



# Hardware/Software Synthesis of Formal Specifications in Codesign of Embedded Systems

VINCENZA CARCHIOLO, MICHELE MALGERI, and GIUSEPPE MANGIONI  
Università di Catania

---

CoDesign aims to integrate the design techniques of hardware and software. 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 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 Templates T-LOTOS. Hardware modules are translated into a register transfer-level language that manages some signals in order to drive synchronization, while 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 to ensure that the implemented system is the same as the specified one.

Categories and Subject Descriptors: B.0 [**Hardware**]: General; B.5.2 [**Register-Treansfer-Level Implementation**]: Design Aids; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages

General Terms: Design, Verification

Additional Key Words and Phrases: Embedded system, hardware and software synthesis, codesign

---

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

Authors' address: Istituto di Informatica e Telecomunicazioni, Università di Catania, Viale Andrea Doria, 6, Catania, I95125, Italy; email: {Vincenza.Carchiolo; Michele.Malgeri; Giuseppe.Mangioni}@iit.unict.it.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 2000 ACM 1084-4309/00/0700-0399 \$5.00

## 1. INTRODUCTION

The availability of more and more complex electronic devices at a low price has boosted the industry of embedded systems considerably. 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.

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 to reduce development time and optimizing the hardware/software tradeoff. 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 methodology is the technique used to define the requirements of the system, since it affects all 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 behavior 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, as specification language, uses 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 system requirements 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, etc.). Besides, in the description of the system, TTL permits a structured approach, allowing 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 is to ensure that the implemented system is the same as the specified one, through a direct translation of the language operators (syntax-direct translation approach).

This ensures that the device obtained has the same behavior as the specified one. The problem with 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 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, for effectiveness, the derived operators have been translated directly in all the cases in which it was convenient.

In our approach, a TTL module can be directly translated into an RT-level language or into C (for 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 (rendezvous). 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; finally, 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 that can be found in the 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 that make TTL interesting in CoDesign. Section 4 summarizes all the phases of the CoDesign methodology, including 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 the literature divide the design process into the following subproblems:

- (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 the most common 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, intro-

duce the concept of nondestructive communication; that is, the written information can be read by the receiver several times. In behavioral 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 nonzero 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 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].

Several formal languages have been used in CoDesign, most of which are based on the FSM model. The one that has recently obtained the 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 behavior of the system becomes highly predictable, since there is no problem of either synchronization or interleaving of concurrent processes. One of the main drawbacks with this approach is that the resulting FSM can have a large 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]. However, 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 Harel 1989]. Finally, several high-level textual languages have been used in CoDesign, e.g.,  $C^x$  [Ernst1993a]; Hardware-C [Ku and Micheli 1990]; Verilog [Sternheim et al. 1993], and Promela [Wenban et al. 1993].

The second step consists of validating the specification; simulation is still the most widely used approach. Many techniques have been proposed in the literature; they differ in the 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 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 Melham 1992] and finite automata-based tools [Thomas 1990].

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

The synthesis is generally carried out starting with 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 (from the partitioning phase).

The synthesis of the hardware parts usually takes place according to the classic techniques of logical synthesis (see De 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 that do not require a scheduler.

The other approaches 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 classical 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 De 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 that 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 (*Language Of Temporal Ordering Specification*). 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 following are the main features of TTL:

- Formal basis*: by using a mathematical approach, it allows us to check that the specification possesses useful properties such as deadlock freedom and liveness.
- Concurrency*: makes it possible to model systems made up of various parts that evolved in parallel; a situation typical of hardware systems.
- Modularity*: 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*: allows us to concentrate on what is to be done without being affected by problems regarding actual implementation. This guarantees that the language is suitable for describing both hardware and software, regardless of the target architecture.

TTL was 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 behavior 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, based on ACT ONE [Ehrig and Mahr 1990], a language for describing abstract data types (ADTs). In the following, we only discuss the behavioral part of the TTL. The data structure part is the same as in LOTOS; for a complete description, see ISO-IS-8807 [1988] or Bolognesi and Brinksma [1987].

The basic hypothesis of TTL is that the behavior 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, whose sequence is described by TTL behavioral 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) that interact with each other and the environment. The atomic forms of interaction with the outside world are called events. The syntax of a process in TTL is

```
process <process-identifier> <parameter-list> :=
    <behavior-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;
    <behavior-expression> are the TTL expressions that
        define the behavior of the process

```

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

### 3.1 Basic Operators

**3.1.1 Action Prefix.** This operator produces a new behavioral expression from an existing one, prefixing it with the name of an event. If **B** is a behavioral 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, indicated as **i**, that represents an internal action, i.e., an action that 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) that identifies the point of interaction (gate), and a finite list of attributes. Two types of attributes are possible: a value declaration and a 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 nontimed events. The time attributes are quite general, so as to permit the modeling 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 behavioral expressions, then **B1 [] B2** denotes a process that can behave both as **B1** and **B2**. Choosing between them is made by the environment: if it offers an event that is the initial event **B1**, then the former is selected; if, on the other hand, it offers an event **B2**, the latter is selected.

Choice, action prefix, and **stop** are often called *basic operators* because they can specify any behavior. All other operators, therefore, can be expressed in terms of the basic ones, so all the operators 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 that synchronize on a given set of events. It is denoted with the expression **P1** |[ $a_1, \dots, a_n$ ]| **P2**.

The *hiding* operator internalizes actions. If **B** is a behavioral expression and  $a_1, \dots, a_n$  are events, then the expression **hide**  $a_1, \dots, a_n$  in **B** represents an expression that behaves like **B**, but the events  $a_1, \dots, a_n$  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 modeling 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 that normally executes **P1**, but 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 behavioral expression can be preceded 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 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 to reduce develop-



Table I. TTL Operators

Name	Syntax
inaction	stop
termination	exit exit( $E_1, \dots, E_n$ )
choice	$B_1 \square B_2$
action-prefix	$g;B$ $i;B$
parallel-composition	$g_{d_1} \dots g_{d_n} [SP];B$ where $d_i$ is of the form $?x:t$ or $!E$ $B_1 \upharpoonright [g_1, \dots, g_n] \upharpoonright B_2$ $B_1 \upharpoonright\upharpoonright B_2$ $B_1 \parallel B_2$
hiding	hide $g_1, \dots, g_n$ in $B$
instantiation	$p [g_1, \dots, g_n] (E_1, \dots, E_n)$
guarding	$[GP] \rightarrow B$
disabling	$B_1 [> B_2$
enabling	$B_1 >> B_2$ $B_1 >> \text{accept } x:t_1, \dots, x:t_n \text{ in } B_2$
local-definition	let $x:t_1=E_1, \dots, x:t_n=E_n$ in $B$ let $x:t=E$
sum-expression	choice $g$ in $[g_1, \dots, g_n] \square B$ choice $x:t \square B$
par-expression	par $g$ in $[g_1, \dots, g_n] \upharpoonright [a_1, \dots, a_n] \upharpoonright B$ par $g$ in $[g_1, \dots, g_n] \upharpoonright\upharpoonright B$ par $g$ in $[g_1, \dots, g_n] \parallel B$
loop-expression	loop( guard; value-expression; $B_1$ )

ment time by using parameterized blocks which may be present in libraries.

TTL modules are a collection of processes that 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.

TTL templates are a further extension of process concepts. Templates allow us to parameterize the names of processes and the types of gates (in the gate list). Thus, 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 control predominates dataflow).

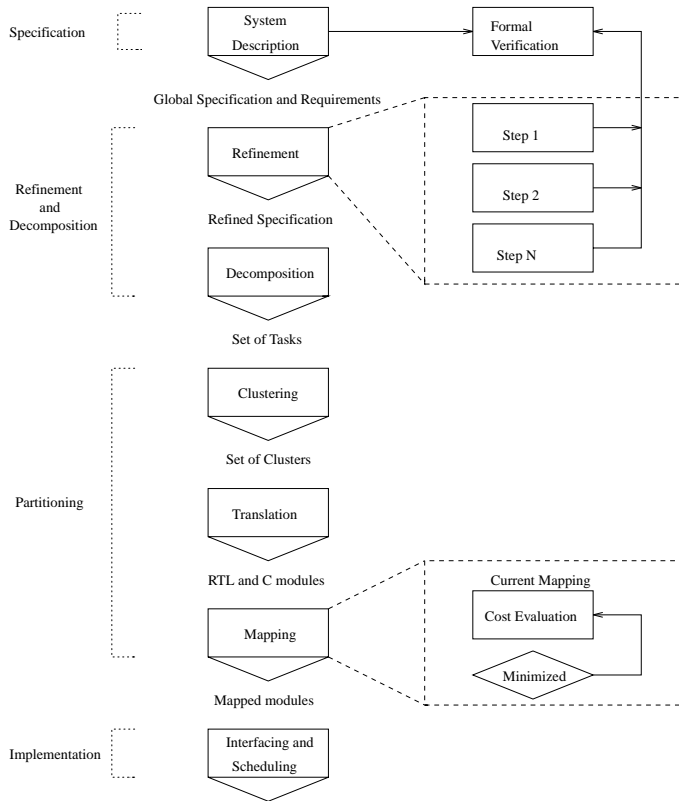


Fig. 1. Overview of methodology.

In this CoDesign methodology we can outline four fundamental phases, each consisting of different steps (see Figure 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 with TTL. The purpose of this phase is to express the requirements of the system given by the client in (plain) TTL, to verify its correctness, and to give a quick prototype to the client for his or her approval. In fact, a TTL specification can easily be simulated using a TTL interpreter.

During the whole specification phase we need to verify that the behavior as described corresponds to what we want to specify (this operation is known as model 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 behavior. The result is that correctness is assured only for the verified test patterns.

The use of a formal model as a specification technique gives us the opportunity to exploit the mathematical base to carry out more complete verification. 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, verify that a given desired configuration is adopted by the system. The combination of these two properties also allows us to verify even very complex situations.

The tools for formal verification 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 semiautomatically), meaning that they ensure 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 automata, permits the verification process to be automatized, even if this approach is often not feasible due to problems related to state explosion (see Kurshan [1994]).

## 4.2 Refinement and Decomposition

The refinement and decomposition 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 specify the requirements in a form that is easily and effectively implementable. During this step, several styles of specification are usually adopted. The style of specification (resource-oriented, state-oriented, constrain-oriented, or monolithic) for the different phases of the design is discussed in Brinksma et al. [1987] and Vissers et al. [1988]; 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 behavior of the system. The final goal is to obtain a specification that is detailed enough to be effectively implemented, but also correctly describes the requirements of the system.

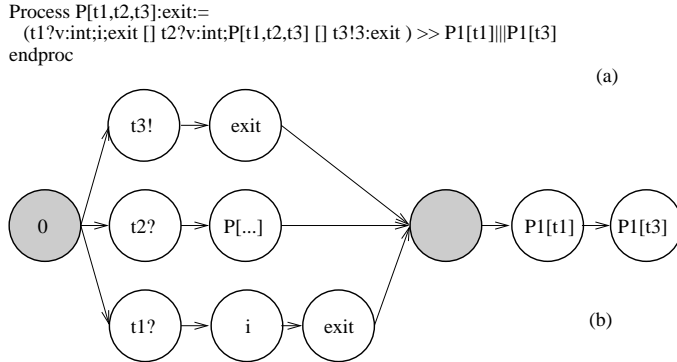


Fig. 2. Example of IGM.

The equivalence between what is 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, such as TTL, allows us to use tools that assist the designer during the refinement phase, and provides 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 provide an easily manageable representation for 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. Figure 2(a) contains an example of a TTL specification, whereas Figure 2(b) represents the behavioral 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 behavioral graph indicates the point from which to start.

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 behavior of the system. 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

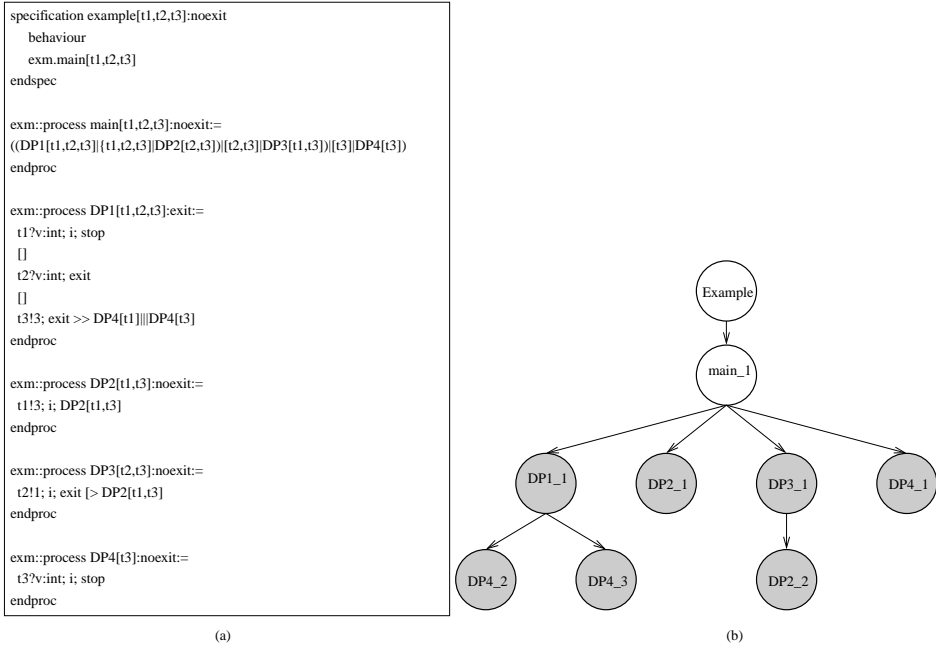


Fig. 3. Example of IT.

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) System decomposition. The system is decomposed in several sub-systems, called “tasks.” 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. Figure 3(a) shows a TTL specification, whereas in Figure 3(b) the relative IT with the labelled nodes is shown. In Figure 3(b) the hatched nodes are the tasks resulting from the decomposition process.

### 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 two steps: *clustering* (not connected to 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 in our methodology is to minimize the *degree of coupling* between two tasks; this parameter is defined as the *number of interactions between 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, the higher the cost of communication (and therefore of the interfaces) among the modules. 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; while if the number of clusters is too small, the mapping algorithm has few chances to obtain a good hardware/software tradeoff [Carchiolo et al. 1998a].

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

- Performance*. This parameter affects the entire 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) favors the software realization, since software can be modified more easily.
- 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  $\mathcal{J}$  that takes the parameters mentioned above into account. A practical approach is to allocate all blocks in hardware, and then to move those tasks whose  $\mathcal{J}$  remains within a given value to software (usually called a hardware-oriented approach, see for example, Gupta and De Micheli [1993]). Of course, we can also do the opposite, as in Ernst [1993b].

In order to evaluate  $\mathcal{J}$  correctly, we need to determine the cost of each task. Some of the approaches proposed in the literature are based on cosimulation [Wilson 1994; Keutzer 1994]; some are based on soft-computing techniques [Catania et al. 1995]; and still others on the knowledge of computed values of  $\mathcal{J}$ , with reference to libraries of components [Axelsson 1996]. Another approach simultaneously develops the scheduler and evaluates the impact of the allocation (hardware or software) on  $\mathcal{J}$  [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  $\mathcal{J}$ , and therefore apply some of the techniques that we find in the 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.

The target architecture is selected in this phase. 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 with a unique correspondence to 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 RT-level description. The synthesis into software components is more complex due to the limits imposed by the nature of software.

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

The presence of software and hardware components imposes the necessity for a scheduler responsible for the following tasks:

- (1) it must synchronize all the modules of the system;
- (2) it must serialize the software modules that 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 an event, the scheduler is also defined on the

same basis. In fact, it does not select the next process to be activated, but does select 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 (rendezvous). The synchronization protocol was implemented into a hardware component for this reason. 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 is the opportunity to synthesize complex behavior by using only a few translation rules, since 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 section we present some of the hypothesis on which the synthesis process is based, and in particular the 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 rendezvous mechanism. Three different types of interaction are present in TTL: *value matching*, *value generation*, and *value passing*. The only one that is meaningful for our application is value passing because it corresponds to the physical reality of devices, and thus it is the only one we take into account.

The instantiation of processes in TTL plays a fundamental role in the specification of systems; some situations that 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 hence has no correspondence in hardware. Moreover, self-instantiation is only allowed if the gate list is not modified.

In implementing the choice operator, we have to solve the nondeterminism 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 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



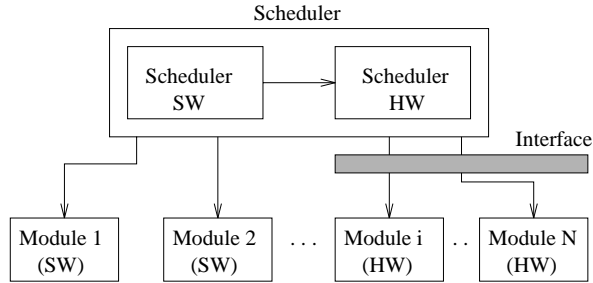


Fig. 4. Scheme of the synthesized system.

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.

Figure 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 serialize the software modules represented in the specification by concurrent processes.

The scheduler consists of software and a hardware parts. 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. As we see below, 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 rendezvous protocol of the TTL.

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

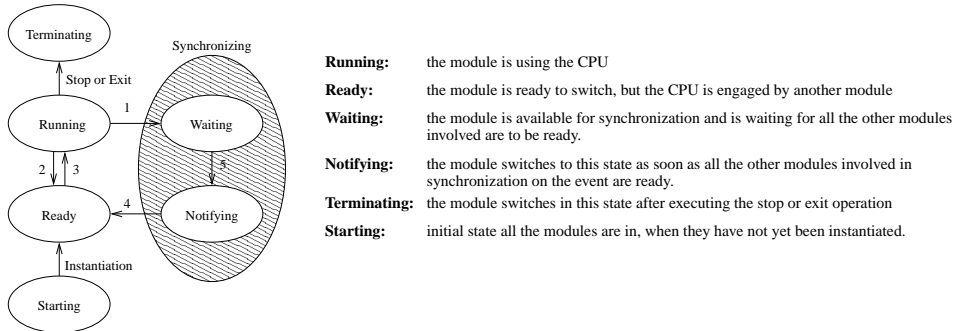


Fig. 5. Diagram of the states of a module.

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 that is higher (on average) than the technique based on interrupt.

The scheduling algorithm that we use belongs to the second group of techniques; but 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 Figure 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 Figure 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 an event only when they are receiving. As we see below, this difference is due to the solution adopted for the implementation of synchronization in hardware.

```

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.

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 Figure 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 verifies whether 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 Figure 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 reception of the *ack* signal, the hardware modules proceed autonomously until 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 retested 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 section 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, while 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 Figure 5), while in the second step it is actually synchronized with other modules (NOTIFYING state in Figure 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

```

...
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.

value (condition expressed by the symbol “!”), while the other accepts a value (expressed by “?”).

This situation is expressed in TTL as

$$\begin{aligned}
 T &:= \dots \cdot g!v; \dots \\
 R &:= \dots \cdot g?v; \dots
 \end{aligned}
 \tag{1}$$

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

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.

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

*One-to-One.* First, we 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

```

...
SWITCH (STATE[instanceNumber]) {
.
.
.
CASE y:
  notifies the scheduler to be ready for the synchronization;
  STATE[instanceNumbr]++;
  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.

signals: one for the exchange of the data itself, and two others to manage the synchronization (a ready and an acknowledgment signal). The need for two signals for synchronization is due to the fact that communication in TTL is a rendezvous between events.

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

The signal  $in_i$  ( $in_j$ ) represents 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 Section 3). Translation of block  $i$  into hardware is represented in Figure 10, using the RTL language described in Appendix A.

As we can see from Figure 10, the transmitter waits for the receiver to be available for synchronization, after which it acknowledges the synchronization and exchanges the value (if any). Figure 11 represents the translation into RTL of the block  $j$ .

The behavior of the receiver complements that of the transmitter. In Figure 10,  $v_T$  represents the variable containing the value to be transferred, which in RTL is equivalent to a register. Likewise in Figure 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 four RTL steps and the receiver in three 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 Figure 11. Coding of the transmitter is similar to that of the previous case (shown in Figure 10), the only exception is that this time the transmitter has to ascertain, before sending the *ack* signal, that all the receivers are available for synchronization.

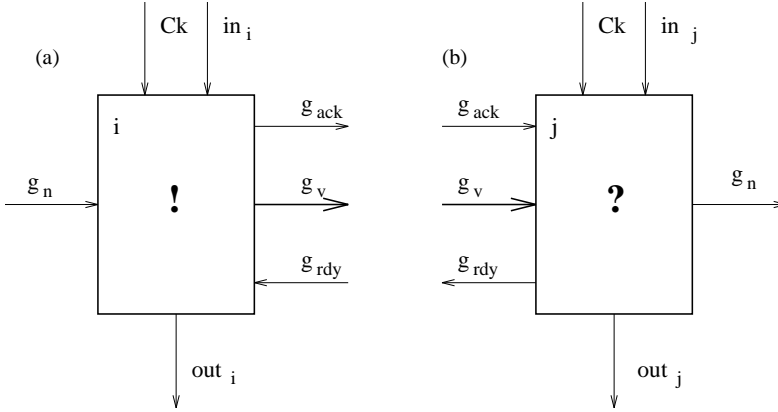


Fig. 9. Scheme of basic interaction events.

```

... :
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.

```

... :
y   : g_rdy = 1; if (not g_ack; g_ack) goto(y; y + 1)    warns the transmitter to be ready
                                                         for synchronization and simultane-
                                                         ously sends an ack signal
y + 1 : g_n = 1                                         informs transmitter that synchro-
                                                         nization has actually occurred
y + 2 : v_R := g_v                                     accepts the value
...  :
    
```

Fig. 11. Translation of receiver.

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 Figure 12.

The case described in Figure 12 implements a transmitter that 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 Figure 12) is generalized by using the functional block  $cond(g_{rdy1}, \dots, g_{rdyn})$ , which in RTL implements the verification condition.

**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 that is occurring. A process

...	:	
$x$	:	$if(not(g_{rdy1} \text{ and } g_{rdy2})); (g_{rdy1} \text{ and } g_{rdy2}) \text{ goto}(x; x+1)$
		waits for all the receivers to be ready for synchronization
$x+1$	:	$g_{ack} = 1$
		acknowledges the synchronization
$x+2$	:	$if(not(g_{n1} \text{ and } g_{n2}); g_{n1} \text{ and } g_{n2}) \text{ goto}(x; x+3)$
		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.

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 takes place is given according to the availability of the other processes. As usual, the job of synchronization is given to the scheduler, which must also inform the process that has executed the choice expression of the event on which actual synchronization took place, in order to determine the way to proceed.

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

$$\begin{aligned}
 T &:= \dots((g_1!v_1; \dots) \square (g_2!v_2; \dots) \square (\dots) \square (g_m!v_m; \dots)) \dots \\
 R_1 &:= \dots g_1?v_1; \dots \\
 R_2 &:= \dots g_2?v_2; \dots \\
 &\quad \vdots \\
 R_m &:= \dots g_m?v_m; \dots \tag{2}
 \end{aligned}$$

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 Figure 13.

*RTL implementation.* In explaining the RTL coding algorithm, we assume, for the sake of simplicity, that we are dealing with a two-way choice (for extension to more general cases, see below). Three different situations can occur, according to the type of event (accepting or offering a value) in the choice.

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

$$T := \dots(g_1!v; \dots \square g_2!v; \dots) \dots$$



```

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.

$$\begin{aligned}
 R_1 &:= \dots \cdot g_1?v; \dots \\
 R_2 &:= \dots \cdot g_2?v; \dots
 \end{aligned}
 \tag{3}$$

That is, a process  $T$  that is able to synchronize both on the event  $g_1!v$  and on the event  $g_2!v$ , and two processes that can accept a value, the first one on gate  $g_1$  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, since from their point of view it makes no difference whether the process with which they synchronize contains a choice or not. Figure 14 shows the RTL translation of  $T$ .

A fundamental point in coding the choice operator is the instruction in step  $x$  in Figure 14; otherwise, the coding can be considered 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 (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 nondeterministic choice between the two events it is possible to

```

...      :
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.

synchronize. In our implementation this indeterminism is solved by choosing one of the possible events *a priori*. In Figure 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 by the combinatorial function  $cond(g_{11rdy}; g_{12rdy}; \dots)$ ; this is repeated for  $g_{2rdy}$ ,  $g_{1n}$  and  $g_{2n}$ .

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

$$\begin{aligned}
 R &:= \dots \cdot (g_1?v; \dots \square g_2?v; \dots) \cdot \dots \\
 T_1 &:= \dots \cdot g_1!v; \dots \\
 T_2 &:= \dots \cdot g_2!v; \dots
 \end{aligned} \tag{4}$$

The RTL translation of processes  $T_1$  and  $T_2$  proceeds as above. Figure 15 shows the coding of the process  $R$ .

Here again, the RTL translation can be considered as the 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  $g_2!v$ .

```

...      :
x        :  g1rdy = 1; g2rdy = 1;
          :  if((not g1ack) and (not g2ack); g1ack; (not g1ack) and g2ack) goto(x; x + 1; x + k)
x + 1    :  g1n = 1
x + 2    :  vR := g1v
...      :
x + k    :  g2n = 1
x + k + 1 : vR := g2v
...      :
    
```

Fig. 15. Translation of process R.

```

...      :
x        :  g1rdy = 1;
          :  if((not g1ack) and (not g2rdy); g1ack; (not g1ack) and g2rdy) goto(x; x + 1; x + k)
x + 1    :  g1n = 1
x + 2    :  vR := g1v
...      :
x + k    :  g2ack = 1
x + k + 1 : if(not g2n; g2n) goto(x; x + k + 2)
x + k + 2 : g2v = vT
...      :
    
```

Fig. 16. Translation of process TR.

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

$$\begin{aligned}
 TR &:= \dots \cdot (g_1?v; \dots \square g_2!v; \dots) \cdot \dots \\
 T &:= \dots \cdot g_1!v; \dots \\
 R &:= \dots \cdot g_2?v; \dots
 \end{aligned} \tag{5}$$

Here again the RTL translation of  $T$  and  $R$  presents no difficulties, being the same as previous cases. Figure 16 gives the RTL coding of  $TR$ . If  $g_2!v$  is involved in a one-to-many synchronization,  $g_{2rdy}$  has to be substituted by  $cond(g_{21rdy}; g_{22rdy}; \dots)$ , 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 is that more than two processes must synchronize with 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 be computed easily. To clarify the procedure, we give an example that 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 Figure 14, but the condition is of the following form:

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

**6.2.5 Synthesis of Derived Operators.** This section 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 behavior of the system. As stated previously, 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 Implementation.* Let us suppose we have  $N$  processes. Their sequential composition is expressed in TTL as  $P1 >> P2 >> \dots PN$ , whereas their parallel composition is expressed as  $P1 \parallel P2 \parallel \dots PN$ . 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 relevant process of its successful termination, the scheduler then 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 figures, we use a box with a thick border. Figure 17(a) shows the result of synthesis in RTL of the behavior expression  $P1 >> P2 >> \dots PN$ , whereas Figure 17(b) shows the result of the synthesis of  $P1 \parallel P2 \parallel \dots PN$ .

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:

$$\begin{aligned}
P & := P1 \parallel P2 \parallel P3 \\
\text{where} \\
P1 & := g^1!v_1; P1 [] g^2!v_2; P1 \\
P2 & := g^1?x : \text{integer}; P2 \\
P3 & := P4 >> P5
\end{aligned}$$

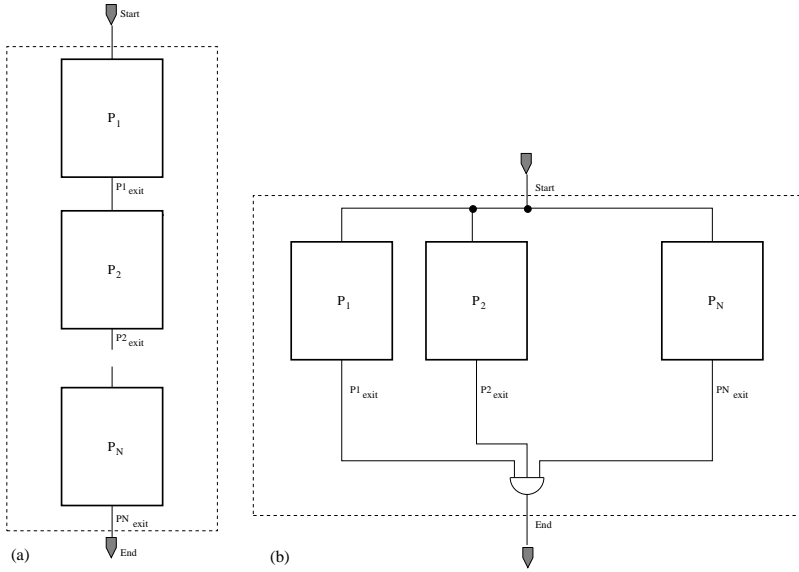


Fig. 17. Scheme of sequential and parallel composition.

where  $P4 := (g^2?y : integer; exit)$   
 $P5 := g^2?z : integer; P5$

Figure 18 shows the result of the synthesis process. The blocks labelled with a question mark are the translations 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

This paper presents all the phases of the methodology, but its focus is on the synthesis phase.

The proposed design path is presented in depth, pointing out the aspects that make it interesting in the context of embedded systems. The use of a formal technique to specify the system is the key point characterizing the methodology.

Moreover, the kind of formal language we use has a strong impact on the synthesis phase; the difficulties related to this problem have been stressed throughout the entire paper; and some toy examples are presented to clarify the algorithm (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 minimizing the size of the code generated for the software part. Moreover, some studies are

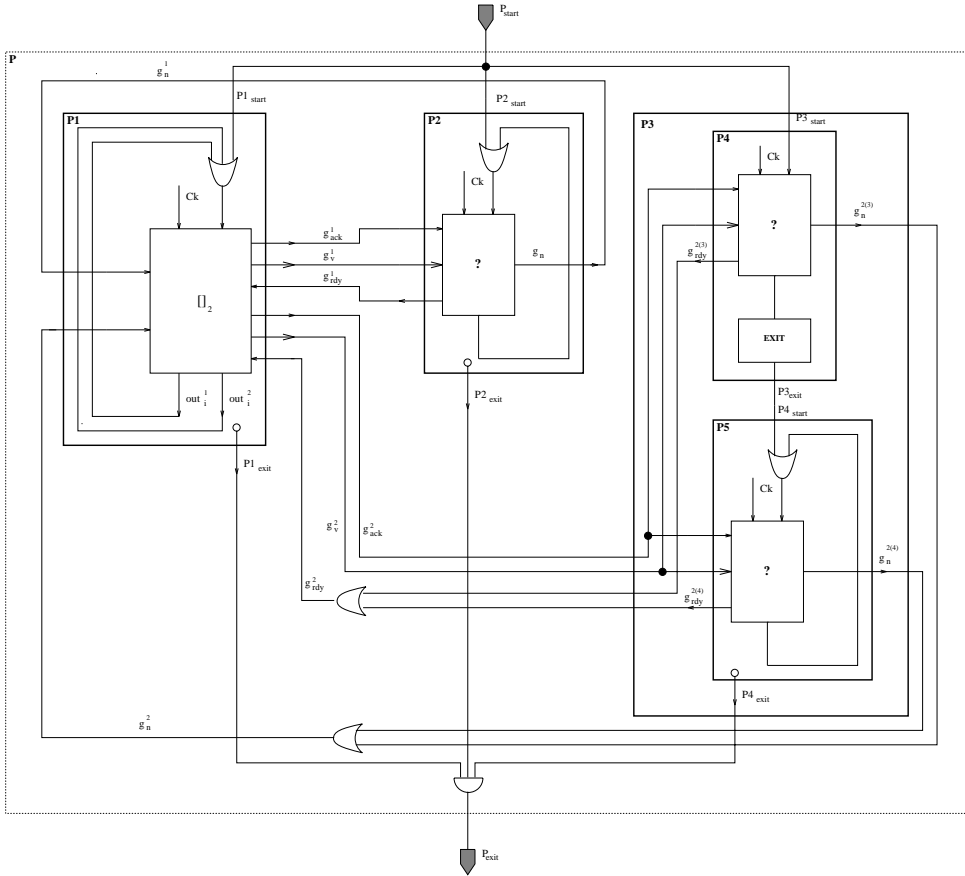


Fig. 18. Scheme of an Example.

devoted to optimal synthesis of the hardware part, mainly in the implementation of interfaces between hardware and software. Finally, we are working on some other languages for register-level hardware description in order to exploit the large number of available commercial tools.

APPENDIX

A. THE TARGET RTL LANGUAGE

The RTL language in this paper can define the structure of a generic digital system. Any digital system is modeled using a functional block that receives information from the external environment via signals, and processes them to produce output signals (in 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

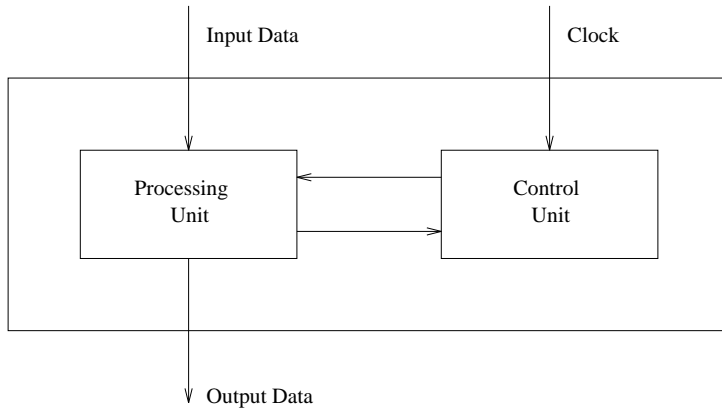


Fig. 19. Logical scheme of a digital system.

be stable before the clock cycle finishes. Figure 19 represents the logical scheme of a generic digital system.

The RTL module is defined by the following:

- Components: contains the declaration of the components that make up the processing unit.
- Control sequence: defines the internal command sequence that must be emitted by the control unit.
- Permanent assignment: defines an operation that 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 that are executed in parallel. All the commands belonging to a step are separated by ;. Thus, 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 is also used to describe the assignment to output lines.

## 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 this paper.

## REFERENCES

- ADAMS, J. K. AND THOMAS, D. E. 1995. Multiple-process behavioral synthesis for mixed hardware-software systems. In *Proceedings of the Eighth International Symposium on System Synthesis* (Cannes, France, Sept. 13–15, 1995), P. G. Paulin and F. Mavaddat, Eds. ACM Press, New York, NY, 10–15.
- AXELSONN, J. 1996. Hardware/software partitioning aiming at fulfillment of real-time constraints. *J. Syst. Architecture* 42, 6-7 (Dec.), 439–464.
- BERRY, G., COURONNE, P., AND GONTHIER, G. 1991. The synchronous approach to reactive and real-time systems. *Proc. IEEE* 79.
- BERRY, G. AND TOUATI, H. 1993. Optimized controller synthesis using Esterelle. In *Proceedings of the International Workshop on Logic Synthesis* (May),
- BOLOGNESI, T. AND BRINKSMA, E. 1987. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.* 14, 25–59.
- BOYER, R. S., KAUFMANN, M., AND MOORE, J. S. 1995. The Boyer-Moore theorem prover and its interactive enhancements. In *Computers & Mathematics with Applications* Pergamon Press, Inc., London, UK, 27–62.
- BRINKSMA, E., SCOLLO, G., AND VISSERS, C. 1987. Experience with and future of LOTOS as a specification language. Tech. Rep. INF-87-17. Twente University.
- CARCHIOLO, V., MALGERI, M., AND MANGIONI, G. 1996. TTL: A LOTOS extension for system description. In *Proceedings of the Conference on Basys* (Basys '96, Lisbon, Portugal),
- CARCHIOLO, V., MALGERI, M., AND MANGIONI, G. 1998. Formal codesign methodology with multistep partitioning. *J. Comput. Aided VLSI Des.* 7, 4.
- CARCHIOLO, V., MALGERI, M., AND MANGIONI, G. 1998. Synthesis of TTL specification: A case study. In *Proceedings of the MultiConference on CESA98-IMACS*,
- CASPI, P., PILAUD, D., HALBWACHS, N., AND PLAICE, J. A. 1987. LUSTRE: A declarative language for real-time programming. In *Proceedings of the Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (POPL '87, Munich, Germany, Jan. 21–23, 1987), ACM Press, New York, NY, 178–188.
- CATANIA, V., MALGERI, M., AND RUSSO, M. 1995. A methodology for codesign based on fuzzy logic and genetic algorithms. In *Proceedings of the 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, 62–70.
- CHIDO, M., GIUSTO, P., JURCSKA, A., LAVAGNO, L., SENTOVICH, E., HSIEH, H., SUZUKI, K., AND SANGIOVANNI-VINCENTELLI, A. 1995. Synthesis of software programs for embedded control application. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation* (DAC '95, San Francisco, CA, June 12–16), B. T. Preas, Ed. ACM Press, New York, NY, 587–592.
- CHIDO, M., GIUSTO, P., JURCSKA, A., HSIEH, H. C., SANGIOVANNI-VINCENTELLI, A., AND LAVAGNO, L. 1993. A formal specification model for hardware/software codesign. In *Proceedings of the International Workshop on Hardware-Software Codesign* (Boston, MA, Sept.),
- CHOU, P. H., ORTEGA, R. B., AND BORRIELLO, G. 1995. The Chinook hardware/software co-synthesis system. In *Proceedings of the Eighth International Symposium on System Synthesis* (Cannes, France, Sept. 13–15, 1995), P. G. Paulin and F. Mavaddat, Eds. ACM Press, New York, NY, 22–27.
- CHOU, P., WALKUP, E. A., AND BORRIELLO, G. 1994. Scheduling for reactive real-time systems. *IEEE Micro* 14, 4 (1994), 37–47.
- DE MICHELI, G. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., New York, NY.
- DRUSINSKI, D. AND HAR'EL, D. 1989. Using statecharts for hardware description and synthesis. *IEEE Trans. Comput.-Aided Des.* 8.



- EHRIG, H. AND MAHR, B. 1990. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Springer-Verlag, New York, NY.
- ERNST, R. 1993. Hardware-software codesign of embedded controllers based on hardware extraction. In *Proceedings of the International Workshop on Hardware-Software Codesign* (Boston, MA, Sept.),
- ERNST, R., HENKEL, J., AND BENNER, T. 1993. Hardware-software co-synthesis for microcontrollers. *IEEE Des. Test* 10, 4 (Dec. 1993), 64–75.
- GORDON, M. J. C. AND MELHAM, T. F., Eds. 1993. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY.
- GUERNIC, P. L., BENVENISTE, A., BOURNAT, P., AND GAUTHIER, T. 1985. A data flow oriented language for signal processing. IRISA Tech. Rep. 246. IRISA, Rennes, France.
- GUPTA, R. K., COELHO, C. N., AND DE MICHELI, G. 1992. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proceedings of the 29th ACM/IEEE Conference on Design Automation* (DAC '92, Anaheim, CA, June 8-12), D. G. Schweikert, Ed. IEEE Computer Society Press, Los Alamitos, CA, 225–230.
- GUPTA, R. K., COELHO, C. N., AND DE MICHELI, G. 1994. Program implementation schemes for hardware-software systems. *IEEE Computer* 27, 1 (Jan. 1994), 48–55.
- GUPTA, R. K. AND DE MICHELI, G. 1993. Hardware-software cosynthesis for digital systems. *IEEE Des. Test* 10, 3 (Sept. 1993), 29–41.
- 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*,
- HALANG, W. AND STOYENKO, A. 1991. *Constructing Predictable Real Time Systems*. Kluwer Academic Publishers, Hingham, MA.
- HEISH, H., LAVAGNO, L., PASSERONE, C., SANSE, C., AND GIOVANNI-VINCENTELLI, A. 1997. Modeling microcontroller peripherals for high-level co-simulation and synthesis. In *Proceedings of the Fifth International Workshop on Hardware/Software Codesign* (Braunschweig, Germany, Mar. 1997),
- HENKEL, J.-R., BENNER, T., ERNST, R., YE, W., SERAFIMOV, N., AND GLAWE, G. 1994. COSYMA: a software-oriented approach to hardware/software codesign. *J. Comput. Softw. Eng.* 2, 3 (1994), 293–314.
- HOARE, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice-Hall, Inc., Upper Saddle River, NJ.
- HOLTZMANN, G. 1991. *Design and Validation of Computer Protocols*. Prentice-Hall, New York, NY.
- ISO. 1988. Information processing systems, open system interconnection LOTOS, A formal description technique based on the temporal ordering of observational behaviour. ISO-IS-8807.
- KEUTZER, K. 1994. Hardware/software co-simulation. In *Proceedings of the 31st Annual Conference on Design Automation* (DAC '94, San Diego, CA, June 6–10, 1994), M. Lorenzetti, Ed. ACM Press, New York, NY, 439–440.
- KOHAVI, Z. 1978. *Switching and Finite Automata Theory*. McGraw-Hill, Inc., New York, NY.
- KU, K. AND MICHELI, G. D. 1990. Hardware C: A language for hardware design version 2.0. Tech. Rep. CSL-TR-90-419. Stanford University, Stanford, CA.
- KUMAN, S. AND ALII, 1993. A framework for hardware/software codesign. *IEEE Computer* 26, 12 (Dec.), 39–45.
- KURSHAN, R. P. 1994. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ.
- LOGRIPPO, L., MELANCHUCK, T., AND DUWORS, R. 1990. The algebraic specification language LOTOS: An industrial experience. In *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development* (Napa, CA, May 9–11, 1990), M. Moriconi, Ed. ACM Press, New York, NY, 59–66.
- MILNER, A. 1989. *A Calculus of Communicating Systems*. Springer Lecture Notes in Computer Science, vol. 92. Springer-Verlag, New York, NY.

- QUEMADA, J., PAVEN, S., AND FERNANDEZ, A. 1989. State exploration by transformation with LOLA. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science, vol. 407. Springer-Verlag, New York.
- SARACCO, R., SMITH, J. R. W., AND REED, R. 1989. *Telecommunications Systems Engineering Using SDL*. Elsevier North-Holland, Inc., New York, NY.
- SHIN, Y. AND CHOI, K. 1997. Enforcing schedulability of multi-task systems by hardware-software codesign. In *Proceedings of the Fifth International Workshop on Hardware/Software Codesign* (Braunschweig, Germany, Mar. 1997).
- STERNHEIM, E., MADHAVEN, R., SINGH, R., AND TRIVEDI, Y. 1993. *Digital Design and Synthesis with Verilog HDL*. Automata.
- TAKACH, A., WOLF, W., AND LEESER, M. 1995. An automaton model for scheduling constraints in synchronous machines. *IEEE Trans. Comput.* 44, 1 (Jan. 1995), 1–12.
- THOMAS, W. 1990. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, J. van Leeuwen, Ed. MIT Press, Cambridge, MA, 133–191.
- VAHID, F. 1997. Modifying min-cut for hardware and software functional partitioning. In *Proceedings of the Fifth International Workshop on Hardware/Software Codesign* (Braunschweig, Germany, Mar. 1997).
- VAN EIJK, P. 1989. Tools for LOTOS specification style transformation. Tech. Rep. Memo. 89-35.
- VISSERS, C. A., SCOLLO, G., AND VAN SINDEREN, M. 1988. Architecture and specification style in formal description of distributed systems. In *Proceedings of the Conference on Protocol Specification, Testing and Verification* (Amsterdam), North-Holland Publishing Co., Amsterdam, The Netherlands, 189–204.
- WENBAN, A. S., O'LEARY, J. W., AND BROWN, G. M. 1993. Codesign of communication protocols. *IEEE Trans. Comput.*, 46–52.
- WILSON, J. 1994. Hardware/software selected cycle solution. In *Proceedings of the International Workshop on Hardware-Software Codesign*.

Received: December 1997; accepted: December 1997