Hardware Software Synthesis of Formal Specifications in Codesign of Embedded Systems

Vincenza Carchiolo
and
Michele Malgeri
and
Giuseppe Mangioni

Istituto di Informatica e Telecomunicazioni - Universita' di Catania
Viale Andrea Doria, 6 - I95125 Catania (Italy)
E-mail {car, mm, gmangion} @ iit.unict.it

The attempt to integrate the design tecnique of hardware and software is the aim of CoDesign. In this work, we present a CoDesign methodology based on a formal approach to embedded system specification. This methodology uses the Templated T-LOTOS language to specify the system during all the design phases. Templated T-LOTOS is a formal language based on CCS and CSP models. Using Templated T-LOTOS a system can be specified by observing the temporal ordering in which the events occur from the outside.

In this paper, we focus on the synthesis of system specified by Templated T-LOTOS. The proposed synthesis algorithm takes advantage of peculiarities of the Templates T-LOTOS. Hardware modules are translated into an Register Transfer Level language managing some signals to drive the synchronization, whilst the software modules are translated into C according to a finite state model whose operations are controlled by a scheduler.

The synthesis of the Templated T-LOTOS specification is based on the direct translation of the language operators, in order to assure that the implemented system is the same as the specified one.

Categories and Subject Descriptors: B.0.0 [**Hardware**]; B.5.2 [**Hardware**]: Design Aids; C.3 [**Computer System Organization**]: Special-Purpose and Application-Based Systems; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages

General Terms: Codesign, Synthesis, Methodologies

Additional Key Words and Phrases: Embedded System, Hardware and Software Synthesis, Codesign

---

1. INTRODUCTION

The availability of more and more complex electronic devices at a low price has considerably boosted the industry of embedded systems. Consequently, the complexity of such systems has increased, and their field of application has spread. An embedded system generally consists of hardware and software components that can be either physically separate or implemented on the same physical device.

Work carried out under the financial support of the Ministero dell'Universita' e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the Project *Design Methodologies and Tools of High Performance Systems for Distributed Applications*.

Hw/Sw CoDesign is an attempt to integrate hardware and software design techniques in a single framework. Its purpose is to give a homogeneous approach to the design of embedded systems aiming at reducing development times and optimizing the hardware/software trade-off. A CoDesign methodology must support the designer during the whole development of the design (that is, from the specification of the requirements to the implementation of the modules that form the system and the corresponding communication interfaces).

Several design methodologies have been proposed in literature [Gupta et al. 1994] [Chou et al. 1995] [Heish et al. 1997]. In general, they agree on the presence of a specification phase, a partitioning phase, and a synthesis phase. One of the fundamental aspects of any CoDesign methodologies is the technique used to define the requirements of the system,because it affects the other phases. A specification technique must allow the designer to specify the system completely and without interpretation errors. The use of a formal language to specify the behaviour of a system has some interesting properties, in particular it allows the correctness of the design process to be validated using mathematical methods.

The methodology proposed use, as specification language, Templated T-LOTOS (TTL) [Carchiolo et al. 1996], a formal technique based on CCS [Milner 1980] and CSP [Hoare 1985] algebras. In TTL the system is described through its interactions with the surrounding environment. Thanks to its formal basis, TTL permits the requirements of the system to be described very precisely, and above all makes it possible to verify that the system has some key properties (absence of deadlock, liveness property, ...). Besides, in the description of the system, TTL permits a structured approach to be used allowing the subdivision of a specification into modules.

After describing the system and verifying its correctness, we need to implement it, so that it respects the requirements described through the specification. This operation implies the choice of the system architecture and the division of the system into modules to be allocated either to hardware or to software. In order to complete the design of the system, we also need to implement the interfaces that permit the exchange of information between modules and definition of the rules governing concurrency between modules.

The synthesis technique presented in detail in this paper takes advantage of TTL. The purpose of the synthesis algorithm proposed is to assure that the implemented system is the same as the specified one, through a direct translation of the language operators (syntax-direct translation approach). This assures that the device obtained has the same behaviour as the specified one. The problem of this approach is connected to the need to define, for each operator in the language, a translation rule for both hardware and software. This makes the development of the translation tool more complex. In the approach proposed in this paper, this difficulty has been dealt with by taking advantage of the fact that all TTL operators can be expressed by a limited number of so-called basic operators. However, the derived operators have been made translated directly in all the cases in which it is convenient for effectiveness reasons.

In our approach, a TTL module can be directly translated into RT level language or into C (in the cases of hardware and software respectively), with no need to pass through intermediate formalisms. Hardware and software modules are translated

according to a finite state model whose execution is controlled by a scheduler. The scheduler is implemented in software, except for the hardware module initialization part. One of the main problems in translating a TTL specification is to respect its synchronization semantics (rendez-vous). When synchronization takes place between hardware modules, it is obtained through appropriate signals; when it takes place between software and hardware modules it is obtained through an interface and the scheduler; lastly, where only software modules are involved it is solved by the scheduler.

This paper deals with a synthesis approach used in the CoDesign methodology developed by the authors [Carchiolo et al. 1998a].

Section 2 presents a short overview of the approaches to CoDesign which can be found in literature, in particular regarding the specification and synthesis phases. Section 3 introduces the TTL language chosen to specify the system being developed, and points out the peculiarities which make TTL interesting in CoDesign. Section 4 summarizes all the phases of the CoDesign methodology which includes the proposed synthesis approach. Section 5 discusses some problems associated with the correct synthesis of a TTL specification. Section 6 presents the synthesis approach, the algorithms for translation into C and RTL and the characteristics of the scheduler.

2. RELATED WORK

The CoDesign methodologies proposed in literature divide the design process into the following sub-problems:

(1) specification;
(2) verification and/or simulation;
(3) mapping on the target architecture.

Several models and languages have been used for the specification phase.

One of most common models used is the Finite State Machine (FSM) [Kohavi 1978]. It models a system through an input/output function that is evaluated by a finite automaton. Starting from basic FSMs, several extensions have been proposed. Extended FSMs (EFSM) [Holtzmann 1991], for example, introduce the concept of non-destructive communication; that is, the written information can be read by the receiver several times. In Behavioural FSMs (BFSM) [Takach and Wolf 1995] inputs and outputs are partially sorted according to time, so time constraints can be expressed. CoDesign FSMs (CFSM) [Chiodo et al. 1993] differ from FSMs because there is an unbounded non-zero quantity of time between the input event and the emission of the output event. The transformation of a CFSM into an FSM implies a choice for the set of unbounded delay values. Another model used for the specification of systems is the Control Data/Flow Graph (CDFG). The model consists of nodes and arcs; nodes indicate operations, while arcs indicate relations of dependence between the nodes. Several CoDesign methodologies are based on the CDFG [Gupta et al. 1994]. The models used to describe systems also include those based on process networks, such as the networks of SDL processes [Saracco et al. 1989], and the networks of Communicating Sequential Processes (CSP) [Hoare 1985].

In the field of CoDesign several formal languages have been used, most of which are based on the FSM model. The one that has recently obtained most attention is Esterel [Berry et al. 1991], which belongs to the group of synchronous languages (which also includes Lustre [Caspi et al. 1987] and Signal [Guernic et al. 1985]). The hypothesis of perfect synchrony, on which the language is based, implies that the system reacts to its environment quickly enough to be considered instantaneous. This means that computation and internal communication take no time. Thanks to this hypothesis, an Esterel description can be transformed into a single FSM. The advantage is that the behaviour of the system becomes highly predictable, since there is no problem of either synchronization or interleaving of concurrent processes. One of the main problems of this approach is that the resulting FSM can have a high number of states. This becomes a problem when the specification is big and makes great use of concurrency. Esterel is currently used as a specification language in CoDesign methodology developed at Berkeley University (POLIS) [Heish et al. 1997]. Besides, many studies have been carried out in order to find an effective hardware implementation of an Esterel specification [Berry and Touati 1993].

StateChart is a graphic specification language based on FSM that permits (among other things) the hierarchical decomposition of the specification, the specification of time constraints and concurrency [Drusinski and Har'el 1989]. Finally, several high-level textual languages have been used in CoDesign, e.g. C^x [Ernst and J. 1993a], Hardware-C [Ku and Micheli 1990], Verilog [Sternheim et al. 1993], and Promela [Wenban et al. 1993].

The second step consists of the validation of the specification; simulation is still the most widely used approach. Many tecniques have been proposed in literature; they differ in their method of coupling hardware and software components. For example, in [Gupta et al. 1992] 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 [Wilson 1994].

The use of models or formal languages permits a better approach to the validation, since it is possible to use the mathematical basis of the language to carry out more complete verifications. The tools available for formal verifications can be divided into two categories: theorem proving-based [Boyer et al. 1995] [Gordon and editors Melham 1992] and finite automata-based tools [Thomas 1990].

The third problem is the mapping on the target architecture. It consists of partitioning the specification into hardware and software parts, and then synthesing them. Several partitioning techniques have been proposed in literature, for example, see [Catania et al. 1997] [Vahid 1997].

The synthesis is generally carried out starting from a graph representation of the specification (which can be handled much more easily), and from a possible allocation of the various components into hardware or software (coming from the partitioning phase).

The synthesis of the hardware parts usually takes place according to the classic techniques of logical synthesis (see [Micheli 1994]).

Conversely, in embedded systems the synthesis of software parts highlights new problems. A scheduler to manage software parts is nearly always required, due to the need to sequentialize a set of tasks that are generally concurrent in the specification (however, there are some exceptions, as in Esterel [Berry et al. 1991]).

The scheduler in embedded systems must respond to criteria of great simplicity and effectiveness, considering the small dimensions of the system. In this area, much of the knowledge acquired in the field of operating systems, especially Real-Time Operating Systems (RTOS), has been applied. For an overview of scheduling methods see [Halang and Stoyenko 1991].

As we said above, in [Berry et al. 1991] the authors propose an approach that takes a single FSM that solves the problem of communication and concurrency between the modules starting from the specification in Esterel of the system as a set of concurrent modules, which do not require the presence of a scheduler.

The other approaches followed in CoDesign tend to subdivide the system into a set of concurrent tasks, and require the implementation of a scheduler. In this sense, it is possible to use classicial scheduling algorithms [Shin and Choi 1997] or to develop ad hoc algorithms. This approach is followed, for example, in [Chou et al. 1994], where an algorithm for a feasible scheduling (respecting timing-constraints) is developed starting from a specification in Verilog.

In [Gupta et al. 1994] and [Gupta and Micheli 1994], starting from a specification in HardwareC, a CDFG is derived; several threads are extracted from it and a scheduling algorithm is proposed.

In [Chiodo et al. 1995] a synthesis methodology is proposed which starts from a CFSM specification of the system. Using this specification model it is possible to obtain an effective hardware implementation. In this approach software synthesis takes place by using an acyclic CDFG obtained from the CFSM specification. This software implementation requires the presence of a scheduler, even if it is quite simple.

3. TTL

Templated T-LOTOS (TTL) is derived from T-LOTOS, which is a timed extension of standard LOTOS (**L**anguage **O**f **T**emporal **O**rdering **S**pecification). LOTOS is a Formal Description Technique (FDT) standardized by ISO (International Standards Organization) between 1981 and 1988 ([ISO-IS-8807 1988]). LOTOS was specifically developed for Open Systems Interconnection, but is applicable to the description of any system, especially concurrent and/or distributed ones (see also [Bolognesi and Brinksma 1987] [Logrippo et al. 1990]).

The main features of TTL are:

—*formal basis*: it allows us to check that the specification possesses useful properties, like deadlock freedom and liveness, by the use of a mathematical approach.

—*concurrency*: this feature makes it possible to model systems made up of various parts which evolve in parallel, a situation typical of hardware systems.

—*modularity*: this allows time to be saved in the specification phase and leads to more efficient design thanks to the reuse of already developed, and thus carefully tested and optimized, components.

—*high degree of abstraction*: it allows us to concentrate on what is to be done without being affected by problems regarding actual implementation. This guarantees the language is suitable for describing both hardware and software regardless of the target architecture.

TTL has been developed in such a way as to use all the existing tools for T-LOTOS (e.g. LOLA [Quemada et al. 1989]) with few restrictions.

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 [Milner 1980] and CSP [Hoare 1985] models; the second is the description of data structures and expressions, and is based on ACT ONE [Ehrig 1985], a language for the description of Abstract Data Types (ADTs). In the following we will only discuss the behavioural part of the TTL. The data structure part of the TTL is the same of LOTOS one; for a complete description see [ISO-IS-8807 1988] or [Bolognesi and Brinksma 1987].

The basic hypothesis of TTL is that the behaviour of the system can be specified by observing from the outside the temporal order in which events occur. In practice, the system is seen as a *black box* which interacts with the environment by means of events, the sequence of which is described by TTL behavioural expressions.

In TTL a system is described in terms of processes; the system as a whole is represented as a process, but it may consist of a hierarchy of processes (often called subprocesses) which interact with each other and the environment. The atomic forms of interaction with the outside world take the name of events. The syntax of a process in TTL is:

```
process  <process-identifier> <parameter-list> :=
        <behaviour-expression>
endproc
where:
     <process-identifier>    is the name to be assigned to the process;
     <parameter-list>        is the list of events with which the process can
                             interact with the environment;
     <behaviour-expression>  are the TTL expressions which define the behaviour
                             of the process
```

The recursive occurrence of a process-identifier in a behavioural expression makes it possible to define infinite behaviour (both self and mutual recursion are possible). A special process which models a completely inactive process, i.e. one which cannot execute any event, is referred to as **stop** .

3.1 Basic operators

3.1.1 *Action Prefix.* This operator produces a new behavioural expression from an existing one, prefixing it with the name of an event. If **B** is a behavioural expression and **a** is the name of an event, the expression **a;B** indicates that the process containing it first takes part in the event **a** and then behaves as indicated by the expression **B**. The possible events include one in particular which is indicated as **i** and represents an internal action, i.e. an action which can occur without interaction with the environment.

The introduction of types makes it possible to describe structured events. They consist of a label (gate name) which identifies the point of interaction (gate), and a finite list of attributes. Two types of attributes are possible: value declaration and variable declaration.

—A value declaration consists of a TTL data item preceded by an exclamation mark. The expression **g!E;B**, for example, means that the process offers the value **E** through the gate **g** and then behaves as indicated in **B**;

—A variable declaration is of the type **?x:t**, where **x** is the name of the variable and **t** is its sort. The expression **g?x:int;B**, for instance, means that the process accepts a value of sort **int** through gate **g**, stores it in **x** and then behaves as **B**.

TTL also allows us to describe timed events, by associating a time attribute to the gate name, that is, a time interval in which it can take place. In the same specification we can have both timed and non-timed events. The time attributes are quite general, such as to permit the modelling of a wide range of situations, including those typical of control-dominated embedded systems. It is, for instance, possible to model fixed delays, min/max constraints and periodic events [Gupta et al. 1994].

3.1.2 *Choice.* If **B1** and **B2** are behavioural expressions, then **B1** [] **B2** denotes a process which can behave both as **B1** and **B2**. Choice between them is made by the environment: if it offers an event which is the initial event of **B1**, then the former is selected; if, on the other hand, it offers an event of **B2**, the latter will be selected.

Choice, action prefix and **stop** are often called *basic operators* because they can specify any behaviour. All other operators, therefore, can be expressed in terms of the basic ones, so all the operators which are described in the following can be viewed as *derived operators*.

3.2 Derived Operators

The *arbitrary interleaving* operator represents the independent composition of two processes, **P1** and **P2** and is indicated as **P1** ||| **P2**. If the two processes have some event in common, **P1** ||| **P2** indicates their capacity to synchronize with the environment but not with each other.

The *parallel* operator is indicated as **P1** || **P2** and it means that the two processes have to synchronize with each other in all events. **P1** || **P2** can take part in an event if and only if both **P1** and **P2** can participate.

The *general parallel composition* is a general way of expressing the parallel evolution of several processes which synchronize on a given set of events. It is denoted with the expression **P1** $| [a1, ..., an] |$ **P2**.

The *hiding* operator internalizes actions. If **B** is a behaviourial expression and $a_1, ..., a_n$ are events, then the expression **hide** $a_1, ..., a_n$ in **B** represents an expression which behaves like **B**, but the events $a_1, ..., a_n$ have been made internal, i.e. they have become unobservable and occur spontaneously, without the participation of the environment.

Sequential composition of two processes, **P1** and **P2**, is indicated as **P1** >> **P2** and it means that when the execution of **P1** terminates successfully (when, for example, no deadlock situations have occurred), **P2** is executed (">>" is also known as an enabling operator). To mark successful termination, there is a special TTL process called **exit** . When **exit** is reached by the first process, control passes to the second.

The *disruption* operator was introduced to facilitate the modelling of situations such as a sudden fall of a connection or the occurrence of an error (in general, all situations in which a given action *disrupts* the normal execution of operations). Given two processes, **P1** and **P2**, **P1** [> **P2** defines a process which normally

executes **P1**, but which can be interrupted at any time by execution of any initial action of **P2**. After the occurrence of such an event, control passes to **P2**.

Every TTL behavioural expression can be preceeded by a boolean condition, a *guarding* operator, which determines whether the expression is to be executed or not.

Table I lists all the operators in the language. Several of them have already been illustrated; for those not analyzed the reader is referred to [Carchiolo et al. 1996].

3.3 Modules and templates

TTL, thanks to modules and templates, allows the designer to create and use libraries of components. This feature is very useful because it permits the designer to model already existing components and also to reduce the development time by using parameterized blocks which can be present in libraries.

The TTL modules are a collection of processes which can be used at any time. A TTL module comprises a declarative part and a definition part. The first part exports all the information needed for the designer to use the process; this part permits us to perform a complete static analysis of the specification. The second part represents the implementation of the processes declared in the previous part. Both a public and a private section can be defined for each module: the former will be exported and used whereas the latter is used for internal matters.

The TTL templates are a further extension of process concepts. Templates allow us to parameterize the name of processes and the type of gates (in the gate list). Thus the templates are concretized into processes as required. The use of the modules and of the template is discussed in [Carchiolo et al. 1996]

4. CODESIGN PATH

The CoDesign methodology, in which we find the synthesis approach presented in this paper, has been developed for the design of control-dominated embedded systems (that is, systems in which the control part is predominant with respect to the data-flow part).

In this CoDesign methodology we can outline 4 fundamental phases, each consisting of different steps (see Fig. 1):

(1) specification;
(2) refinement and decomposition;
(3) partitioning;
(4) implementation;

Below we will briefly discuss the phases of the methodology.

4.1 Specification

The first phase of the methodology is specification of the system, which, as we said above, is carried out by using TTL. The purpose of this phase is to express the requirements of the system given by the client in (plain) TTL, to verify their correctness, and to give a quick prototype to be accepted by the client. In fact, a TTL specification can easily be simulated using a TTL interpreter.

During the whole specification phase we need to verify that the behaviour described corresponds to what we want to specify (this operation is known as model

<table>
<tr><th>Name</th><th>Syntax</th></tr>
<tr><td>inaction</td><td>stop</td></tr>
<tr><td>termination</td><td>exit
exit(E1,...,En)</td></tr>
<tr><td>choice</td><td>B1 [] B2</td></tr>
<tr><td>action-prefix</td><td>g;B
i;B
$g_{d1...dn}$ [SP];B where d_i is of the form ?x:t or !E</td></tr>
<tr><td>parallel-composition</td><td>B1 |[g1,...,gn]| B2
B1 ||| B2
B1 || B2</td></tr>
<tr><td>hiding</td><td>hide $g_1, ..., g_n$ in B</td></tr>
<tr><td>instantiation</td><td>p[$g_1, ..., g_n$] (E1,...,En)</td></tr>
<tr><td>guarding</td><td>[GP] $\rightarrow$ B</td></tr>
<tr><td>disabling</td><td>B1 [> B2</td></tr>
<tr><td>enabling</td><td>B1 >> B2
B1 >> accept x:t1,...,x:tn in B2</td></tr>
<tr><td>local-definition</td><td>let x:t1=E1, ..., x:tn=En in B
let x:t=E</td></tr>
<tr><td>sum-expression</td><td>choice g in [g1,...,gn] [] B
choice x:t [] B</td></tr>
<tr><td>par-expression</td><td>par g in [g1,...,gn] |[a1,...,an]| B
par g in [g1,...,gn] ||| B
par g in [g1,...,gn] || B</td></tr>
<tr><td>loop-expression</td><td>loop(guard; value-expression; B1)</td></tr>
</table>

Table I. TTL Operators

validation). The model that we used for specification allows us to deal with the problem differently from traditional simulation methods.

In fact, verification through simulation is carried out by using test patterns as inputs of the system, and by verifying that the outputs are the expected ones. This method, however, only allows partial verification of the system's behaviour. The result is that correctness is assured only for the verified test patterns.

The use of a formal model as a specification technique gives the opportunity to exploit the mathematical base to carry out more complete verifications. For example, we can verify the so-called *safety properties*, that is, verify that the system, whatever the inputs might be, never ends up in undesired states. We can also verify the *liveness properties*, that is, to verify that a given desired configuration is adopted by the system. Besides, the combination of these two properties allows us to verify even very complex situations.

The tools for formal verifications can be divided into two categories: theorem proving-based tools and finite automata-based tools. The former are tools that assist the designer during mathematical demonstrations (generally semi-automatically), meaning that they assure the correct use of the basic theorems of the model (see [Boyer et al. 1995]). The second category of tools, which exploit the theory of automatons, permit the verification process to be automatized,

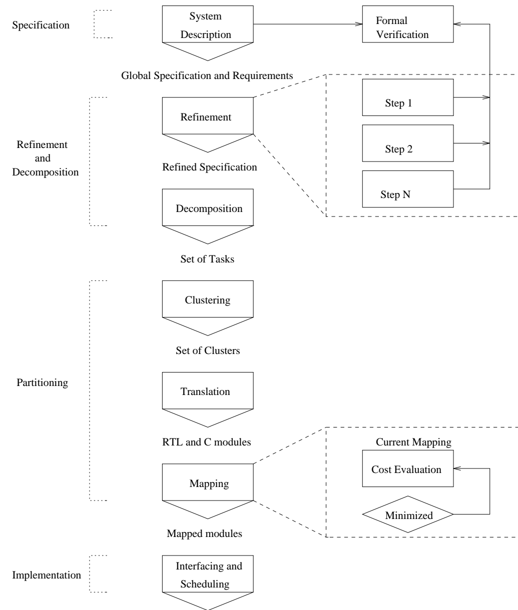

Fig. 1. Overview of Methodology

even if this approach is not often feasible due to problem related to state explotion (see [Kurshan 1994]).

4.2 Refinement and decomposition

This phase consists of three steps:

(1) refinement;
(2) translation into an intermediate format;
(3) decomposition of the system.

The purpose of the first step is to specificy the requirements in a form that is easily and effectively implementable. During this step, several styles of specification are usually adopted. In [Brinksma et al. 1987] and [Vissers et al. 1988] the problem of the style of specification (resource-oriented, state-oriented, constrain-oriented and monolithic) to be adopted in the different phases of the design is discussed; [van Eijk 1989] discusses the problem of transformation from one style of specification to another.

At each refinement step, some functional blocks are divided into simpler blocks, without changing the behaviour of the system. The final purpose is to obtain a specification which is detailed enough to be effectively implemented, but correctly describes the requirements of the system.

The equivalence between what has been specified at one level of refinement and the specification at the next level usually takes place through simulation. The use of languages with a formal base, like TTL, allows us to use tools assisting the designer during whole the refinement phase, and giving the mathematical certainty that the descriptions at the different levels are equivalent.

In the second step of this phase the specification is translated into an intermediate format called the Intermediate Graph Model (IGM). This representation is used during the decomposition and synthesis phases. The purpose of the IGM is to have an easily manageable representation for the translation into both hardware and software.

Each process that forms the specification is translated into an IGM. Each node of an IGM represents an operator of the TTL language, directly synthesizable, or a reference to a process. Fig. 2 (a) contains an example of a TTL specification, whereas Fig. 2 (b) represents the behavioural graph that is associated to it. Shaded nodes do not represent an operator, but are used to facilitate interpretation of the graph. The node labelled 0 in the behavioural graph indicates the point from which to start.

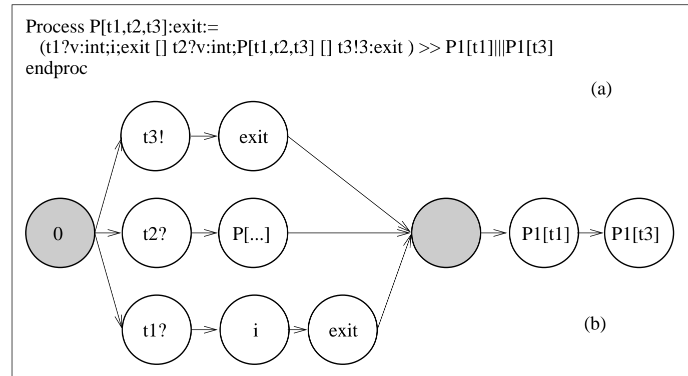

Fig. 2. Example of IGM

In the last step, called decomposition, the specification is divided into a set of elements called *tasks*, according to their parallelism. In this step no hypothesis about the target architecture is necessary, nor do we need to add constraints on the mapping of tasks. Thus we do not need to reduce the level of autonomy in the partitioning phase.

The decomposition algorithm operates as follows:

(1) Classification of the IGM into two sets:
 —**PARA**: indicates an instance relating to a process consisting of only the parallel composition of several processes;
 —**NOPARA**: indicates all the other processes.

(2) Construction of the instance tree (IT). It represents the behaviour of the system as a whole. Each node of the instance tree represents an instance of a formal process, and the nodes are connected according to the order of instantiation. In conclusion, the instance tree is made by composing the IGMs (already classified into **PARA** or **NOPARA**) of each formal process in the specification, after replacing the formal parameters with the actual ones (if necessary).

(3) Decomposition in task. The tasks are all the processes that are classified as **NOPARA** in the IT.

At the end of the decomposition algorithm, the initial specification is divided into a series of tasks. Fig. 3 (a) shows a TTL specification, whereas in Fig. 3 (b) the relative IT with the labelled nodes is shown. In Fig. 3 (b) the hatched nodes are the tasks resulting from the decomposition process.

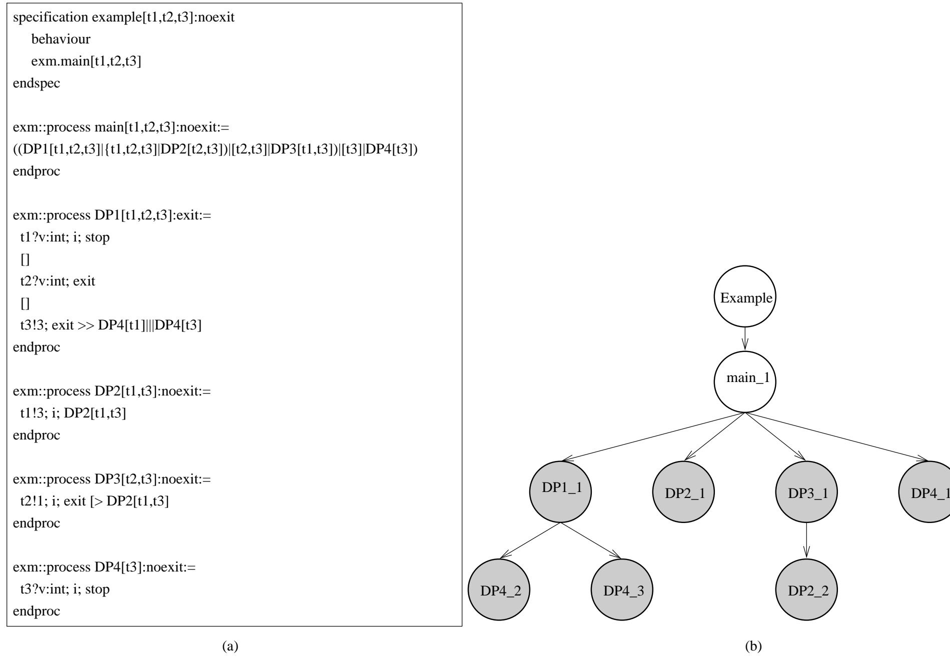

Fig. 3. Example of IT

4.3 Partitioning

The purpose of the partitioning phase is to choose which of the tasks resulting from the decomposition step must be allocated to either hardware or software. The partitioning phase consists of 2 steps: *clustering* (which is not connected with the architecture), and *mapping*.

Clustering reduces the number of tasks below a fixed threshold, in order to reduce the complexity and therefore the cost (in computational terms) of the next phase. The purpose of the clustering algorithm used in our methodology is to minimize the *degree of coupling* between two tasks; this parameter is defined as the *number of interactions between the tasks* [Carchiolo et al. 1998a].

The degree of coupling is a critical factor for the implementation of the final device, mainly because the higher its value is, the higher the cost of the communication (and therefore of the interfaces) among the modules will be. The degree of coupling seems to be an effective heuristic method for reducing the complexity of the problem. In fact, tasks characterized by a considerable amount of interaction will be grouped in the same cluster, and will therefore be mapped in the same

partition. The number of clusters generated by the clustering step is of great importance for the next step, i.e. mapping. In fact, if the number of clusters is too high the problem of mapping is too complex, whilst if the number of clusters is too small the mapping algorithm has few chances to obtain a good hardware/software trade-off [Carchiolo et al. 1998a].

The step called *mapping* consists of choosing the best allocation for the clusters, according to a number of factors, the most important of which are:

—*Performance*. This parameter affects the whole design of the system. According to this principle, a cluster must be allocated in hardware, in order to obtain an improvement in performance.

—*Implementation cost*. The choice of a good allocation for the modules has a considerable impact on production costs, since the difference between the hardware and software realizations is considerable. If some hardware resources can be shared, this factor must be taken into account.

—*Modifiability*. This parameter (which is difficult to quantify) favours the software realization, because software can be more easily modified.

—*Communication*. We need to consider the additional cost due to the exchange of data among the blocks allocated in different partitions. In some cases, this cost can be considerable.

The problem of mapping is to decide whether to implement a module in hardware or in software according to the evaluation of a cost function $\Im$ that takes the above mentioned parameters into consideration. A practical approach is to allocate all blocks in hardware and then to move to software those tasks whose $\Im$ remains within a given value (usually called hardware-oriented approach, see for example [Gupta and Micheli 1993]). Of course, we can also do the opposite, as in [Ernst and J. 1993b].

In order to evaluate $\Im$ correctly, we need to determine the cost of each task. Some of the approaches proposed in literature are based on cosimulation [Wilson 1994] [Rowson 1994], some are based on soft-computing techniques [Catania et al. 1995], and others on the knowledge of computed values of $\Im$, with reference to libraries of components [Axelsonn 1996]. Another approach simultaneously develops the scheduler and evaluates the impact of the allocation (hardware or software) on $\Im$ [Kuman and Alii 1993].

In this paper we do not expressly deal with the problem of mapping. However, since we can synthesize the tasks (both in hardware and in software) at a low computational cost, we can evaluate some of the basic parameters for calculating $\Im$, and therefore apply some of the techniques that we can find in literature.

4.4 Implementation

The purpose of the last phase of CoDesign is the synthesis of hardware and software modules, the interface between these modules and the scheduling algorithm.

In this phase, the target architecture is selected. It usually consists of a microprocessor and several hardware components (FPGAs, for example). One or more modules building the system will be mapped on to these elements.

The implementation phase is discussed thoroughly in the following sections.

5. IMPLEMENTATION OF THE SPECIFICATION

The main purpose of the synthesis algorithm presented in the following paragraphs is to obtain a device having a unique correspondence with the specification processed during the previous phases of the methodology. This allows us to avoid intermediate translations into representations which have no formal base and which therefore might not assure the consistency of the final result with the specification input.

In the case of hw the system is directly synthesized into an RTL level description. The synthesis into software components is more complex due to the limits imposed by the nature of software.

In embedded application the software components must reduce the use of dynamic allocation of memory and the use of stack in order to simplify the architecture of the target device and make the amount of memory needed predictable. The language chosen to synthesize TTL is C, due to its diffusion and, therefore, the availability of tools and compilers for any microprocessor.

The presence of software components together with hardware ones imposes the presence of a scheduler; it has the following tasks:

(1) it must synchronize all the modules of the system;
(2) it serializes the software modules which have to share the (single) microprocessor;
(3) it must be simple and reliable;
(4) it has to use minimal resources (processor, memory, etc.).

The scheduler is subdivided into two parts: the first one manages the activation of all the modules and the serialization of software; the second is in charge of the initialization of hardware components. Since the language is based on the concept of event, the scheduler has also been defined on the same basis. In fact, it does not select the next process to be activated, but the next event that has to take place.

The synthesis technique used to synchronize the modules is imposed by the language chosen for the specification: it implies that an event can take place only when all the processes that want to synchronize are ready to execute it (rendez-vouz). For this reason, the synchronization protocol has been implemented into hardware component. The presence of buffers is not necessary, as the exchange of signals is absolutely synchronous. The main disadvantage of this technique is the presence of a higher number of signals (and therefore of wires) between the modules.

Another characteristic of the synthesis algorithm proposed is the opportunity to synthesize complex behaviour by using only a few translation rules, because the TTL operators can all be expressed by the basic operators. This fact allows us to synthesize the device as a set of distinct components that will only be joined later.

5.1 Restrictions

In this paragraph we present some of the hypotheses on which the synthesis process is based, and in particular which restrictions we have had to impose in the use of TTL.

The synthesis of a TTL process can be carried out both after reducing all the derived operators into basic operators (fine-grained approach) [Henkel et al. 1994],

and by acting on the general form (coarse-grained approach) [Adams and Thomas 1995] directly.

The basic element of TTL is the event, which consists of the interaction between processes based on a rendez-vous mechanism. Three different types of interaction are present in TTL: *value matching, value generation* and *value passing.* The only one which is meaningful for our application is value passing because it has a correspondence with the physical reality of devices, thus it is the only one we have taken into account.

The instantiation of processes in TTL plays a fundamental role in the specification of systems; some situations which are syntactically correct cannot be used due to the static nature of hardware. The main limit imposed on the use of processes lies in the use of recursion and in the form of the gate list. Mutual recursion must be avoided because it is a dynamic structure and therefore it has no correspondence in hardware. Moreover, self-instantiation is only allowed if the gate list is not modified.

In the implementation of the choice operator we have to solve the non-determinism typical of this operator because it cannot be easily implemented in either hardware or software.

We are working on discarding some of the above limitations, such as the avoidance of recursion.

6. TRANSLATION

The translation of the TTL specification into the target languages (C and RTL) is made by operating on the IGM and on the IT created during the refinement and decomposition phase. The way an IGM is synthesized depends of course on whether it is implemented in software or in hardware.

In the case of software, each TTL process is translated into a procedure that implements an FSM obtained from the relative IGM representation. Each procedure executes the typical instructions of a state in atomic mode; this means that their execution cannot be interrupted either by the scheduler or by other modules. This fact allows us to respect the synchronization semantics of TTL.

In the case of hardware synthesis, each IGM state corresponds to a certain number of registers that are activated through an appropriate control sequence. In this case, the atomicity of each state is assured through appropriate signals to synchronize the registers.

Fig. 4 shows the architecture of the synthesized system and the relations among its parts, that is: the scheduler, hardware and software modules, and the interfaces that will be described in detail later.

6.1 Scheduling

The scheduler is the component of the system that manages activation and the synchronization of the various modules, whether they are hardware or software. It is necessary to sequentiale the software modules which are represented in the specification by concurrent processes.

The scheduler consists of a software and a hardware part. The hardware scheduler is a very simple component that has the task of activating hardware modules (during the initialization), according to the information sent by the scheduler software.

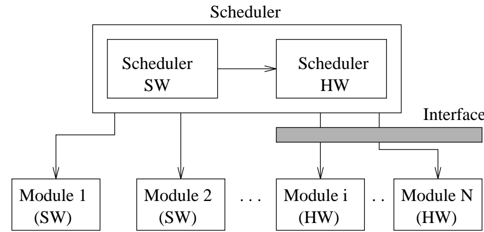

Fig. 4. Scheme of the synthesized system

As we will see below, the synchronization on events, in which only hardware modules participate, is solved by implementing the synchronization protocol directly in hardware. The synchronization on events involving software modules is solved by the scheduler, whose task is, among other things, to implement the complex rendez-vous protocol of the TTL.

6.1.1 *Software scheduler.* The choice of the appropriate scheduling algorithm has been affected by the need for a complete and reliable manager of the device, that must avoid the excessive use of resources (memory and CPU time in particular).

Two possible types of scheduling algorithm can be chosen: polling and interrupt driven. The interrupt-driven technique schedules hardware and software tasks using interrupts: each task generates an interrupt, which is managed by the related Interrupt Service Routine (ISR). This technique introduces some complexity regarding the saving of contexts and the management of priorities. Moreover, it requires additional memory to store the contexts, and, in general, a dedicated circuit to manage several interrupt lines. The techniques based on polling algorithms are characterized by less implementation and management complexity, but by a response time which is higher (on average) than the technique based on interrupt.

The scheduling algorithm that we use belongs to the second group of techniques, but it is different from classical polling algorithms, because scanning is carried out on events rather than on tasks. It is implemented by an infinite loop, in which synchronization on the various events is checked. When a synchronization on an event is found out, the processes involved are activated. In order to explain the operation of the scheduler better, we need to introduce a logical model of software modules. In Fig. 5 we represent the state diagram of a software module, and the possible transitions from one state to the others.

The most significant transitions of Fig. 5 are described below. Transition 1 takes place when a module is ready to synchronize on a given event. Transitions 2 and 3 are caused by the scheduler. Transition 2 takes place when the module returns control to the scheduler. Transition 3 takes place when the scheduler sends the module back to RUNNING. Finally, transition 4 takes place when the synchronization on a given event has finished, and the module can go back to the READY state.

Hardware and software modules implement synchronization in different ways. When software modules are ready to take part in an event (both transmitting and receiving), they notify their availability to the scheduler. This does not occur for hardware modules, which notify the scheduler of their availability to take part in

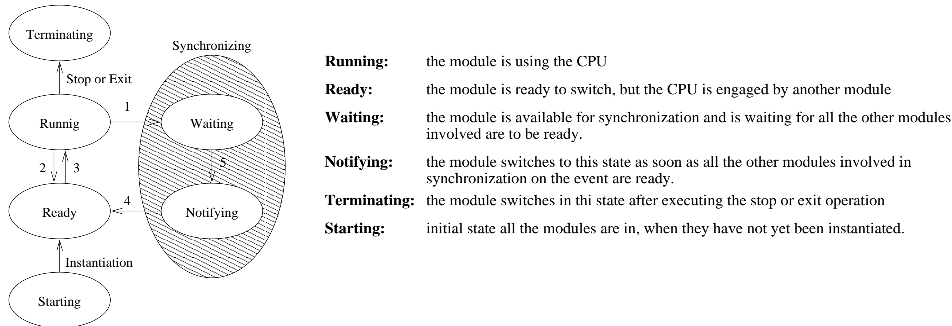

Fig. 5. Diagram of the states of a module

```
FUNCTION Scheduler() {
   LOOP FOREVER {
      FOR EACH Event {
         IF (all Modules which are sensitive to the event are ready to synchronize) {
            IF (the transmitter Module is sw) <CASE1>
            ELSE <CASE2>
         }
      } //Runs all the Modules that are Ready to Run
      FOR EACH Module {
         IF (the Module is (Sw && Ready to Run))
            Run the Module
      }
   }
}
```

Fig. 6. Scheduling algorithm

an event only when they are receiving. As we will see below, this difference is due to the solution adopted for the implementation of synchronization in hardware.

The most general case of synchronization is *one-to-many* (a transmitter and several receivers), where both hardware and software modules are involved. Two cases may occur:

(1) the process offering the event (transmitter) is of the software type;
(2) the process offering the event (transmitter) is of the hardware type.

The two cases of synchronization are managed in a different way. In Fig. 6 the general scheduling algorithm is shown, while the management of the two subcases mentioned before is described below.

The scheduling algorithm is based on two scanning cycles: one for events and the other for modules. The first one verifes if all the modules involved in synchronization on a given event are ready to take part in the event (that is, they are all in the SYNCHRONIZING state). Should this be true (transition 5 in Fig. 5), there are two possible scheduler procedures: CASE1 is called when the transmitter module offering the event is hardware, CASE2 is called when the transmitter module is software.

Procedures CASE1 and CASE2 first manage the exchange of data between the transmitter and the receivers, and then set the modules in the READY state. The scanning cycle of the modules sets all the software modules that are in the READY

state (that is, those that are not already involved in a synchronization) in the RUNNING state.

The CASE1 procedure carries out three basic operations.

(1) The software modules that are synchronizing on the given event are activated. During this activation, an exchange of values between the transmitter and the receivers takes place.
(2) An *ack* is sent to all the hardware receivers involved in the synchronization. The synchronization signals are sent to the hardware modules thanks to an appropriate external circuitry. On the reception of the *ack* signal, the hardware modules proceed autonomously till they synchronize.
(3) All the modules are set in the READY state, because they are ready to take part in another event.

The basic steps of the CASE2 procedure are as follows:

(1) The *ready* signal is sent, through an interface, to the transmitter hardware module. If it is ready to synchronize, it sends back an *ack* signal, which is copied in a register of the interface used to notify the scheduler of the availability to synchronize. Thus the scheduler starts the synchronization management procedure. Conversely, if the hardware module is not ready to synchronize, the management procedure terminates. The availability of the hardware module to synchronize will be re-tested subsequently.
(2) The same as step 1 of the CASE1 procedure.
(3) The same as step 3 of the CASE1 procedure.

The exchange of values between the transmitter and the receivers takes place within the CASE1 and CASE2 management procedures, in a different way according to the type of modules involved. In particular, the receiver modules read the value transmitted as follows.

(1) A hardware receiver module reads the value transmitted:
    —through a register, previously initialized to the value by the transmitter module, if it is software;
    —through a direct connection, if the transmitter is hardware.
(2) A software receiver module reads the value transmitted from a vector in memory (if the transmitter is software), or from an external register (if the transmitter is hardware).

6.2 Module Translation

The following subsection shows the way hardware and software modules are translated. The main problem is preserving the synchronization semantics of TTL in the target modules. The software translation manages the synchronization thanks to the scheduler whilst the hardware translation uses signals to synchronize the modules.

6.2.1 *Synchronization.* A generic module, both software and hardware, is synchronized (with the other modules) on an event through an operation carried out in two steps: in the first step it notifies the scheduler that it is ready to take part

in the event (WAITING state in Fig. 5), while in the second step it is actually synchronized with other modules (NOTIFYING state in Fig. 5). The last step takes place only when the scheduler finds out that all the modules involved in a synchronization are ready; then this information is notified to the modules involved. The exchange of information among the modules that are synchronizing takes place through a vector, which contains an element for each gate of the system.

Let us assume we have two processes, T and R, which, at a certain time, respectively offer and are able to accept a value v through a gate g. In this case two gates are involved in the synchronization, one of which offers a value (condition expressed by the symbol *"!"*) while the other accepts a value (expressed by *"?"*).

This situation is expressed in TTL as:

$$\begin{array}{l} T := \cdots g!v; \cdots \\ R := \cdots g?v; \cdots \end{array} \tag{1}$$

6.2.2 *Implementation of the synchronization in C.* The translation of the event $g!v$, in the case of software, is shown in Fig. 7.

```
...
SWITCH (STATE[instanceNumber]) {
.
.
.
CASE x:
      notifies the scheduler to be ready for the synchronization;
      VALUE[gateId]=v;
      STATE[instanceNumber]++;
      RETURN; //After the RETURN the module is in the WAIT state
   .
   .
   .
} //end SWITCH
...
```

Fig. 7. Translation of the transmitter into C

Process T notifies the scheduler to be ready to take part in the event and, at the same time, sets the value v to be transmitted to the appropriate entry of the vector VALUE, so that, when the synchronization has taken place, the other processes involved in the synchronization can read this value.

Fig. 8 shows the translation of the event $g?v$. Process R notifies the scheduler of its availability for synchronization and, after updating the state, returns the control to the scheduler. Once the synchronization has taken place (NOTIFYING state), the scheduler activates all the modules involved in the synchronization. All the receiver modules read the value transmitted by the transmitter, and immediately execute the code relating to the next operator. The receiver reads this item and carries out the code relating to the next state atomically. The implementation does not change in the case of N receivers (one-to-many synchronization).

6.2.3 *Implementation of the synchronization in RTL.* In RT level synthesis we have to distinguish between two possible cases: synchronization between two pro-

```
...
SWITCH (STATE[instanceNumber]) {
.
.
.
CASE y:
   notifies the scheduler to be ready for the synchronization;
   STATE[instanceNumebr]++;
   RETURN; //After the RETURN the module is in the WAIT state
CASE y+1:
   v=VALUE[gateId]; //NOTIFY state
   STATE[instanceNumber]++;
   break;
.
.
.
} //end SWITCH
...
```

Fig. 8. Translation of the receiver into C

cesses (one-to-one) and the synchronization of one process with many others (one-to-many).

One-to-One. First we will deal with the problem of one-to-one synchronization with a value exchange, irrespective of the value actually exchanged.

Translation of the complex TTL synchronization into RTL requires the use of several signals to guarantee the semantic correctness of the translation. So each synchronization operation (event) is associated with three signals: one for the exchange of the data itself, and two others to manage the synchronization (a ready and an acknowledgement signal). The need for two signals for synchronization is due to the fact that communication in TTL is a rendez-vous between events.

Let us assume we have the same processes, T and R, shown in the previous subsection. Schematically, translation of the event $g!v$ can be represented as in Fig. 9 (a). Fig. 9 (b) is the scheme of the event $g?v$. The signal in_i (in_j) represents

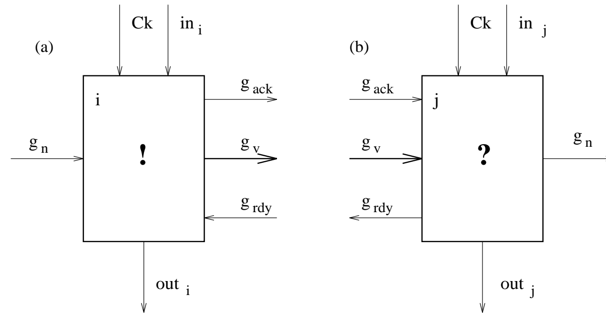

Fig. 9. Scheme of basic interaction events

the signal enabling execution of block i (j) and signal out_i (out_j) the termination of block i (j) (which coincides with the signal enabling execution of the block $i+1$). The signal g_n is needed when a choice operator is involved in the synchronization (as can be seen in subsection 3.1.2). Translation of block i into hardware is represented

in Fig. 10 using the RTL language described in appendix A. As we can see from

...	:	
x	: $if\ (not\ g_{rdy};\ g_{rdy})\ goto(\ x;\ x+1\ )$	waits for the receiver to be ready to synchronize
$x+1$	: $g_{ack}\ = 1$	acknowledges the synchronization
$x+2$	: $if\ (not\ g_n;\ g_n\ )\ goto(\ x;\ x+1\ )$	waits for the synchronization to be correctly concluded by the receiver ($g_n\ = 1$)
$x+3$	: $g_v = v_T$	
...	:	

Fig. 10. Translation of Transmitter

Fig. 10, the transmitter waits for the receiver to be available for synchronization, after which it acknowledges the synchronization and exchanges the value (if any). Fig. 11 represents the translation into RTL of the block j. The behaviour of the

...	:	
y	: $g_{rdy} = 1; if\ (not\ g_{ack};\ g_{ack})\ goto(y;\ y+1)$	warns the transmitter to be ready for synchronization and simultaneously sends an ack signal
$y+1$	: $g_n\ = 1$	informs transmitter that synchronization has actually occurred
$y+2$	: $v_R := g_v$	accepts the value
...	:	

Fig. 11. Translation of Receiver

receiver complements that of the transmitter. In Fig. 10 v_T represents the variable containing the value to be transferred, which in RTL is equivalent to a register. Likewise, in Fig. 11 v_R is the register which, following synchronization, will contain the value exchanged.

According to the translation scheme used, the transmitter is translated in 4 RTL steps, the receiver in 3 steps.

One-to-Many. In one-to-many synchronization a transmitter synchronizes with several receivers, which all have to be available for the event. The RTL coding of the receivers remains unchanged with respect to the one shown in Fig. 11. Coding of the transmitter is similar to that of the previous case (shown in Fig. 10), the only exception being that this time the transmitter has to ascertain, before sending the *ack* signal, that all the receivers are available for synchronization.

Let us assume we have three processes, one transmitter and two receivers; they are the same processes shown above. The RTL coding of the transmitter is given in Fig. 12. The case described in Fig. 12 implements a transmitter which is able to synchronize with two receivers. In the case where more than two receivers are present the condition $(g_{rdy1}\ and\ g_{rdy2})$ (see line labelled x in Fig. 12) is generalized by using the functional block $cond(g_{rdy1}, \ldots, g_{rdyn})$ which in RTL implements the verification condition.

...	:	
x	: $if(\,not(g_{rdy1}\; and\, g_{rdy2}))\,;\,(g_{rdy1}\; and\, g_{rdy2})\; goto(x;\, x+1)$	waits for all the receivers to be ready for synchronization
$x+1$	: $g_{ack}\; = \;1$	acknowledges the synchronization
$x+2$	: $if(\,not(g_{n1}\; and\, g_{n2})\,;\, g_{n1}\; and\, g_{n2})\; goto(x;\, x+3)$	waits for all the g_{nx} signals to be equal to 1
$x+3$	: $g_v = v_t$	exchanges the value
...	:	

Fig. 12. Translation of Transmitter

6.2.4 *Implementation of the Choice Operator.* The choice operator allows branches to be introduced into the specification. The branch to be taken is chosen according to the type of event occurring. A process carrying out a several-branch choice offers its availability to take part in all the events indicated by the different branches of the choice. The event on which the actual synchronization will take place will be given according to the availability of the other processes involved. As usual, the job of the synchronization is given to the scheduler, which will also have the task of informing the process that has executed the choice expression of the event on which actual synchronization has taken place, in order to determine the way to be followed.

Implementation in C. An n-way choice is represented in TTL below.

$$
\begin{array}{l}
T := \cdots((g_1!v_1;\cdots)[](g_2!v_2;\cdots)[](\cdots)[](g_m!v_m;\cdots))\cdots \\
R_1 := \cdots g_1?v_1;\cdots \\
R_2 := \cdots g_2?v_2;\cdots \\
\vdots \\
R_m := \cdots g_m?v_m;\cdots
\end{array} \tag{2}
$$

The translation of the choice into C uses a vector called BRANCH, containing as many elements as there are branches in the choice expression. Each element of BRANCH stores the next state from which execution begins when the related event takes place. The translation into C is shown in Fig. 13.

RTL implementation. In explaining the RTL coding algorithm we will assume, for the sake of simplicity, that we are dealing with a choice between two possible events (extension to more general cases is dealt with below). Three different situations can occur, according to the type of event (accepting or offering a value) present in the choice.

Case 1. Let us assume we have three TTL processes of the following kind:

$$
\begin{array}{l}
T := \cdots(g_1!v;\cdots \quad [] \quad g_2!v;\cdots)\cdots \\
R_1 := \cdots g_1?v;\cdots \\
R_2 := \cdots g_2?v;\cdots
\end{array} \tag{3}
$$

That is, a process T which is able to synchronize both on the event $g_1!v$ and on the event $g_2!v$, and two processes which can accept a value, the first one on gate

```
VOID ProcessName(int p) {
   STATIC INT BRANCH[NumEvents];
   WHILE(1) {
      SWITCH(STATE[instanceNumber]) {
      CASE -1:
         //Updates the state according to the value of BRANCH[p](where p is a
         //parameter passed by the scheduler, and which indicates on which gate
         //the synchronization has taken place).
         STATE[instanceNumber]=BRANCH[p];
         break;
      :
      CASE x:
         //Event of the first way of the choice expression;
         Notifies the scheduler to be ready for synchronization on g1;
         VALUE[g1]=v1;
                      :
         BRANCH[g1]=STATO_1;
         //Event of the m-th way of the choice expression;
         Notifies the scheduler to be ready for synchronization on gm;
         VALUE[gm]=vm;
         BRANCH[gm]=STATO_m;
         STATE[instanceNumber]=-1;
         RETURN;
      :
      }
   }
}
```

Fig. 13. Choice Implementation

g_1 and the second on gate g_2. The RTL coding of the two processes, R_1 and R_2, is identical to that seen previously, as from their point of view it makes no difference whether the process with which they synchronize contains a choice or not. Fig. 14 shows the RTL translation of T. A fundamental point in coding the

...	:
x	: $if((not\, g_{1rdy})\, and\, (not\, g_{2rdy});\ g_{1rdy};\ (not\, g_{1rdy})\, and\, g_{2rdy})\ goto(x;\, x+1;\, x+k)$
$x+1$	: $g_{1ack} = 1$
$x+2$	: $if(not\, g_{1n};\, g_{1n})\ goto(x;\, x+3)$
$x+3$	: $g_{1v} = v_T$
...	:
$x+k$	: $g_{2ack} = 1$
$x+k+1$	: $if(not\, g_{2n};\, g_{2n})\ goto(x;\, x+k+2)$
$x+k+2$	: $g_{2v} = v_T$
...	:

Fig. 14. Translation of Process T

choice operator is the instruction in step x in Fig. 14; otherwise, the coding can be considered to be the union of two synchronization operations of the kind seen previously. The conditional jump in step x makes it possible to implement the semantics of the choice operator. If, in fact, neither process R_1 nor process R_2 is ready to synchronize (condition expressed by the fact that both *ready* signals are set to zero) the instruction jumps to position x, restarting execution of the same instruction. If, on the other hand, either of the two *ready* signals goes high, the algorithm executes a *goto* and the instruction starts synchronization with the

gate emitting the *ready* signal. If both *ready* signals go high at the same time (i.e. during the same clock cycle) the semantics of the choice operator allows a non-deterministic choice between the two events it is possible to synchronize with. In our implementation this indeterminism is solved by choosing one of the possible events a priori. In Fig. 14, for instance, if both g_{1rdy} and g_{2rdy} have a value of 1, the synchronization is on g_1. In the more general case of one-to-many synchronization, the conditions have to be verified on the logical *and* of all the *ready* signals, i.e. as explained in the case of one-to-one synchronization, g_{1rdy} is substituted with the combinatorial function $cond(g_{11rdy}; g_{12rdy}; ...)$; this is repeated for g_{2rdy}, g_{1n} and g_{2n}.

Case 2. The second case which may occur is complementary to the first one, i.e. where the types of event are acceptance of a value. Let us assume we have the following TTL processes:

$$\begin{array}{l} R := \cdots (g_1?v; \cdots \quad [] \quad g_2?v; \cdots) \cdots \\ T_1 := \cdots g_1!v; \cdots \\ T_2 := \cdots g_2!v; \cdots \end{array} \tag{4}$$

The RTL translation of processes T_1 and T_2 proceeds as above. Fig. 15 shows the coding of the process R. Here again, the RTL translation can be considered as the

...	:	
x	:	$g_{1rdy} = 1;\ g_{2rdy} = 1;$ $if((not\, g_{1ack})\, and\, (not g_{2ack});\ g_{1ack};\ (not\, g_{1ack})\, and\, g_{2ack})\ goto(x;\ x+1;\ x+k)$
$x+1$	:	$g_{1n} = 1$
$x+2$	:	$v_R := g_{1v}$
...	:	
$x+k$	:	$g_{2n} = 1$
$x+k+1$	:	$v_R := g_{2v}$
...	:	

Fig. 15. Translation of Process R

union of two operations of synchronization with the acceptance of a value. The meaning of the conditional instruction in step x is the same as in Case 1. This time the signals g_{1n} and g_{2n} play a fundamental role. If, after the emission of the signals g_{1rdy} and g_{2rdy} (which signal the availability of process R to participate both in the event $g_1?v$ and in the event $g_2?v$), the processes T_1 and T_2 are ready for synchronization (expressed by the emission of the signals g_{1ack} and g_{2ack}), an indeterminate situation will occur and, as in the previous case, will be solved in favour of the first event. However, if there were no signal g_{1n} to confirm that process T_1 has been chosen for synchronization, both processes, T_1 and T_2, would reach the synchronization instruction. This situation is semantically incorrect for T_2: not having been chosen for synchronization, T_2 has to remain in the ready state for synchronization on the event $g_2!v$.

Case 3. The last case that can occur is a mixture of the previous ones. Let us assume we have the following TTL processes:

$$\begin{array}{l} TR := \cdots (g_1?v; \cdots \quad [] \quad g_2!v; \cdots) \cdots \\ T := \cdots g_1!v; \cdots \\ R := \cdots g_2?v; \cdots \end{array} \tag{5}$$

Here again the RTL translation of T and R presents no difficulties, being the same as previous cases. Fig. 16 gives the RTL coding of TR. If $g_2!v$ is involved in a one-

...	:	
x	:	$g_{1rdy} = 1;$ $if((not\, g_{1ack})\, and\, (not\, g_{2rdy});\, g_{1ack};\, (not\, g_{1ack})\, and\, g_{2rdy})\; goto(x;\, x+1;\, x+k)$
$x+1$	:	$g_{1n} = 1$
$x+2$	:	$v_R := g_{1v}$
...	:	
$x+k$	:	$g_{2ack} = 1$
$x{+}k{+}1$	:	$if(not\, g_{2n};\, g_{2n})\; goto(x;\, x+k+2)$
$x{+}k{+}2$	:	$g_{2v} = v_T$
...	:	

Fig. 16. Translation of Process TR

to-many synchronization, g_{2rdy} has to be substituted with $cond(g_{21rdy};\, g_{22rdy};\, \ldots)$, as does g_{2n}.

In the more general case of a choice among several events, the RTL coding method is exactly the same as explained above. The only difference lies in the fact that more than two processes must synchronize whith each other. Thus the condition in the first conditional jump (corresponding to step x in the previous cases) is more complex, because it takes into account all the signals exchanged among all the processes, but it can easily be computed. To clarify the procedure we will give an example which extends the situation shown in Case 1; the same procedure can be applied to extend the other cases. Let us extend Case 1 to one in which m processes are able to receive a value and one process can synchronize with them, as shown in expression(2).

The RTL code of process T is the same as Fig. 14 but the condition is of the following form:

$$\begin{array}{l} if(\\ \quad \bigwedge_{i=1}^{n}(not\ cond(g_{i1rdy}; g_{i2rdy};\, \ldots; g_{imrdy}); \\ \quad cond(g_{11rdy}; g_{12rdy};\, \ldots; g_{1mrdy}); \\ \quad (not\, cond(g_{11rdy};\, g_{12rdy};\, \ldots; g_{1mrdy}))\, and\, cond(g_{21rdy};\, g_{22rdy};\, \ldots; g_{2mrdy}); \\ \quad \ldots \\ \quad \bigwedge_{i=1}^{n-1}(not\ cond(g_{i1rdy}; g_{i2rdy};\, \ldots; g_{imrdy}))\, and\, cond(g_{n1rdy}; g_{n2rdy};\, \ldots; g_{nmrdy}) \\ ) \end{array}$$

6.2.5 *Synthesis of derived operators.* This subsection presents some examples of the synthesis of derived operators, that is, the TTL operators which can be defined in terms of other TTL operators. The derived operators permit the user to simplify specifications and to gain a clearer understanding of the behaviour of the system. As said above, they are all obtained by simple composition of the basic operators. In

particular in this section we show the synthesis of the enable and parallel operators described in Section 3.

C Inplementation. Let us suppose we have N processes. Their sequential composition is expressed in TTL as $P1 >> P2 >> \ldots PN$, whereas their parallel composition is expressed as $P1 \parallel P2 \parallel \ldots PN$. To implement the enable and parallel operators in software we use the scheduler.

—In the implementation of the enable operator the scheduler is informed by the involved process of its successful termination, then the scheduler deactivates the current process and enables the next one. After this operation the current process is put in the TERMINATION state and the next process in the READY state.

—The parallel operator is managed directly by the scheduler, which provides the necessary mechanism for sequentializing parallel execution.

RTL Implementation. Each process has an input signal, P_{start}, which starts the process and an output signal P_{exit} which is emitted when the process ends successfully. When the signal P_{exit} is never emitted (as for example in a recursive process) we represent it with a floating signal. To represent a process P graphically in the

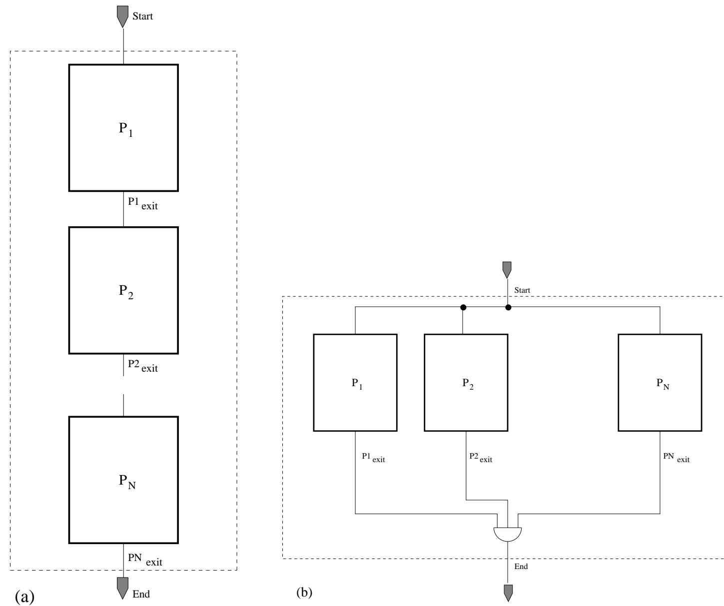

Fig. 17. Scheme of sequential and parallel composition

figures we use a box with a thick border. Fig. 17 (a) shows the result of synthesis in RTL of the behaviour expression $P1 >> P2 >> \ldots PN$ whereas Fig. 17 (b) shows the result of synthesis of $P1 \parallel P2 \parallel \ldots PN$.

For the sake of clarity we provide a simple example of the synthesis of a composition of processes using the previously shown operators. The TTL specification is

as follows:

$$
\begin{array}{llll}
P & := P1 \parallel P2 \parallel P3 & & \\
where & & & \\
 & P1 & := g^1!v_1; P1 \, [] \, g^2!v_2; P1 & \\
 & P2 & := g^1?x : integer; P2 & \\
 & P3 & := P4 >> P5 & \\
 & where & P4 := & (g^2?y : integer; exit) \\
 & & P5 := & g^2?z : integer; P5
\end{array}
$$

Fig. 18 shows the result of the synthesis process. The blocks labelled with a

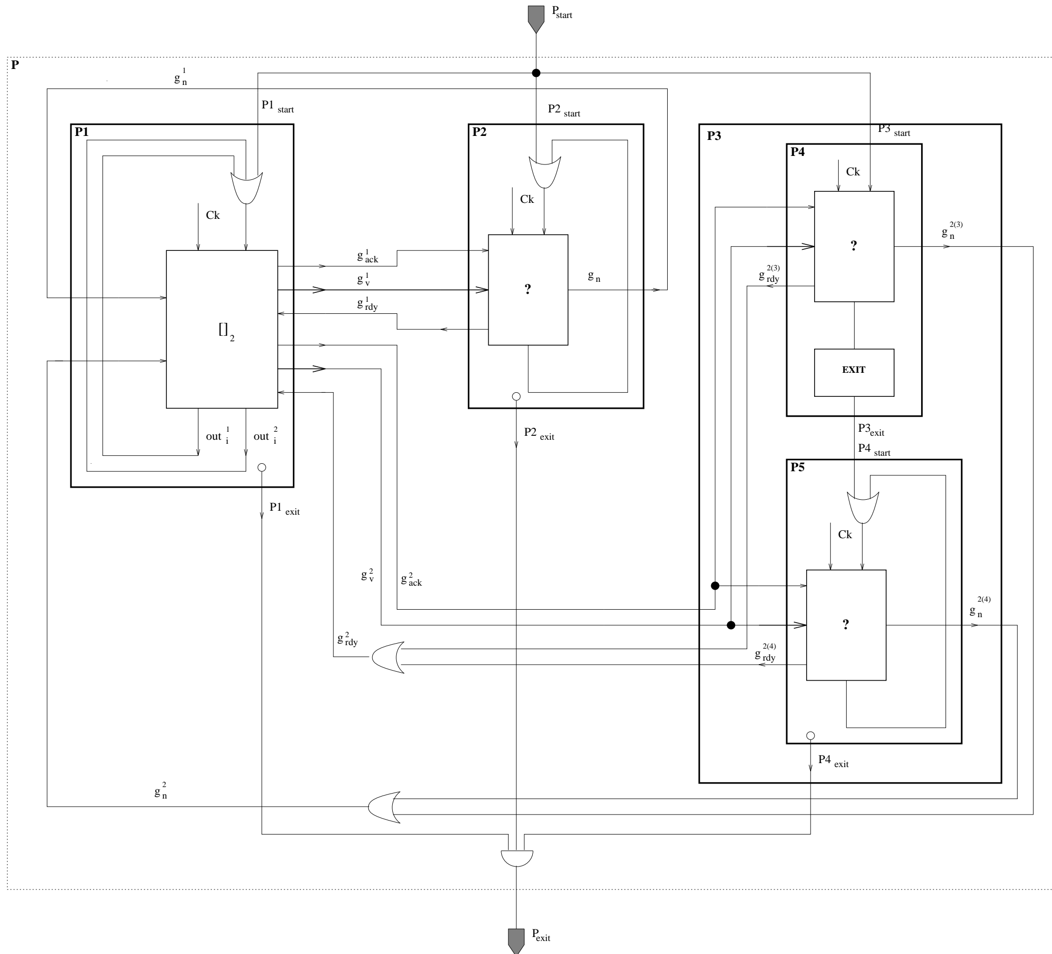

Fig. 18. Scheme of example

question mark are the translation of the basic receiver and the block labelled with the symbol $[]_2$ represents the translation of a two-way choice; lastly the block

labelled EXIT is the implementation of the **exit** operator. The figure also shows the interconnections among the blocks and the input/output signals.

7. CONCLUSIONS

The paper presents all the phases of the methodology, but mainly focus on the synthesis phase.

The proposed design path is deeply presented pointing out the aspects making it interesting in the context of embedded system. The use of a formal technique to specify the system is the key point characteriziong the methodology.

Moreover the kind of formal language has a strong impact on the synthesis phase; the problems related to this problem has been stressed throughout the whole paper and some toy examples are presented to clarify the algorithm used (some other more complex examples can be found in [Carchiolo et al. 1998b]).

Further studies are needed to optimize the synthesis of both hardware and software. A critical parameter in embedded systems is memory usage by software modules. For this reason we are working on minimization of the size of the code generated for the software part. Moreover, some studies are being devoted to optimal synthesis of the hardware part, mainly in the implementation of interfaces between hardware and software. Finally, we are working on the use of some other languages for register level hardware description in order to exploit the large amount of commercial tools available.

ACKNOWLEDGMENTS

The authors are indebted to Prof. Giovanni De Micheli for many helpful discussions. We also thank the anonymous referees and the editor for many helpful comments on the presentation of the paper.

References

Adams, J. and Thomas, D. 1995. Multiple-process behavioural synthesis for mixed Hardware-Software systems. In *Proceedings of International Symposium on System Synthesis* (September 1995). pp. 10–15.

Axelsonn, J. 1996. Hardware/Software partitioning aiming at fulfillment of real-time constraints. *Journal of System Architecture 42*, 6–7 (December), 439–464.

Berry, G., Couronné, P., and Gonthier, G. 1991. The synchronous approach to reactive and real-time systems. *IEEE Proceeding 79*.

Berry, G. and Touati, H. 1993. Optimized controller synthesis using Esterelle. In *Proceedings of the International Workshop on Logic Synthesis* (May 1993).

Bolognesi, T. and Brinksma, E. 1987. Introduction to the ISO specification language LOTOS. *Computer Networks and ISD Systems 14*, 25–59.

Boyer, R. S., Kaufmann, M., and Moore, J. S. 1995. The Boyer-Moore theorem prover and its interactive enhancement. *Computer & Mathematics with Applications*, 27–62.

Brinksma, E., Scollo, G., and Vissers, C. 1987. Experience with and future of LOTOS as a specification language. Technical Report INF-87-17, Department of Computer Science, Twente University.

Carchiolo, V., Malgeri, M., and Mangioni, G. 1996. TTL: A LOTOS extension for system description. In *Proceedings of Basys '96* (Lisboa, Portugal, 1996).

Carchiolo, V., Malgeri, M., and Mangioni, G. 1998a. Formal codesign methodology with multistep partitioning. *VLSI Design Journal 7*, 4.

Carchiolo, V., Malgeri, M., and Mangioni, G. 1998b. Synthesis of TTL Specification: a Case Study. *CESA98 - IMACS MultiConference.*

Caspi, P., Pilaud, D., Halewachs, N., and Plaice, J. 1987. LUSTRE A declarative language for the real-time programming. In *Proceedings of Conference on Principles of Programming Languages* (Munich, 1987).

Catania, V., Malgeri, M., and Russo, M. 1995. A methodology for codesign based on fuzzy logic and genetic algorithms. In *Proceedings of 8th International Conference* (Melbourne, Australia, June 1995).

Catania, V., Malgeri, M., and Russo, M. 1997. Applying fuzzy logic to codesign partitioning. *IEEE Micro 17*, 3 (May-June), 62–70.

Chiodo, M., Giusto, P., Hsieh, H., Jurecska, A., and alii. 1995. Synthesis of Software programs from CFSM specifications. In *Proceedings of the Design Automation Conference* (June 1995).

Chiodo, M., Giusto, P., Jureska, A., Hsieh, H. C., Sangiovanni-Vincentelli, A., and Lavagno, L. 1993. A formal specification model for Hardware/Software codesign. In *Proceeding of International Workshop on Hardware-Software Codesign* (Boston, September 1993).

Chou, P., Ortega, R., and Borriello, G. 1995. The Chinook Hardware/Software cosynthesis system. In *International Symposium on System Synthesis* (Cannes, France, September 1995).

Chou, P., Walkup, E., and Borriello, G. 1994. Scheduling for reactive real-time systems. *IEEE Micro 14*.

Drusinski, D. and Har'el, D. 1989. Using statecharts for Hardware description and synthesis. *IEEE Transaction on Computer Aided Design 8*.

Ehrig, B. M. H. 1985. *Fundamentals of Algebraic Specifications, 1 EATCS Monographs on Computer Science*. 1 EATCS Monographs on Computer Science. Springer-Verlag.

Ernst, R. and J., H. 1993a. Hardware-Software codesign of embedded controllers based on Hardware extraction. In *Proceeding of the International Workshop on Hardware-Software Codesign* (Boston, September 1993).

Ernst, R. and J., H. 1993b. Hardware-Software cosynthesis for microcontrollers. *IEEE Design and Test Computers 10*, 4 (December), 29–41.

Gordon, M. J. C. and editors Melham, T. F. 1992. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press.

Guernic, P. L., Benveniste, A., Bournat, P., and Gauthier, T. 1985. A data flow oriented language for signal processing. Technical Report IRISA report 246, IRISA, Rennes, (France).

Gupta, R. K., Coelho Jr, C. N., and De Micheli, G. 1992. Synthesis and simulation of digital systems containing interacting Hardware and Software components. In *Proceeding of Design Automaton Conference* (June 1992).

Gupta, R. K., Coelho Jr, C. N., and Micheli, G. D. 1994. Program implementation schemes for Hardware-Software systems. *IEEE Computer*.

Gupta, R. K. and Micheli, G. D. 1993. Hardware-Software cosynthesis for digital systems. *IEEE Design and Test Computer*.

Gupta, R. K. and Micheli, G. D. 1994. Constrained Software generation for Hardware-Software systems. In *Proceedings of the International Workshop on Hardware-Software Codesign* (1994).

Halang, W. and Stoyenko, A. 1991. *Constructing predictable real time systems*. Kluwer Academic Publishers.

Heish, H., Lavagno, L., Passerone, C., Sansoe, C., and San Giovanni-Vincentelli, A. 1997. Modeling microcontroller peripherals for high-level co-simulation and syntheis. In *Proceeding of Fifth International Workshop on Hardware/Software Codesign* (Braunschweig, Germany, March 1997).

Henkel, J., Benner, T., Ernst, R., Ye, W., Serafimov, N., and Glawe, G. 1994. COSYMA: A Software-orienetd approach to Hardware-software codesign. *The Journal of Computer and System Architecture 2*, 3, 293–314.

Hoare, C. A. R. 1985. *Communicating Sequential Processes, International Series in Computer Science*. International Series in Computer Science. Prentice-Hall.

Holtzmann, G. 1991. *Design and Validation of Computer Protocols*. Prentice-Hall.

ISO-IS-8807. 1988. *Information Processing Systems, Open System Interconnection, LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. ISO.

Kohavi, Z. 1978. *Switching and Finite Automata Theory*. McGraw-Hill, New York.

Ku, K. and Micheli, G. D. 1990. HardwareC - A language for Hardware design version 2.0. Technical Report No. CSL-TR-90-419 (April), Standford University.

Kuman, S. and Alii. 1993. A framework for Hardware/Software codesign. *Computer 26*, 12 (December), 39–45.

Kurshan, R. P. 1994. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press.

Logrippo, L., Melanchuck, T., and DuWors, R. 1990. The Algebraic Specification Language LOTOS: An Industrial Experience. In M. Moriconi, Ed., *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development (Napa, CA.)* (1990). pp. 59–66.

Micheli, G. D. 1994. *Synthesis and optimization of digital circuits*. McGraw-Hill.

Milner, R. 1980. *A Calculus of communicating systems, LCNS 92*. LCNS 92. Springer Verlag, New York.

Quemada, J., Paven, S., and Fernandez, A. 1989. State exploration by transformation with LOLA. *Workshop on Automatic Verification Methods for Finite State Systems*.

Rowson, J. 1994. Hardware/Software co-simulation. In *Proceedings of Design Automation Conference* (1994). pp. 439–440.

Saracco, R., Smith, J. R. W., and Reed, R. 1989. *Telecommunications systems engineering using SDL*. North-Holland Elsevier.

Shin, Y. and Choi, K. 1997. Enforcing Schedulability of Multi-Task Systems by Hardware-Software Codesign. In *5th International Workshop on Hardware Software Codesign* (Braunschweig, Germany, March 1997).

Sternheim, E., Singh, R., Madhaven, R., and Trivedi, Y. 1993. *Digital Design and Synthesis with Verilog HDL*. Automata Publishing Company.

Takach, A. and Wolf, W. 1995. An automaton model for scheduling constraints in stnchronous machines. *IEEE Transaction on Computer 1*, 44 (January), 1–14.

Thomas, W. 1990. *Automata on infinite objects, handbook of Theoritical Computer Science*. handbook of Theoritical Computer Science. Elsevier.

Vahid, F. 1997. Modifying min-cut for Hardware and Software functional partitioning. In *5th International Workshop on Hardware Software Codesign* (Braunschweig, Germany, March 1997).

van Eijk, P. 1989. Tools for LOTOS specification style transformation. Technical Report Memoranda 89-35 (June), Twente University.

Vissers, C. A., Scollo, G., and van Sinderen, M. 1988. Architecture and specification style in formal description of distributed systems. In *Proceedings of Conference of Protocol Specification, testing and Verification* (Amsterdam, 1988). North-Holland, pp. 189–204.

Wenban, A. S., O'Leary, J. W., and Brown, G. M. 1993. Codesign of Communication Protocols. *IEEE Computers*, 46–52.

Wilson, J. 1994. Hardware/Software selected cycle solution. In *Proceeding of International Workshop on Hardware-Software Codesign* (1994).

APPENDIX

A. THE TARGET RTL LANGUAGE

The RTL language used throughout this paper is a language which can define the structure of a generic digital system. Any digital system is modelled by using a functional block which receives information from the external environment by using signals and processes them producing output signals (the response to the environment). Each functional block is implemented by the *control unit* and the *processing unit*. The first unit provides the signal to synchronize the operations performed by the second. The full system is based on a single *clock* which provides

the synchronization. The basic hypothesis is that the circuit must be stable before the clock cycle finishes. Fig. 19 represents the logical scheme of a generic digital system.

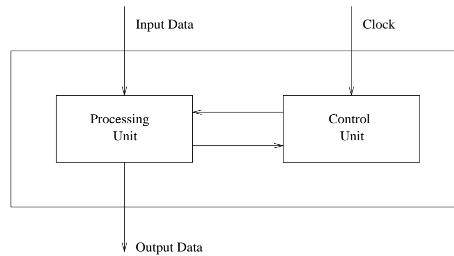

Fig. 19. Logical scheme of a Digital System

The RTL module is defined by the following sections:

—components: it contains the declaration of the components which make up the processing unit.

—control sequence: it defines the internal command sequence which must be emitted by the control unit.

—permanent assignment: it defines an operation which must be repeated every clock cycle.

The control sequence is made up of steps; each one is numbered and must be executed in a single clock unit. Each step is made up of one or more commands which are executed in parallel. All the commands belonging to a step are separated by *;*. Therefore the control sequence has the following form:

```
i:  op1; op2; op3
j:  op4; op5
```

where i and j are the generic step i and step j and op_i are the commands.

The main constructs of the language are the assignment and the conditional. The first represents the transfer of a value between two registers. The right hand side of the operation can contain any Boolean operation. The two operators are represented as follows:

```
i:  targetRegister := sourceRegister
j:  targetRegister := sourceRegister_1 and sourceRegister_2 or ...

k:  if( c1; c2 ) then (op1; op2)
h:  if( c3; c4 ) goto (n; m)
```

To describe a direct connection between elements, the language allows us to describe the assignment of a value to a line; in this case the assignment is only valid for one clock cycle. It is described by the operator "=" and it is also used to describe the assignment to output lines.