VLSI DESIGN
1998, Vol. 7, No. 4, pp. 401–423
Reprints available directly from the publisher
Photocopying permitted by license only

Formal Codesign Methodology with Multistep Partitioning

VINCENZA CARCHIOLO*, MICHELE MALGERI and GIUSEPPE MANGIONI

Istituto di Informatica e Telecomunicazioni, Facoltá di Ingegneria – Universitá di Catania, Viale Andrea Doria, 6 I 95125 Catania

(Received 22 February 1996; In final form 26 August 1996)

A codesign methodology is proposed which is suitable for control-dominated systems but can also be extended to more complex ones. Its main purpose is to optimize the trade-off between hardware performance and software reprogrammability and reconfigurability. The methodology proposed intends to cover the development of the whole system. It deals in greater detail with the steps that can be made without the need for any particular assumption regarding the target architecture. These steps concern splitting up the specification of the system into a set of individually synthesizable elements, and then grouping them for the subsequent mapping stage. In order to decrease the complexity of each partitioning attempt, a two step algorithm is proposed, thus permitting a wide exploration of possible solutions. The methodology is based on the TTL language, an extension of the T-LOTOS Formal Description Technique which provides a large amount of operators as well as a formal basis. Finally, an example pointing out the complete design cycle, excepting the allocation stage is provided.

Keywords: Codesign, formal description technique, embedded systems, partitioning, top down method

1. INTRODUCTION

The design of complex systems comprising hardware and software elements is of considerable interest on account of its extremely varied applications, thanks to the current availability of low-cost hardware devices. It is of fundamental importance to optimize both the cost and the performance of such systems; various studies have been carried out on this kind of design, which is commonly called codesign. Codesign is an approach to the development of systems composed by both hardware and software modules [1, 2]. Its main purpose is to optimize the trade-off between hardware performance and software reprogrammability and reconfigurability. Moreover the aim of codesign is to be able to design a whole system without excessive preliminary constraints on mapping the module onto hardware and software parts [3, 4].

*Corresponding author.

At present the sector which seems to offer most prospects of codesign methodology application is that of embedded control-dominated systems, thanks to their low complexity. In these systems output signals are caused directly by input signals, which generally means that the systems do not require extremely complex processing of the input signals.

Embedded systems are often used in life-critical situations, where reliability and safety are more important criteria than performance. For this reason we believe that the design approach should be based on the use of a formal model to describe the behaviour of the system before a decision on its implementation is taken.

In this paper we propose a codesign methodology which is not only suitable for the above-mentioned systems, but can also be extended to more complex ones. The systems to which our codesign methodology is applied are control-dominated ones.

In order to achieve the final partitioning it is necessary to define the processor, hardware components and interfaces – generally referred to as the target architecture. The methodology proposed currently refers to an architecture including a single general-purpose processor and a few application-specific hardware components (ASIC or FPGA), a single-bus master software component and a single-level memory hierarchy [5–7].

As the methodology proposed intends to cover the development of the whole system, that is, from the specifications in terms of both time and behaviour to implementation of its components (the software components by using a programming language, the hardware ones by synthesis) certain choices have to be made, especially that of the technique used to describe the system.

The language used for specification of the system is TTL [8] (Templated T-LOTOS), an extension of T-LOTOS [9] specially developed for use in codesign. TTL is also a valid tool for the subsequent stages of development, on account of its formal bases and the operators it provides. TTL allows consistency to be tested using mathematical properties instead of simulation approaches; in this sense the methodology is said to be formal.

The issues this paper deals with in greater detail are the way in which the specification of the system is split up into a set of individually synthesizable elements, and the way in which they are grouped prior to the mapping stage. These choices are made without the need for any particular assumption regarding the target architecture. It will need to be chosen before mapping. However, the paper does not deal in detail with the problem of mapping, because it is possible to use most of the approaches in literature.

The final part of the paper presents a case study in order to evaluate the proposed design methodology.

The steps needed to go from specification to implementation are sketched in Section 2. Section 3 gives a brief description of the formal technique used in the design. Section 4 describes in more detail the process of specification and decomposition. Section 5 explains preclustering, which is a part of partitioning. Section 6 discusses implementation and Section 7 introduces a case study for the present methodology. Section 8 provides the authors' conclusions.

2. AN OUTLINE OF CODESIGN METHODOLOGY

Figure 1 outlines the main steps which go from specifications to implementation of the system.

The first step in the methodology is, therefore, the development of the specifications, using TTL. The **specification** stage is followed by the splitting stage which is subdivided into two steps: refinement and decomposition. The first splitting step, called **refinement**, makes the description of the system less abstract, thus passing from specification of the requirements of the system (maximum abstraction) to a structured representation (minimum abstraction). The refinement step, in which specifications are made less abstract, in reality includes several cycles of subsequent refinement.

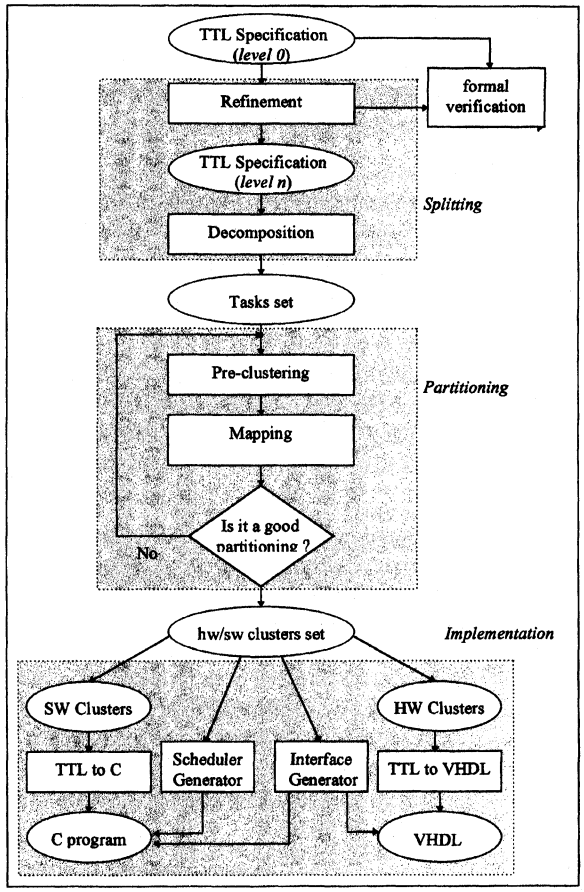


FIGURE 1 Methodology overview.

The TTL language provides adequate support during the whole stage, thanks to its operators and formal basis.

The second splitting step, called **decomposition**, consists of dividing the specifications up into a set of elements (called tasks) which can be synthesized separately. Decomposition is based on syntactic and semantic specification characteristics, as discussed in greater detail in Subsection 4.2 (a similar approach to specification can be found in [10]). Thanks to this approach the computational complexity required is quite low.

At this stage, however, there are still no constraints on whether an element is to be implemented in hardware or software.

In the methodology proposed, the set of tasks obtained from the decomposition step undergoes two further stages which together perform partitioning. In the first (**preclustering**) the number of tasks is reduced below a certain threshold (grouping them into so-called clusters) in order to make the subsequent mapping stage less computationally complex. Preclustering is followed by a **mapping** stage in which the clusters are classified as hardware or software, grouped together if necessary and mapped on the target architecture; this last stage performs the functions usually referred to as partitioning in codesign. Dividing partitioning into two stages speeds the operation up and improves the cost-performance trade-off of the system being developed.

The software partitions obtained are then translated into C, the hardware partitions into synthesizable VHDL. There is, however, nothing to prevent the choice of other languages; the current choice was dictated by the wide availability of tools for these languages. If the results of the mapping phase are unsatisfactory, the clustering process can be repeated starting from any stage, varying, for instance, the number of clusters produced by preclustering. This fact is a peculiarity of the methodology proposed thanks to the choice of TTL and the use of the same language throughout design. Finally, the interface and the software scheduler are also generated, on the basis of the target architecture and the hardware and software partitions.

3. TTL: THE SPECIFICATION LANGUAGE

In literature the problem of the technique to be used to describe a system has been widely discussed. Several specification methods have been proposed, including, FSM [11, 12], Petri nets [13] and high level languages [5, 14, 15, 16].

In this paper we use TTL (Templated T-LOTOS) as the specification language. It is an extension of T-LOTOS ([9, 17]) which is suitable for the codesign approach.

The main extensions TTL introduces to T-LOTOS are modularization (allowing, for example, the use of libraries), use of templates (allowing the definition of a generic process) and the introduction of an iterative construct (**loop**) [8].

The main features of TTL are:

- A high degree of abstraction. This makes it possible to concentrate on what is to be done without being affected by problems regarding actual implementation. For example, the high degree of abstraction of TTL guarantees that the language is suitable for describing both hardware and software, regardless of the target architecture.
- Concurrency. This feature makes it possible to model systems made up of various parts which evolve in parallel, a situation typical of hardware systems.
- The possibility of inserting time references. This makes it possible to specify the timing cons traints and estimate the evolution in time of the system. This feature is necessary for real time systems.
- The possibility of using component libraries. This allows time to be saved in the specification stage and leads to more efficient design thanks to the re-utilization of already developed and thus carefully tested and optimized-components.
- Formal basis. This allows a mathematical approach (as opposed to a simulation one) to be used to test the consistence of each refinement step with respect to the previous one. Moreover, the formal basis allows us to check that the specification possesses useful properties like deadlock freedom, liveness, respect of time constraints.

To manage the time constraints, we have identified two kinds of time attributes which can describe a wide range of situations: min/max and rate constraints [5]. In TTL min/max constraints can be directly expressed by the time attributes of TTL actions. This is written in TTL as:

- *min constraint of t_1 time units on a given action* ***a.*** This means that action a has to occur after a delay of at least t_1 time units. This is written in TTL as $a\{t_1 \ldots \infty\}$.
- *max constraint of t_1 time units on a given action* ***a.*** This means that action a has to occur within a maximum of t_1 time units. This is written in TTL as $a\{0 \ldots t_1\}$.
- *min/max constraint of t_1 time units on a given action* ***a.*** This is a combination of the two previous cases. If the minimum delay is t_1, and the maximum t_2, it will be written as $a\{t_1 \ldots t_2\}$.
- *fixed delay of t_1 time units on a given action* ***a.*** It is possible to fix a definite delay by writing $a\{t_1\}$.

The language does not allow rate constraints on actions to be specified directly. This is not a problem, however, as for practical purposes a rate delay can always be expressed as min/max or a fixed delay [18].

TTL has been developed in such a way as to use all the existing tools for T-LOTOS (e.g., Lola [19]). In fact it is possible to translate a TTL specification into T-LOTOS using only syntactical transformations. TTL can be supported by a set of graphic tools which allow the designer to specify the behaviour of the system in a simple, immediate, familiar way. A possible approach would be like the one followed in [20], which illustrates a technique by which it is possible to go from specification of the system by time diagrams to a T-LOTOS specification; since TTL is a superset of T-LOTOS a similar tool can be built.

The language has two components: the first is the description of the behaviour of processes and their interaction, and is mainly based on the CCS [21] and CSP [22] models; the second is the description of the data structure and expressions, and is based on ACT ONE [23], a language for the description of Abstract Data Types (ADTs).

The syntax of the most important TTL operators is summarized in Table I; a complete description of TTL syntax and semantics can be found in [24].

4. SPLITTING

4.1. Refinement

The process of specifying a system is generally composed of several refinement steps. It starts with

TABLE I Name and syntax of TTL operators

Name	Syntax
inaction	**stop**
termination	**exit** **exit**($E_1, \ldots, E_n$)
choice	B_1[ ] B_2
action-prefix	g;**B** i;**B** $g\ d_1, \ldots, d_n$[SP];B where d_i is ?$x:T$ or !E
parallel-composition	B_1|[$g_1, \ldots, g_n$]| B_2 B_1|||B_2 B_1||B_2
hiding	**hide** $g_1, \ldots, g_n$ **in** B
instantiation	p[$g_1, \ldots, g_n$] ($E_1, \ldots, E_n$)
guarding	[GP]- > B
disabling	B_1[> B_2
enabling	$B_1 \gg B_2$ $B_1 \gg$ **accept** $x:t_1, \ldots, x:t_n$ **in** B_2
local-definition	**let** $x:t_1 = E_1, \ldots, x:t_n = E_n$ **in** B $x:t = E$
sum-expression	**choice** g **in** [$g_1, \ldots, g_n$][ ]B **choice** $x:t$[ ] B
par-expression	**par** g **in** [$g_1, \ldots, g_n$]|[$a_1, \ldots, a_n$]| B **par** g **in** [$g_1, \ldots, g_n$]||| B **par** g **in** [$g_1, \ldots, g_n$]|| B
loop-expression	**loop** (guard; value-expression; B1)

a system-level description and proceeds by splitting the system into increasingly smaller pieces, until it reaches a level at which the single pieces can either be constructed by combining library components or are described directly.

Figure 2 shows the refinement process during the preliminary stages of codesign.

Level 0 coincides with top-level system specification: at this level it is preferable to describe the system in as abstract a way as possible. The next n steps go from the abstract description of the system to a concrete one: at each refinement step the functional blocks are split into more elementary ones, leaving the behaviour of the system unchanged. Consistency between the description of the system at level n and that at level $n-1$ is verifiable thanks to the formal base of the language. Traditionally the consistency between what is specified at level n and level $n-1$ was checked by simulation. The use of a language like TTL supports a better approach to system specification.

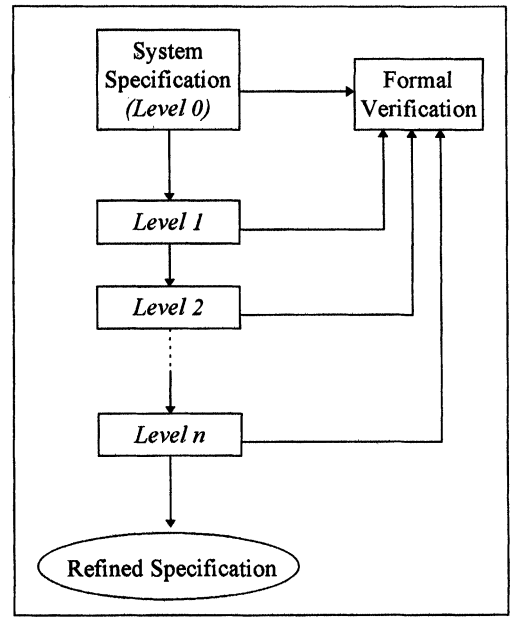


FIGURE 2 Refinement Steps.

The aim of the refinement step is to obtain specifications which can be efficiently implemented and, at the same time, represent the same system as level 0.

The division into modules also requires definition of the signals that have to be exchanged among the modules, which are usually called *internal* signals.

Exploiting the formal basis of the language, TTL aids the designer throughout the refinement process, giving mathematical certainty that the descriptions in the various steps are consistent.

The modularity of TTL also makes it possible to implement and include library components which have already been tested and used. This plays an important role in this step of the methodology, as the replacement of certain blocks with library modules which have a hardware counterpart notably increases the efficiency of the system.

4.2. Decomposition

The aim of the system decomposition stage is to identify the main functional blocks of the system

being developed. These blocks represent the bricks which will be used in the subsequent stages to perform partitioning. Decomposition consists of splitting up the TTL specifications until a set of tasks which can be synthesized separately is found.

The main aim of codesign methodologies is to identify the blocks which permit the trade-off between performance and manufacturing costs to be optimized. It is, however, not desirable at this stage to make choices that constrain when a block must be mapped onto hardware or software. This would reduce the degree of freedom in the partitioning phase and therefore the possibility of obtaining a near-optimal system. In addition, making implementation choices at this stage would reduce the possibility of re-designing the system. Therefore to obtain the maximum independence from the final implementation, the decomposition criteria used must not involve choices which depend on the target architecture and considerations concerning implementation. On the basis of the considerations made so far, therefore, the data used must be obtained from the characteristics of the specifications alone.

Given the features of TTL there are two possible alternatives for the choice of parts we consider to be elementary.

- Considering the single TTL constructs to be elementary, i.e., considering the operators which make up the behavioural expressions (external offer, choice, etc.).
- Considering the processes to be elementary.

The first hypothesis can be discarded straight away as it would lead to an excessively high number of tasks, thus introducing too high a degree of complexity and fragmentation. The second is more plausible, also in view of subsequent translation from TTL into the language which will be used to implement the system.

Due the tool currently used to translate the specification into synthesizable languages, if other processes are instanced inside a given process they have to be part of it. This hypothesis can be discarded using an ad hoc developed TTL synthesizer or some other translator which also accepts generic processes.

The decomposition process starts from the main specification and decomposes it according to the parallelism between the various processes. Figures 3a and 4a give some examples of decomposition into tasks. In the first example three tasks are obtained as they instance no other processes and are each parallel with the other two. In the second example the result of the decomposition process is two tasks, as process P_2 instances P_3 and so they constitute a single atomic element.

Decomposition can be performed automatically by means of a recursive algorithm which applies the considerations made previously.

The starting point of the decomposition algorithm is the tree which represents the hierarchy of TTL processes according to how they are instanced; the trees for the processes in the example given above are shown in Figures 3b and 4b. Each node in the tree can be labelled with an attribute which indicates whether it is made up of a parallel combination of other processes. The possible values of this attribute are:

- **para**, which indicates that the node is made up of a parallel combination of other processes (as in the specification in the first example in Fig. 3);

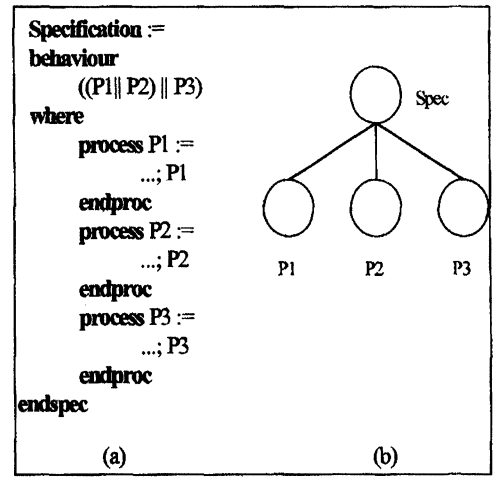


FIGURE 3 Decomposition example of parallel process only.

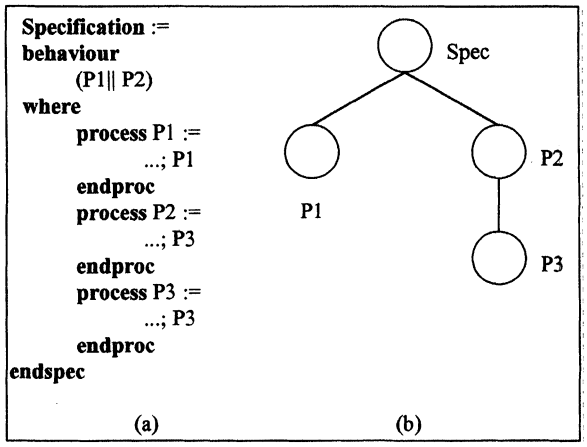


FIGURE 4 Decomposition example of process instancing another process.

- **nopara**, which means the opposite of **para**. The attribute **nopara** is also assigned to processes which instance themselves (like process P_1 in Fig. 3).

Figure 5 shows an example of a tree of processes, where the single nodes are marked with the relative attribute. The algorithm for the decomposition into tasks is described in Figure 6 (the function do_task is described in C-like language).

In the algorithm, *attrib*(*node*) indicates the function which returns the value of the node attribute,

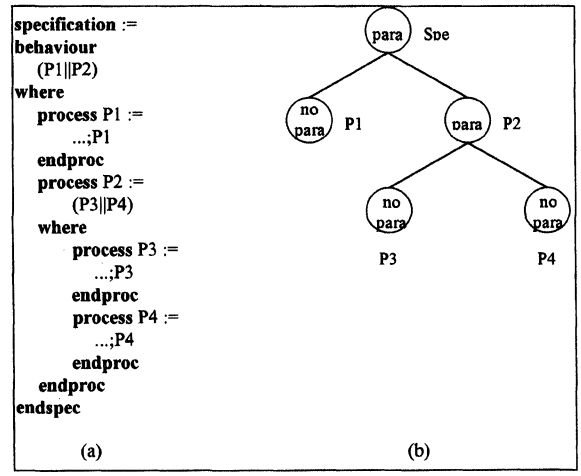


FIGURE 5 Example of decomposition with tree labelling.

- step 1. Create Process Tree from the Specification and labelling of nodes.
- step 2. Set i=0 and calling function do_task(*tree*).

```
where:
do_task(alb *root){
    if (attrib(root)==PARA){
        for each branch k-th of root{
            do_task(branch_k-th(root))
        }
    }
    else if (attrib(root)==NOPARA){
        if (root is selfinstancing process){
            i=i+1, Task_i=root, return
        }
        else{
            i=i+1, Task_i=Σ P_j, where P_j are instanced
                in every branch of root , return
        }
    }
}
```

FIGURE 6 Decomposition algorithm.

while $branch_{k\text{-th}}(node)$ indicates the function which gives the k-th branch of the node and Task_i is the i-th task.

When applied to the example in Figure 5 the algorithm gives the following results:

$$T_1 = P_1, \quad T_2 = P_3, \quad T_3 = P_4.$$

By adopting this algorithm it is possible for a given specification to be decomposed into a low number of tasks. However, too low a number of tasks would mean few alternatives in the partitioning stage and therefore little chance of exploring hw/sw trade-offs.

The optimal case is when the specification is composed of a hierarchy of processes of two types:

1) Processes which are only a parallel combination of other processes; and
2) Processes which instance themselves. In this case, in fact, the tasks are equivalent to the branches of the process tree, and so the maximum possible number.

To achieve close to optimal results, the initial specification of the system has to be made in a style that will favour the partitioning process. In

practice, it has to be made in such a way that the processes fall into one of the following categories:

- Processes which instance themselves;
- Processes obtained by means of a parallel combination of several processes.

It should be pointed out that these rules should only be taken as suggestions as to the specification style to be adopted and not as TTL constraints.

5. PARTITIONING

After splitting the specifications up into tasks according to the criteria outlined above, the partitioning stage starts. It aims to map tasks onto hardware or software components. In our approach the partitioning is divided into two stages in order to reduce the complexity and the computational cost, which are critical in developing complex systems; these stages are called preclustering and mapping. The main difference between the two stages is that mapping is made after choosing the target architecture (e.g., the type of processor, hardware circuits, bus etc.), according to the actual delay introduced by the modules and their manufacturing (monetary) costs, while preclustering groups the tasks together according to their "coupling degree". We focus our attention on the preclustering stage showing an algorithm which is able to perform it at a very low computational cost. The results of preclustering can be used by most mapping strategies, to be found in literature, without any change.

5.1. Pre-clustering

The aim of preclustering is to reduce the number of tasks to be partitioned with the purpose of reducing the complexity of the problem of partitioning.

The number of "sets of tasks" (which will be called clusters) generated by preclustering is obviously of critical importance for mapping. If this number is too high the complexity of the problem is not significantly reduced; whereas if it is too low, the mapping will not achieve a good cost-performance trade-off. This is due to the fact that the only stage where delays and manufacturing costs are taken into account is mapping.

A lower computational cost would suggest executing preclustering until is possible to reach such a low number of clusters that they will be allocated without making any choices in the mapping stage. On the other hand, the greater number of parameters taken into account during the mapping stage would suggest giving it as many optimization chances as possible by providing a large number of clusters. The best solution is probably a compromise between the two strategies. The most suitable number of clusters that preclustering has to provide the mapping with is very difficult to establish a priori and up to now our methodology has proceeded by trial and error. However, we are working on partially automating this choice, basing it on data collected during previous design cycles and interactions with designers.

The preclustering algorithm adopted to group tasks attempts to minimize the coupling degree among the tasks defined as the "*number of interactions between two tasks*". We believe the coupling degree is critical for implementation of the final device, mainly because the higher it is, the higher the communication will be, which increases the cost connected with interfaces. The preclustering stage works on the system before the choice of target architecture, so it is not possible to know the manufacturing cost or the delay cost. The coupling degree, instead, can be evaluated and it appears to be a valid heuristic method to reduce the complexity of problems: in fact, by reducing the coupling degree tasks with higher interactions will be grouped together and will be mapped on the same partitions (either software or hardware).

On the basis of the tasks output by the decomposition process, the preclustering algorithm constructs a weighted (with respect to the coupling degree) graph of the various tasks and works on this to group them into separate clusters.

Construction of the graph is preceded by classification of the task interaction point by identifying the type of data exchanged with the other tasks. Then each type of data is associated with a weight which depends on the amount of interaction introduced by the transaction.

The weighted graph has a biunique correspondence with the set of tasks output by the decomposition process. More specifically:

- Each task corresponds to a vertex in the graph;
- Each interaction point corresponds to an edge with a weight given by the function *coupling Degree* (*interactionPoint*) which gives the weight associated with the type of gate.

Figure 7 gives a simple example to clarify the concept. We assign a weight of one to the Boolean type and a weight of sixteen to the Int type; thus the resulting graph is shown in Figure 8. Therefore the values returned by the function are the following:

$$couplingDegree(g1) = couplingDegree(g4) = 16$$
$$couplingDegree(g2) = couplingDegree(g3) = 1$$

Given a graph with p nodes $v_1, v_2, \ldots, v_p$, it is possible to associate with it an adjacency (or distance) matrix, $p \times p$ in size, in which the element a_{ij} is equal to the weight of the edge which connects nodes v_i and v_j (if the edge does not exist we assume that it has a weight of 0).

If, in the previous example, we decide to provide two tasks as input to the mapping stage algorithm, it would be natural to combine task (2) and task (3) as they are the ones which interact the most.

Figure 9 shows the preclustering algorithm written using a *C*-like syntax.

When execution terminates, the set C will contain the n clusters which minimize the total coupling degree function, defined as:

$$globalCouplingDegree(C) = \frac{1}{2}\sum_{h \in C}\sum_{\substack{k \in C \\ h \neq k}} A[h,k]$$

Figure 10 shows the various steps of the algorithm when applied to the simple example in Figure 7, with $n=2$. Figure 11, on the other hand, shows a more complex example, in which $p=5$ and $n=3$.

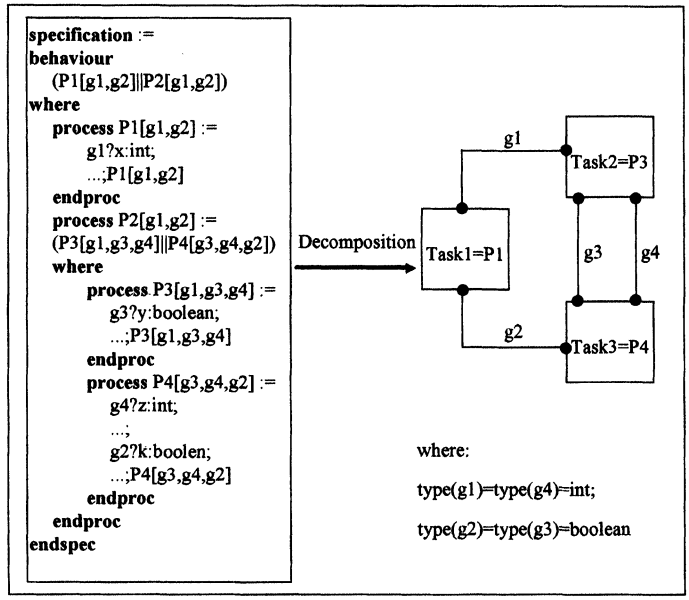
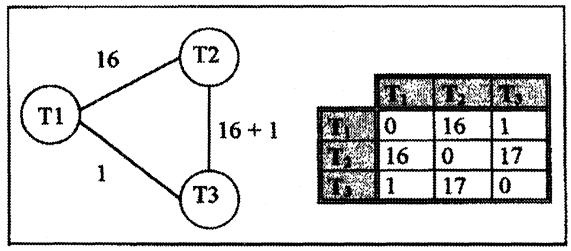


FIGURE 7 Example of gate classification after decomposition.



	T_1	T_2	T_3
T_1	0	16	1
T_2	16	0	17
T_3	1	17	0

FIGURE 8 Weighted graph.

The proposed algorithm is optimal at minimizing the *globalCouplingDegree* function with the same number of final clusters, in the sense that the final configuration is one in which the function reaches the absolute minimum. There may, however, be several configurations in which the *global CouplingDegree* takes on the minimum value.

In the algorithm, the clusters r and s, to be grouped together, are those for which the element $A[r, s]$ is the maximum of all the elements in the matrix. If there are several elements which take on the maximum value, the algorithm used in the examples chooses one at random. Some enhancement could be made in order to improve the effectiveness of the algorithm in choosing the best element. To make this clearer, let us consider the example shown in Figure 12, which only reproduces the portion of the graph we are interested in. On the basis of the algorithm illustrated above, task (2) could be clustered with either (1) or (3, 4) obtained by a previous iteration of preclustering. A variation to the algorithm suggests clustering task (2) with task (1) as this solution, in minimizing the coupling degree, produces final clusters with a

$A[i,j]_{pxp}$ is the weigthed graph matrix (of p nodes), where $n \leq p$ is the final cluster number and C is the set of all cluster.

Initially let C containts all the tasks provided by decomposition, and let $c=p$.

While $(n < c)$ {
  Merge cluster r and cluster s obtaining the cluster (r,s):
    $C \Leftarrow C\text{-}(r)\text{-}(s)$;
    $C \Leftarrow C + (r,s)$ where r and s make true $A[r,s] = \max\{A[i,j] \mid i,j \in C\}$

  Update $A[i,j]$:
    $\forall t \in C$: $A[t,(r,s)] = A[t,r] + A[t,s]$
    $c = c\text{-}1$
}

FIGURE 9 Pre-clustering algorithm.

step	c	A[i,j]	r	s	C	gCouplingDegree
0	3	(1) 0 16 1 (2) 16 0 17 (3) 1 17 0	-	-	(1,2,3)	34
1	2	(1) 0 17 (2,3) 17 0	2	3	(1,(2,3))	17

FIGURE 10 Example of simple pre-cluster algorithm application.

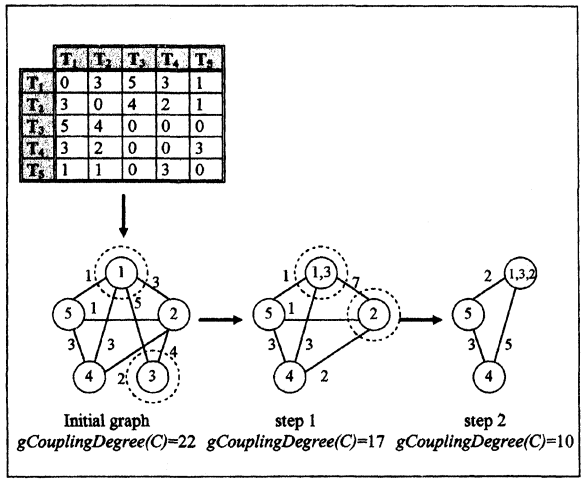


FIGURE 11 Another example of pre-cluster algorithm application.

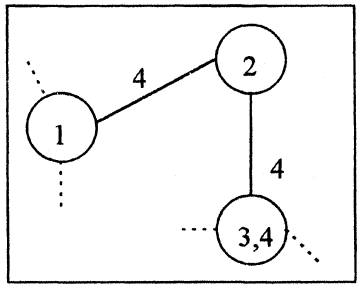


FIGURE 12 A sample of graph showing enhanced algorithm.

lower number of tasks for each. This modification, along with some others, has not been shown for the sake of simplicity but they are implemented in the working program. Having *smaller* clusters means it is easier to explore hw/sw trade-offs and consequently obtain a better final solution.

5.2. Mapping

This is the stage where the various clusters output by the preclustering stage are classified as hardware or software and allocated to the target architecture. The purpose of this stage is to allocate each module either to software or hardware trying to maximize the performance of the system and minimize the cost (in terms of money) of manufacturing. To achieve this result the system must find the best allocation for each module. The mapping is influenced by the target architecture chosen because it imposes requirements on the dimension (of the hardware part, memory available, etc.) and on the interfaces between hardware and software. Moreover, the scheduling algorithm has a strong impact on the performance of the system [25, 26].

The methodology has been devised in such a way as to leave a wide choice of partitioning methods. The mapping problem is not addressed by this paper; some interesting strategies can be found in [27] and [28], each one can easily be integrated in our methodology and benefits from the reduction in the number of input tasks.

6. IMPLEMENTATION

6.1. Scheduler, Interfaces, I/O and Software Module Implementation

The last stage in the development of a device, performed using typical codesign techniques, is the definition of the interfaces (i.e., software drivers and hardware counterpart) between the modules and the scheduling algorithm needed to manage the active tasks.

Such an algorithm is needed because the various modules allocated to software use shared resources, such as the CPU, and also because it is required to manage the exchange of information between hardware and software.

Choice of the interfaces affects the performance of the system as a whole and is closely correlated with the scheduling algorithm. Interfaces and scheduling algorithms can indeed be said to represent a single feature of the system and they have to be chosen when the target architecture is defined [29].

The scheduling algorithm is essentially the operating system of the device being developed and its main aim is to activate all the software tasks correctly and in the right sequence and, at the same time, manage synchronization of the

hardware modules; all these operations have to be performed in such a way as to respect the time constraints of the device (max delay, max response time, etc.). Choice of the appropriate scheduling algorithm has to reach a compromise between the need for a complete, reliable manager and the need to avoid using excessive resources, especially memory and CPU time, to manage itself. This last point is even more important when the device comes under the category of control-dominant systems, where the management routines for single signals are relatively simple and so do not require very long processing times.

Two possible kinds of scheduling algorithm are interrupt-driven and soft-managed.

The interrupt-driven technique is based on the use of classical interrupt management techniques to schedule both software and hardware tasks. Each task (hardware or software) which requires the exchange of an output signal generates an interrupt which activates the related routine. Even though it is logically simple and immediate, the interrupt-driven algorithm introduces the complexity inherent in the problem of saving the context of any routine which may be active when the interrupt occurs and managing priorities. In addition, this algorithm requires memory in which to store context information and a device to manage several interrupt lines so as to be able to cope with all the hardware tasks present.

The soft-managed technique is based on a simple polling algorithm, modified to deal with synchronizing the various tasks. Here the resources needed to manage the algorithm itself are very few, but care must be taken to prevent the time required by the polling cycle from introducing an excessive delay in the management of signals. The technique also has to allow the parallel evolution of all the hardware clusters at least until they require input/output from other modules. The algorithm may also allow some tasks to be queried more frequently if their delay requirements are greater.

Choice of the scheduling technique also affects how the software modules are translated from TTL to *C* because, according to the choice made, different management interfaces and different signal synchronization techniques will have to be inserted. It will be necessary to follow the rendezvous rules imposed by the TTL synchronization protocol, as happens with all the techniques of the same family.

The scheduler can be described in TTL and so it is possible to check that the system comprising the scheduler and the modules behaves correctly before passing on to actual synthesis of the hardware modules, which is costly in terms of time, by simulation of global behaviour. In the future, by exploiting TTL's capacity to describe time quantitatively, it will be possible to obtain the scheduler program in such a way as to respect the time constraints by construction.

6.2. From TTL to VHDL

As said above, a TTL specification comprises a behaviour and a data part. These two parts require different translation procedures.

Data Part

This part is translated by establishing a relation between the types of data in TTL and those of VHDL, in the sense that each type in one language is made to correspond to a type in the other.

Behaviour Part

This part of a TTL description is made up of a set of processes combined by binary operators. It is possible to identify three types of semantic elements to be translated: events, processes and operators.

- *Events*: Synchronization in TTL is achieved by means of multi-way rendezvous. VHDL, on the other hand, achieves synchronization by using signals. It is therefore necessary to decompose the sophisticated TTL rendezvous into VHDL signals.

- *Processes*: A TTL process is quite similar to a VHDL entity where the PORTS can be seen as low-level gates.
- *Operators*: TTL operators are translated into the instructions provided by VHDL.

In this phase of the methodology it is possible to use a tool comprising two modules; the first translates from TTL into T-LOTOS (an extended version of LOTOS including time) and the second from T-LOTOS into VHDL.

The first step involves expanding the modules, templates and loops of TTL to obtain the specification in standard LOTOS with explicit time constraints (which in turn is quite easy to translate into T-LOTOS). For the second step it is possible to use Harpo [30], which accepts T-LOTOS in input and outputs VHDL. Harpo is currently being developed but already presents interesting features, such as the possibility of generating a synthesizable VHDL code. A drawback, however, is the fact that the code generated is too large.

7. EXAMPLE: PONDAGE POWER PLANT CONTROLLER

As an example of application of the method proposed, we present a system to control the production of electricity in a hydroelectric plant. The aim of the example is to show the applicability of the method to quite complex real systems.

7.1. Specifications

The controller essentially has two functions: it has to check the level of the reservoir to make sure it does not exceed a certain limit, and then directly control the production of electrical power.

The system provides for two functioning modes, manual and automatic. In the first mode the parameters involved in power production are supplied manually from the outside, while in the second mode everything is controlled automatically by a daily production program.

The controller presented comprises several blocks: the clock, the daily program, the control panel, the regulator and a set of actuators. Figure 13 shows the structural interconnection between the various blocks, which we reached after performing several refinement steps on the abstract specifications of the system. Figure 14 shows the main TTL specification of the system.

In giving a detailed description of the features of the individual blocks, we will make use of the modularity offered by TTL.

Control Panel The Control Panel sets the functioning mode for the system (manual or automatic). Figure 15 shows the declaration of the Control Panel. It comprises a public process called main and three private processes (which cannot be exported). Figuer 15 gives a definition of the main process. As can be seen, the Control Panel module is in turn a parallel combination of four processes; CNTRL, B1, B2 and B3.

The CNTRL process (a definition of which is given in Fig. 16) has two functions:

- it detects the occurrence of the signal *auto_t* (as opposed to *manu_t*) and informs the regulator of the automatic (as opposed to manual) functioning mode by emitting the signal *auto* (*manual*);
- it memorizes emission of the signals *prgstart*, *prgstop* and *prgwidth*, so as to restore normal functioning when passing from manual to automatic.

The processes B1, B2 and B3 perform a sort of logical OR on the input signals, so as to guarantee correct functioning both in the automatic mode and during the transition from manual to automatic. Figure 17 gives a definition of these three processes.

Daily Program This block manages the daily automatic production of electricity. Figure 18 shows the declaration of the module.

The Daily Program module is a parallel combination of two processes, DP1 and DP2 (see Fig. 18); the first turns the plant on and off, while

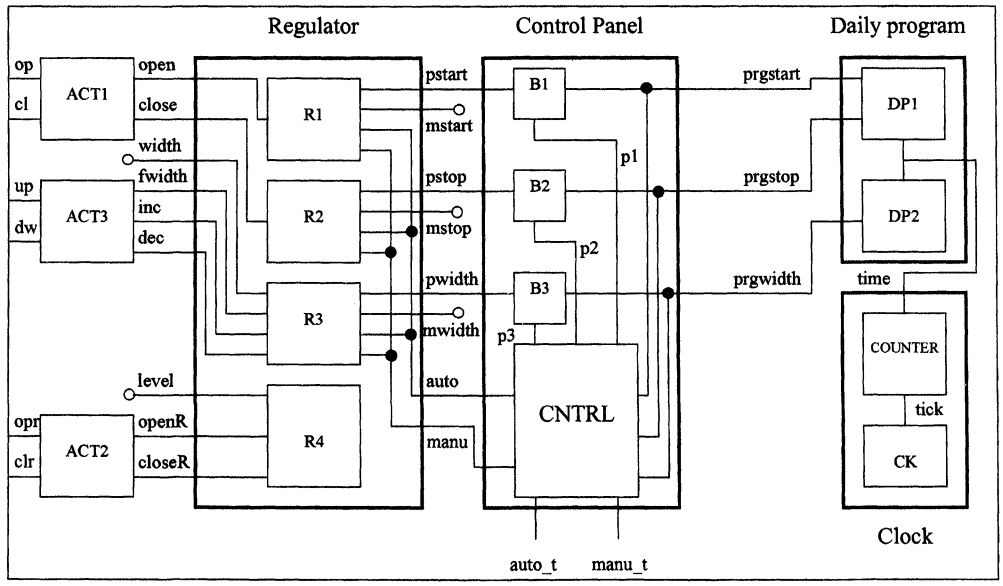


FIGURE 13 Complete scheme of a pondage power plant controller.

```
use comp.dec                                        (* module declaration file *)
specification power_plant[op,cl,width,up,dw,opr,clr,level,mstart,mstop,mwidth,auto_t,manu_t]:noexit
behaviour
  ((((((Regulator.main[open,close,width,inc,dec,level,openR,closeR,pstart,mstart,pstop,mstop,pwidth,mwidth,fwidth,auto,manu]
  |[ pstart,pstop,pwidth,auto,manu]| Control_Panel.main[pstart,pstop, pwidth,auto,auto_t,manu,manu_t])
  |[prgstart,prgstop,prgwidth]|Daily_Program.main[prgstart,prgstop,prgwidth,time])
  |[time]|Clock.main[time])
  |[openR,closeR]|Act.main[openR,closeR,opr,clr])          (* ACT2 *)
  |[fwidth,inc,dec]|Act_step.main[inc,dec,fwidth,up,dw])   (* ACT3 *)
  |[open,close]|Act.main[open,close,op,cl])                (* ACT1 *)
endspec
```

FIGURE 14 Pondage power plant controller main specification.

```
module Control_Panel is
private:
  process B[s1,s2,s3];
  process B3[pwidth,prgwidth,p3];
  process CNTRL[auto_t,manu_t,auto,manu,p1,prgstart,p2,prgstop,p3,prgwidth](x:int,y:int4);
public:
  process main[pstart,pstop, pwidth, auto,auto_t,manu,manu_t];
end Control_Panel

Control_Panel::main[pstart,pstop, pwidth, auto,auto_t,manu,manu_t]: noexit:=
  (((CNTRL[auto_t,manu_t,auto,manu,p1,prgstart,p2,prgstop,p3,prgwidth](0,0)
  |[prgstart,p1]|B[pstart,prgstart,p1])          (* B1 *)
  |[prgstop,p2]|B[pstop,prgstop,p2])             (* B2 *)
  |[prgwidth,p3]|B3[pwidth,prgwidth,p3])
endproc
```

FIGURE 15 Control panel module declaration and main process definition.

```
Control_Panel::
  process CNTRL[auto_t,manu_t,auto,manu,p1,prgstart,p2,prgstop,p3,prgwidth](x:int, y:int4): noexit:=
  auto_t;auto;(
        [x=1]->p1;p3!y;CNTRL[auto_t,manu_t,auto,manu,p1,prgstart,p2,prgstop,p3,prgwidth](x,y)
        []
        [x=0]->p2;CNTRL[auto_t,manu_t,auto,manu,p1,prgstart,p2,prgstop,p3,prgwidth](x,y)
        )
  []
  manu_t;manu;CNTRL[auto_t,manu_t,auto,manu,p1,prgstart,p2,prgstop,p3,prgwidth](x,y)
  []
  prgstart;CNTRL[auto_t,manu_t,auto,manu,p1,prgstart,p2,prgstop,p3,prgwidth](1,y)
  []
  prgstop;CNTRL[auto_t,manu_t,auto,manu,p1,prgstart,p2,prgstop,p3,prgwidth](0,y)
  []
  prgwidth?y:int4;CNTRL[auto_t,manu_t,auto,manu,p1,prgstart,p2,prgstop,p3,prgwidth](x,y)
endproc
```

FIGURE 16 Control panel CNTRL process definition.

```
Control_Panel::process B[s1,s2,s3]: noexit:=
  s2;s1;B1[s1,s2,s3]
  []
  s3;s1;B1[s1,s2,s3]
endproc

Control_Panel::process B3[pwidth,prgwidth,p3]: noexit:=
  prgwidth?x:int4;pwidth!x; B3[pwidth,prgwidth,p3]
  []
  p3?x:int4;pwidth!x; B3[pwidth,prgwidth,p3]
endproc
```

FIGURE 17 Control panel B e B3 process definition.

the second manages differentiated production of electricity according to the time of day.

On the basis of the time signal, the process DP1 (see Fig. 18 for a definition) turns the plant on and off. It is turned on at 6.00 am (by emitting the *prgstart* signal) and turned off at 9.00 pm (by emitting the *prgstop* signal).

The process DP2 (see Fig. 18) uses the time signal to regulate the level of production of electricity. It acts indirectly on the aperture of the valve (signal *prgwidth*: 0 completely closed, 10 completely open) which regulates the flow of water into the power plant. According to the daily requirements, production is divided into three time bands:

- From 9.00 pm on the previous day to 5.00 am on the next day, aperture 0, corresponding to no production of electricity;
- From 6.00 am to 6.00 pm, 70% of maximum production;
- From 7.00 pm to 8.00 pm 50% of maximum production.

Clock Automatic management of production requires knowledge of the real time, which is provided by the block called Clock.

Figure 19 shows the statement of the module and definition of the main process. As can be seen, the Clock block has been decomposed into a parallel combination of two processes – counter and ck.

```
module Daily_Program is
private:
  process DP1[prgstart,prgstop,time];
  process DP2[prgwidth,time];
public:
  process main[prgstart,prgstop,prgwidth,time];
end Daily_Program

Daily_Program:: main[prgstart,prgstop,prgwidth,time]: noexit:=
  (DP1[prgstart,prgstop,time] |[time]| DP2[prgwidth,time])
endproc

Daily_Program:: DP1[prgstart,prgstop,time]: noexit:=
  time?x:int5 in        (
                        [x=6]->prgstart;DP1[prgstart,prgstop,time]
                        []
                        [x=21]-> prgstop,DP1[prgstart,prgstop,time]
                        []
                        [(x!=6) and (x!=21)]->DP1[prgstart,prgstop,time]
                        )
endproc

Daily_Program:: DP2[prgwidth,time]: noexit:=
  time?x:int5;  (
                [(x>=0) and (x<=5)]-> prgwidth!0; DP2[prgwidth,time]
                []
                [(x>=6) and (x<=18)]-> prgwidth!7; DP2[prgwidth,time]
                []
                [(x>=19) and (x<=20)]-> prgwidth!5; DP2[prgwidth,time]
                []
                 [(x>=21) and (x<=23)]-> prgwidth!0; DP2[prgwidth,time]
                )
endproc
```

FIGURE 18. Daily program declaration and definition.

```
module Clock is
private:
  process counter[tick,time](x:int);
  process ck[tick];
public:
  process main[time];
end Clock

Clock:: process main[time]: noexit:=
  (counter[tick,time](0) |[tick]| ck[tick])
endproc

Clock:: process ck[tick]: noexit:=
  tick{1};ck[tick]
endproc

Clock:: process counter[tick,time] (x:int5): noexit:=
  let i:int=0 in
  time!x;
  loop (i<3600; i+1; tick;exit);>> counter[ck,time]((x+1) mod 24)
endproc
```

FIGURE 19 Clock declaration and definition.

Figure 19 also gives a definition of the ck process, which produces a tick every second, exploiting the possibility TTL offers of inserting quantitative time references into the description.

For the purpose of automatically managing the production of electricity, it is sufficient for the time signal to be emitted every hour. We therefore implemented the counter process (see Fig. 19) which uses a typical construct of TTL (loop) to create a counter which puts out the information every 3600 *ticks*.

Regulator This block deals directly with control of the plant. It has two main functions:

- Checking that the level of the reservoir does not exceed a certain emergency threshold, in which case it tries to restore normality by acting on an outlet valve;
- Checking the level of production, and turning the plant on and off by manual or automatic controls.

Figure 20 describes the statement of the regulator module and the main process. Let us analyze the functioning of the processes which make up the regulator.

The processes R1 and R2 function in a similar way (see Fig. 21). The former turns the plant on by opening the valve (*open signal*) of the duct which goes from the reservoir to the plant; the latter turns the plant off by closing the valve (*close signal*). Both processes deal with correct management of the input signals in relation to the functioning mode (manual or automatic).

The process R3 (See Fig. 22) manages the level of production in the plant by acting on the valve which regulates the flow entering the plant. By means of the *width* signal, a sensor communicates the current aperture of the valve, which is compared with what has been programmed (in the automatic functioning mode) or set manually (manual functioning mode). On the basis of the difference between the two values, it acts on the motor which regulates the valve, emitting the signals *inc* and *dec* (which indicate the direction in which the engine has to move to increase or decrease the angle of aperture) and the signal *fwidth* (which indicates the relative angle of rotation of the valve). This process is also constructed in such a way that the automatic and manual functioning modes are managed appropriately.

The process R4 (Fig. 23) controls the level of the reservoir. The current level is provided by the signal *level* (0 to indicate that the reservoir is empty, 10 that it is completely full). The safety level is set to a value of eight, which corresponds to 80% of the maximum capacity. If this level is

```
module Regulator is
private:
  process R[u,i_a,i_m,a,m](x:int);
  process R3[width,pwidth,mwidth,fwidth,auto,manu,inc,dec](x:int);
  process R4[level,openR,closeR];
public:
  process main[open,close,width,inc,dec,level,openR,closeR,pstart,mstart,pstop,mstop,pwidth,mwidth,auto,manu];
end Regulator

Regulator::
main[open,close,width,inc,dec,level,openR,closeR,pstart,mstart,pstop,mstop,pwidth,mwidth,auto,manu]: noexit:=
  (((R[open,pstart,mstart,auto,manu](1)                    (* R1 *)
  ||[auto,manu]||R[close,pstop,mstop,auto,manu](1))        (* R2 *)
  ||[auto,manu]||R3[width,pwidth,mwidth,auto,manu,inc,dec](1))
  |||R4[level,openR,closeR])
endproc
```

FIGURE 20 Regulator module declaration and main process Definition.

```
Regulator:: process R[u,i_a,i_m,a,m](x:int):noexit:=
        [x=1]->(i_a;u;R1[u,i_a,i_m,a,m](x)
                [] m; R1[u,i_a,i_m,a,m](0)
                )
        [x=0]->(i_m;u;R1[u,i_a,i_m,a,m](x)
                [] a; R1[u,i_a,i_m,a,m](1)
                )
endproc
```

FIGURE 21 Regulator R process definition.

exceeded the process R4 activates signals to open an outlet valve so as to bring the situation back to normal.

Act1, Act2 and Act3 The blocks Act1, Act2 and Act3 deal with interfacing between the control system and the actuators which drive the valves. More specifically:

- Act1 runs the motor which controls the valve of the duct going from the reservoir to the plant. There are two possible positions for this valve – open and closed.
- Act2 controls the outlet valve which serves to keep the level of the reservoir below a certain safety level. Here again there are only two possible positions – open and closed.
- Act3 serves as an interface between the system and the stepper motor which controls the inlet valve. There are eleven positions for this valve, from zero to ten, which correspond to 0% and 100% of the angle of aperture of the valve (and therefore indirectly to the level of production).

Act1 and Act2 have a similar structure, the statement of which is given in Figure 24 where definition of the main process is also given.

Act3, as said above, serves as an interface with a stepper motor which can move by steps towards increasing or decreasing angles, according to whether a signal *up* or *dw* is sent. The aim of Act3 is to send as many *up* (or *dw*) signals as the steps supplied by *fwidth*. Figure 24 gives the declaration of the module and a definition of the main process.

7.2. Decomposition

The specification of the system is given in such a way as to obtain the maximum number of tasks in the decomposition phase. Figure 25 shows the tree which represents the hierarchy of TTL processes on the basis of how they are instanced.

Applying the decomposition algorithm, we obtain the following tasks:

$$T_1 = \text{B1},\ T_2 = \text{B2},\ T_3 = \text{B3},\ T_4 = \text{CNTRL},$$
$$T_5 = \text{Act1},\ T_6 = \text{R1},\ T_7 = \text{R2},\ T_8 = \text{R3},$$
$$T_9 = \text{R4},\ T_{10} = \text{Act2},\ T_{11} = \text{DP1},\ T_{12} = \text{DP2},$$
$$T_{13} = \text{Counter},\ T_{14} = \text{Ck},\ T_{15} = \text{Act3}$$

```
Regulator:: process R3[width,pwidth,mwidth,fwidth,auto,manu,inc,dec](x:int):noexit:=
   width?y:int4;(
   [x=1]->    (pwidth?z:int4;(
                        [z>y]->inc; fwidth!(z-y);R3[width,pwidth,mwidth,fwidth,auto,manu,inc,dec](x)
                        []
                        [z<=y]->dec;fwidth!(y-z); R3[width,pwidth,mwidth,fwidth,auto,manu,inc,dec](x)
                        )
              [] manu; R3[width,pwidth,mwidth,fwidth,auto,manu,inc,dec](0)
              )
   []
   [x=0]->    (mwidth?z:int4;(
                        [z>y]->inc;fwidth!(z-y); R3[width,pwidth,mwidth,fwidth,auto,manu,inc,dec](x)
                        []
                        [z<=y]->dec;fwidth(y-z); R3[width,pwidth,mwidth,fwidth,auto,manu,inc,dec](x)
                        )
              [] auto; R3[width,pwidth,mwidth,fwidth,auto,manu,inc,dec](1)
              )
endproc
```

FIGURE 22 Regulator R3 process definition.

```
Regulator:: process R4[level,openR,closeR]:noexit:=
        level?x:int4;  (
                       [x>8]->openR;R4[level,openR,closeR]
                       []
                       [x<8]->closeR;R4[level,openR,closeR]
                       []
                       [x=8]->R4[level,openR,closeR]
                       )
endproc
```

FIGURE 23 Regulator R4 process definition.

```
module Act is
public:
 process main[in1,in2,out1,out2];
end Act

Act:: process main[in1,in2,out1,out2]: noexit:=
 in1;out1;main[in1,in2,ou1,ou2]
 []
 in2;out2;main[in1,in2,out1,out2]
endproc
```

```
module Act_step is
public:
 process main[inc,dec,fwidth,up,dw];
end Act_step

Act_step:: process main[inc,dec,fwidth,up,dw]: noexit:=
 let i:int=0 in
 inc;fwidth?x:int4;loop(i<x;i+1;up); main[inc,dec,fwidth,up,dw]
 []
 dec; fwidth?x:int4;loop(i<x;i+1;dw); main[inc,dec,fwidth,up,dw]
endproc
```

FIGURE 24 Act and act_step process declaration and definition.

7.3. Partitioning

Pre-Clustering Application of the pre-clustering algorithm passes through construction of the adjacency matrix for the weighted graph. We assume that the function *couplingDegree* has the following values:

- 5 for the *time* signal (minimum number of bits required to represent the 24 hours of the day);
- 4 for *prgwidth*, *pwidth*, p3, *fwidth* (needed to represent the 11 positions of the valve);
- 1 for all the other signals.

In this case the matrix of the graph is the one shown in Figure 26. Applying the clustering algorithm to this matrix with $n=10$, we get the following clusters:

$$C_1 = T_1,\ C_2 = T_2,\ C_3 = T_3+T_4+T_{11}+T_{12}+T_{13},$$
$$C_4 = T_5,\ C_5 = T_6,\ C_6 = T_7,\ C_7 = T_8 + T_{15},$$
$$C_8 = T_9,\ C_9 = T_{10},\ C_{10} = T_{14}$$

In this example, we chose to reduce the number of tasks from 15 to 10, on account of particular efficiency requirements. As mentioned previously, in fact, the final number of clusters has to be chosen in such a way as to:

- Reduce the number of tasks as far as possible (and consequently the complexity of the subsequent mapping phase);

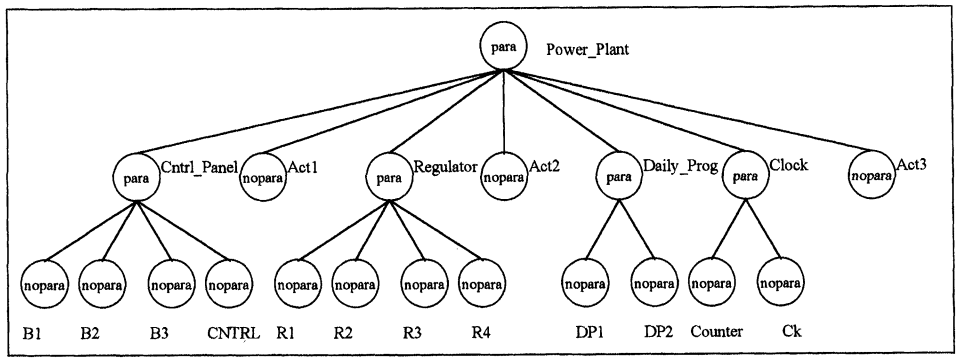


FIGURE 25 Labelled process tree of pondage power plant Controller.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}
T_{15}	0	0	0	0	0	0	0	6	0	0	0	0	0	0	
T_{14}	0	0	0	0	0	0	0	0	0	0	0	0	1		
T_{13}	0	0	0	0	0	0	0	0	0	0	5	5			
T_{12}	0	0	4	4	0	0	0	0	0	0	0				
T_{11}	1	1	0	2	0	0	0	0	0	0					
T_{10}	0	0	0	0	0	0	0	0	2						
T_9	0	0	0	0	0	0	0	0							
T_8	0	0	4	2	0	0	0								
T_7	0	1	0	2	1	0									
T_6	1	0	0	2	1										
T_5	0	0	0	0											
T_4	1	1	4												
T_3	0	0													
T_2	0														
T_1															

FIGURE 26 Weighted graph matrix of pondage power plant controller.

- Not impose constraints on the mapping of the target architecture.

If we had decided to take the final number of clusters to be used as input for the mapping algorithm down to seven, the pre-clustering algorithm would have put out the following clusters:

$$C_1 = T_1 + T_2 + T_3 + T_4 + T_8 + T_{11} + T_{12} + T_{13} + T_{15}, C_2 = T_5, C_3 = T_6, C_4 = T_7, C_5 = T_9, C_6 = T_{10}, C_7 = T_{14}$$

For reasons linked to minimization of the coupling degree, the cluster C_1 is composed of nine tasks; such a critically large cluster which would represent a hard constraint in the mapping stage. This means that a given cluster could be mapped without taking into account the parameters directly linked with the target architecture, which should be decisive for mapping.

In traditional design methodologies, the way in which specification of the system was made was a constraint for the subsequent mapping on the target architecture, as there was a tendency to map the blocks which functionally constituted the specification (e.g., in this case the regulator, the control panel, the daily program, etc.) directly onto hardware or software. In our case, instead, the composition of a cluster is not linked to the functions it performs but is a result of application of an algorithm which minimizes the degree of coupling between the parts of the system. For example, it would have been hard to envisage a cluster like C_3, which is made up of processes belonging to different functional blocks and which in substance represents an optimal choice with respect to the coupling degree parameter.

8. RELATED WORK AND CONCLUSIONS

In this section we will examine the different approaches that can be found in literature to solve each aspect of codesign.

Several techniques have been proposed to tackle the specification of hardware and software; in the following we will sketch the characteristics of some of them.

Esterel [31] is a synchronous language based on FSM. The synchronous hypothesis states that time is described as a sequence of instants, between which no action can takes place. This hypothesis permits the system to be modelled using only a single FSM exhibiting a totally predictable behaviour. Unfortunately the resulting FSM is generally fairly large, thus making it difficult to specify systems with a large amount of concurrency.

Another technique belonging to FSM is State-Charts [32]. It is a graphical specification language which allows hierarchical decomposition, timing, concurrency and subroutines. It allows a concise specification and a clear documentation, but it lacks in specification of software submodules.

Among the other languages used for co-specification we can cite two examples: C^x, the entry language for COSYMA [33], which extends ANSI *C* with delays, tasks and task communication, and Hardware *C* [34] which can be translated into a flow graph.

In the methodology introduced in this paper the specification language used is TTL. It is derived from T-LOTOS, an FDT based on the CCS and CSP process algebras. TTL appears suitable for describing control-dominated systems, as discussed throughout the paper.

As shown in the paper a key problem in codesign methodologies is the validation of the model of the system being developed. Simulation is still the main tool used for this purpose and consists of comparing the model against a set of specifications. Many methods have been proposed in literature, they differ in their method of coupling hardware and software components. For example, in [35] a single custom simulator is used for both hardware and software, whereas another approach proposes using a software process running on a host computer loosely connected with a hardware simulator [36].

TTL aims to perform *verification* on the specification. Formal verification is the process of checking that the behaviour of the system satisfies a given property, also described using a formal method. This approach has been widely adopted to verify the correctness of protocols and it appears useful in hardware/software property checking. It also allows the congruence between two successive refinement steps to be checked without using a simulation approach. For these reasons we refer to our methodology as a "*formal codesign methodology*".

Several solutions to the partitioning problem are proposed in literature. Some use a graph model to represent the operations performed by devices and associate a cost to them [33]. Others perform the partitioning together with the implementation of the scheduling algorithm as, for instance, in [29] where the specification is made with a hardware description language and synthesis tools are used to estimate the costs. The basic idea of performing scheduling and partitioning together is to minimize the response time.

Our methodology divides the partitioning stage into two steps. The first (preclustering) is based only on the properties of the system and aims to reduce the complexity of problems. This is obtained by a simple algorithm whose complexity is very low especially compared with that of the mapping algorithm. The second step groups the remaining clusters and maps onto the target architecture. The strategy used to reduce the complexity of mapping is based on minimization of the interaction among clusters.

Finally some problems dealing with mapping have been discussed, including the choice of the scheduling algorithms needed to allow hardware and software modules to coexist. Proper choice of the scheduling algorithm is, however, an open problem to which further studies must be devoted.

References

[1] De Micheli (Aug. 1994). *Computer-Aided Hardware-Software Codesign.* IEEE Micro.

[2] Hardt, R. and Camposano (Oct. 1993). *Trade-Offs in HW/SW Codesign.* Proc. International Workshop on Hardware-Software Codesign.

[3] Barros, W. and Rosenstiel, X. Xiong (Oct. 1993). *Hardware/Software Partitioning with UNITY.* Proc. International Workshop on Hardware-Software Codesign.

[4] Chiodo, P., Giusto, A., Jurecska, H. C., Hsieh, A. and Sangiovanni-Vincentelli, L. Lavagno (Aug. 1994). *Hardware-Software Codesign of Embedded Systems.* IEEE Micro.

[5] Gupta, R. K. and De Micheli, G. (September 1993). *Hardware-Software Cosynthesis for Digital Systems.* IEEE Design and Test Computer.

[6] Gupta, R. K., Coelho, C. N. and De Micheli, G. (January 1994). *Program Implementation Schemes for Hardware-Software Systems.* IEEE Computer.

[7] Bolognesi, T., D: Latella and Pisano, A: "*Toward a graphic syntax for LOTOS*", Proc. of EUTECO'88, Vienna April 1988, North-Holland.

[8] Carchiolo, V., Malgeri, M. and Mangioni, G. "TTL: A LOTOS Extension for System Description", on Proc. of Basys '96, Lisboa, Portugal.
[9] Quemada, J. and Fernandez, A. (1987). *Introduction of Quantitative Relative Time into* LOTOS IFIP Workshop on Protocol Specification, Testing and Verification VII North Holland.
[10] Eide, A. (March 1993). "Compiling UNITY programs to parallel processes in a coupled environment", Master Thesis, University of Trondheim and FZI, Karlsruhe.
[11] Alur, A. and Dill, D. (1990). "Automata for modelling Real Time Systems", In *Automata Languages and Programming: 17th annual Colloquium*, **443** of LNCS.
[12] Chiodo, P., Giusto, A., Jurecska, H. C., Hsieh, A., Sangiovanni-Vincentelli and L. Lavagno (October 1993). *A Formal Specification Model for Hardware/Software Codesign.* Proc. International Workshop on Hardware-Software Codesign.
[13] Murata (April 1989). *Petri nets: Properties, analysis and applications.* Proc. IEEE.
[14] Carchiolo, V., Di Stefano, A., Faro, G. and Pappalardo (April 1989). *ECCS and LIPS: Two Languages for OSI Systems Specification and Verification.* ACM Transactions on Programming Languages and Systems, **11**(2), pp. 284–329.
[15] Tiedemann, W. D., Lenk, S., Grobe, C. and Grass, W. (1993). *Introducing Structure into Behavioural Descriptions obtained froma Timing Diagram Specification.* Microprocessing and Microprogramming 38, North-Holland.
[16] McCaskill, A. and Milne, G. J. (June 1992). *Hardware description and verification using the CIRCAL-System.* Technical Report HDV-24-92, University of Strathclyde, Department of Computer Science, Glasgow.
[17] ISO IS 8807, *Information Processing Systems, Open System Interconnection, LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* ISO, June 1988.
[18] Chou, P. and Borriello, G. (June 1994). *Software Scheduling in the Co-Synthesis of Reactive Real Time Systems* proceeding of the Design Automation Conference, San Diego CA.
[19] Quemada, S. and Pavón, A. Fernández (June 1989). *State Exp loration by Transformation with LOLA.* Workshop on Automatic Verification Methods for Finite State Systems, Grenoble.
[20] Tiedemann, D., Lenk, S., Grobe, C. and Grass, W. (1993). *Introducing Structure into Behavioural Descriptions obtained from a Timing Diagram Specification.* Microprocessing and Microprogramming 38, North-Holland.
[21] Milner, R. (1980). A calculus of communicating systems. LCNS 92, Springer-Verlag, New York.
[22] Hoare, C. A. R. (1985). "Communicating Sequential Processes". *International Series in Computer Science.* Prentice Hall.
[23] Ehrig, H. and Mahr, B. (1985). *Fundamentals of Algebraic Specifications 1* EATCS Monographs on Computer Science, Springer.
[24] Carchiolo, V., Malgeri, M. and Mangioni, G. July 1995. "TTL: Templated T-LOTOS", Internal report of the University of Catania.
[25] Takach, W. Wolf. (January 1995). *An automaton model for scheduling constraints in synchronous machines.* IEEE Transactions on Computers.
[26] Axelsson (June 1995). "Analysis and Improvement of Task Schedulability in Hardware/Software Codesign", Internal Report Linkping University, Sweden, LITH-IDA-R-95, 24.
[27] Kalavade and Lee, E. A. (June 1995). "The extended Partitioning Problem: Hardware/Software Mapping and Implementation-Bin Selection", Proc. of Inter. Workshop on Rapid Prototyping, Chapel Hill, NC.
[28] Lopez, M. Jan. 1995. *Reference Manual for the LOTOS to VHDL translation tool.* Internal report of FORMAT/ESPRIT Project No. 6128.
[29] Olokutun, K., Helaihel, R., Levitt, J. and Ramirez, R. (August 1977). A software-hardware cosynthesis approach to digital system simulation. IEEE Micro, **14**(4), 48–58.
[30] Delgado Kloos, de Miguel Moro, T., Valladares, T. R., Filho, G. R. and Lopez, A. M. (1993). *VHDL generation from a timed extension of the formal description technique LOTOS within the FORMAT project.* Microprocessing and Microprogramming 38, North-Holland.
[31] Berry, G., Couronne', P. and Gonthier, G. (September 1991). The synchronous approach to reactive and real-time systems. *IEEE Proceeding*, 79.
[32] Drusinski, D. and Har'el, D. (July 1989). Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design*, **8**(7).
[33] Ernst, R. and Henkel, J. (September 1992). Hardware-software codesign of embedded controllers based on hardware extraction. *In Proceeding of the International Workshop on Hardware-Software Codesign*, Boston.
[34] Ku, D. and De Micheli, G. (1992). *High level synthesis of ASICs under timing and synchronization constraints.* Kluwer Academic Publisher.
[35] Gupta, R. K., Coelho Jr. C. N. and De Micheli, G. (June 1992). *Synthesis and simulation of digital systems containing interacting hardware and software components.* In Proceeding of the Design Automaton Conference.
[36] Wilson (1994). Hardware/software selected cycle solution. In *Proceeding of the International Workshop on Hardware-Software Codesign.*

Authors' Biographies

Vincenza Carchiolo is currently associate professor of Computer Science in Institute di Informatica e Telecommunicazioni at University of Catania. Her research interests include distributed system, formal language, embeded system design, CAD methodology. She received a degree with Honors in Electrical Engineering from University of Catania, Italy in 1983. She is member of ACM.

Michele Malgeri is researcher in Institute di Informatica e Telecommunicazioni at University of Catania. His research interests include distributed system, formal language, embeded system design, CAD methodology and networks. He received a degree with Honors in Electrical

Engineering from University of Catania, Italy in 1983.

Giuseppe Mangioni received a degree with Honors in Information Engineering from University of Catania, Italy in 1995. He is currently a Ph.D., candidate of Catania. His research interests include distributed system, formal language and their application in Codesign.