabilities as well as code generation like C, Fortran and VHDL for simulation purposes. In this approach the main difficulty is that SIGNAL produces implementation independent specifications but to reach an implementation *non-functional* requirements need to be considered as well as target architecture details. Somehow this information has to be introduced and thus make SIGNAL specifications *implementation aware*. It can be argued that the SIGNAL environment in its current state presents a gap between the tools for specification and the tools for code generation. The proposed methodology intends to bridge this gap and develop the link between these two important phases of *system development*. This link is nothing more than the definition of a methodology that prompts the creation of new tools to complement existing ones and the modification of existing tools in order to fit into the methodology. The definition of this methodology is guided by a number of important decisions that have to be taken at an early stage and have to do with the targeted application domain acknowledging the fact that no single methodology can be general and efficient at the same time for the whole spectrum of computing applications. Other decisions include the chosen implementation target architecture, the internal design representation, algorithms, degree of automatization etc.

The rest of this paper is structured as follows: in Section 1 the general context of this effort identifying where the SIGNAL language and its environment fit in a *development lifecycle*. Section 2 describes the choices made in terms of target architecture, the necessary tools to conduct design space exploration, and finally an overview of the methodology that integrates these tools and assists the designer in the disciplined development of *feasible* and *cost-effective* systems. Section 3 demonstrates the methodology through a small but complete example. Finally we conclude by summarizing what has been done and what remains to be done in respect to the initial goals and the ones that sprung out during the evolution of this work.

1 System development in the SIGNAL environment

In this section the stage is set so that the reader may get the broader picture. In the beginning we felt that the most important thing was to form a vision of *system development* that would allow us to gain insight on what we were trying to accom-

plish, where we were standing and finally how we should proceed in order to attain our goals.

1.1 The SIGNAL language and its environment

The SIGNAL language[1] is a dataflow oriented language based on the synchrony hypothesis[3]. It belongs to the family of synchronous languages [4] and it is used for the functional specification of reactive R/T systems for control and DSP applications. Using the language expressions the user is programming in an equational style and thus each program is a system of equations. The SIGNAL compiler resolves these systems and performs checks (see [1][5]) useful in discovering possible sources of error and functional unsafety. Around the SIGNAL language and its compiler exists a variety of tools that constitute the SIGNAL environment. There is a Graphical User Interface for program (system design) entry, C and Fortran code generators to generate code used for functional simulators, intermediate code generators to access formal verification [6] and other third-party development tools [9]. Finally there exists a VHDL code generator [7][8] that enables us to access hardware synthesis tools.

Before introducing the basic elements of the SIGNAL language let us give some basic definitions. In SIGNAL terminology a *signal* is an infinite sequence of data where each element is implicitly indexed by time. At any given logical instant a signal may be *absent* or *present*. Presence is denoted by the signal's value at that instant. The *clock* of a signal is the set of the logical instants that it is present (and thus carries a value). Clocks can be considered as equivalence classes between signals. When two signals are present at the same logical instants they possess the same clock; otherwise their clocks are different.

The SIGNAL language is defined by a small kernel of statements. Each statement has formally defined semantics and defines a clock equation and the data dependencies of the participating signals. Its small size allows for mathematical manipulation of these equations enabling formal verification of program properties. The basic language constructs are shown in the table below along with an informal description. For a more detailed description of the language, its semantics and applications the reader is referred to [2] and [6]. A last point of interest is the SIGNAL language features that extend its use in the domain of DSP applications by supporting array data types and vector operations.

Language Construct	**SIGNAL syntax**	**Description**
step-wise extensions	X := A op B	where op: arithmetic/relational/ boolean operators
delay	ZA := A $x	memorization of the x^{th} past value of A

Language Construct	SIGNAL syntax	Description
extraction	R := A when B	R equal to A when B is present and true
priority merging	R := X default Y	if X present R:=X else if Y present R := Y else R absent
process composition	(| P | Q |)	processes are composed common names correspond to shared signals
useful extensions		
	when B	the clock of the true instants of B
	event X	the presence instants of X
	synchro {A, B}	clock of A equal with clock of B

The basic data structure of the SIGNAL environment is the *Dynamic Graph* (DG) which is constructed during the compilation process [5]. This graph is the internal representation of a SIGNAL program and consists of nodes and dependencies among them. Both are tagged by a clock meaning that they are active only at certain logical instants. There are two types of nodes, clock nodes and signal nodes. Clock nodes are organized as a hierarchy and constitute the control of the program. Each clock node may have a dataflow subgraph attached to it containing the signals that carry a value at the clock's presence. In other words this subgraph contains the operations that are to be executed when the clock is present. This graph structure, where the control is explicit, permits to carry out the necessary processing so that several *static* properties of the system's functionality can be verified at compilation time. Briefly once compilation is over it is certain that the program is deterministic, does not exhibit contradictions, has no cycles (circular data/control dependencies), no constraints are set on the inputs and that desired functional properties (coded as SIGNAL equations) are satisfied.

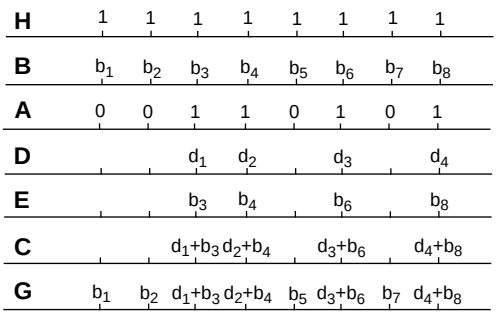
Finally the DG exposes the potential parallelism in the specification which later has to be constrained when a particular implementation is considered. In the next section we present what was done to enhance this environment in order to be able to proceed from the specification phase to code generation for systems that also satisfy non-functional requirements related to performance and cost.

Example:

To specify the functionality of our system we use the SIGNAL programming language and we capture the functional requirements as a SIGNAL program. For such a simple case we can directly write the program shown in Figure 1a. For more complex systems the use of the *SIGNAL Graphical User Interface* can make the task a lot easier.

```
process TESTWHEN=
     { ? event H;
         logical A;
         integer B;
         integer D
       ! integer C;
         integer G }
     (| E := B when A
      | G := C default B
      | C := D+E
      | synchro { A, H, B }
      |)
     where
          integer E
end
```

(a) SIGNAL program



(b) temporal and value relationships of the inputs and outputs of the program

FIGURE 1. Functional specification in SIGNAL

In Figure 1b we can see what is produced as output of the system in response of the inputs and when this output is produced in respect to the occurrence of the inputs. For example the value of the output G depends on the value of B if A is *false* or on the value of C if A is *true*. We can also see that D carries a value only at the logical instants that A is *true*. In Fig2 the DG representation of this example is given in two forms: in (a) the clock hierarchy and the corresponding sub-graphs and in (b) the complete dependency graph where nodes having the same clock are shaded with the same pattern, and dependencies are tagged by the clock defining their presence.

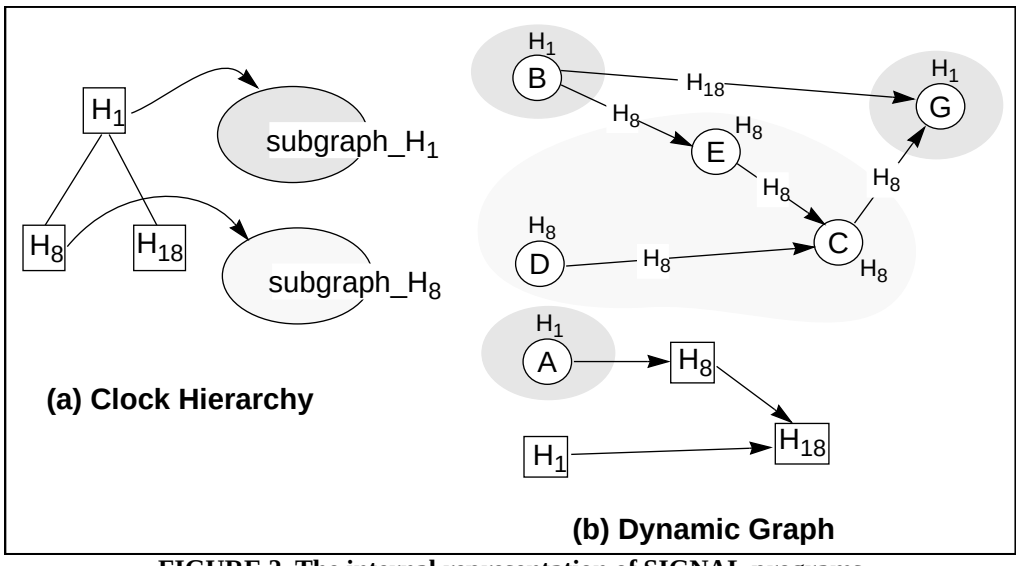


FIGURE 2. The internal representation of SIGNAL programs

1.2 System development

In our view, System Development is a complex iterative process in which from a given set of requirements a fully functional system is produced. This process (depicted in Figure 3) shows how customer requirements and designer experience can be transformed into a functional system. In this process we distinguish three main stages indicated in the shaded parts. These are:

I. *Specification*: system requirements (functional and non-functional) are identified

II. *Design Space Exploration:* contains two activities, the choice of a particular target architecture and the partitioning of system functionality among the components of this architecture, so that the requirements (functional and non-functional) are satisfied

III. Implementation

Bubbles correspond to development phases and the tagged arrows correspond to the data that needs to be communicated between phases. The dashed arrows imply iteration through a set of phases until some pre-established criteria are satisfied and we can move to the subsequent phase(s).

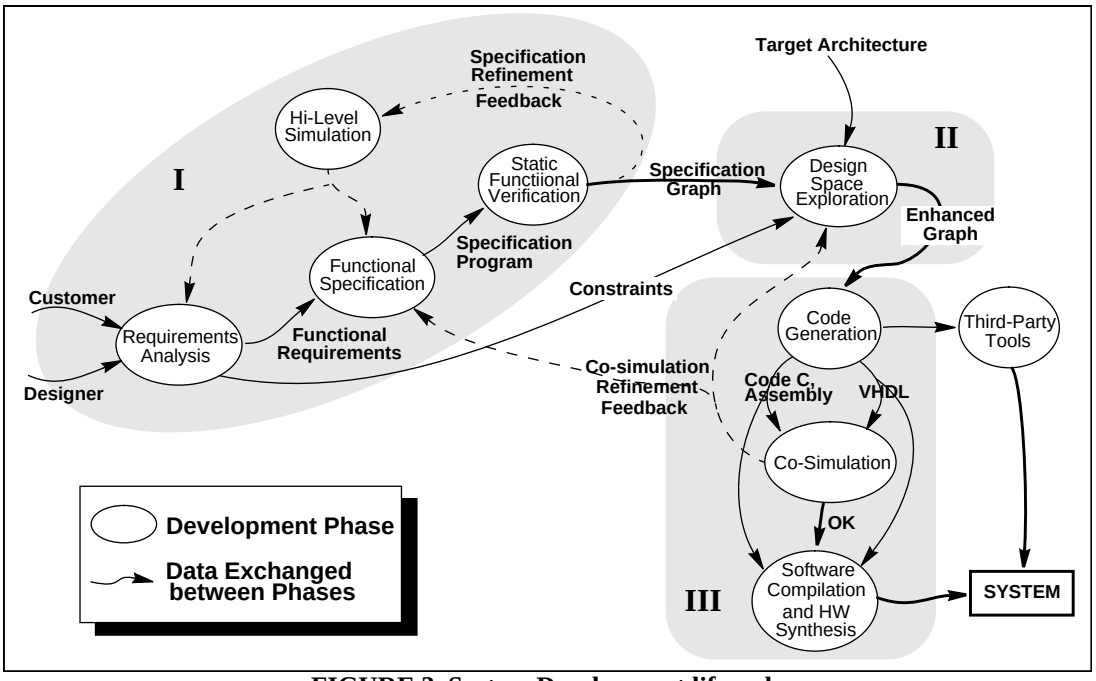


FIGURE 3. System Development lifecycle

If we attempt to map the lifecycle of Figure 3 to the context of the SIGNAL language the functional specification results to a SIGNAL program, Static *Function-*

al Verification maps to the SIGNAL compilation and the *Specification Graph* maps to the *Dynamic Graph* (DG) that the SIGNAL compiler produces. Furthermore, aspects of code generation map to existing SIGNAL code generators or a modified version of them. Consequently SIGNAL and its tools are examined in the broader context of a *System Development Lifecycle*. The SIGNAL synchronous dataflow language has the ability to produce implementation independent specifications due to its principal hypothesis *(synchrony hypothesis)* which states that operations take no time [3]. It also offers a variety of tools for formal verification, simulation, code generation [1] as well as the means to access the Syndex [9] environment for multiprocessor code distribution. By means of VHDL code generation [7] hardware simulation and synthesis tools [8] can be accessed. Finally the SIGNAL environment offers a *Graphical User Interface* by which the user is assisted in the specification of a system by organizing it as a collection of functionally independent components [2], in a top-down approach.

2 A co-design approach

In the development of the methodology the initial goal was to take advantage of the maturity of the SIGNAL environment in the domain of functional specification, formal verification and the work that has been already done in code generation and to integrate these tools in a complete framework, by which the user will be assisted in the full development of systems. Once the methodology is defined it can be considered as a policy of using the various tools in a disciplined fashion. Through this process we identified the additions and modifications that we had to incorporate in the SIGNAL environment so that tool cooperation could be achieved. We were also prompted to take the necessary high level decisions that guided not only the structure of the methodology but specific tool implementation details as well. As an example we can mention the choice of a particular *Target Architecture* that prompted us on what information was necessary to include in the graph and the characteristics of the code that the code generators should produce.

2.1 Target Architecture

In hw/sw co-design one of the most important decisions is the choice of the target architecture to be used for systems that are to be treated by a particular co-design methodology. It is an important decision as it affects the targeted application domain and the complexity of the tools that are used during the different phases of the methodology (algorithms, models etc.). The target architecture we use for our purposes may be classified as an *heterogeneous mixed* one and is graphically represented in Figure 4. By heterogeneous we mean the use of processing elements with different execution models and by mixed the combination of software and hardware elements.

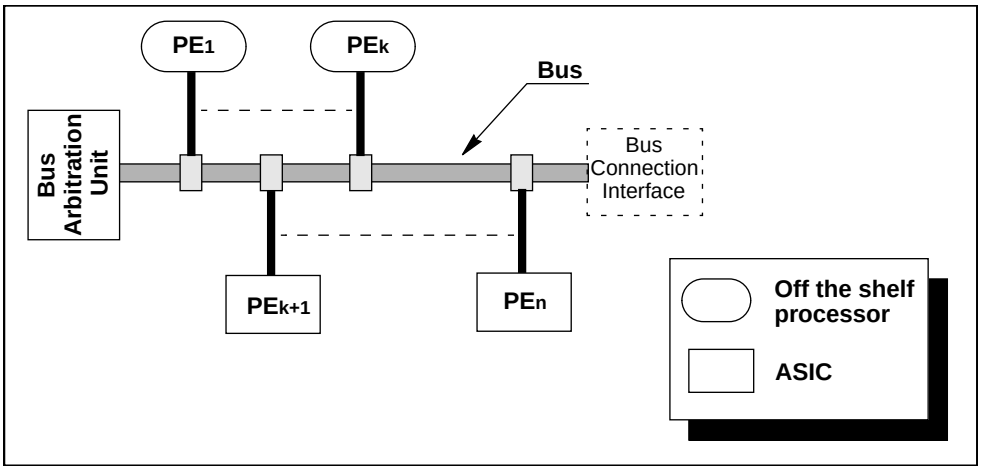
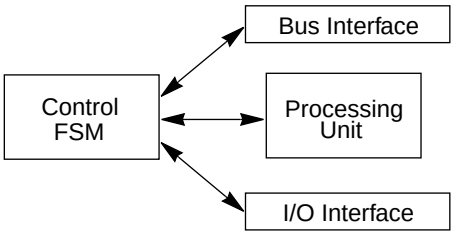


FIGURE 4. The Target Architecture

In this scheme we have *Processing Elements (PE's)* that can be either off-the-shelf processors, (microprocessors, microcontrollers etc.) or specialized hardware components synthesized to perform certain functions. In the remainder they are collectively referred to as PE's having in mind that the former type constitutes the software dimension of a system while the latter its hardware dimension. As interconnection medium a bus is chosen. The bus is managed by a special *Bus Arbitration Unit* that has the sole purpose of managing (ordering) the access to the shared medium. Its functionality depends on the application which means that for different applications we need different bus arbiters. Let us briefly describe the structure of the PE's. The specific hardware Processing Elements may consist of:

i. a Control FSM
ii. a Processing Unit (PU)
iii. an interface to the bus
iv. an interface to the external world



Let us note that this is only one realization out of many possible ones. In Figure 5 we present the different parts that make up the Hardware PE's analysing in some detail its two main components the *Control FSM* and the *PU*. The different types of memories correspond to different data types that may be found in a SIGNAL program. Bit memory corresponds to boolean valued signals of type *logical* or *event*. Constants remain unchanged throughout the execution. *Volatile memory* corresponds to instantly valued signals for which the value is not to be used after the logical instant of their presence. Finally, *permanent memory* accommodates the explicit memorization of past signal values offered by the SIGNAL language. The final point

to mention has to do with the way the PE's interface with the shared interconnect (bus). Interfacing to the bus is achieved by FIFO's. One FIFO for each communicated value with a depth that depends on the application.

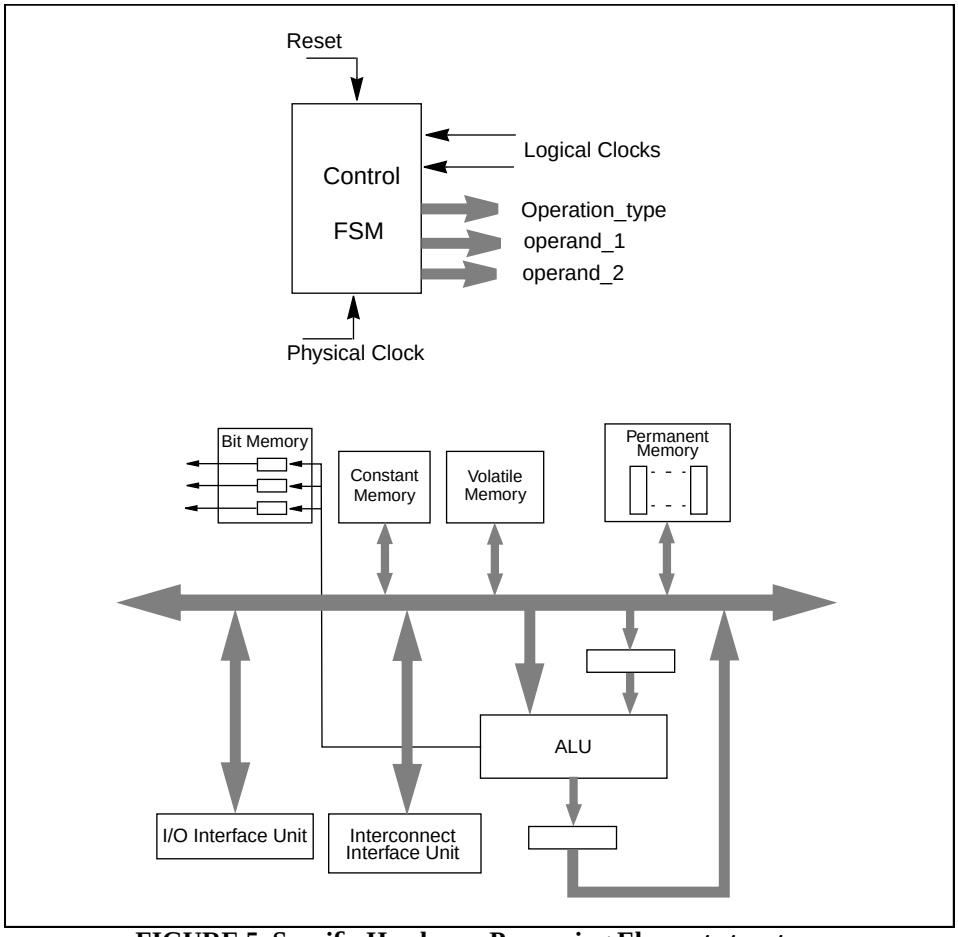


FIGURE 5. Specific Hardware Processing Element structure

2.2 HW/SW co-design and SIGNAL

In the development of a system what drives the realization of a particular implementation are the constraints that the system has to meet. There are performance constraints and cost (budget) constraints and in short, the end product it is desirable to meet the performance requirements at a minimum cost. A system realization can be hardware, software or a mix and its execution or more generally speaking its realization has an associated cost in terms of the resources it requires in order to perform its task. That is for a set of inputs produce a set of outputs. The cost can be thought as:

i. the time it is required to do so,

ii. the memory demands (code & data) and

iii. the silicon area, components, chips etc. required.

For software implementations the precious resource is processor cycles whereas in hardware ones is silicon. From a marketing point of view the cost of a system includes parameters like design time and effort and other interesting stuff. An illuminating discussion on the parameters and the trade-off's that drive the designer's implementation decisions can be found in [11], [14].

Form the discussion above it is clear the designers have the difficult task to perform what is called in the literature Design Space Exploration. Speaking of design space exploration we have to define two terms: *design space* and *exploration*. The design space is the set of all possible implementations that can be used to implement the functionality of a specified system. It is the space of all possible solutions to the problem of implementing a system. The question is why this is a problem in the first place. There are constraints to be respected which belong to two main categories: *performance* and *cost*. That means that a possible implementation might satisfy both, one or none of the constrain types. Acceptable solutions are the ones that satisfy both though it has to be noted that for R/T systems performance constraints are of higher importance. From the set of acceptable solutions we have to choose the best one. Exploration is the process of navigating through an unknown space with the goal of finding something. This process is not arbitrary. There is a starting point and tools used in order to conduct it with discipline. In our case the unknown territory is the space of possible solutions, the starting point is given by the initial performance and cost constraints and the tools are methods and tools that help to evaluate possible solutions against the imposed constraints. By evaluating various possibilities we narrow down the space that has to be searched, to the space of acceptable solutions. Within this sub-space we might further search for an optimal solution that represents the best trade-off. The design space may be à-priori constraint due to the limitations of a method to support a certain class of design architectures. If for example the supported target architecture class consists of one off-the-shelf processor and custom hardware then the design space exploration consists in choosing the right components and technologies and finding an optimal partition of system functionality in hardware and software. Furthermore if the supported components and hardware technologies are limited too the design exploration becomes the search of the right hardware-software partition.

2.3 *Profiling* of SIGNAL programs as a tool for design space exploration

In the context of SIGNAL the question is how to incorporate a mechanism that will permit the evaluation of the execution cost of a program on various implementations without leaving the powerful formalism of the initial SIGNAL specification. In other words obtain a facility for design space exploration. This question is very

interesting if one remembers that the foundation of SIGNAL (and all the synchronous languages in general) is the *synchrony hypothesis* (operations take no time) that makes a specification implementation independent. To answer this question we implemented a tool that transforms the original SIGNAL program into an equivalent profiled one [17]. To do this we had to modify the original DG in order to include some necessary information. This information consists of the partitioning and the total ordering of operations and is provided by the *Springplay* [20] *static scheduling algorithm*. The resulting instrumented program is implementation sensitive meaning that we have place holders in the form of parameters whose values reflect the designer's choices in terms of target architecture and its constituent parts. That means that ultimately we shall have a new program that has to be executed in order to get the evaluation results. Making available specific target architecture information (that can be found in architecture dependent libraries) the generated program will update the arrival dates of the input signals in order to produce the production dates of the outputs of the system taking into account the operation costs for the target architecture under evaluation. In this way we succeed in introducing implementation specific information in a SIGNAL specification which is by nature implementation independent. Furthermore this is a tool for evaluating possible implementations early in the system development lifecycle something highly desirable as we have already mentioned. Along with this tools we decided on how to model a target architecture and make this modelling information available for the evaluation. Due to lack of space we briefly mention the main aspects of such a facility:

- Modelling target architecture components in a component library database. Having a mapping of SIGNAL instructions to generic machine instructions and for each machine a mapping of these generic instructions to specific machine instructions. Provide the interface to the database in order to access this information.
- Introducing implementation details in the specification and accessing the libraries. Schedule operations on processing elements and get partitioning and total order dependencies for each element as well as involved communications.
- Producing the profiled SIGNAL program with the appropriate parameter values according to the designer's choices.
- Definition of an Evaluation Methodology. Execution of a profiled simulation, data collection and check against constraints.

2.4 A high level description of the methodology

Our methodology has for goal the integration of SIGNAL tools and their cooperation within a well defined context, in order to produce functionally correct and constraint satisfying systems. In Figure 6 we present this methodology in the form of a *dataflow diagram*. The shaded part is the one that realizes our *co-design approach* (it corresponds to phase *II. Design Space Exploration* of Figure 3) but we included some additional elements so that the reader can get the broader picture. Bubbles cor-

respond to transformations of the design data flowing in and out of them as indicated by the tagged arrows. The numbers in them indicate the relative sequence in their invocation. Dashed arrows signify iteration through a set of phases until established criteria are satisfied and we can move to a subsequent phase in the methodology. Rectangles represent external entities (e.g. Designer, Requirements Document, etc.). The entity *Designer* signifies that the methodology requires assistance from the user in order to progress towards a desired direction.

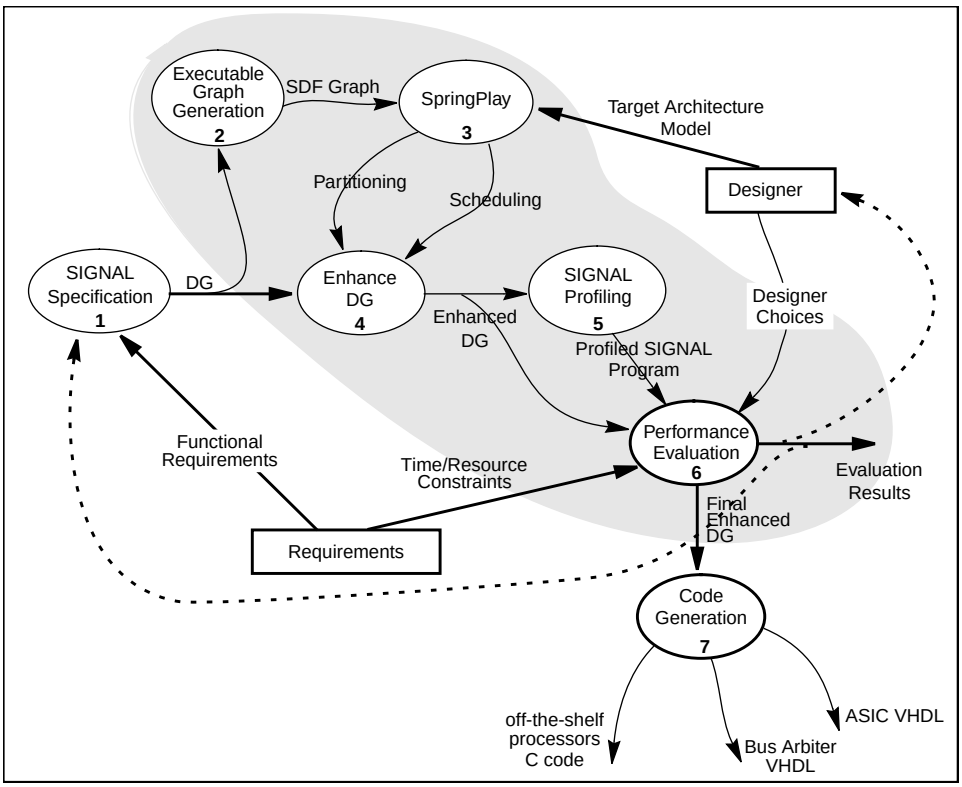


FIGURE 6. Co-design methodology outline

The SIGNAL specification is a complex phase that includes some sub-phases. From a document that contains functional and non-functional requirements we capture the functional ones in a SIGNAL program and finally we produce a Dynamic Graph *(DG)*. This graph has to be enhanced with implementation information regarding the *Target Architecture model* chosen by the designer. To make this enhancement we need the assignment of graph nodes to processors the communications between PE's and the total ordering of potentially parallel nodes that execute on the same PE. This information is provided by performing the *Springplay scheduling algorithm* [20] (Phase 3). For that we need to provide a model of the Target Architecture (roughly operation times for each processing element, communication cost etc.) and an SDF Graph (resulting from Phase 2) equivalent to our initial DG. The *Enhanced*

Dynamic Graph is used in order to produce an implementation aware equivalent SIGNAL program (profiling, Phase 5) whose execution provides timing measurements in processor cycles that permit the evaluation of the design choices made (Phase 6) up to this point. For this evaluation we use the constraints captured as non-functional requirements. If the evaluation is not favourable the designer has to reiterate through Phases 3, 4, 5 and 6 until he obtains a favourable evaluation. At this point we have to note that design space exploration is achieved relatively early in the *Development Lifecycle*. No software or hardware is generated as yet. This is the purpose of the code generation phase (Phase 7) that takes as input the *Enhanced DG (Final)*, that produced the favourable evaluation, and produces C code for the software parts and their interfacing to the bus and VHDL code for the *Bus Arbitration Unit* and the hardware components. From there on a retargetable compiler (like lcc [18]) may be used to produce object code for the off-the-shelf processors and synthesis tools (like Synopsys etc.) to produce netlists. Things like co-simulation and manufacturing may follow but for the moment they are out of the scope of our effort.

3 A complete example

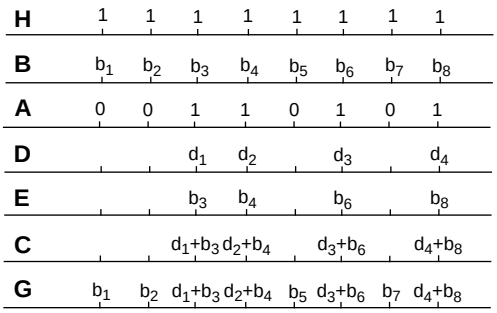
To illustrate our approach we give an elementary but complete example of all the methodology steps from specification down to implementation. For each phase we explain what information we use, how we use it and what we get as a result.

3.1 Functional specification: the SIGNAL program

To specify the functionality of our system we use the SIGNAL programming language and we capture the functional requirements as a SIGNAL program. As a working case we shall use the program presented in Section 1.1. To facilitate the reader in Figure 7 we have copied the example program and its chronogram.

```
process TESTWHEN=
     { ? event H;
         logical A;
         integer B;
         integer D
       ! integer C;
         integer G }
     (| E := B when A
      | G := C default B
      | C := D+E
      | synchro { A, H, B }
      |)
     where
          integer E
end
```

(a) SIGNAL program



(b) temporal and value relationships of the inputs and outputs of the program

FIGURE 7. Functional specification in SIGNAL

3.2 Enhancing the dynamic graph with implementation details

The successful compilation of this program produces the DG which is the internal representation of a functionally correct system. This representation is deprived of any implementation specific information as well as any constraints (performance or cost) affecting the implementation choices. The second step consists in introducing in the DG part of this information in order to produce a profiled SIGNAL program that will permit us to explore the design space by changing the implementation parameter values and performing a profiled simulation. The information we need is mainly the *assignment-of-nodes-to-processing elements* (processors) which reflects the *partitioning* of the system in different components, and the necessary *ordering dependencies* between nodes in each component that will enforce a total order on the graph. Finally we need to know the communications taking place between processing elements as well as the participating nodes of the DG. This goal is achieved by applying the *Springplay scheduling algorithm* in order to obtain a best static schedule of the operations on a set of processing elements. As we already have said (Section 2.4) Springplay works on an *Static Data Flow (SDF)* graph so before its invocation we transform our DG into its equivalent SDF graph.

```
(hinput rH1 rH1 logical !H1 !H1 !H1 !H1 !H1 !H1 !H1 !H1 !H1)
(function rA rA 10 logical ?H1 !A !A !A !A !A !A !A !A)
(function rB rB 10 logical ?H1, integer !B !B)
(function rD rD 10 logical ?A ?H1, integer !D)
(function wC wC 10 integer ?C, logical ?A ?H1)
(function wG wG 10 integer ?G, logical ?H1)
(default G integer ?eC ?eB, logical ?A ?H1, integer !G)
(when E logical ?eA, integer ?eB, logical ?A ?H1,integer !E)
(function add1 add 10 integer ?eD ?eE, logical ?A ?H1, integer !XZX)
(when C logical ?eA, integer ?X, logical ?A ?H1, integer !C !C)

(connect rA/A G/A)            (connect rH1/H1 rA/H1)
(connect rA/A E/A)            (connect rH1/H1 rB/H1)
(connect rA/A add1/A)         (connect rH1/H1 rD/H1)
(connect rA/A C/A)            (connect rH1/H1 wC/H1)
(connect rB/B G/eB)           (connect rH1/H1 wG/H1)
(connect rB/B E/eB)           (connect rH1/H1 G/H1)
(connect rD/D add1/eD)        (connect rH1/H1 E/H1)
(connect G/G wG/G)            (connect rH1/H1 add1/H1)
(connect E/E add1/eE)         (connect rH1/H1 C/H1)
(connect add1/XZX10 C/X1)     (connect rA/A E/eA)
(connect C/C G/eC)            (connect rA/A C/eA)
(connect C/C wC/C)            (connect rA/A rD/A)
                              (connect rA/A wC/A)
```

FIGURE 8. SDF Graph description file. Nodes (up) and Connections (down)

As it can be seen in Figure 8 the SDF graph is represented by a two-part file. The first part describes each node in the graph (operation it performs, its inputs and

outputs as well as the duration of its execution). The second part describes how these nodes are connected in order to form the graph.The resulting graph is graphically depicted in Figure 9 where we see that each node has a control part (black oval) which is responsible for the triggering of the execution of the node's operation. At the bottom right of the figure the corresponding SIGNAL DG is given.

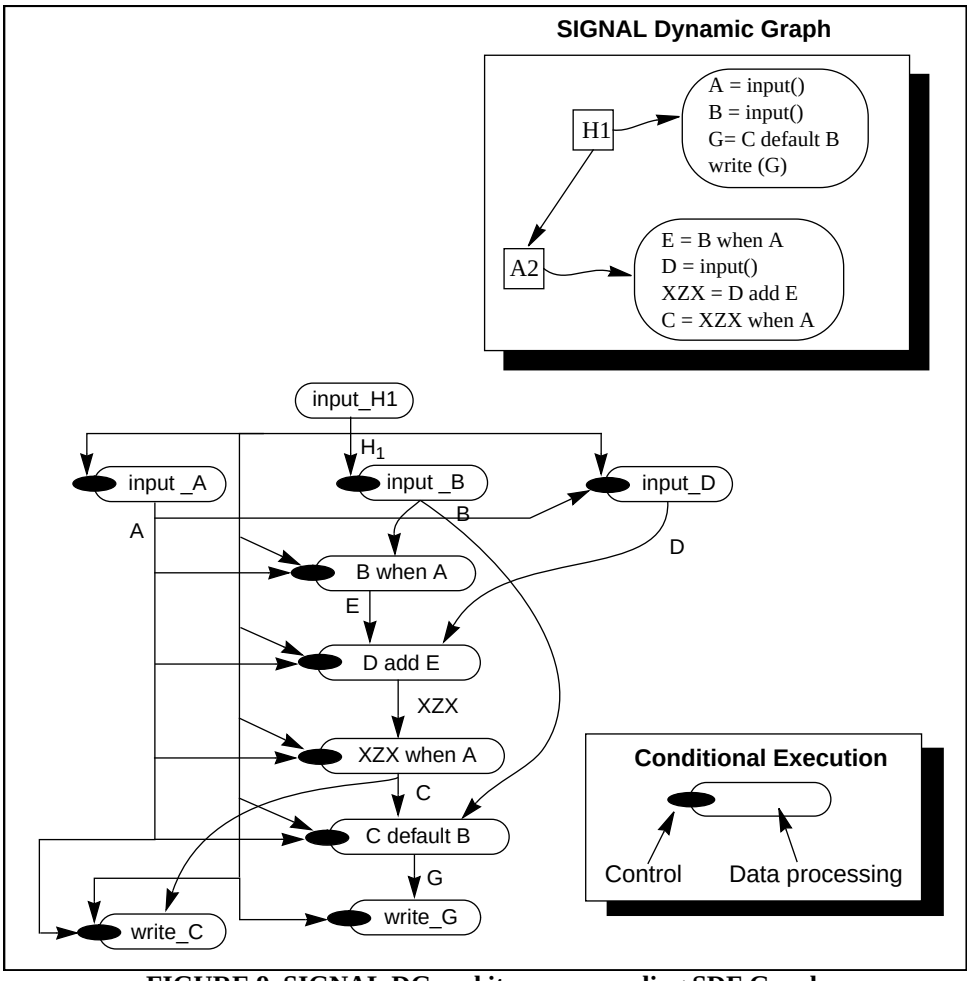


FIGURE 9. SIGNAL DG and its corresponding SDF Graph

This SDF graph and the Model of the Target Architecture that the graph is to be executed is passed to Springplay for static scheduling. The result is an assignment of nodes to PE's as well as the static scheduling of the communication between these PE's. The result of Springplay as far as the communications are concerned is given below where each line has the following format:

source node	*destination node*	*destination processor number*	*source processor number*	*communication start time*

```
rH1  rA 1 2 1.0     (1)          rH1  add1 4 2 21.79 (redundant)
rH1  rB 3 2 1.10    (2)          rA   add1 4 1 21.99 (redundant)
rA   E 2 1 11.1     (3)          E    add1 4 2 22.29 (7)
rB   E 2 3 11.19    (4)          add1 C 2 4 32.5     (8)
rH1  rD 4 2 11.29   (5)          rA   G 2 1 33.70    (redundant)
rA   rD 4 1 11.49   (6)          rB   G 2 3 33.80    (redundant)
rH1  wG 3 2 11.7    (redundant)  C    wC 1 2 33.90   (9)
rH1  wC 1 2 12.0    (redundant)  G    wG 3 2 35.0    (10)
rA   C 2 1 13.09    (redundant)
```

At this point it is necessary to identify the redundant communications. A communication is defined as redundant if it involves the transfer of a value between two processors, that has already been transferred between them for a previous computation. We consider that in such a case only the first communication is necessary and that for the subsequent ones we only have to read the value from the destination PE's communication buffer where it is stored for as long as it is needed in that PE. In the Springplay result text files we present redundant communications in *italics* and we tag useful ones with a sequence number - <n>.

```
>PE 1                                    >NODE:C 32.70 33.70
<1>COMM:R rH1->rA 2->1 1.0 1.10          < >COMM:R rA->G 1->2 33.70 33.80
>NODE:rA 1.10 11.1                       < >COMM:R rB->G 3->2 33.80 33.90
<3>COMM:S rA->E 1->2 11.1 11.19          <9>COMM:S C->wC 2->1 33.90 34.0
<6>COMM:S rA->rD 1->4 11.19 11.5         >NODE:G 34.0 35.0
< >COMM:S rA->add1 1->4 11.5 11.80       <10>COMM:S G->wG 2->3 35.0 35.10
< >COMM:S rA->C 1->2 11.80 11.9          >PE 3
< >COMM:S rA->G 1->2 11.9 12.0           <2>COMM:R rH1->rB 2->3 1.10 1.20
< >COMM:R rH1->wC 2->1 12.0 12.1         >NODE:rB 1.20 11.19
<9>COMM:R C->wC 2->1 33.90 34.0          <4>COMM:S rB->E 3->2 11.19 11.29
>NODE:wC 34.0 44.0                       < >COMM:S rB->G 3->2 11.29 11.39
>PE 2                                    < >COMM:R rH1->wG 2->3 11.79 11.89
>NODE:rH1 0.0 1.0                        <1>COMM:R G->wG 2->3 35.0 35.10
<1>COMM:S rH1->rA 2->1 1.0 1.10          >NODE:wG 35.10 45.10
<2>COMM:S rH1->rB 2->3 1.10 1.20         >PE 4
<3>COMM:R rA->E 1->2 11.1 11.19          <5>COMM:R rH1->rD 2->4 11.29 11.49
<4>COMM:R rB->E 3->2 11.19 11.29         <6>COMM:R rA->rD 1->4 11.49 11.79
<5>COMM:S rH1->rD 2->4 11.29 11.49       >NODE:rD 11.79 21.79
< >COMM:S rH1->add1 2->4 11.49 11.69     < >COMM:R rH1->add1 2->4 21.79 21.99
< >COMM:S rH1->wC 2->1 11.69 11.79       < >COMM:R rA->add1 1->4 21.99 22.29
< >COMM:S rH1->wG 2->3 11.79 11.89       <7>COMM:R E->add1 2->4 22.29 22.49
>NODE:E 11.89 12.89                      >NODE:add1 22.49 32.5
<7>COMM:S E->add1 2->4 12.89 13.09       <8>COMM:S add1->C 4->2 32.5 32.70
< >COMM:R rA->C 1->2 13.09 13.19         >LENGTH: 45.10
<8>COMM:R add1->C 4->2 32.5 32.70
```

In Figure 10 below we give a graphical representation of the Springplay result after the elimination of redundant communications *(dotted arrows)*. The *numbered*

circles on the necessary communications *(solid arrows)* represent the ordering of communications during execution.

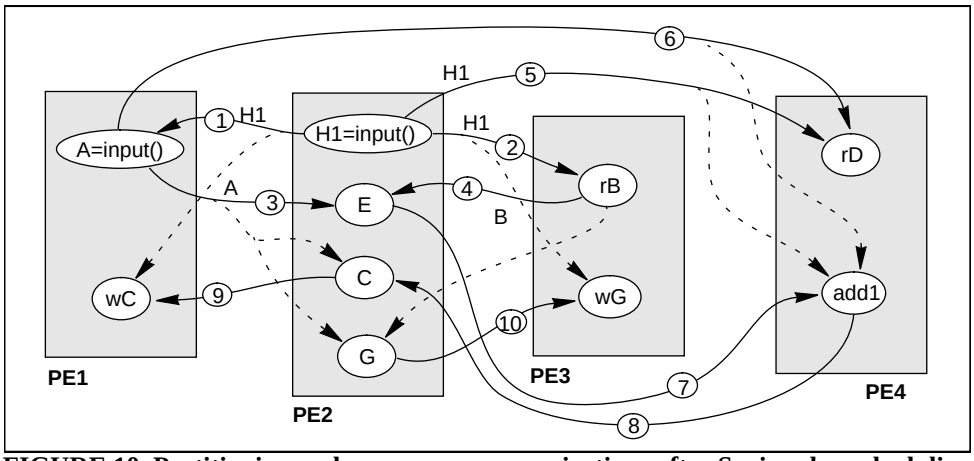


FIGURE 10. Partitioning and processor communications after Springplay scheduling

This ordering of communications prompts the specification of the *FSM* of the *Bus Arbitration Unit* which is responsible for managing the access to the shared medium. The resulting state transition diagram is shown in Figure 11. Each state is numbered with the number of communication that it handles. To make things easier to read we have included in the figure *(white rectangles)* the participating processors and the transferred data during each communication. As it can be seen The communicated values are also stored locally in the arbitration unit as they may affect subsequent communications.

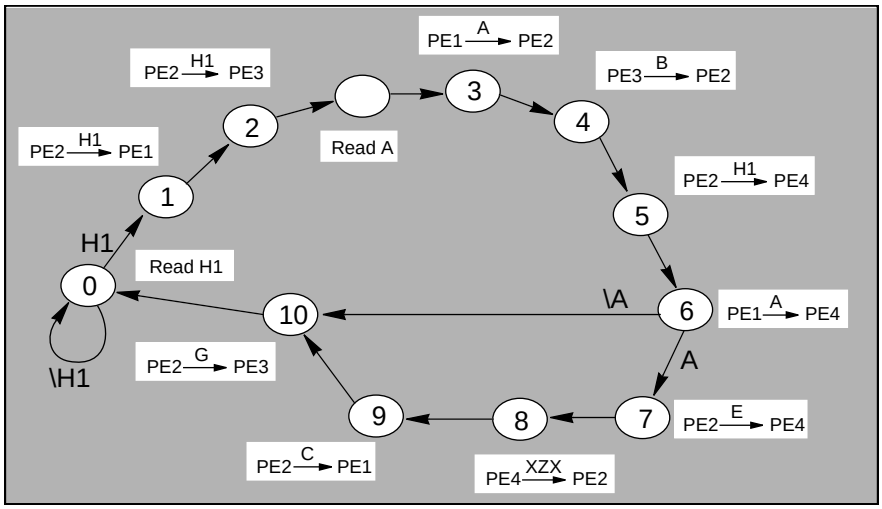


FIGURE 11. The state diagram for the FSM of the *Bus Arbitration Unit*

3.3 Design performance evaluation

During this step we introduce in the DG the information produced by the scheduling steps in order to produce the equivalent profiled program. During its execution it provides measurements pertaining to the expected performance of such an implementation. The profiled program is linked together with a library front-end that permits the query of the appropriate component library files during the simulated profiled execution. As a result we obtain for a set of input dates (dates that the system inputs become available) a set of output dates. Based on these measurements we evaluate our performance constraints. If not met that prompts the designer to alter the choice of specific components (processors) in an attempt to meet the constraints without reiterating through the previous phases of the design process. In case that this attempt fails then it will be necessary to choose a different instantiation of our Target Architecture meaning that even though the same abstract scheme of Figure 4 is to be used we have to use more PE's in order to exploit more of the available parallelism.

The SIGNAL process that instruments the original SIGNAL specification with a date mechanism for each input/output signal is given below. Cxxx () {} denote library front-ends that access the appropriate component libraries in order to account for the cost of the operation they represent. In Figure 10 we give an example of such a library front end. An exception is CCOMM which accounts for the cost of communicating data from one PE to another and its parameters signify source and destination PE's, type of communicated data etc.

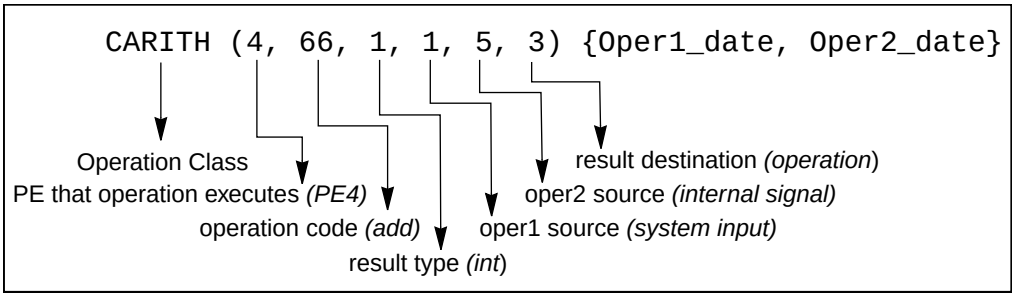


FIGURE 12. Library front-end usage example

```
process TESTWHEN_COS=
    { ? event H_1,B_3_EV,D_4_EV;
        integer H_1_date,A_2_date,B_3_date,D_4_date;
        logical A_2
      ! event C_5_EV,G_6_EV;
        integer C_5_date,G_6_date
     }
(| (| synchro {H_1,H_1}
    | synchro {H_1,A_2,A_2_date,B_3_EV,B_3_date,G_6_EV,G_6_date}
    | H_1_date := CIN(2,1968,6,1,1,5){H_1_date} when H_1
    |)
 | (| H_8_H := when A_2
```

```
    | synchro {H_8_H,D_4_EV,D_4_date,C_5_EV,C_5_date}
    | H_8_H_date := CWHEN1(1,24,6,1,0,5){CIN(1,1968,5,1,0,3){
           MAX(){A_2_date,CCOMM (1,2,1,...) {H_1_date}}} when H_8_H
    |)
 | (| C_5_date := COUT(1,1969,1000,5,3,1){CCOMM (2,1,1,...) {
                  CWHEN(2,45,1,5,3,5){CADD(4,66,1,1,5,3){
                  CIN(4,1968,1,1,0,3){MAX(){D_4_date,H_1_date}},
                  CCOMM (2,4,1,..) {E_8_date}},H_8_H_date}}}
                  when H_8_H
    | E_8_date := CWHEN(2,45,1,5,1,5){CCOMM (3,2,1,...) {
                    CIN(3,1968,1,1,0,3){MAX(){B_3_date,
                    CCOMM (2,3,ev,...) {H_1_date}}},
                H_8_H_date} when H_8_H
    |)
 | (| H_18_H := H_1 ^- H_8_H
    | H_18_H_date := CHCOMPL(2,76,6,5,5,5){H_8_H_date,H_1_date}
                      when H_18_H
    |)
 | (| G_6_date := COUT(3,1969,1000,5,3,1){CCOMM (2,3,1,...) {
                    CDEFAULT(2,54,1000,3,3,5){
                    CWHEN(2,45,1,5,5,3){C_5_date,H_8_H_date},
                    CWHEN(2,45,1,5,1,3){CIN(3,1968,1,1,0,3){
                    MAX(){B_3_date,CCOMM (2,3,ev,...) {H_1_date}}},
                    H_18_H_date}}}} when H_1
    |)
 |)
where
    integer H_18_H_date, H_8_H_date, E_8_date
    event H_18_H,H_8_H,E_8_EV;
end %TESTWHEN_COS%
```

3.4 Code generation

Implementation can start once we have reached the point that a set of design choices produces a favorable evaluation of the performance constraints. The first step of the implementation is the *code generation* phase of our methodology (Figure 6). Code generation means the production of *C code* for the system parts that are to be implemented in *off-the-shelf processors*, *C code* for their *FIFO interface* to the bus, *VHDL code* for the *hardware elements* and their *FIFO interface* to the bus and *VHDL code* for the *Bus Arbitration Unit.* Below we give samples of the results of code generation. We present the software to realize the functionality of PE1, PE2 and PE4 that correspond to the software dimension of our system, followed by the VHDL code that implements the functionality of the control FSM of hardware PE3.

Once the code is produced we may proceed in machine code generation [18] for the processors and in netlist generation for the hardware elements. From there on the road is open for low-level testing/debugging, co-simulation and finally manufacturing.

```
/******************/
/* C code for PE1 */
/******************/
#include <...>
#define A2PORT ...
#define C5PORT ...

void main ()
{
int H1, A2, C5;
while (1) {
   H1 = CommInput ();
   if (H1) {
      A2 = Read (A2PORT);
      CommOutput (A2);
      CommOutput (A2);
      if (A2) {
         C5 = CommInput ();
         Write (C5, C5PORT);
         }
      }
   }
}
```

```
/******************/
/* C code for PE2 */
/******************/
#include <...>
#define H1PORT ...

void main ()
{
int H1, A2, B3, G6, temp;
while (1) {
   H1 = Read (H1PORT);
   if (H1) {
      CommOutput (H1);
      CommOutput (H1);
      CommOutput (H1);
      A2 = CommInput ();
      B3 = CommInput ();
      if (A2) {
        CommOutput (B3);
        temp = CommInput ();
        CommOutput (temp);
        }
```

```
     G6 = CommInput ();
     if (A2) {
       G6 = temp;
       }
     CommOutput (G6);
     }
   }
}


/******************/
/* C code for PE4 */
/******************/
#include <...>
#define D4PORT ...

void main ()
{
int H1, A2, B3, E8, temp;
while (1) {
  H1 = CommInput ();
  if (H1) {
    A2 = CommInput ();
    if (A2) {
      temp = Read (D4PORT);
      E8 = CommInput ();
      temp = temp + E8;
      CommOutput (temp);
      }
    }
  }
}


-- VHDL for Control FSM of PE3
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_components.all;

entity AUTOMAT_proc_3 is
port ( H1 in : std_logic;
     operation_type out : std_logic_vector(5);
     operand_1 out : std_logic_vector(1);
     operand_2 out : std_logic_vector(1);
     reset in : std_logic;
```

```
     clock in : std_logic; );
end AUTOMAT_proc_3;

architecture BEHAVIORAL of AUTOMAT_proc_3 is
type state_automat is ( STATE_0,
                STATE_1,
                STATE_2,
                STATE_3,
                STATE_4 );
signal state, next_state : state_automat;
begin
process
  begin
    wait until clock = '1';
    if reset = '1' then state <= next_state;
    end if;
end process;

process ( H1, reset, state )
begin
case state is
  when STATE_0 =>
    if reset = '1' OR H1 = '0' then
     next_state <= STATE_0;
    else next_state <= STATE_1;
    end if;
  when STATE_1 =>
    next_state <= STATE_2;
  when STATE_2 =>
    next_state <= STATE_3;
  when STATE_3 =>
    next_state <= STATE_4;
  when STATE_4 =>
    next_state <= STATE_0;
end process;

process ( state )
begin
case state is
  when STATE_0 =>
    operation_type <= '00000'; -- CommInputLogical
    operand_1 <= '0'; -- H1
    operand_2 <= '0';
  when STATE_1 =>
    operation_type <= '00011'; -- ReadInteger
    operand_1 <= '0'; -- B3PORT
    operand_2 <= '0'; -- temp
```

```
  when STATE_2 =>
    operation_type <= '00101';
-- CommOutputInteger
    operand_1 <= '0';
    operand_2 <= '0'; -- temp
  when STATE_3 =>
    operation_type <= '00001';
-- CommInputInteger
    operand_1 <= '0';
    operand_2 <= '0'; -- temp
  when STATE_4 =>
    operation_type <= '00101';
-- CommOutputInteger
    operand_1 <= '0';
    operand_2 <= '0'; -- temp
end process;
end BEHAVIORAL;
configuration CFG_AUTOMAT_proc_3_BEHAVIORAL of
AUTOMAT_proc_3 is
  for BEHAVIORAL
  end for;
end CFG_AUTOMAT_proc_3_BEHAVIORAL;
```

4 Conclusions

After having reached this point it is time for us to evaluate the results of our first steps in co-design. The most important thing is to test our methodology on a larger scale application and gather the necessary data to evaluate quantitatively the quality of both the results and the methodology. As a first conclusion we can say that we have managed to combine SIGNAL's advantages in functional specification, production of functionally correct specifications, formal verification tools, code generation tools by defining a methodology for the development of mixed hw/sw systems. At the same time we achieved at a first degree to integrate the tools, discipline their cooperation and facilitate the designer throughout the development lifecycle from specification down to implementation. Right now we are working towards the more detailed modelling of components into a library database, consider other possible target architectures suitable for other application domains, the transparent use of the collection of our tools in the context of an integrated methodology framework.

REFERENCES

[1] *"Programming Real Time Applications with SIGNAL"*, Paul Le Guernic, Michel Le Borgne, Thierry Gautier, Claude Le Maire, IRISA Research Report No 1446, Jun. 1991

[2] *"For a new Real-Time Methodology"*, Thierry Gautier, Paul Le Guernic, Olivier Maffeis, IRISA Internal Publication No 870, Oct.1994

[3] *"The Synchronous Approach to Reactive and Real-Time Systems"*, Albert Benveniste, Gerard Berry, Proceedings of the IEEE, vol.79, no.9, Sep. 1991

[4] Special section: R/T Programming, Proceedings of the IEEE, vol.79, no.9, Sep. 1991

[5] *"Compilation de SIGNAL: horloges, dependances, environment"*, Loic Besnard, PhD thesis, University of Rennes I

[6] "The SIGNAL dataflow methodology applied to a production cell", T.P. Amagbegnon, P. Le Guernic, H. Marchand, E. Rutten, Lecture Notes in Computer Science LNCS No 891, Springer Verlag, Jan. 1995

[7] "VHDL & SIGNAL: A Cooperative Approach", M. Belhadj, Proceedings of International Conference on Simulation and Hardware Description Languages, pgs 76-81, Jan. 1994

[8] "Using VHDL for Link to Synthesis Tools", M. Belhadj, Proceedings of North Atlantic Test Workshop, Nimes France, Jun. 1994

[9] *"The SynDEx software environment for real-time distributed systems design and implementation"*, C. Lavarenne, O. Seghrouchni, Y. Sorel, M. Sorine, ECC91 European Control Conference, Grenoble, France Jul. 91 pp 1684-1689

[10] *"Cout d' execution de programmes SIGNAL"*, Simon de Givry, Rapport de Stage, DEA Informatique, University of Rennes I

[11] *"System Synthesis via Hardware-Software Codesign"*, R.K. Gupta, Giovanni De Micheli, Computer Systems Laboratory Technical Report, CSL-TR-92-548

[12] *"A Hardware-Software Codesign Methodology for DSP Applications"*, Asawaree Kalavade, Edward A. Lee, IEEE Design & Test of Computers, pgs 16-28, Sep. 1993

[13] *"A Formal Approach to Managing Design Processes"*, Reid A. Baldwin, Moon Jung Chung, IEEE Computer, pgs 54-63, Feb. 1995

[14] *"A Global Criticality / Local Phase Driven Algorithm for the Constrained Hardware / Software Partitioning Problem"*, A. Kalavade, E.A. Lee, IEEE Proceedings, Intl. Conf. on HW/SW Codesign, pgs 42-48, 1994

[15] *"Synthesis Steps and Design Models for Codesign"*, Tarek Ben Ismail, Amine Jerraya, IEEE Computer, pgs 44-52, Feb. 1995

[16] *"Design Methodology Management"*, S. Kleinfeldt, M. Guiney, J.K. Miller, M. Barnes, Proceedings of the IEEE, vol.82, no.2, Feb. 1994

[17] *"A Methodology and a Tool for the Evaluation of possible Implementations for Systems specified in SIGNAL"*, A. Kountouris, Draft for an IRISA Research Report, June 1995

[18] "A Retargetable Compiler for ANSI C", C.W. Fraser, D.R. Hanson, SIGPLAN Notices 26, pgs 29-43, Oct. 1991

[19] "A Code Generation Interface for ANSI C", C.W. Fraser, D.R. Hanson, Software Practice and Experience, v.21, pgs 963-988, Sep. 1991

[20] "SPRINGPLAY: A New Class of Compile-Time Scheduling Algorithm for Heterogeneous Target Architectures", G. Paller, C. Wolinski, 3rd IEEE RTAW, Florida US, Nov. 1995

[21] "Hardware-Software Codesign", Guest Editors Introduction, Giovanni De Micheli IEEE Micro, pgs 8-9, Aug. 1994

Unité de recherche INRIA Lorraine, technopôle de Nancy-Brabois, 615 rue du jardin botanique, BP 101, 54600 VILLERS-LÈS-NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, domaine de Voluceau, Rocquencourt, BP 105, LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105 LE CHESNAY Cedex (France)

ISSN 0249-6399