DESIGN METHODS FOR REACTIVE REAL-TIME SYSTEMS CODESIGN

by

Massimiliano Chiodo and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M92/116

12 October 1992

DESIGN METHODS FOR REACTIVE REAL-TIME SYSTEMS CODESIGN

by

Massimiliano Chiodo and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M92/116

12 October 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DESIGN METHODS FOR REACTIVE REAL-TIME SYSTEMS CODESIGN

by

Massimiliano Chiodo and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M92/116

12 October 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Design Methods for Reactive Real–Time Systems CoDesign

Massimiliano Chiodo
Marelli Autronica, Pavia, Italy

Alberto Sangiovanni–Vincentelli,
University of California, Berkeley

Abstract

In this paper, we propose a design methodology for real–time mixed hardware–software system that stresses system design issues rather then specific software or hardware aspects. We propose that a unified mathematical model based on interacting FSMs be used as a representation of the behavior. This underlying model is used as the basis to perform all steps of the design process, namely verification, simulation, software–hardware partitioning, synthesis, and technology mapping. The common model allows the description to be technology independent thus enhancing the flexibility of the design. Our approach is based on the knowledge of the techniques used to design embedded controllers for automotive applications and on the synthesis and verification of finite state machines. We demonstrate the method on a few examples from the automotive industry.

1. Introduction

Although it has been common practice in industrial systems design for decades, mixed Hardware–Software systems design is still in its infancy as a recognized discipline. In addition, systems which require the joint design of hardware and software range from the portable CD player controller to the navigation control unit of a battle aircraft, and it is difficult to think of design methods that can serve in such a wide array of applications. Therefore, we concentrate on the field of Reactive Real–Time Systems [1,4] (RRTS) which will be defined more formally further on. In this paper, we propose a design methodology that stresses the *system* design issues rather then specific software or hardware aspects.

In our method, a system is described as a network of interacting Finite State Machines (FSM). The actual representation of this network, which is clearly not unique, consists of an *Intermediate Format* (FSM–IF) into which an input language is mapped. We believe that the choice of the input language is relatively unimportant, as long as the languages used can properly express the intended behavior and can be mapped into the FSM–IF which carries no knowledge of what entry language was used. The FSM–IF carries all the information on how the system reacts to external stimuli. The FSM–IF is the input representation on which we can perform all further steps of the design process. We want to work as much as possible on the FSM–IF in order to remove from the system description every possible source of problems that could affect the final implementation, and to gain flexibility in the process of actually producing and evaluating the implementation before we build it. In practice, we want to do *Formal Verification* to check whether the system satisfies a number of properties that specify its correctness, *Simulation* to check whether the systems responds correctly

to the stimuli that the environment is supposed to produce, and *Automatic Synthesis* to produce, with a maximum of flexibility, an implementation consistent with some user–defined criteria.

The reason why we choose a FSM based representation is that FSMs are well known mathematical objects, relatively easy to handle. Manageable algorithms for formal verification and automatic circuit and code synthesis exist that work on some type of FSMs. In particular, formal verification is now becoming of age and the most viable techniques, namely model checking and language containment, are based on FSMs. We allow the use of non–determinism in the specification of a system to allow a stepwise refinement design process, as proposed by Kurshan [28], and to provide a means to specify the interaction with the environment which is generally not fully defined. Non–determinism also allows for abstraction mechanisms which are a powerful means to cope with the complexity of systems. However, non–determinism in system specification must always be resolved once an implementation is to be devised.

In the implementation, a major issue is obviously what criteria are to be considered in software–hardware partitioning. Ideally, the software solution provides the maximum of flexibility during the design process. However, an all–software solutions may be too slow with respect to the timing requirements. Hardware solutions are faster but more risky, because a poorly designed circuit is not nearly as easy to fix as code. Regarding cost, software solutions tend to be cheaper for low to moderate production volumes, whereas hardware solutions are to be preferred for large volumes. In addition, the criteria can change over time (e.g. variations in production volume, or new technology avaliable) and call for a rather different implementation option. In such a case, the FSM–IF based model provides a neutral system description on which, if all the proper steps of verification and simulation have been applied, only the implementation sythesis must be re–done.

In perspective, our methodology would be embodied in a design framework (as depicted in Figure 1) consisting of a number of specification languages, including graphical interfaces, that will be compiled into a common FSM–IF; an array of design aid tools like verification systems and simulators; a set of "extended" technology mapping tools. All of these will exchange information through FSM–IFs.

Finally, it must be mentioned that the flexibility introduced in the design process by the principles we advocate (i.e. technology independent specification) allow a clear trade–off between specification and actual implementation. Within the array of all possible implementation option for a given initial specification, we may not find one that will produce a viable product. This means that the specification needs to be changed, which in turn produces a new set of implementation options.

The paper is organized as follows. In section 2 we define the class of systems of interest. In section 3 we explain the ideas of our method and illustrate them on a simple example. In section 4 we use a simplified version of an industrial system to demonstrate how the method can actually work. In section 5 we address the issue of specification–implementation trade–off by examining a real industrial project.

2. Reactive Real–time Systems and Embedded Systems

Pneuli [1], along with others, introduced the name Reactive Systems (RS) to designate systems that react to external signals from their environment by sending themselves signals to the environment. As opposed to "Transformational" or "Terminating" systems which, given a set of inputs, compute a (possibly non–deterministic) set of outputs, RS ideally never terminate. The purpose for which they are run is not to obtain a final result but rather to maintain some interaction with the environment. In general, the evolution of this interaction is not known a priori. That is, the intermediate signals from the environment to the system depend on the intermediate signals from the system to the environment. In other words, the environment can be modelled as a reactive system itself whose behaviour is not completely known.

The notion of RS captures the notion of Real–Time (RT) system. As well pointed out [4], it is commonly accepted to call *real–time* a program or system that receives external interrupts or reads sensors connected to the physical world and outputs commands to it. Real–time Systems (RTS) design and programming is an essential industrial activity whose importance keeps increasing. Factories, plants, transportation systems,

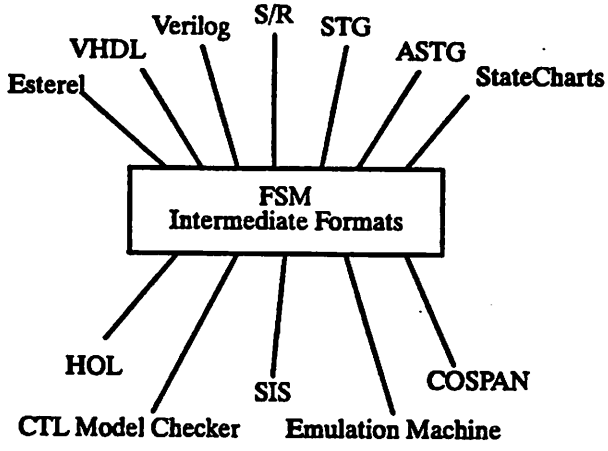

Figure 1. CoDesign Framework

cars, and a wide variety of everyday objects are controlled by electronic systems. With[4], we will call *reactive* any system that maintains a permanent interaction with its environment, and reserve the term *real-time* to those reactive systems that in addition are subject to externally defined timing constraints.

In this paper we will use the term Reactive Real-Time System (RRTS) to characterize the specific class of systems of interest. According to [14], RRTS characteristics are as follows:

- *The problem formulation for a RRTS system is drawn from science and engineering disciplines.* Real-time problems are formulated in language drawn from biophysics, chemistry, automotive or avionic engineering rather then in language of computers. The vocabulary distinction is significant since it is correlated with the separation of the problem formulation and the details of implementation in system development problems.

- *The environment of a RRTS contains devices that act as the "senses" and as the "arms" of the system.* A RRTS is typically attached to sensors such as termocouples, optical scanners, contact probes, and thus collects a continuous stream of relatively unstructured data. The environment of a RRTS contains also devices that can produces change the physical world in a literal way by changing temperatures, valve positions, and so on.

- RRTS require concurrent processing of multiple inputs. A RRTS has to process an input/event when it occurs, not when it is ready to do so.

As examples of RRTS we mention:

- Most Automotive electronics and Avionics applications, among which
  - engine control system (injection/ignition and emission control)
  - breaks anti-blocking systems
  - vehicle automatic navigation systems
- Industrial processes monitoring and control systems (like automated assembly lines, automatic storage plants)
- Home and office appliances

Safety is a crucial concern for RRTS due both to the particular type of applications they are used for (think of automotive or aircraft controllers) and to the fact that they cannot be fixed once installed and sold (if a malfunction is detected a market share of the manufacturer may be in jeopardy.) In some systems even the slightest delay or wrong synchronization make results unacceptable and may have dangerous or catastrophic consequences.

RRTS are mostly used to provide some degree of automation in common widespread goods like cars (Automotive) and consumer electronics, and in other fields like Avionics. In these types of products, (for example on board of any kind of vehicle) an inexpensive, small, light, robust device is needed. Since they must be inexpensive, they only come with the capabilities strictly necessary for the function they have been thought for (the smallest memory, the smallest CPU, etc.) Consequently, most RRTS are implemented in the form of Embedded systems (ES). ES are single–purpose devices designed to perform a single well–defined function life long. In other words, Embedded Systems are systems whose behaviour is built in with the system itself and can never be changed. The typical ES is a μP based system that comes with a fixed program in its ROM. For ES, special devices have been developed, called micro–controllers (μC) or single chip computers, which consist of a μP and a number of support devices like I/O devices, memories, A/D converters, timers etc., all integrated in a single chip. This methodology has been targeted for the design of embedded controller applications mainly.

We believe that the demand for embedded controllers will increase more and more in the near future both in terms of range of applications and sophistication of the functions performed. This will be particularly true in some areas like in–vehicle automotive applications. In this specific sector of consumer goods, high production volumes go along with a demand for high quality (since the end customer pays for a fairly expensive good), low costs, and time–to–market issues. Thus, there the demand for flexible design methods is extremely high. In hardware design, the problem of automatic synthesis of circuits is well defined, so that numerous synthesis techniques and tools have been developed. This is not the case for general software design. However, the type of algorithms that most automotive applications implement is relatively simple from a computational point of view, which makes it possible to address the problem of effective software synthesis. Moreover, the automotive sector is on the edge of a technological revolution, namely the appearance of a new generation of hardware platforms like 32–bit micro–controllers, that stirs a lot of interest from the industry for effective software–hardware codesign. For these reasons, we choose the automotive ES as a privileged area to explore the problems involved in software–hardware codesign and envision possible solutions.

In our experience, the design of embedded controllers suffer from the lack of a well–defined design flow to minimize errors and to improve time–to–market. Often the specifications are not given formally and are subject to continuous revisions. In addition, the decisions on how to implement the system are often taken a priori on the basis of experience or intuition resulting in suboptimal solutions. Hardware and software design proceed almost independently so that only at integration time system errors can be caught. Even if the micro–controllers that are the hart of embedded controllers have been constantly improving in terms of speed their power is not fully exploited since new control algorithms are difficult to analyze and to implement with the present design approach.

Embedded controllers are in general less complex than computers, or telecommunication apparati and, hence, are more amenable to formal design methods. The importance of developing these methods cannot be overemphasized: The number of designs is very high and it will increase dramatically over the next few years since most home appliances, transportation and industrial equipment will have at least one embedded controller that performs a specific task. Given the big production volumes of this type of goods, it is apparent that the consequences of a poor design choice can be dramatic in commercial terms. This, with the increasing complexity of the systems calls for a more structured approach to this problem.

The design methodology proposed here capitalizes on a set of results recently obtained in formal verification and automatic synthesis of FSMs. We emphasize the view that a general technique for system design including hardware–software trade–offs is not practical today. Our methodology is applicable to an important

class of real world designs but we do not claim that this technique could be used universally with success. We also point out that the theoretical basis for this work are known, the contribution here consists of putting together in a coherent frame a set of tools and techniques that are not used in the embedded controller design world. We believe that the choice of FSMs as the representation of the design is a most natural one for embedded controller design: in most of the applications we are aware of some form of FSM formalism is used to represent designs, whence the recent interest placed by designers on tools such as StateCharts from iLogix that provides a powerful editor for FSM description.

While automatic partitioning between hardware and software is a most interesting research topic, in the present form of our methodology, it is done interactively. In automotive applications, flexibility and cost are main issues. Hence, the ideal solution would be to implement as much of the application as possible on the micro-controller. If this solution is not feasible because of performance and capacity constraints, then parts of the control system is downloaded to some additional hardware (either a Digital Signal Processor, a Field Programmable Gate-Array or a gate-array) until the performance and capacity constraint are met. In this situation, it is important to have tools that map a given subalgorithm onto the chosen hardware accurately, efficiently and rapidly since this step is key to evaluate a given partition. Since the number of feasible partitions is limited, an interactive session is certainly possible.

3. A Design Methodology for RRTS

System Specification

The objective of this phase is to produce an algorithm that models the behavior of the system. In general, we need some entry language that enables us to describe the detail of the algorithm with a given degree of detail/abstraction. No single input language is absolutely better than another, thus we may want to be able to use several different languages. For all of them the semantics is defined in terms of a general mathematical model which provides the capability to put different types of specifications together. In other words, each description of the parts of the system must be unambiguously mapped into an intermediate format which is the input to the subsequent steps of the design process (verification, simulation, synthesis, etc.)

Notice that different languages may have different expressive powers. For example, we can have languages based on synchronous event-reaction like Esterel [3], or on non-deterministic dicrete selection/resolution like S/R [31], or on asynchronous event-reaction like Signal-Transition Graphs [7,8], or on Petri nets, or a generic language like VHDL (provided a subset with a formal semantics is used, as described in [5]), and so on. Each language is apt to specify a type of application (or *domain*) and the communication between domains is made possible by the underlying common semantic model which subsumes the expressive powers of all the languages used. This approach is different from the one used in other systems like Ptolemy [6] where each domain has its own model, although some similarities exist across domains. To allow communication between domains in their approach specific interfaces have to be written.

Proper expressiveness is the most imporant feature of a specification language. This does not mean that the best language is the most expressive one. In fact, a language which is too expressive can often turn out to be confusingly cumbersome. Even though we want to be able to describe the widest possible array of behaviors, we do not want to do this all at once and by using the same tool. As an analogy, consider the different types of domains and corresponding techniques used in specifying the parts of an automobile: the type of specification used for the engine mechanical design would be both redundant and useless for the description of the electrical connections. Therefore, each language should have a restricted expressiveness that allows to concentrate on the characteristic issues of a given domain. For example, it is pretty easy to design a pure controller using Esterel, but if in the same system some data processing needs to be done we had better turn to some other language.

Regarding the differences between information domains, a major distinction can be made between *control* and *data*. Some claim that this distinction is artificial and misleading, therefore advocating the use of description techniques that do not distinguish between the two. We believe that the distinction is certainly

arbitrary and depends on what models are used. When using a FSM–based model, it is necessary to draw a boundary between the two: control is everything that can be expressed directly by a FSM; data is everything which stays on the side. For example, consider the following specification:

```
if (ok(x)) then y=foo(x);
```

the `if...then` part is a control operation. `ok()` and `y=foo()` are data operations. These two aspects have different expressive needs.

In most cases, in a RRTS we can always distinguish among three basic parts: data acquisition, reaction, actuation. These three parts have different characteristics. In data acquisition we group together the computational (i.e. numerical) complexity that results in the ackowlegment of the occurence of some input events. In reaction (or decision) phase the system selects the output actions that appropriately respond to the input events. In actuation the actuators are driven. In general, these three parts belong to different domains, and will be specified with different techniques.

Another desirable characteristic of every input specification language, along with the expressiveness, is user–friendliness, that is the ability to make understandable what the description stands for. This is by no means a secondary aspect of the design process. Human ability to handle information is affected by the amount of information that has to be processed; the fewer the information, the easier it is to understand it. Therefore, economicity is a highly desirable feature of a language. Since economicity does not go with expressiveness, we need a wide array of languages of relatively limited expressiveness rather than a rich and tortuous labyrinth of intricacies such as a unique all–purpose language would be. In a sense, this idea is analogous to the concept of information hiding (as in Object Oriented programming and design methodologies.) In fact, in OO the quantity of information is organized into a structure that makes it tractable, whereas here the diverse qualities (or "flavours") of information are presented in different fashions. Roughly speaking, we can see each domain as an object type, and the general model as the super–class of which all domains are special cases. As to the specific languages that can be effectively used to specify a RRTS, it is only a matter of taste, as long as they fit in the general semantic model.

We believe that the underlying mathematical model should be based on *Multiple Interacting FSMs*. As better explained in [43], there are numerous advantages for using such a model, mainly in terms of simplicity, flexibility and expressiveness. Virtually every type of sequential behaviour can be expressed as a FSM of some kind. It is important to point out though that the FSM model, although fairly general, may not be practical in the case of more complex data processing algorithms for the size of the FSMs may grow too big. It is indeed perfectly suited for control algorithms where mainly signals are exchanged. An even stronger argument in favour of the use of FSM is that we have algorithms for sequential system manipulation (namely verification, optimization, code and circuit synthesis) that work on networks of synchronous FSMs.

The idea of representing RRTS as FSMs is not new. Several slightly different variations on this theme have been proposed. In the field of mixed software–hardware systems, different communities (e.g. CAD vs CASE) stress different aspects of the design problem. Several CASE product tend to address the problem of how to help a designer to come up with a set of meaningful specifications. Most CAD product tend to take the specification for granted and devise methods that automatically synthetize, verify, and test implementations. Both approaches are correct in their own place. Techniques like *Structured Analysis* [14] and the tools that implement it (for example *Cadre*'s *TeamWork*, *EDI*'s *Software Through Pictures*) can help in the very early stages of defining a set of specifications, but do not make for a formal unambiguous (i.e. compilable) description that allows the use of a number of design aids from verification to synthesis. A very interesting idea is to make Data Flow Diagrams formal in order to use them as a customizable input specification language, as shown in [15]. SDL is a formalism invented for software design that the authors claim can be effective for hardware design as well [40]. The communication between processes is based on a queuing mechanism. *Communicating Real–time State Machines* (CRSM), proposed by Shaw [41], are an attempt to introduce a similar notation which is also executable. Communication between FSMs is based on message passing. *Statecharts*, proposed by Harel and Druzinski [12,13], are a most interesting way to organize a FSM based

system description into a manageable hierarchy. Communication between FSMs in done by broadcasting. Commercial tool like *i–Logix's Statemate* and *Express VHDL* implementing this technique are already available. Wolf [9,10,11] proposes using FSM networks for behavioral synthesis of control–dominanted ASICs and systems.

In our approach, we propose the use of two layers of FSM–IF. The upper level (UIF) provides a convenient general formalism to combine the different types of specifications. In particular, it must be able to represent event/reaction as well as value/function semantics. The partitioning into implementation domains is done on this level. The lower level (LIF) is a network of synchronous Moore–like FSMs on which we actually do verification, simulation, and hardware synthesis.

In our model of UIF, a FSM (or *process*) is an entity that consists of a set of states and a set of transitions. A transition $u \rightarrow (^{t}/_{a}) \rightarrow v$ specifies that a system M will move from state u to state v when the event t occurs, and the action a will instantaneously be executed. The model of time is similar to that proposed by Berry [3]. It is assumed that nothing interesting happens between event occurrences, therefore the time is discretized into intervals of uninteresting length. Each interval is identified as being before some event and after some event. The model of parallel composition is based on broadcasting; each process can see the value of the states of every other process but can modify only its own. Each event can be sensed by every process and a given event can be emitted by different processes. However, a process is not sensitive to events it has emitted in order to avoid temporal paradoxes.

We can express such a system of FSMs using BLIF–MV [43], a very simple format whose base semantic is purely synchronous. BLIF–MV is an extension of the *Berkeley Logic Interchange Format* (BLIF) [44] . The base elements of BLIF–MV are multi–valued latches, and multi–valued functions. These elements can be nested and organized into a hierarchy by using a sub–system construct. The communication is based on broadcasting. At a meta–level (or interpretive level) it is capable of modelling synchronous as well as asynchronous behavior, delays, interleaving semantics as well as true concurrency through the use of non–determinism. The particular interpretation we are going to define, which we call *BLIF–MV, must be able to express the type of behaviour we have described above, and should also be suitable to be translated into software and hardware primitives.

In *BLIF–MV, we interpret every signal whose name begins by "*" as a discrete event. We impose that an event can take values 1 or 0 only. When as input, an event triggers the function that uses it. When as output, it is a (re)action that is executed synchronously (or in zero time) with the occurence of the input trigger(s). Non–event signals are interpreted as pure static values. Function can be classified into two types: *triggered functions* modify output values when the trigger occurs; *event transformers* produce actions depending upon input values when triggers occur; A triggered function can modify the same variable it accepts as input; the input is interpreted as the value before the trigger occurence, and the output is interpreted as the value after the trigger occurrence. Consequently, latches are not explicitly needed but can be used as a special case of triggered functions. An output action can be a trigger for another function. The model of time is discrete, thus it is admissible that several triggers occur at the same time. The output event can be selected non–deterministically (see Figure 2.)

The UIF is unambiguously mapped into LIF which represents a network of synchronous interacting FSMs whose behavior is consistent with the behavior of the upper layer. The LIF is technically a transparent implementation of the UIF and it is obviously not unique. At the LIF level, a discrete events is typically interpreted as a state of an event carrier flow. That is, if an event carrier reads 1, it means that the associated event is present (i.e. it occured sometime within the last clock cycle). If an event carrier stays 1 for n cycles in a row, this is interpreted as n successive occurrences of the event. This makes it possible to apply many FSM manipulation algorithms based on the Moore FSM model.

The main disadvantage of a FSM based model is the potential for an explosion in the size of the representation. In fact the size (number of states) of a FSM with n state variables each of cardinality k can grow as big as k^n. For this reason, we recommend that the model be based on multiple machines rather than on a single

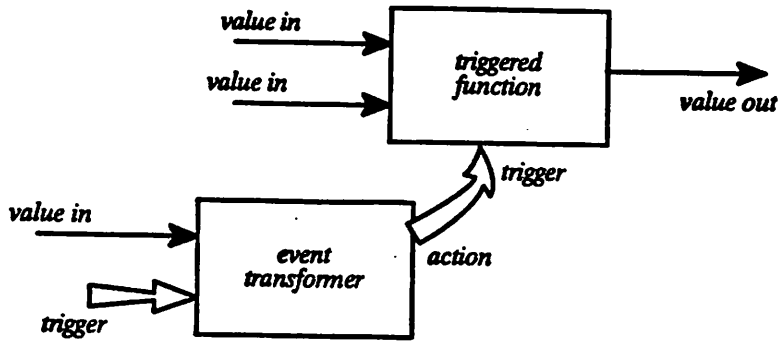

Figure 2. *BLIF–MV constructs

machine. That is, a system should be specified as a set of interacting components that should be kept separate (i.e. not explicitely collapsed.) However, the structure of the system can be changed, for example re–partitioned into different components, to fit better in a verification or implementation scheme. As a consequence, the FSM representation that can be associated to a given behavior is not unique. As for the LIF in particular, it is convenient to use an encoded format rather than a symbolic one.

The following example illustrates how a simple system specified across diverse domains can be specified using different languages and mapped into *BLIF–MV. Suppose we want to specify a simple safety function of an automobile: the alarm that beeps when the seat belt is not fastened. A typical specification, as it would be given to a system engineer, would be: *"Five seconds after the key is turned on, if the belt has not been fastened, an alarm beep will sound for ten seconds or until the key is turned off."* As a first step, the engineer would transform this sentence into a corresponding FSM like the one depicted in Figure 3. Assuming that a graphic interface (like *Statecharts* [12]) is not available, s/he would translate it into some formal language. For example, s/he could write the following pure (i.e. signals only are used) Esterel code:

```
input BELT_ON, END_10_SECONDS, END_5_SECONDS, KEY_ON, KEY_OFF;
output START_5_SECONDS, START_10_SECONDS, ALARM_ON, ALARM_OFF;
relation END_5_SECONDS # END_10_SECONDS # BELT_ON # KEY_OFF # KEY_ON;
loop
   await KEY_ON;
   do
      emit START_TIME;
      await END_5_SECONDS;
      emit ALARM_ON;
      await END_10_SECONDS;
      emit ALARM_OFF;
   watching [BELT_ON or KEY_OFF]
   timeout
      emit ALARM_OFF
   end
end.
```

Note that this behavior can be best modelled by an asynchronous semantics. Esterel is a language with synchronous semantics that can approximate asynchrony by imposing that some inputs cannot occur at the same time. This constraint is implemented by the statement

```
relation END_5_SECONDS # END_10_SECONDS # BELT # KEY_OFF # KEY_ON;
```

A *BLIF–MV mapping of this specification is

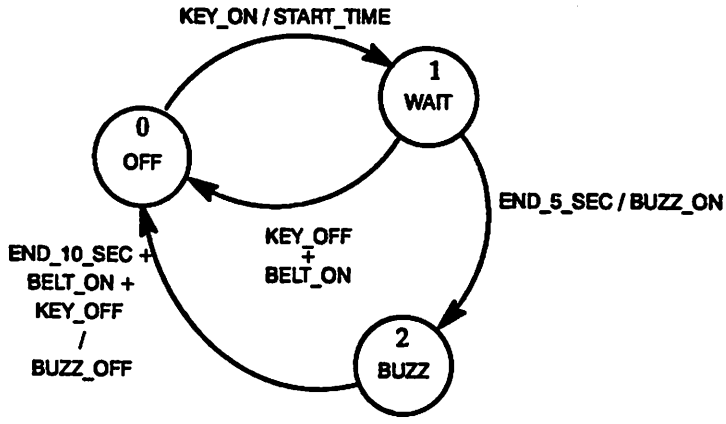

Figure 3. Belt example.

```
.module belt
.inputs *BELT_ON *END_10_SECONDS *END_5_SECONDS *KEY_ON *KEY_OFF
.outputs *START_5_SECONDS *START_10_SECONDS *ALARM_ON *ALARM_OFF
.mv __STATE 3

.names __STATE *KEY_OFF *KEY_ON *END_10_SECONDS *END_5_SECONDS *BELT_ON __STATE
0 - 1 - - -  1
1 1 - - - -  0
1 - - - - 1  0
1 - - - 1 -  2
2 1 - - - -  0
2 - - 1 - -  0
2 - - - - 1  0

.names __STATE *KEY_OFF *END_10_SECONDS *BELT_ON *ALARM_OFF
2 1 - -  1
2 - 1 -  1
2 - - 1  1

.names __STATE *END_5_SECONDS *ALARM_ON
1 1  1

.names __STATE *KEY_ON *START_TIME
0 1  1
```

Notice how __STATE appears as both an input and an output in the first function. The timer that counts 5 and 10 seconds, can be specified by a language that features both arithmetic capability and non-determinism to model the occurence of the external event SEC. A possible way to do this is by writing S/R code.

```
proc count_5_10
      import  s
      selvar  #: (0,SEC,END_5_SEC,END_10_SEC)
      stvar   n: integer
      init    0
    trans
      10          {END_10_SEC}
```

```
            ->0    : s=START_TIME
            ->n    : else;

      5             {END_5_SEC}
            ->0    : s=START_TIME
            ->a5   : true;

      a5            {0: SEC}
            ->6    : #=SEC;
            ->n    : else;

      true          {0: SEC}
            ->0    : s=START_TIME
            ->n+1  : #=SEC
            ->n    : else;

   end /* COUNT 5 AND 10 SEC */
```

A straighforward *BLIF–MV mapping of this trivial algorithm is

```
.model count_5_10
.inputs *SEC *START_5_SEC *START_10_SEC
.outputs *END
.mv i

.names *SEC *START_TIME i i
- 1 -  0
1 0 0  0
1 0 1  2
1 0 2  3
1 0 3  4
1 0 4  5
1 0 5  6
1 0 6  7
1 0 7  8
1 0 8  9
1 0 9  10

.names *SEC i *END_5_SEC
1 5  1

.names *SEC i *END_10_SEC
1 10  1
```

The semantics is immediate. The statement

```
.names *SEC i *END_5_SEC
1 5  1
```

in `count_5_10` means that, at each `*SEC` tick, `*END_5_SEC` must be emitted if the value of `i` is 5. The statement

```
.names __STATE *KEY_OFF *KEY_ON *END_10_SEC *END_5_SEC *BELT_ON __STATE
....
```

```
1 - - - 1  2
....
```

in `belt` means that the system makes a transition from `__STATE=1` to `__STATE=2` when `*END_5_SEC` occurs.

These *BLIF–MV procesess are mapped into LIF which is then validated through formal verification and simulation. The mapping from UIF to LIF is done by explicitely introducing latches triggered by a common clock tick. Every discrete events is associated to an event carrier line, abd the presence of the event is interpreted as a state (typically 1) of the carrier. BLIF or an immediate synchronous interpretation of BLIF–MV can be used as a LIF. A BLIF–MV synchronous mapping of the belt system is the following:

```
.module belt
.inputs *BELT_ON *END_10_SECONDS *END_5_SECONDS *KEY_ON *KEY_OFF
.outputs *START_5_SECONDS *START_10_SECONDS *ALARM_ON *ALARM_OFF
.latch __NEXT_STATE __STATE
.mv __STATE 3

.names __STATE *KEY_OFF *KEY_ON *END_10_SECONDS *END_5_SECONDS *BELT_ON __next_state
0 - 0 - - -  0
0 - 1 - - -  1
1 0 - - 0 0  1
1 1 - - - -  0
1 - - - - 1  0
1 - - - 1 -  2
2 0 - 0 - 0  2
2 1 - - - -  0
2 - - 1 - -  0
2 - - - - 1  0

.names __STATE *KEY_OFF *END_10_SECONDS *BELT_ON *ALARM_OFF
1 - - -  0
0 - - -  0
2 0 0 0  0
2 1 - -  1
2 - 1 -  1
2 - - 1  1

.names __STATE *END_5_SECONDS *ALARM_ON
1 1  1
0 -  0
2 -  0

.names __STATE *KEY_ON *START_TIME
0 1  1
1 -  0
2 -  0
```

Notice that new transitions have been added (highlighted).

A similar LIF mapping is made for the other FSM, that is the counter. We can now put these two FSMs together just by associating the signals by name. The resulting network of *BLIF–MV processes is a general representation of the behavior we expect from the system.

In this example, we explicitely coded a 10 step counter in S/R. In the real world it would be very unconvenient for a designer to do so. Instead, it would be useful to have libraries of pre-built functions, (like adders, counters, etc.) already optimized and mapped in LIF that the designer can instanciate by name expansion and use as sub-components in a larger design. In this way the network manipulation is done on the LIF description of the global system. The manipulation of the UIF, instead, must treat these blocks as black boxes.

Specification verification

Once a system has been specified, we want to be able to verify whether it satisfies a set of properties before we move to an implementation. This can be done by means of formal verification, simulation, and fast prototyping. All of these algorithms take as input a network of synchronous FSMs (i.e. the LIF), so the first step is to map the system into such a representation. No hypothesis is made so far on the type of final implementation. Whatever the result of the verification, it tell us something on how the behaviour has been specified. If the specification fails at this stage, it means that no implementation of it will work.

Among the possible approaches to formal verification of sequential systems [22], *model checking* of branching time temporal logic properties [35] seems to be the most viable one for an industrial design scenario. It subsumes other approaches, like language containment [29,30], and provides a base for more sophisticated techinques like real-time verification [36,37,38]. The verification of real-time properties, that is *quantitative* timing constraints, is the fundamental target of the research in this field. However, the complexity of the algorithms is still a major drawback.

For example, let us consider the seat belt example again. Suppose we want to verify that if the alarm goes on, it will eventially go off. We can express this property in CTL [17], a formalism to represent branching time temporal logic properties. The property above can be expressed in CTL as follows:

$$\forall G(^*ALARM\_ON \Rightarrow \forall F(^*ALARM\_OFF)).$$

The algorithm that checks this property, takes as input the LIF representation of the system, and computes the set of states that satisfy each sub-formula of the nested expression by examining the sequences of states that are consistent with the sub-formula. In this case, it first extracts the states in which *ALARM_ON is true and that lie on paths that take to a state where *ALARM_OFF is true, and then it extracts the states that lie on paths where all states have the former property. This property is clearly satisfied by this simple system.

The main obstacle to the formal verification of large industrial systems is the problem of the explosion of the size of the representation. In fact, most known algorithms for formal verification work on a single FSM whose size can grow exponentially in the number of variables. The use of implicit representations, like BDDs [18] and MDDs [19], allows to handle systems of greater complexity [23]. Yet, in many real cases these techniques cannot handle the complexity of the system. For this reason many reduction and minimization techniques are being considered [26,27,25,20,21,30,24,32,33,34].

Specification Simulation and Fast Prototyping

Simulation is particularly useful when the type of interaction with the external environment is well know. In this way we can provide sequences of inputs that the system can expect in the real world and check if the simulated system responds correctly. Simulation and verification are complementary aspects of the design process. In section 4 it is shown how the verification can give hints on what situations need to be simulated, and simulation gives feed-back to the verification.

By Fast Prototyping (FP) we mean building a program that behaves as a part of the intended system of which we want to have a flavour before we move forward to all other design phases. This is especially interesting when the system has a user interface of some sort, and it is mainly used to test the user interaction which cannot be evaluated in any way but trying it out. In other words, in most cases the user/system interaction cannot be measured but only experimented with. For the specific case of the belt system, it may be interesting

for the user to try whether s/he likes the timing sequence of the alarm. Is 5 seconds too short? Is 10 seconds too long? Unless there are specific regulations on the subject, the only answer can be an empiric one.

System Implementation and Partitioning

The problem of implementation is somehow dual to that of system specification: given a multiple FSM description how can we map it into a set of different modules chosing what type of technology and style of design (e.g. interrupt–based or sequential software) we should use for each of them, and how can we make them communicate with each other. Conceptually, for each of the possible implementation solutions we need an algorithm that automatically transforms a FSM into a given technology. As it is the case for system specification, the different types of implementations are not interchangeable. That is, a given function (part of the algorithm that describes the system) may or may not be mappable into a given implementation domain. For example, a recursive graph–traversal algorithm cannot be implemented as a circuit because hardware cannot be instantiated dynamically; a fast event–sampling function cannot be implemented as sequential software if the required sampling frequency is over a given ratio of the actual response time of the software.

In software–hardware mixed systems the software–hardware partition is a most critical phase. Traditionally, the partition was done in the early phases of the design process, which affects all the following steps. This can be a jump in the dark. For example, if in the early phases of the design of an automotive dashboard controller we decide that a given µC will be used, we set a limit on the amount of program and data memory we are allowed to use. If, for some reason, new functions need to be added, or we realize that the code that implements the required algorithm is too big, we may not be able to come up with a functioning product. In addition, in some fields (like some automotive products) the specifications are subject to frequent changes. Thus it would be useful to delay as much as possible making definitive decision on the structure of the system that one may not be able to undo.

In general, to implement a system as software as much as possible gives the most flexibility and is therefore preferable. This is even more true now that even almost every µP or µC can be programmed with standard languages (various C versions usually). Changes are made easier to make, and, if some coding conventions that ensure portability are observed, comparisons between implementations on different CPUs are made possible. In most cases, using a standard product like a µC will be the least expensive solution. However, software evaluation is a critical point in evaluating an implementation option. If it is easy to determine a priori whether a given µC has a I/O configuration that matches the system specifications, it is much harder to estimate if a program can fit in its memory, or if the software will run fast enough to meet all timing requirements. In most cases, it will be convenient to start with an all–software solution. If it works fine the job is done. If it does not, that is the program is either too slow or too big, it will be necessary to replace some sub–routines with hardware parts and try this new solution. The process is repeated until a satisfactory solution is found. Unfortunately, trying hardware is not as easy as trying software. In this respect, the use of hardware emulation machines (like *PIE* or *QUICKTURN*) introduces great flexibility. It gives the designer information on how a given partition is going to work. Therefore it can help to evaluate different possible partitions and implementation options before one actually builds prototypes.

In our method, we restrict our attention to one single implementation option: a synchronous system comprised of one or more µP's (or µC's) surrounded by synchronous sequental logic. There is one base clock for the entire system, although the clock used by the µP's can be a multiple of the base clock. We propose that algorithms be developed to translate the LIF into two standard intermediate specialized formats: BLIF for hardware parts, and a standard pseudo–language (PL) for software. The partition must be provided by the user who will divide the UIF or the LIF format into groups of elements. Each part will be input to a BLIF compiler or PL compiler that will automatically provide the interface to the other domain. A triggered function can be implemented in hardware as circuit composed of a multi–valued latch and a combinational logic block. In Figure 4 a "naive" implementation of a generic next state function is shown. Notice that in the circuit the potential non–determinism of the *BLIF–MV table (and therefore of the LIF) must be resolved. An event

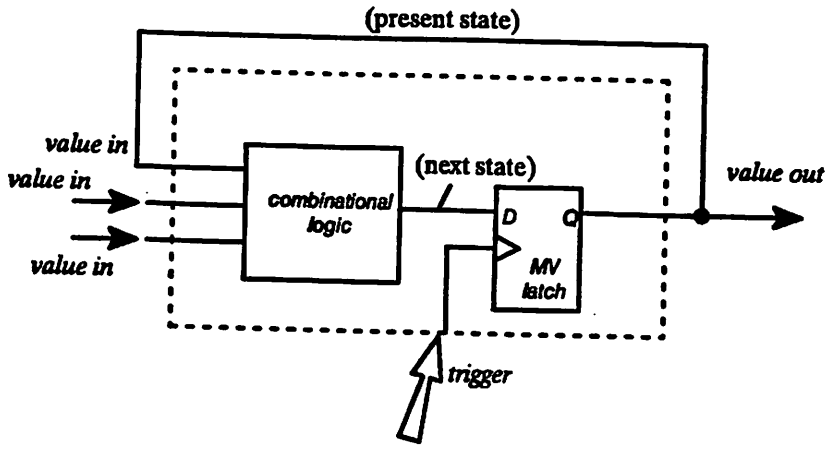

Figure 4.Implementation of a *BLIF–MV triggered function

transformer is implemented just as combinational logic thank to the discrete event LIF implementation. The software is synthetized in an explicit state coding style to allow direct translation from the FSM format.

A major and yet still largely unsolved problem is the communication across implementation domains. For example, consider the problem of synchronizing software and hardware implementations of parts of a *BLIF–MV description. The time being discretized, it is necessary to provide mechanisms to guarantee that contemporaneity of events in the specification (i.e. *BLIF–MV and LIF format) will imply contemporaneity of events in the real system. For example, it is the responsibility of the hardware compiler to check that the time span between two clock signals is long enough to allow the propagation of all events. But in software every single operation takes a discrete amount of time, therefore introducing "states" in the implementation that we did not specify in the specification format.

The discrete event model of communication of *BLIF–MV is aimed to help coping with this problem. *BLIF–MV functions communicate with each other by explicitely synchronizing on discrete events. The idea is to define a standard interpretation and implementation of discrete event detection and emission for each domain (namely software or synchronous hardware) and put standard interface blocks at the border between domains. For example, in a software implementation a trigger can either be an interrupt occurrence or the presence of a signal detected by polling a line, and an action is the statement execution of a (possibly complex) statement. In hardware, it is just a "1" (or a "0") detected on a line labelled as event carrier. An example of interfacing software and hardware modules in a discrete event model of communication is shown in figure 5. The event x emitted by the software module A is implemented as a simple sequence of two lines that set a bit and immediately resets it. This operation, on a typical µC like the Motorola MC68HC05, takes 4 clock cycles. The interface block *x_to_e* transforms it into a waveform that last for a single clock cycle. This works under the assumption that the software can output events at a rate slow enough. Conversely, the event s emitted by the hardware module B is latched in the interface block *s_to_f* until the software module acknowledges it. If the event f were of higher priority, it could be fed to an interrupt line therefore eliminating the need for an explicit interface module.

A further step will be to map the intermediate specialized formats into actual industrial technologies (i.e. assembly code for a given CPU, specific FPGA cells, specific DSP code, etc.) This can be seen as an extension of the problem of technology mapping for hardware.

Once an implementation has been done, it is still useful to check whether the result is consistent with the initial specification. This problem is called *Implementation Verification*. In a hardware system, implementation verification can be done relatively easily, at least in principle, by checking FSM equivalence. However, the size of a real system can make the verification impossible even though algorithms and programs do exist.

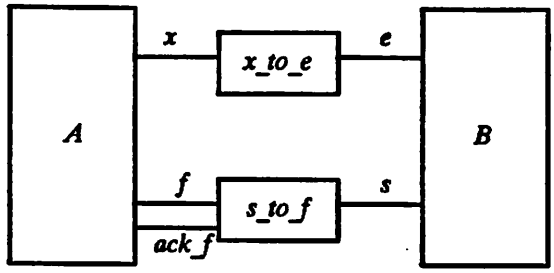

Figure 5. Interfacing heterogeneous modules

When we deal with software, it is even harder since the complexity of even the simplest programs can be much greater as compared to that of circuits. In addition, at the implementation level abstraction techniques based on non−determinism cannot be applied. As a matter of fact, very few simple systems can be effectively verified by the existing tools. We believe that the approach to implementation verification of mixed systems cannot be separated from the general problem of managing the complexity of large systems. That is, only with viable verification techniques based on compositional verification [32] or compositional minimization [24,25] we can think of performing implementation verification on real industrial systems.

4. An Example: CAN Controller

As an example of a non−trivial RRTS, consider an automotive in−vehicle network. In−vehicle networks are being introduced to replace the electrical wirings of cars which have become a major concern for car makers. The complexity of electrical wiring on cars has increased steadily since the late 1970s at a rate of 5−10% per year. An average car produced in the 1990s carries about 4 km of wires. This is mainly due to the increasing introduction of computer−controlled functions, (i.e. engine control, emission control, brakes control, etc.) which come with their sensors and actuators. This gives rise to a number of obvious problems, like increased weight, increased assembly cost, risk of erroneous connections. In−vehicle networking is being intoduced to get over these inconvenients. Automotive networks are classified into three classes, according to the time constraints of the application they serve:

Class	Application	Latency time	Bit rate
Class A	Body electronics (e.g. locks,	10 − 50 ms	< 10 kbit/s
Class B	Information sharing (e.g. diagnostics)	1 − 10 ms	10 − 125 kbit/s
Class C	Real−time control (e.g. engine control)	< 1 ms	0.125 − 1 Mbit/s

Several protocols have been proposed to implement automotive networks. Among them J1850 developed by SAE [45], and CAN (Controller Area Network), developed by Bosch [46], which we use here as an example.

The CAN network architecture consists of a number of nodes that communicate with each other by broadcasting messages on a serial bus. Although the nodes are not initially synchronized, the system can be modeled as a synchronous system since they use the same baud rate and re−synchronize periodically on edges detected on the bus. A major characteristic of the bus protocol is the automatic arbitration of transmission collisions, which is important for safety critical reactive systems such as automotive applications. Whenever a node wants to begin a transmission, it has to wait for the bus to be available. The bus is available whenever an end−of−message (EOM) sequence is detected (seven bits at "1"). As soon as the EOM is detected the node

sends a start–of–message (SOM) (one bit at "0"), and then begins to transmit a message ID (eleven bits where the seven "1"s sequence is forbidden). One or more nodes can start the transmission at the same time. When a collision arises (that is, some nodes are writing "1", and some are writing "0") the bus reads "0". In this case, the nodes writing "0" do not detect the collision and continue the transmission, whereas the nodes transmitting "1" switch to receiving mode. Since no two nodes share the same IDs, the collisions are automatically arbitrated, and only one node completes the transmission of the ID and can continue with the transmission of the full message. In this way the bus is never idle with pending requests.

The system described here implements the part of ID transmission and arbitration of one CAN node. The structure of a node is shown in Figure 6. The node is organized into three functions: a main unit *node_control* which controls the activities of the other functions, *detect_eom* which monitors the bus to detect a EOM, and *id_register* which outputs the 11 bits of the ID and monitors the bus to check whether a collision has occurred.

These subsystems have different characteristics and can be effectively described by different formalisms. The module *node_control* is a typical reactive element and can be easily specified in pure Esterel. Anyone who knows the Esterel language will notice that signals TX_ON, RX_ON, SENT_MSG, REC_MSG have been "latched" to states which are redundant to the specification of the behavior. The reason is that Esterel is based on a Mealy–like FSM model, and the Esterel–to–BLIF compiler does not use the same representation of discrete events as we need for LIF. Therefore we need to force these states to appear in the BLIF implementation.

```
module node_ctrl:
input REQUEST, EOID, EOM, COLLISION;
output TX_ON, RX_ON, START_ID, SENT_MSG, REC_MSG;
  loop
    await REQUEST;
    await EOM;   emit START_ID;          % at EOM, start tx ID
    await tick;  emit TX_ON;             % tx begins
    await [COLLISION or EOID];           % tx ID until EOID or COLLISION
    present COLLISION then
      await tick; emit RX_ON;            % tx interrupted, rx begins
      await EOID;                        % rx ID until EOID
      await tick; emit REC_MSG;          % rx completed
    else
      await tick; emit SENT_MSG;         % tx completed
    end
end.
```

The other parts of the system have also been specified in Esterel. For this example, only

For the remaining steps, we are going to use the *sis* synthesis system [42], which has been extended with a CTL Model Checker [34], a capability for the formal verification of CTL formulae. Since so far *sis* accepts BLIF as input format, each part of the system is compiled into a BLIF description. The connections between the various modules are resolved by name and the resulting output is the LIF representation of the entire system. The fact that BLIF is mainly a hardware description format is irrelevant. We use it in this example because it is the only format through which the Esterel compiler and the *sis* system can communicate.

Before we move forward to attempt an implementation, we need to verify if the behavior we have specified is the behaviour we expect. Some properties we want to verify on this network are: (1) whenever a node attempts a transmission (TX_ON), it will eventually complete it (SENT_MSG) or will receive an ID from another node (REC_MSG), (2) one node will never complete a transmission and a reception at the same time. These properties can be expressed in CTL as follows:

(i) $\forall G(TX\_ON \Rightarrow \forall F(SENT\_MSG + REC\_MSG))$

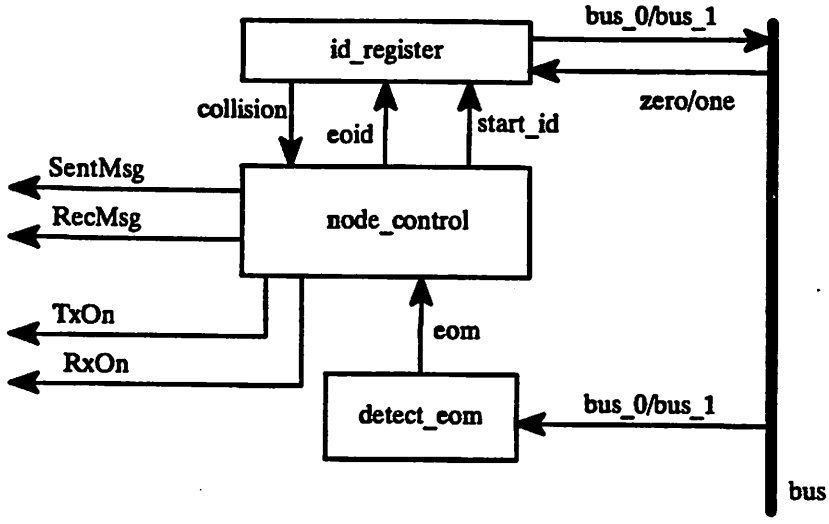

Figure 6. CAN controller.

(2) $\forall G \neg(SENT\_MSG \cdot REC\_MSG)$

If we were to consider a network of several nodes, we would have to verify properties concerning the interaction between them, like that two nodes will never complete a transmission at the same time. In addition, we could impose some fairness constraints. For example, we can require that, regardless of what point in time the system attempts a transmission, the EOM signal will be found true sometimes later, thus allowing the transmission to start (This is an example of *justice* property [2].) Also, it would be necessary to verify some real−time properties, like whether the specification satisfies a constraint on the maximum latency time. We are not exploring these issues in this example.

We start the session by calling the *sis* system.

```
cad 45: msis
UC Berkeley, SIS Release 1.1 (compiled 21-May-92 at 7:34 PM)
```

The Esterel source code has been previoulsy compiled into BLIF format.
We read the BLIF file into *sis*.

```
sis> rl can1.blif
sis> wl
.model can_one
.inputs BIT ZERO_EXT REQUEST
.outputs TX_ON RX_ON SENT_MSG REC_MSG
....
(blif format omitted)
....
```

Next, we invoke the CTL model checker to verify whether the system satisfies properties 1 and 2 above by calling the `_mc` command. The file `can1_1.ctl` contains the two CTL formulae.

```
sis> _mc can1_1.ctl
Reading encoding info ... done: 5 fsms, 34 bits
Computing fsm_reach()... done: TRs are NOT EQUAL. ini=166450, fin=2444
```

Notice that by computing the set of reachable transitions (`fsm_reach`) the size of the system is dramatically

reduced. In this case the reachable FSM is about 68 times smaller than the original machine.

```
M->name: can1
M->Inital states:
    cube:
    #  1  ---0100100000000000011000000010000000----------------------------

CTL Compositional Model Checker
Current options:
  * CTL Input File [ can1_1.ctl ]
  * Debug
  * Product machine
  * Reachability computation

Verifying formula:
 TX_ON=SH140_; ->(!(EG(!( SENT_MSG=SH157_; + REC_MSG=SH154_; ))))

....
(session omitted)
....

Final set Q:

cube: (size = 4)
    #  1  ----------------------------------0----------------------------
    #  2  ----------------------------------10---------------------------

mc OK! The initial state satisfies the property.

Continue? [y/n]: y

Verifying formula:
!((E[ f=1; R(!( TX_ON=SH140_; ->(!(EG(!( SENT_MSG=SH157_; + REC_MSG=SH154_;
   ))))))])+(!( TX_ON=SH140_; ->(!(EG(!( SENT_MSG=SH157_; + REC_MSG=SH154_;
   )))))))

....
(session omitted)
....

Final set Q:
    cube: 0123456789ABCDEF (size = 195)
    #  1  ---00000000000000----------------------------------------------
    #  2  ---000000000000010---------------------------------------------
    #  3  ---000000000000011000------------------------------------------
    #  4  ---00000000000001100100000000000-------------------------------
    #  5  ---00000000000011001000000000000101----------------------------
    ....
    (list of 625 cubes omitted)
    ....

mc FAULT! The initial state DOES NOT satisfy the property.

Continue? [y/n]: n
```

```
sis>
```

The mutual exclusion between reception and trasmission is verified (property 1.) But it is not guaranteed that a transmission or a reception will be completed once started (property 2). To find the cause, a litte simulation can help.

```
sis> print_stats
can_one        pi= 3   po= 4   nodes=300   latches=34
lits(sop)= 707   lits(fac)= 707
sis> simul -ns -f can1_1.sim
Network simulation
(Inputs: EXT_ZERO REQUEST BIT)
(Outputs: TX_ON RX_ON SENT_MSG REC_MSG)
Inputs:  Outputs:
0 1 0    0 0 0 0   ♦ REQUEST
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    1 0 0 0   ♦ TX_ON
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 0 0
0 0 1    0 0 1 0   ♦ SENT_MSG
0 0 1    0 0 0 0
....
....
sis>
```

The simulation shows that the system can actually reach a state where SENT_MSG is emitted, but the input BIT (i.e. the implicit trigger of every bit received or transmitted) must be provided "sufficiently often." In fact, this specification works under the following fairness assumption: at least 11 bits must be read/written in order to reach the end of the transmission/reception. We can guarantee this assumption by imposing the stronger constraint that BIT will occur infinitely often (which is an "impartiality" constraint.) Since Esterel is a totally deterministic language, we had to provide a mechanism to guarantee the periodic occurrence of BIT. If we had used S/R we could have exploited the *pause* construct to express this behaviour. In a next version, our CTL model checker will handle fairness constraints.

```
sis> rl can1.blif
sis> _mc can1_1.ctl

CTL Compositional Model Checker
Current options:
```

```
   * CTL Input File [ can1_1.ctl ]
   * Debug
   * Product machine
   * Reachability computation
```

Verifying formula:

```
!((E[ f=1; R(!( TX_ON=SH140_; ->(!(EG(!( SENT_MSG=SH157_; + REC_MSG=SH154_;
   ))))))])+(!( TX_ON=SH140_; ->(!(EG(!( SENT_MSG=SH157_; + REC_MSG=SH154_;
   )))))))....

(session omitted)
....

Final set Q:
    cube: 0123456789ABCDEF (size = 2)
    #  1  ----------------------------------------------------------------
mc OK! The initial state satisfies the property.

Continue? [y/n]: n

sis>
```

Now we try to synthetize an implementation. Since the system is very small we can use *sis* to synthetize a circuit. First, observe that BLIF is already a circuit description in a specialized format. However the BLIF output generated by the Esterel compiler produces a redundant circuit. We minimize it, and check that the implementation is still consistent with the specification. First, we read the system description.

```
sis> rl can1.blif
sis> ps
can_one          pi= 2  po= 4  nodes=300  latches=34
lits(sop)= 707  lits(fac)= 707
```

We build a state−transition graph from the BLIF format ...

```
sis> stg_extract -e
Total number of states = 155
Total number of edges = 220
Total time = 4.82
Checking to see that the STG covers the network...
```

... and minimize it.

```
sis> state_minimize
Running stamina, written by June Rho, University of Colorado at Boulder
Number of states in original machine : 155
Number of states in minimized machine : 122
```

Then, we extract a new circuit description from the minimized STG.

```
sis> state_assign
Running nova, written by Tiziano Villa,  UC Berkeley
sis> ps
can_one          pi= 2  po= 4  nodes=163  latches= 7
lits(sop)=1270  lits(fac)=1270  #states(STG)= 122
```

The new circuit has fewer nodes (163 out of 300) and fewer latches (7 out of 34). Finally, we verify the implementation. That is, we check whether the minimized circuit has the same sequential behavior as the original one.

```
sis> verify_fsm can1.blif
The finite state machines have the same sequential behavior.
sis> quit

cad 46:
```

Finally we could produce a real–world technology implementation, for example by using the "xilinx" library of functions embedded in *sis*. Or we could produce a software implementation by running the *occ* program which compiles Esterel into C code. But the discussion of these further steps would be beyond the scope of this paper.

In this case we have worked only on a fairly simple part of a larger industrial system, and the implementation verification step consists only of applying the *fsm_verify* procedure to check whether the optimized circuit is functionally equivalent to the one generated by the BLIF input. This was possible because the initial description and the implementation were given in the same format.

5. A Case Study: Engine Control Unit

The process of choosing an implementation solution for a mixed software–hardware systems is affected by a number of interrelated elements whose nature is often non–technical and hardly quantifiable. The type of function that needs to be implemented determines the algorithm, which determines the possible implementations. Considerations of a more commercial order (i.e. cost, time–to–market, and manufacturability), affect the spectrum of possible implementations, which in turn affect the algorithms that can be considered. We discuss the case of an Engine Control Unit prototype, developed by MAGNETI MARELLI in the CMA project [48].

An Engine Control Unit (ECU) is an electronic device whose task is to control the torque produced by the engine. This is done by controlling the duration and timing of the fuel injection and timing of the spark as a function of the physical state of the engine. Traditionally, (i.e. over the last 5–10 years) an ECU was an open loop system using an empirical look–up table to derive its control output. Engine speed and mass air flow/manifold pressure are sampled and the resulting data are fed to the processor. Look–up table addresses are derived from this information, and the look–up table data are then used to modify spark timing and fuel injector pulse width (see Figure 7.) An example of this type of ECU is the MM IAW 06 [47], which is based on the µC Motorola 68HC11. The advantages of such a system are its simplicity and its low cost. However, this type of solution has less then optimum operating characteristics and cannot compensate for errors due to manufacturing variability, fuel quality, aging, engine wear, and transient conditions such as "cold" engine or steep accelerations.

One major cause of bad fuel efficiency it that the real volume of each cylinder of the same engine can vary up to ± 10%. For example, to have a precise value of the amount of fuel to be injected into a specific cylinder of a 4 cylinder engine with maximum speed of 6000 RPM, it would be necessary to know the actual volume of the cylinder and compute the output every 5ms or less. A traditional ECU program has a response time of approximately 15ms. Therefore the injection time is computed as a function of an average non–optimal value, and is never exact. To compute the exact injection time for each cylinder the ECU should be able to elaborate the input data and access the look–up table every 6ms or less. Another cause of bad fuel efficiency is the engine wear that causes the actual volume of cylinder to vary over time. Since the variation of the actual volume of the cylinder can not be measured, if we want to keep the performance of the engine constant over time, the air/flow must be computed in a different way.

The target of CMA project was to develop an ECU powerful enough to be used as a platform to experiment with new control algorithms that improve the operating conditions of the engine, especially in terms fuel efficiency and emission level. The system has been sub–divided into three parts as follows: a data acquisi-

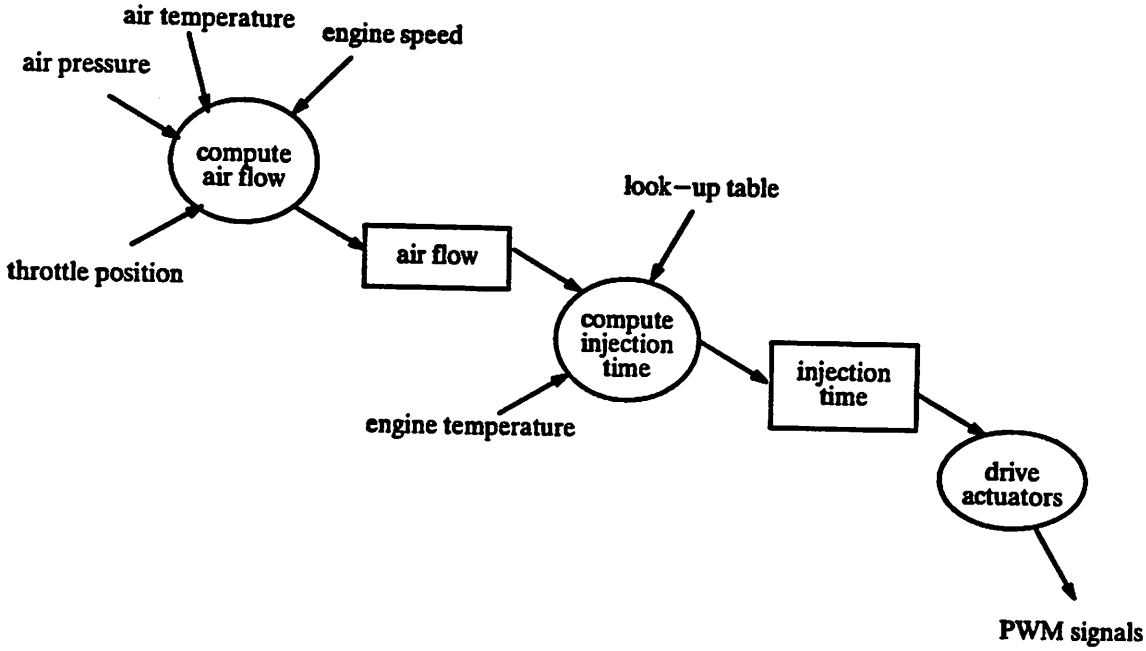

Figure 7. Simplified Data Flow Diagram of an ECU

tion part computes the air flow; a reaction part applies the kernel of the control algorithm; an actuation part consists of producing a number of PWM signals to drive electrovalves controlling the fuel injection, the canister, the compressor turbine (in turbo versions), and the VAE valve. The frequencies needed are in the range 10Hz–512Hz.

Initially, the proposed algorithm would be an open loop one where the volumetric efficiency of each cylinder is dynamically computed before every single ignition by sampling the air/flow, measured by a debimeter, every 2ms during the aspiration stroke. The actual volume is computed by integrating the flow over the duration of the aspiration stroke. The implementation of such algorithm would not have been possible on any standard µC without adding an extra unit, like a DSP (the Texas Instrument TMS320 was considered for the job). A cost analysis ruled this solution out for the time being. Instead, it was estimated that the main 32–bit CPU (Motorola 68332) would be capable of running an open loop algorithm with an expected response time of 4ms which computed separate outputs for each cylinder.

The generation of PWM signals needed by the actuators is a trivial task that can absorb a lot of CPU time. Therefore, it would be best implemented by an FPGA which has been developed in–house for that purpose. As a matter of fact, as opposed to the initial plan, the engine on which the CMA prototype will first be mounted will not need any of the PWM signals generated by the FPGA. The µC Motorola 68332 comes with a PWM generator which will be used for the main fuel injector.

This experience shows how four different implementation scenari would be possible for different versions of the same system: (1) CPU only, (2) CPU and DSP, (3) CPU and FPGA,(4) CPU, DSP and FPGA. The actual decision was mainly made on grounds of cost. The initial intended specification was dismissed in favour of a simpler solution that is yet a sensible advancement with respect to the existing product.

6. Conclusions

We have outlined a design methodology for the restricted area of RRTS.

We propose a framework where an array of specification languages and design tools can interact via FSM based intermediate formats which subsume the expressiveness of all languages used. In the field of system specification, we advocate the use of different languages in each of which a relatively limited expressiveness is traded off for economicity. In this manner, we can handle each face of the complexity of a system in a simple way, and nevertheless build specifications of highly complex systems.

The common underlying FSM based semantic model allows both formal verification and behavioral simulation of the system. The objective is to refine the *system* description as much as possible before an implementation is attempted.

The common FSM based model is a convenient way to address the issue of implementation and partitioning since every type of implementation can be derived from a subset of that model. The problem of trying different system partitioning and evaluating the trade–offs is reduced to trying different system composition schemes, and applying a technology mapping approach to each component of this scheme.

Among the unsolved problems, there is the issue of how to make the different implementation domains communicate. Moreover, how and to what extent the evaluation of different implementation options can be done automatically, is still an open question. However, by using emulation machines this task can be made easier for the user.

References

[1] D. Hare, A. Pneuli, "On the Development of Reactive Systems," *Logics and Models of Concurrent Systems*, NATO ASI series,, vol. 13, K. R. Apt ed., New York: Springer–Verlag, pp.477–489, 1985.

[2] A. Pneuli, "Application of Temporal Logic to the Specification and Verification of Reactive Systems: A survey of Current Trends," in: J. W. De Bakker, W. P. De Roever, and G. Rosenberg, eds., *Current Trends in Concurrency: Overviews and Tutorials*, LNCS, Vol. 224, 1986.

[3] G. Berry, P. Couronn\'{e}, G. Gonthier, "Synchronous Programming of Reactive Systems," In *France–Japan Artificial Intelligence and Computer Science Symposium*, 1986.

[4] A. Benveniste, G. Berry, "The synchronous Approach to Reactive and Real–Time Systems," Procceding of the IEEE, Vol. 79, no. 9, sept. 1991.

[5] W. Baker, "A Synchronous Semantics of VHDL for Synthesis and Verification Applications," UCB/EECS 290H Project report, Dec. 1991.

[6] J. Buck, S. Ha, E. A. Lee, D. G. Messershmitt, "Ptolemy: A Platform for Heterogeneous Simulation and Prototyping, " In *Proc. European Simulation Conf.*, Copenhagen, June 1992.

[7] T. A. Chu, "Synthesis of Self–timed VLSI Circuits from Graph–theoretic Specifications. PhD thesis, MIT, June 1987.

[8] \bibitem {STG2} L. Lavagno, C. W. Moon, R. K. Brayton, A. Sangiovanni–Vincentelli, "A Nover Framework for Solving the State Assignment Problem for Event–based Specifications," UCB Memorandum No. UCB/ERL M92/19, 1992.

[9] W. Wolf, "The FSM Network Model for Behavioral Synthesis of Control–Dominated MAchines," ICCAD, November, 1990.

[10] W. Wolf, "An Algorithm for Nearly–Minimal Collapsing of Finite–State Machine Networks," DAC 1990.

[11] \W. Wolf, A. Takach, C. Huang, R. Manno, "The Princeton University Behavioral Synthesis System," DAC, Anaheim, June 1992.

[12] D. Hare "Statecharts: A Visual Formalism for Complex Systems," In *Science of Computer Programming*, North–Holland, 1987.

[13] D. Druzinski, D. Hare "Using Statecharts for Hardware Description and Synthesis," IEEE Transaction on Computer–Aided Design, Vol. 8, No. 7, July 1989.

[14] P.T.Ward et al., Structured Development for Real–Time Systems. New York: Yourdon Press, 1985, 1986.

[15] A. Fuggetta et al., "Formal Data Flow Diagrams," IEEE Transactions on Software Engineering, Feb. 1986.

[16] \bibitem {BPM83} M. Ben–Ari, A. Pneuli, and Z. Manna, "The Temporal Logic of Branching Time," Acta Inf., 20, pp. 207–226, 1983.

[17] E. M. Clarke, E. A. Emerson, and P. Sistla, "Automatic Verification of Finite–State Concurrent Systems using Temporal Logic Specifications," ACM Trans. Prog. Lang. Syst., 8(2):244–263, 1986.

[18] R. E. Bryant, "Graph–based Algorithms for Boolean Function Manipulation," IEEE Trans. Comput., C–35(8), 1986.

[19] T. Kam, R. Brayton, "Algorithms for Discrete Function Manipulation," A. Srinivasan, T. Kam, S. Malik, R. K. Brayton, In *Proc. ICCAD'90*, Santa Clara CA., 1990.

[20] O. Coudert, C. Berthet, and J. C. Madre, "Verification of Sequential Machines Based on Symbolic Execution," In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, LCNS 407, pp. 365–373, 1989.

[21] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni–Vincentelli, "Implicit State Enumeration of Finite State Machines using BDDs," ICCAD, November, 1990.

[22] A. Gupta, "Formal Hardware Verification: A Survey," CMU–CS–91–193, 1991.

[23] J. Burch, E. Clarke, K. McMillan, and D. Dill, "Sequential Circuit Verification using Symbolic Model Checking," DAC, June, 1990.

[24] S. Graf and B. Steffen, "Compositional Minimization of Finite State Systems," In *Proc. 2nd Conference on Computer–Aided Verification*, New Brunswick, NJ, USA, LCNS 531, pp. 186–196, 1990.

[25] E. M. Clarke, D. E. Long, and K. L. McMillan, "Compositional Model Checking," In *Proc. of the 4th IEEE Symposium on Logic in Computer Science*, Asilomar, CA, June, 1989.

[26] P. Wolper and V. Lovinfosse, "Verifying Properties of Large Sets of Processes with Network Invariants," In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, LCNS 407, pp. 68–80, 1989.

[27] R. P. Kurshan and K. L. McMillan, "A Structural Induction Theorem for Processes," In *Proc. 8th ACM Symp., PODC*, 1989.

[28] Z. Har'El, R. P. Kurshan, "Software for Analytical Development of Communication Protocols," AT\&T Technical Journal, Jan. 31, 1990.

[29] E. Clarke, I. Draghicscu, R. P. Kurshan,"A Unified Approach for Showing Language Containment and Equivalence between Various Types of ω–automata," CMU–CS–89–192, 1989.

[30] R. P. Kurshan, "Analysis of Discrete Event Coordination," In *Lecture Notes in Computer Science*, 1990.

[31] Z. Har'El, R. P. Kurshan, "COSPAN User's Guide", AT\&T, Oct., 1987.

[32] O. Grumberg and D. E. Long, "Model Checking and Modular Verification," In *CONCUR, Amsterdam, The Netherlands*, August 1991.

[33] E. M. Clarke, O. Grumberg and D. E. Long, "Model Checking and Abstraction," In *Proc. Principles of Programming Languages*, January 1992.

[34] M. Chiodo, T. R. Shiple, A. Sangiovanni–Vincentelli, and R. K. Brayton, "Automatic Reduction in CTL Compositional Model Checking," In *Proc. CAV'92, 4–th Workshop on Computer Aided Verification*, Montréal, Canada, 1992.

[35] E. Allen Emerson, "Temporal and Modal Logic," In *Handbook of Theoretical Computer Science*, editor J. van Leeuwen, Elsevier Science Publishers B.V., pp. 995–1072, 1990.

[36] D. Dill, "Timing Assumptions and Verification of Finite–State concurrent Systems," In J. Sifakis, ed. *Automatic Verification Methods for Finite–State Systems*, LNCS 407, 1989.

[37] R. Alur, C. Courcubetis, D. Dill, "Model Checking for Real–Time Systems," In *Proc. 5–th IEEE Symposium on Logic in Computer Science*, pp. 414–425, 1990.

[38] F. Balarin, A. Sangiovanni–Vincentelli, "Formal Verification of Timing Constrained Finite–State Systems," In *Proc. CAV'92, 4–th Workshop on Computer Aided Verification*, Montréal, Canada, 1992.

[39] \bibitem {ESTERELHW} G. Berry, "A Hardware Implementation of Pure Esterel," 1986.

[40] \bibitem {SDL} "SDL".

[41] A. Shaw, "Communicating Real–Time State Machines," Technical report CSE No. 91–08–09, University of Washington 1991.

[42] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni–Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," submitted to ICCD, 1992.

[43] R.K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R.P. Kurshan, S. Malik, A.L. Sangiovanni–Vincentelli, E.M. Sentovich, T. Shiple, H.Y. Wang, "BLIF–MV: An Interchange Format for Design Verification and Synthesis" UBC Memorandum No. UCB/ERL M91/97, nov. 1991.

[44] "Berkeley Logic Interchange Format," UCB–EECS, Jun., 1991.

[45] "Road Vehicles – Serial Data Communication for Automotive Applications," Draft Proposal of ISO/TC22/SC3/WG1, Sep., 1990.

[46] D. Arnett, "A High Performance Solution for In–Vehicle Networking – 'Controller Area Network (CAN),'" SAE Technical Paper Series, 870823, April 1987.

[47] "Progetto IAW 06," MAGNETI MARELLI Divisione Elettronica, Torino, Italy.

[48] M.R. Albertocchi, R. Ferrari, G. Marchesi, W. Nesci, G. Uliana, "Progetto Controllo Motore Avanzato," MAGNETI MARELLI Divisione Elettronica, Pavia, Italy, 1992.