Synthesis of Mixed Software-Hardware Implementations from CFSM Specifications

Massimiliano Chiodo, Paolo Giusto, Harry Hsieh,
Attila Jurecska, Luciano Lavagno,
Alberto Sangiovanni-Vincentelli

June 1, 1993

Abstract

Embedded controllers for reactive real-time applications are implemented as mixed software-hardware systems. In [CLJ+93], a formal specification model called Co-design Finite State Machine (CFSM) is introduced. In this paper we present a methodology for partitioning and automatic synthesis of software-hardware systems specified with CFSMs. CFSM networks are partitioned into software and hardware domains. Interfaces between hardware and/or software partitions are defined. An implementation of the entire system is automatically generated and optimization techniques are applied to both software and hardware. To reduce the complexity of the design and the representation, library modules can be predefined and reused. An example from the automotive industry is used to demonstrate the method.

1 Introduction

Embedded controllers are used more and more frequently in the design of electronic systems as the increasing complexity of the applications requires the development of software to implement more sophisticated functions. Since timing requirements may not be met by the software, part of the design has to be implemented in hardware.

As pointed out in [CSV92], mixed Hardware-Software systems design is still in its infancy as a recognized discipline, although it has been common practice in industrial systems design for decades. In addition, the range of systems which require the joint design of hardware and software is so wide that 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. Within this field, we can recognize several critical aspects that are found in the process of designing an embedded system.

1. **Need for natural specification**: we want to be able to concentrate on specifying a function or a part of a system in the most natural possible way without being distracted by implementation concerns. Sometimes, engineers are so familiar with a time-honored implementation that the implementation itself becomes part of the specification. Hence, when a new technology becomes available they cannot take advantage of the opportunity because they do not know how to specify the problem in a non-implementation biased way.

2. **Need for easy partitioning**: we want to be able to explore a wide range of implementation solutions, and choose the one that best suits us in terms of cost, performance, manufacturability, and so on. Too often, the partitioning is done as part of the specification. That is, the initial system description is given in terms of what the software is supposed to do and what the circuitry surrounding the micro-P is going to provide. This can be acceptable when upgrading some old project without changing an established technology, but it is obviously not ideal when a totally new system has to be devised.

3. **Need for software and hardware synthesis**: since system implementation issues, in particular circuit design and test, software coding and debugging, are the most time-consuming activities and the main sources of errors, we want to trade off the optimality of a hand-made implementation for rapid turnaround and reliability.

4. **Need for optimization**: after partitioning, we want to optimize the proposed implementation. This step can only be performed if the behavior is expressed using a formal representation compatible with some optimization algorithm.

5. **Need for validation**: we want to validate both the model before partitioning and implementation, and the system after implementation. The importance of this step is often underestimated especially when designing "small" systems. Designers today seem more concerned with "getting something done" as soon as possible regardless of whether this something is what they really want.

To address the points listed above, we propose to use a non-committed behavioral internal representation based on *Co-Design Finite State Machines*, or CFSMs [CLJ+93]. By a synthesis point of view, the most desirable characteristic of CFSMs is that they can model a significant class of both hardware circuits and software programs. Therefore, they can be used as a representation of a *system's* behavior rather than of a circuit's or a program's.

The main points of our approach, which has been introduced in [CSV92], are:

1. An array of formal languages, each of which is mapped into a neutral FSM based *Intermediate Format* (IF), is used to specify different parts of a system.
2. The IF represents a network of CFSMs and is organized in a way that makes partitioning easy.
3. The IF has semantics in terms of both a hardware representation format and of a generic software architecture to allow for automated synthesis.
4. Optimization can be performed on both the IF, and on the synthesized code and circuit.
5. Validation consists of two tasks:
   (a) specification verification to verify that the specification satisfies a set of user-defined properties, and design verification to verify that an abstracted representation of an implementation satisfies a set of user-defined properties.
   (b) implementation verification to check whether the implementation's behavior is consistent enough with the specification's.

   Validation can be done by means of simulation or formal verification.

The CFSM model (like most FSM-based models) is not meant to be used directly by designers, due to its relatively low level (almost "bitwise") view of the world. Designers will conceivably write their specifications using a higher level language, for example ESTEREL ([BCG91]), StateCharts ([DH89]), Formal Data Flow Diagrams ([FG*et al.*86]) or a subset of VHDL ([Bak93]), that will by directly translated into CFSMs. This translation task is made easier by the formal definition of the model.

The basic idea is to use a network of interacting CFSMs, that communicate through very low-level primitives: *events*. The notions of "time" and "event" that we use are different from the perfectly synchronous model used, e.g., by ESTEREL. We use a *discrete* model of time, where each CFSM takes a *non-zero unbounded* time to perform its task (at least before an implementation is chosen). This model is quite realistic for synchronous systems and lends itself to efficient formal verification techniques (e.g. [Kur90], [Bur92]).

Events directly implement a *synchronous* protocol ([CKN86]) between communicating partners. The receiver waits for the sender to emit the event (which is essential to avoid reading the wrong information), but the sender can proceed after emitting the event without the need to wait. An implicit one-place buffer between the sender and each receiver saves the event until it is detected (or overwritten). This approach has two advantages: it lends itself to a very efficient hardware implementation with synchronous circuits, and can be used very easily to construct the *full handshake* (required, e.g., by a CSP channel or by asynchronous circuits) where the sender waits for the receiver to acknowledge reception of the event.

The notion of communication between CFSMs used by our proposed model implies that the sender does not "remove" the event immediately after emitting it, but only when it emits another

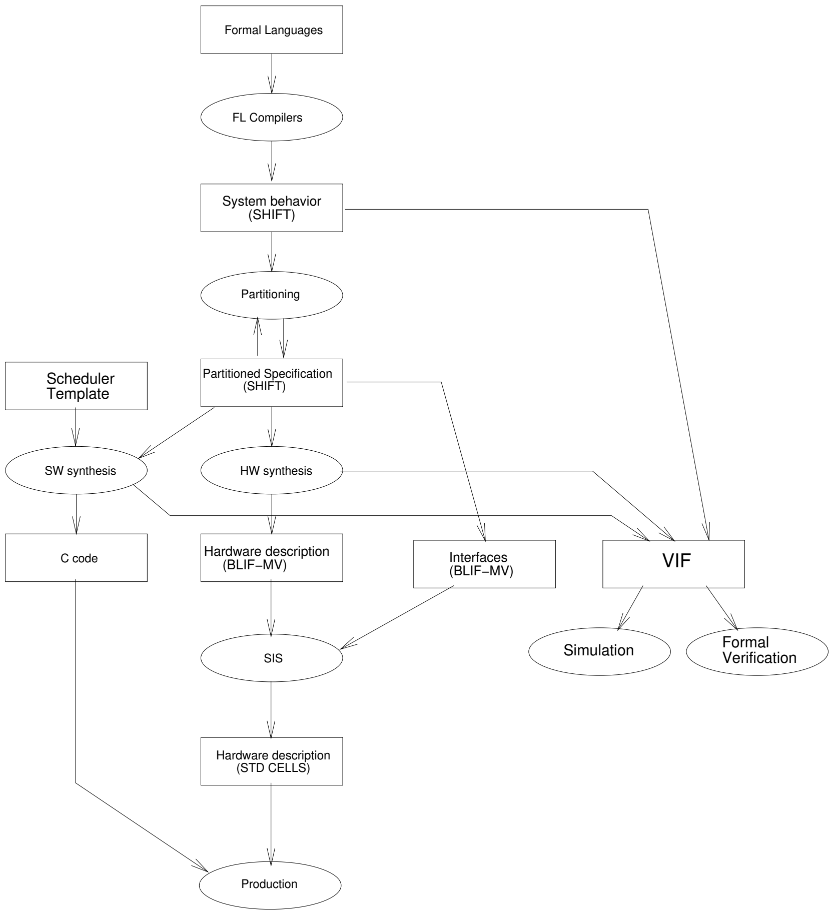

Figure 1: Co-design Framework

one. An event is present and can be detected not only at the time of its emission, but until it is either detected or "overwritten" by another event of the same type. So the event can be correctly received even with the rather unpredictable detection time that is associated with a software implementation. Correct reception is ensured as long as either the sender data rate is lower than the receiver's processing ability, or the designer explicitly introduces a full handshake between the two.

1.1 Overview of the co-design methodology

The design process starts when the designer inputs the system behavior description using one (or more, if different components require different styles) of the high-level languages that can be mapped into a network of CFSMs. We represent CFSM networks by a specific IF called *Software Hardware Intermediate FormaT*, or SHIFT (see Section 2.1).

Some formal properties of the specification, namely those that do not depend on the time required to perform each operation, can be verified at this very early stage using, e.g., model checking for temporal logic ([CLM91]) or language containment ([Kur90]) techniques, using a formal transformation from CFSMs to standard non-deterministic FSMs (denoted by "VIF" in Figure 1).

The second step is *design partitioning*, i.e. the choice of a software or hardware implementation for each component of the system specification. A key point of our approach is that the IF specification is totally implementation independent, and this makes it possible for the designers to experiment with a number of implementation options. Obviously, this flexibility of design can best be taken advantage of in a scenario in which the designer is also allowed to evaluate a possible implementation "on the fly". In this regard, the use of hardware emulation machines in addition to traditional micro-controller emulators is essential in order to exploit the full potential of our method.

The choice of a particular partitioning can be done by hand or by using automated techniques. Automated techniques may be based on I/O data rate requirements ([GJM92]) or on simulation/profiling data ([EH92]). This paper does not address directly the automated partitioning problem, but describes a framework where algorithms to solve it can be easily and transparently embedded.

It is important to remark that, by defining the communication between partitions on discrete event emission and detection, we provide a mechanism that *can help* to specify delay-insensitive partitions. For example, a hand-shaking mechanism can be easily specified in terms of discrete event exchange. In any case, we do not guarantee by any means that the partitions will actually be delay-insensitive. The validity of the system description is not ensured by construction but rather through explicit validation of the system model (i.e. verification and simulation.)

A major issue related to partitioning is how to interface CFSMs that belong to different implementation domains. In Section 6 we describe how a given partitioning affects the way the implementation is synthesized.

The third step is to implement each CFSM in the chosen style. The synthesis algorithms we propose are based on restrictions which are common to the design of most industrial embedded control systems:

- Each hardware partition is implemented as a fully synchronous circuit.
- Each software partition is implemented as a C stand-alone program embedded in one or more micro-controllers.
- All partitions have the same clock.

Hardware implementation is rather straightforward. After mapping a SHIFT specification directly into a netlist representation format, we use the *sis* sequential synthesis system ([SSL+92, SSM+92]) developed at U.C. Berkeley. Note that our unbounded non-zero discrete delay model also allows for pipelined implementations of the very same specification, of course if the designer provides some adequate means for synchronization between components with different degrees of pipelining.

The CFSM delay model plays a key role in making the software implementation possible. Writing a procedure that computes the next state and the output function given some input data and present state information is trivial. But this procedure will be scheduled to run at an almost unpredictable point in time and will perform its task requiring a number of clock cycles that is very hard to control. This type of behavior naturally leads to our definition of events as having a duration, until detected or overwritten, and to non-zero (but bounded) reaction times. Each CFSM is translated into a C function that is activated at the arrival of one (or more) event the CFSM is waiting for. The function computes the next state and, possibly, emits further events. This simplifies drastically the requirements on the real-time operating system that must coordinate the operation of the software components and implement the interface between different software components and between software and hardware (as described in Section 6).

The paper is organized as follows: Section 2 recall the CFSM theory thoroughly introduced in [CLJ+93], Section 3 describes the technique used to partition a system design, Section 4 describes the algorithms used to synthesize the hardware and software implementations, Section 5 describes how we handle system complexity, Section 6 describes how heterogeneous implementation domains are interfaced, Section 7 discusses the issue of software optimization, Section 8 discusses the issue of system validation, Section 9 shows an example of automated synthesis of a system from the automotive industry, and finally Section 10 draws some conclusions and outlines future developments of the project.

2 Background

In [CLJ+93], we thoroughly introduced the theoretical foundation of our approach to Hardware-software Co-Design. Here, we briefly recall the main points.

The basic observable entities that define the behavior of the system that we want to model are *events*. Sequences of events are called *traces*, and the behavior of the system is defined as the set of *traces* that can be observed when it interacts with the environment. The system itself and possibly its environment will be modeled using a set of CFSMs that produce those traces.

An event is defined as triple $e = (e_n, e_v, e_t)$. It is identified by its *name* e_n, or in other terms by the "communication port" on which it occurs. The event is associated with a value e_v on a finite set of possible values e_V. A particular instance of an event is identified by its *time of occurrence* e_t. Some events may not have an "interesting" value.

A *timed trace* is an *ordered* finite or infinite sequence of events, with monotonically non-decreasing times of occurrence, such that no two events with the same name are simultaneous (i.e. each "communication port" can carry only one value at a time).

A *network* of *Co-design Finite State Machines* (CFSMs) is a *finite* object that *generates* a set of timed traces by its evolution in time.

A CFSM $\mathcal{C} = (I, E, O, R, F)$ is basically composed of a set of input events I (each with its associated set of values), a set of output events O (each with its associated set of values and possibly with an initial value in R), and a transition relation F describing how the CFSM *reacts* to input events by causing output events to occur. Each transition is *triggered* by the input events with the appropriate values and *emits* the output events with the appropriate values. The *reaction time* is *unbounded non-zero*. The *state* of the CFSM is constituted by the set of those events that are at the same time *input* and *output* for it. The non-zero reaction time provides the "storage" capability that is required to implement the concept of state.

As a general rule, there is no need to specify a "self-loop" for a CFSM (i.e. a transition from a state to itself), because the "default" behavior is to remain in the present state as long as no events that can cause a transition arrive. A notable exception is the case described above, where without the self-loop specification the definition of "priorities" between events would not have been possible.

A Co-design Finite State Machine is like a "classical" FSM (from now on just FSM) because both transform a set of inputs into a set of outputs by using only a finite amount of internal state. However, a CFSM has no implied "synchronous" hypothesis. The standard definition of interaction between FSMs, based on the concept of *product machine*, assumes that all the FSMs change state exactly at the same time. This can be very different from the actual behavior of a mixed hardware/software system, in which software components can take hundred of clock cycles.

As a practical example, suppose that 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 given to a designer would be:

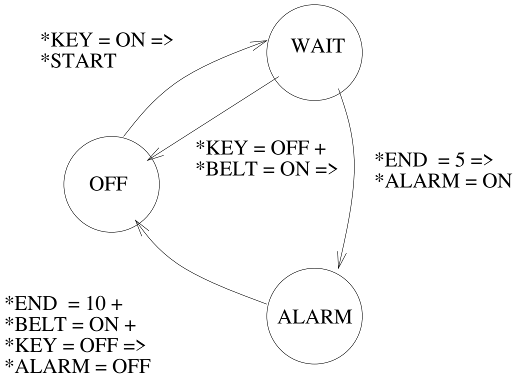

Figure 2: CFSM Specification of a simple system

Example 1 *Five seconds after the key is turned on, if the belt has not been fastened, an alarm will beep for ten seconds or until the key is turned off.*

This example will be used in the following sections of this paper.

The input events of the system are: `*BELT`, with values `ON` and `OFF`, `*KEY`, with values `ON` and `OFF` (from now on, names preceded by "`*`" will denote event names). The output event of the system is `*ALARM`, with values `ON` and `OFF`. Internal events (i.e. events exchanged by the system components) represent the starting of the timer and the fact that 5 or 10 seconds elapsed. They are: `*START`, without "interesting" values, and `*END` with values 5 and 10. A valid timed trace of the system, i.e. a possible sequences of events, can be expressed in tabular form, e.g.:

	0	1	6	7	9	10
`*BELT`					`ON`	
`*KEY`	`ON`					
`*ALARM`				`ON`		`OFF`
`*START`		ϵ		ϵ		
`*END`			5			

This trace describes a driver who does not fasten his seat belt. The alarm, triggered by the timer expiring after 6 seconds, is turned on at time 7. The driver fastens his seat belt at time 9 and the alarm is turned off after 2 seconds.

A CFSM describing the desired event/reaction pattern for the seat belt example is represented in Figure 2 ("`+`" denotes the logic *or* condition, while "`=>`" separates input and output events of a given transition).

The formal description of the same CFSM $\mathcal{C}_1 = (I_1, E_1, O_1, R_1, F_1)$ is as follows:

Example 2

- $I_1 = \{(\texttt{*KEY}, \{\texttt{ON}, \texttt{OFF}\}), (\texttt{*BELT}, \{\texttt{ON}, \texttt{OFF}\}), (\texttt{*END}, \{5, 10\}), (s_1, \{\texttt{OFF}, \texttt{WAIT}, \texttt{ALARM}\})\}$.
- $E_1 = \{(\texttt{*KEY}, \{\texttt{ON}, \texttt{OFF}\}), (\texttt{*BELT}, \{\texttt{ON}, \texttt{OFF}\}), (\texttt{*END}, \{5, 10\})\}$.

- $O_1 = \{(\texttt{*START}, \{\epsilon\}), (\texttt{*ALARM}, \{\texttt{ON}, \texttt{OFF}\}), (s_1, \{\texttt{OFF}, \texttt{WAIT}, \texttt{ALARM}\})\}$.
- $R_1 = \{(s_1, \texttt{OFF})\}$.
- $F_1 = \{(\{(\texttt{*KEY}, \texttt{ON}), (s_1, \texttt{OFF}), \} \Rightarrow \{(s_1, \texttt{WAIT}), (\texttt{*START}, \epsilon)\}),$
  $(\{(\texttt{*KEY}, \texttt{ON}), (\texttt{*BELT}, \texttt{ON}), (s_1, \texttt{OFF}), \} \Rightarrow \{(s_1, \texttt{OFF}), \}),$
  $(\{(\texttt{*KEY}, \texttt{OFF}), (s_1, \texttt{WAIT})\} \Rightarrow \{(s_1, \texttt{OFF})\}),$
  $(\{(\texttt{*BELT}, \texttt{ON}), (s_1, \texttt{WAIT})\} \Rightarrow \{(s_1, \texttt{OFF})\}),$
  $(\{(\texttt{*END}, 5), (s_1, \texttt{WAIT})\} \Rightarrow \{(s_1, \texttt{ALARM}), (\texttt{*ALARM}, \texttt{ON})\}),$
  $(\{(\texttt{*END}, 10), (s_1, \texttt{ALARM})\} \Rightarrow \{(s_1, \texttt{OFF}), (\texttt{*ALARM}, \texttt{OFF})\}),$
  $(\{(\texttt{*BELT}, \texttt{ON}), (s_1, \texttt{ALARM})\} \Rightarrow \{(s_1, \texttt{OFF}), (\texttt{*ALARM}, \texttt{OFF})\}),$
  $(\{(\texttt{*KEY}, \texttt{OFF}), (s_1, \texttt{ALARM})\} \Rightarrow \{(s_1, \texttt{OFF}), (\texttt{*ALARM}, \texttt{OFF})\})\}$

Each CFSM describes a *component* of the system to be modeled. The whole system is described as a *network* $\mathcal{N}^{\mathcal{C}}$ of interacting CFSMs; that is, a set of CFSMs such that no two *different* ones have an output event name in common. Hierarchy can be used (as will be shown in Section 2.1) to bridle complexity, but the formal definition is given here, without loss of generality, in terms of a *flat* view.

To define the set of legal traces produced by a CFSM network, we give a set of conditions ensuring that the transitions of each CFSM are "atomic". That is, all output events of a given transition must be emitted before the next transition can occur. This means that for each timed trace for each CFSM we can find an *ordered sequence* of pairs $(c_0, r_0), (c_1, r_1), (c_2, r_2), \ldots$ such that:

- each (c_i, r_i) identifies events of T that are the *cause* and the *result* respectively the i-th transition of $\mathcal{C}$, and
- every output event has a cause.

We also have the notion of *maximality* of the sets causing each transition to allow the transition relation to test for the *absence* of a particular event at a certain time, and give priorities to the reactions to simultaneously present events. Recall that an event is "absent" for a CFSM at a point in time either if it has not yet been emitted since time 0 or if it is a *triggering* event and it has been used for a previous transition of the same CFSM.

A CFSM network does not enforce *fairness*, because this would have imposed too tight constraints on the software implementation. Input events belonging to a timed trace might legally cause infinitely often a transition of a CFSM, and yet this transition may never occur or may never produce a reaction. We only force *all* the *output* events for a transition to be emitted if one of them is emitted. However, fairness is imposed to the type of software implementation we propose through the use of a fair software scheduler.

As explained in [CLJ+93], we need to map CFSMs into FSMs for synthesis and validation. A CFSM event is mapped in the FSM world as:

- one binary signal that encodes the event occurrence.

- a bundle of n signals that encode the event value. (Obviously, $log_2 k \leq n$ where k is the number of values that the event value can take. n can be 0 if k is 0).

Given a CFSM $\mathcal{C}$ we can derive a network of FSMs $\mathcal{N}^{\mathcal{F}}$ whose behavior is the same, in the sense that they interact with the "rest of the world" in the same way, composed of:

1. One "main" *completely specified* FSM $\mathcal{F} = (I^{\mathcal{F}}, O^{\mathcal{F}}, X^{\mathcal{F}}, R^{\mathcal{F}}, F^{\mathcal{F}})$ such that its transition relation is modified in order to include "self-loop" transitions that express the fact that, when no trigger event occurs, no output events are emitted and no state/output event value is changed.
2. For each input signal pair $*i_n, i_n$ of $\mathcal{F}$ corresponding to an input event of $\mathcal{C}$ there is an FSM such that its inputs are the "external" pair of signals $*i'_n, i'_n$ corresponding to $*i_n, i_n$, its outputs are the "internal" pair of signals $*i_n, i_n$, and its set of states and *non-deterministic* transition relation define the (possibly non-deterministic) delay with which the signals $*i_n, i_n$ follow the external corresponding ones.
3. For each output signal there a FSM somewhat similar to those used for input signals that governs the transmission of the output signal pairs of $\mathcal{F}$ to the external world.

Notice that, the output FSM being a Moore machine, the whole network $\mathcal{N}^{\mathcal{F}}$ is a Moore machine.

In [CLJ+93], we have shown formally the behavioral equivalence between a CFSM network and an FSM network.

This interpretation of a CFSM network $\mathcal{N}^{\mathcal{C}}$ as an FSM network $\mathcal{N}^{\mathcal{F}}$ is extremely useful for synthesis. It ensures consistency between a SHIFT specification and an array of implementation options that differ largely in terms of timing behavior. These software and hardware implementations propose are *correct* in the sense that the set of their timed traces is contained in that of $\mathcal{N}^{\mathcal{F}}$, and hence of $\mathcal{N}^{\mathcal{C}}$. That is, for a CFSM network $\mathcal{N}^{\mathcal{C}}$ described in SHIFT we can derive a hardware and a software implementations I_h and I_s such that the set of timed traces of $\mathcal{N}^{\mathcal{F}}$ includes that of I_s and that of I_h. However, in general the sets of timed traces of I_h and I_s are not equal.

This containment will be ensured by the fact that:

- For a hardware implementation, the input FSMs are reduced only to the s_0 state and the output FSMs delay by exactly one cycle each event.
- For a software implementation, input event buffers that are set whenever an event is sensed from the outside world (possibly overwriting the old value) and output event buffers that are transmitted to the outside world whenever a transition is completed, implement the input-output FSMs. The buffers should not be changed while the C function implementing the transition relation is executing, thus ensuring the correct sequencing of input events and output reactions.

2.1 CFSM Intermediate Format

We can describe a CFSM network using a specific representation format, called *Software-Hardware Intermediate Format* (SHIFT), that we use to exchange CFSM descriptions across tools in the form of files. A complete specification of the SHIFT language has been given in [CLJ+93]. Figure 3 shows how the simple seat belt example described in Figure 2 and Example 2 can be specified using SHIFT.

The example shows how the SHIFT specification of a system is organized into a three-layer hierarchy. At the top (layer 0, or *Inter-partition and interfaces description*), we only have `.subckt`s representing partitions as black boxes; no `.names` constructs are allowed. Each partition is a `.model` at layer 0. Inside each of them (layer 1, or *Partition description*) we can have reactive transformations (i.e. CFSMs) and sub-systems. Some elements of a partition may be elementary components that specify pure combinational functions (layer 2, or *Functional description*) referred to in the reactive CFSM description. Layer 0 is always a flat description, whereas layers 1 and 2 can be organized into a homogeneous hierarchy. Figure 5 depicts how a SHIFT specification (shown in Figure 4) is organized.

Most of the time, it will be the case that a SHIFT hierarchy is not complete, as in Example 3: there the model `INC_4` is missing. These aspects will be addressed in Section 5.

```
# level 0 - inter partition interfaces
.model system
.inputs *BELT BELT *KEY KEY
.outputs *ALARM ALARM
.mv BELT 2 ON OFF
.mv END 2 5 10
.mv KEY 2 ON OFF
.mv ALARM 2 ON OFF
.subckt belt b1 *BELT=*BELT BELT=BELT *END=*END END=END \
*KEY=*KEY KEY=KEY *START=*START *ALARM=*ALARM ALARM=ALARM
.subckt timer t *START=*START *END=*END END=END
.end

# level 1 - this is a partition
.model belt
.inputs *BELT BELT *END END *KEY KEY
.outputs *START *ALARM ALARM
.mv BELT 2 ON OFF
.mv END 2 5 10
.mv KEY 2 ON OFF
.mv ALARM 2 ON OFF
.mv s1 3 OFF WAIT ALARM
.names s1    *KEY KEY *END END *BELT BELT => s1    *START *ALARM ALARM
       OFF   1    ON  -    -   0     -       WAIT  1      0      -
       OFF   -    -   -    -   1     ON      OFF   0      0      -
       WAIT  1    OFF -    -   -     -       OFF   0      0      -
       WAIT  -    -   -    -   1     ON      OFF   0      0      -
       WAIT  -    -   1    5   -     -       ALARM 0      1      ON
       ALARM 1    OFF -    -   -     -       OFF   0      1      OFF
       ALARM -    -   1    10  -     -       OFF   0      1      OFF
       ALARM -    -   -    -   1     ON      OFF   0      1      OFF
.end

# level 1 - this is a partition
.model timer
.inputs *START
.outputs *END END
.mv END 2 5 10
.mv N 16
.mv N_PLUS_1 16
.names *TICK *START N_PLUS_1 => N          *END END
       -     1      -           0          0    -
       1     0      -           (N_PLUS_1) 0    -
       1     0      5           5          1    5
       1     0      10          0          1    10
.names *TICK *START => *TICK
- 1   1
1 -   1
.subckt INC_4 i1 a=N i=N_PLUS_1 c=cc
.end
```

Figure 3: SHIFT Specification of the seat belt example

Top layer - Inter-partition description

```
.model top
.inputs *x
.outputs *y
.mv v 8
.subckt one _1 *a=*x b=v *c=*e
.subckt two _2 *a=*e b=v *c=*y
.end # top
```

Layer 1 - Partition description

```
.model one
.inputs *a b
.outputs *c
.mv b 8
.names *a f f_minus_1 => *p f
1 - -   0 (f_minus_1)
1 0 0   1 N
.subckt three _1_1 *e=*p u=b *y=*c
.subckt DECR _1_DECR i=f o=f_minus_1
.end # one

.model three
.inputs *p u
.outputs *y
.mv u 8
.mv g 8
.names *p u u_minus_1 => *y g
1 - -   0 (u_minus_1)
1 0 0   1 K
.subckt DECR _1_1_DECR i=u o=u_minus_1
.end # three

.model two
....
....
.end # two
```

Layer 2 - Functional description

```
.model DECR
.inputs i
.outputs o
.mv aa 8
.mv bb 8
....
....
.subckt FOO DECR_FOO_1 i=aa o=bb
.end # DECR

.model FOO
.inputs i
.outputs o
....
.end # FOO
```

Figure 4: Example of Hierarchical SHIFT Specification

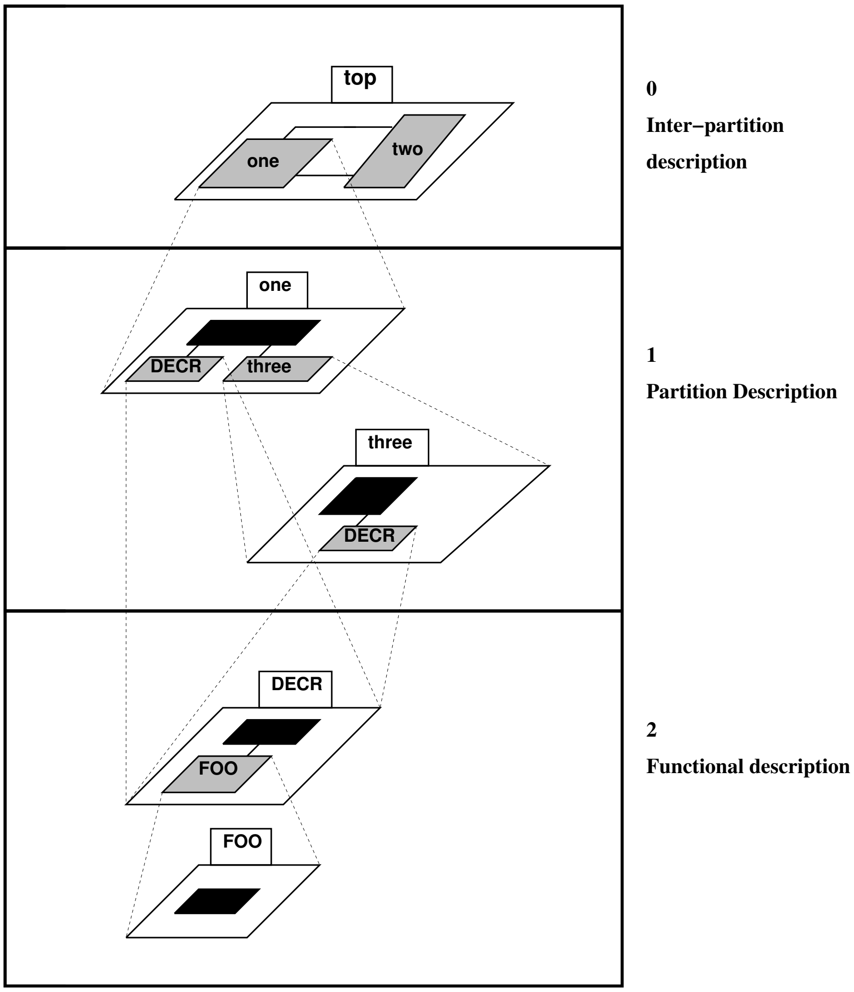

Figure 5: SHIFT Hierarchy

3 System Partitioning

The general embedded system architecture that we envision is shown in Figure 6. The figure describes a system with three partitions, one software (implying one micro-controller) and two hardware (implying two integrated circuits). Each partition is synthesized separately.

- Partition 1 implements two CFSMs, exchanging event e_1 internally (from CFSM 1 to CFSM 2) and events e_1, e_2, e_3 and e_8 externally. I.e. event e_1 is sent (broadcast) to CFSMs 2 and 7, event e_2 is sent to CFSM 5, the value (only) of event e_3 is sent to CFSM 3, and event e_8 is sent to CFSMs 3 and 5.
- Partition 2 implements only CFSM 7, that receives events e_1 and e_4.
- Partition 3 runs on a micro-controller with a simple Operating System (mainly a scheduler and a hardware interface handler) and four CFSMs. These CFSMs logically communicate with each other and with the hardware CFSMs using events. In practice the events are implemented using the OS interprocess communication facilities and its device drivers for the micro-controller ports (Sections 4.2 and 6 describe the interface mechanism implementing events more in detail). For example, ports 1, 3, and 5 of the micro-controller are dedicated to a single event each, while ports 2 and 4 are shared by events e_3 and e_8, and e_6 and e_7 respectively. Ports 1, 2, and 3 are real I/O ports of the micro-controller, whereas ports 4 and 5 are virtual ports used for intra-partition inter-task communication (see Section 6.)

It is useful to remind that when we talk of micro-controller we mean a single-chip computer that includes a CPU, a ROM, a RAM, possibly a EEPROM, and a set of peripherals such as digital I/O ports and A/D converters. In general, in order to keep the cost of the device low, the size of the RAM is very small, and the number of I/O port is limited. For example, a typical micro-controller such as the Motorola MC68HC11K4 has 24K byte ROM, 768 byte RAM, 40 digital I/O lines, plus other devices as PWM outputs, a 16-bit timer system, etc. The I/O ports are generally memory mapped. Thus, the software program of a micro-controller interacts with the external world by reading and writing memory registers.

Another and possibly more general approach is to work with a micro-processor instead of a micro-controller. In that case we would have a bus and shared memory architecture. The communication between the μP and its environment is made more flexible but this type of solution tends to be significantly more expensive both in terms of development cost and product cost and should only be considered in the case of very big production volumes.

The partitioning process takes as input a SHIFT specification and outputs a SHIFT specification in which each `.model` partition (i.e. each `.model` of the layer 0) is labeled as HW or SW. In addition, a mixed hardware-software interfacing mechanism is automatically synthesized (see Section 6) and passed on to the automated synthesis process.

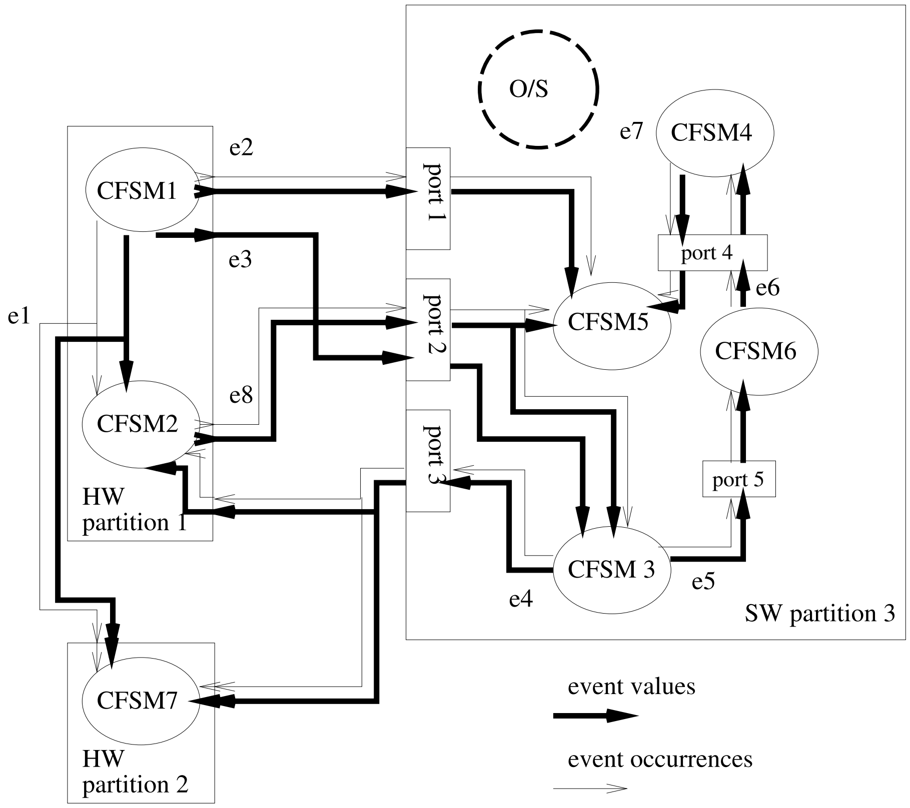

Figure 6: Embedded system architecture

4 Automated Synthesis of SHIFT Specifications

The partitioning process returns a SHIFT specification in which each partition (i.e. a `.model` of layer 0) is labeled as HW or SW. Each HW model is translated into a hardware description format; each SW model is translated into an embedded C program. The syntax of SHIFT is suited for easy translation into synchronous hardware and into a specific software architecture. We specify precisely the semantics of SHIFT in terms of a hardware description format and of a software architecture. The range of hardware and software implementations we can produce may not be the widest possible. But our main concerns in this phase are to ensure the generality of the behavioral specification and the consistency of the derived implementation.

4.1 Hardware Synthesis

In the case of hardware synthesis, we map SHIFT into a hardware description format: a pure synchronous interpretation of BLIF-MV ([BCH+91]). BLIF-MV is basically a network of single-output multi-valued functions of multi-valued variables, possibly with latched outputs [1]. Hence, we project each `.names` onto each of its output components and obtain a set of single-output tables from each SHIFT table. We also latch the output of every function (with the exclusion of library modules described in Section 5) to implement the minimum delay of one time unit required by the CFSM semantics. The result of this translation is a network of synchronous interacting Moore FSMs whose behavior (set of traces) is contained in the behavior of the SHIFT specification[2].

A circuit implementing the safety belt example is specified in BLIF-MV (excluding model `.timer`) as follows:

Example 3

```
.model system
.inputs *BELT BELT *KEY KEY
.outputs *ALARM ALARM
.mv BELT 2 ON OFF
.mv END 2 5 10
.mv KEY 2 ON OFF
.mv ALARM 2 ON OFF
.subckt belt b1 *BELT=*BELT BELT=BELT *END=*END END=END \
*KEY=*KEY KEY=KEY *START=*START *ALARM=*ALARM ALARM=ALARM
.subckt timer t *START=*START *END=*END END=END
.end

.model belt
.inputs *BELT BELT *END END *KEY KEY
.outputs *START *ALARM ALARM
```

[1] BLIF-MV is still under development. The syntax used in this paper is therefore subject to change

[2] This is compatible with the S/R model proposed by Kurshan ([Kur90]), that is also based on Moore machines.

```
.mv BELT 2 ON OFF
.mv END 2 5 10
.mv KEY 2 ON OFF
.mv ALARM 2 ON OFF
.mv s1 3 OFF WAIT ALARM
.names s1 *KEY KEY *END END *BELT BELT _s1
OFF   1 ON  - -  0 -   WAIT
OFF   - -   - -  1 ON  OFF
WAIT  1 OFF - -  - -   OFF
WAIT  - -   - -  1 ON  OFF
WAIT  - -   1 5  - -   ALARM
ALARM 1 OFF - -  - -   OFF
ALARM - -   1 10 - -   OFF
ALARM - -   - -  1 ON  OFF
.names s1 *KEY KEY *END END *BELT BELT _*START
OFF   0 -   - -  - -   0
OFF   1 ON  - -  0 -   1
OFF   1 ON  - -  1 -   0
OFF   1 OFF - -  - -   0
WAIT  - -   - -  - -   0
ALARM - -   - -  - -   0
.names s1 *KEY KEY *END END *BELT BELT _*ALARM
OFF   - -   - -  - -   0
WAIT  0 -   0 -  - -   0
WAIT  0 -   1 5  0 -   1
WAIT  0 -   1 5  1 ON  -
WAIT  0 -   1 5  1 OFF 1
WAIT  0 -   1 10 - -   0
WAIT  1 ON  0 -  - -   0
WAIT  1 ON  1 5  0 -   1
WAIT  1 ON  1 5  1 ON  -
WAIT  1 ON  1 5  1 OFF 1
WAIT  1 ON  1 10 - -   0
WAIT  1 OFF 0 -  - -   0
WAIT  1 OFF 1 5  - -   -
WAIT  1 OFF 1 10 - -   0
ALARM 0 -   0 -  0 -   0
ALARM 0 -   0 -  1 ON  1
ALARM 0 -   0 -  1 OFF 0
ALARM 0 -   1 5  0 -   0
ALARM 0 -   1 5  1 ON  1
ALARM 0 -   1 5  1 OFF 0
ALARM 0 -   1 10 - -   1
ALARM 1 ON  0 -  0 -   0
ALARM 1 ON  0 -  1 ON  1
ALARM 1 ON  0 -  1 OFF 0
ALARM 1 ON  1 5  0 -   0
```

```
ALARM 1 ON  1 5  1 ON  1
ALARM 1 ON  1 5  1 OFF 0
ALARM 1 ON  1 10 - -   1
ALARM 1 OFF - -  - -   1
.names s1 *KEY KEY *END END *BELT BELT ALARM
OFF   1 ON  - -  0 -   -
OFF   - -   - -  1 ON  -
WAIT  1 OFF - -  - -   -
WAIT  - -   - -  1 ON  -
WAIT  - -   1 5  - -   ON
ALARM 1 OFF - -  - -   OFF
ALARM - -   1 10 - -   OFF
ALARM - -   - -  1 ON  OFF
.latch _*START *START
.latch _*ALARM *ALARM
.latch _s1 s1
.end

.model timer
.inputs *START
.outputs *END END
.mv END 2 5 10
.mv N 16
.mv N_PLUS_1 16
.names *TICK *START N_PLUS_1 N
- 1 -  0
1 0 -  (N_PLUS_1)
1 0 5  5
1 0 10 0
.names *TICK *START N_PLUS_1 _*END
0 - -  0
1 0 0  0
1 0 1  0
1 0 2  0
1 0 3  0
1 0 4  0
1 0 5  -
1 0 6  0
1 0 7  0
1 0 8  0
1 0 9  0
1 0 10 -
1 0 11 0
1 0 12 0
1 0 13 0
1 0 14 0
1 0 15 0
```

```
1 1 -  0
.names *TICK *START N_PLUS_1 END
- 1 -  -
1 0 -  -
1 0 5  5
1 0 10 10
.names *TICK *START _*TICK
0 0 0
0 1 1
1 - 1
.subckt INC_4 i1 a=N i=N_PLUS_1 c=cc
.latch _*END *END
.latch _*TICK *TICK
.end
```

Notice that latches and new transitions specifying "self-loop" transitions have been introduced. As explained in [CLJ+93], this is necessary to transform the reactive syntax of CFSMs into the purely synchronous syntax of BLIF-MV.

In the case of hardware synthesis the modeling of delay is very simple. Events are sensed immediately and reacted within the minimum delay. That is, every state is associated with the set of events that are present in that state, and the reaction to an event is present in the immediate successor of the state associated to the triggering event. No non-deterministic delay is possible.

The following table that shows a particular possible path of behavior of this BLIF-MV implementation (only some signals are shown):

```
KEY_ON       . . . 1 . . . . . . . . . . . . . . . . . . . .
START_TIME   . . . . 1 . . . . . . . . . . . . . . . . . . .
STATE        0 0 0 0 1 1 1 1 1 2 2 2 2 2 2 0 0 0 0 0 0 0 0
END_5_S      . . . . . . . . 1 . . . . . . . . . . . . . .
ALARM_ON     . . . . . . . . . 1 . . . . . . . . . . . . .
END_10_S     . . . . . . . . . . . . . . 1 . . . . . . . .
ALARM_OFF    . . . . . . . . . . . . . . . 1 . . . . . . .
n            - - - - - 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 8 0
```

Finally, we need to translate the BLIF-MV description into a circuit. To do this we translate BLIF-MV into BLIF, and pass the BLIF description to a logic synthesis tool like U.C. Berkeley's *SIS* ([SSL+92, SSM+92]) by which we produce the final implementation. The transformation of BLIF-MV is pretty straightforward. A multi-valued signal denoting the *presence* of an event is implemented by a single wire, that is at value 1 when the event is present and at 0 otherwise. Note that if such a wire stays at 1 for n clock cycles, this denotes n occurrences of the event, so there is no overhead associated with event detection in hardware (unlike most other communication paradigms used originally for software specifications, such as channels). A signal denoting the *value* of an event is implemented by a bundle of wires encoding the values.

The only significant issues are: how to resolve the possible non-determinism of BLIF-MV and how to encode multi-valued variables. Currently, we have developed a BLIF-MV to BLIF translator that accepts only deterministic BLIF-MV descriptions and applies the "natural" encoding implied by interpreting each value in a `.mv <name> <val_1> ... <val_n>` statement as an integer. Programs such as NOVA [Vil86], MUSTANG [DMNSV88] or JEDI [LN89] can be used (via *sis*) to do the encoding so that the logic implementing the FSM is minimized. As a further optimization, we can also use multiplexer and bus allocation techniques from high-level synthesis to minimize the consumption of expensive inter-partition communication resources (i.e. I/O pins).

4.2 Software Synthesis

In the case of software, we map the SHIFT specification of a partition into a software structure that includes a number of procedure and a simple operating system. To ensure portability across micro-controllers, we only consider producing a self-contained block of C code that should be compiled and used virtually on any processor.

The customized OS for each micro-controller consists of a scheduler and drivers for the virtual I/O ports, hence it is extremely small and imposes little overhead. It is also easily portable to new micro-controllers.

In order to correctly implement the CFSM behavior the OS must satisfy the following constraints:

1. Each transition of a task must be performed "atomically", i.e. the values of the input event buffers for that task must not change once it has been started.
2. "Consumed" trigger events must be reset before a task is invoked again.
3. Output events from a task must be all transferred to the interfaces (described more in detail in Section 6) by setting the signal or variable denoting the event presence only *after* the corresponding value has been updated.

As explained in Section 3, a micro-controller generally communicates with the external world via memory mapped I/O ports. The software program of a micro-controller interacts with the external world by reading and writing memory registers. In our approach, we generalize this idea in order to make the inter-task and the inter-processor communication as similar as possible. The communication mechanism between a task and the external world (other tasks or external devices) is based on the use of *Virtual I/O ports.* A virtual I/O port is a memory location that both software tasks and the OS use to circulate events. In the case of communication with an external device, such a location will correspond to a real I/O port. In the case of inter-task communication, it will be just a regular r/w memory register that some tasks can write to and some tasks can read from.

Each event is associated with a bit of a virtual port. If the bit is set, the event has occurred and the value of the event, if any, is to be found in another memory location. Once an event has been detected by the OS it is copied to the input buffers of all tasks that are sensitive to it. Event emission and detection are performed by macros that use ROM tables created by the software synthesis program that specify which port and which bits are associated to an event. The `emit(event)` call writes an output event directly to the port. The `occurred(event,InpBuff)` call checks a task's private input buffer. The `assign_events_to_proc_buffs()` procedure checks virtual I/O ports and flushes ports after writing events to each task's input buffer, according to the sensitivity information stored in a ROM table.

The simple scheduling mechanism is as follows: all process buffers are checked in turn. If a buffer is not empty (i.e. the process is enabled) the corresponding software process is invoked. The acyclic structure of the code and the simple structure of the scheduler ensure liveness (i.e an enabled task will be eventually invoked) and fair scheduling of every process (i.e. all enabled tasks will be invoked infinitely often). These are two essential properties of every sound concurrent software system as explained in [BAPM83].

We consider each CFSM as a software concurrent process or *task*. A task is acyclic, without iteration. Each time it is activated it performs *at most one* (possibly none if no triggering event is present) transition of the CFSM. Each line of the `.names` table is thus interpreted as a *condition* $\rightarrow$ *action* statement. Each input event to a CFSM is saved, until it is either consumed or overwritten, in an event buffer that is private to each software task. Assignments are implemented in a straightforward fashion. For example, the following SHIFT specification

```
.names a *x x *z => b *y
- 1 K -    0  1
A 0 - 1   (x) 0
B 0 - 1   (x) 0
```

is mapped in C as

```
void
foo(b)
int b; /* process inp_event buffer */
{
   if (1) {
      if (occurred(c1,b)) {
         if (x == K) {
            b = 0;
            emit(y);
         }
      }
   }
   else
   if (a == A) {
      if (occurred(c2,b)) {
            b = x;
         }
      }
   }
```

```
   else
   if (a == B) {
      if (occurred(c2,b)) {
            b = x;
         }
      }
   }
}
```

This straightforward translation produces an implementation that can be quite inefficient in terms of code size and speed. For example, the fragment above can be reduced to

```
void
foo(b)
int b; /* process inp_event buffer */
{
   if (occurred(c1,b)) {
      if (x == K) {
         b = 0;
         emit(y);
      }
   }
   else
   if ((a == A) || (a == B))
      if (occurred(c2,b)) {
            b = x;
         }
      }
   }
}
```

by realizing that a constant value does not need to be tested and that a set of condition that produce the same action can be collapsed. The issue of software optimization is covered in Section 7.

Other features of the software synthesis method can be found in the following example that represents a software implementation of the CFSM described in Figure 2:

Example 4

```
typedef void (*foop_t)();   /*foop_t = pointer to a function returning void*/

/* process names */
#define _p_belt_n1 0
```

```
#define N_PROCS 1           /* number of processes */

foop_t proc[N_PROCS];       /* array of pointers to functions */
int inp_event[N_PROCS];     /* array of function input buffers (in order)*/

/* event IDs */
#define _e_KEY    0
#define _e_BELT   1
#define _e_END    2
#define _e_START  3
#define _e_ALARM  4

/* physical address of i/o ports */
#define IO_PORT_A ADDR_A
#define IO_PORT_B ADDR_B
#define IO_PORT_C ADDR_C

/* each "column" specifies INP_port and bit in the port for an inp event          */
/*                      0 key       1 belt      2 end       3 start     4 alarm   */
int *event_io_port[] = { IO_PORT_A,  IO_PORT_A,  IO_PORT_B,  IO_PORT_C,  IO_PORT_C };
int  event_io_bit[]  = { 0x01,       0x02,       0x04,       0x01,       0x08      };

/* each "row" specifies an event/proc/bit relation. i.e. the "event" can
   sensed by the "proc" by polling the "bit" */
int event_sens_tabl[] = { _e_KEY,  _p_belt_n1, 0x01,
                          _e_BELT, _p_belt_n1, 0x02,
                          _e_END,  _p_belt_n1, 0x04
                        };

#define N_E_P_REL 3   /* size of event/proc relation = rows/3 of event_sens_tabl*/

/* constants that define which bit of a process' inp buff carry an event */
/* the ..._1 means event must be 1, ...._0 event must be 0               */
/* used by "occurred"    zero  one                                       */
int _c_belt_n1_c1[2] = { 0x02, 0x01 };   /* !BELT * KEY */
int _c_belt_n1_c2[2] = { 0x00, 0x02 };   /* BELT */
int _c_belt_n1_c3[2] = { 0x00, 0x01 };   /* KEY  */
int _c_belt_n1_c4[2] = { 0x00, 0x02 };   /* BELT */
int _c_belt_n1_c5[2] = { 0x00, 0x04 };   /* END  */
int _c_belt_n1_c6[2] = { 0x00, 0x01 };   /* KEY  */
int _c_belt_n1_c7[2] = { 0x00, 0x04 };   /* END  */
int _c_belt_n1_c8[2] = { 0x00, 0x02 };   /* BELT */


/*general macros*/

/* returns != 0 if event is found in io_port */
```

```
#define got_inp_event(event) (event_io_bit[event] & *event_io_port[event])

/* set a bit in PROCs inp_event buffer */
#define put_inp_event(pos) (inp_event[event_sens_tabl[pos+1]] = \
        inp_event[event_sens_tabl[pos+1]] | event_sens_tabl[pos+2])

/* see section "interfaces" */
void
acknowledge_events() {};

/* check io_ports to find events */
void
assign_events_to_proc_buffs()
{
   int n, pos, event;

   for (n=0; n<N_E_P_REL; n++) {
      pos = n * 3;
      event = event_sens_tabl[pos];
      if (got_inp_event(event)) {
         put_inp_event(pos);
      }
   }
   (void) acknowledge_events();
}

/* check inp_event buffer */
#define ZEROES  0
#define ONES    1
#define occurred(cond, e) (((cond[ZEROES] & e) == 0x0) && \
     (cond[ONES] == (cond[ONES] & e)))

/* set io_port or internal port */
#define emit(event) (*event_io_port[event] = \
     *event_io_port[event] | event_io_bit[event])

/* ------------------------------------------------------------------------ */

/*Initialize processes' vars*/
typedef enum { OFF, ON } type_1_t;
typedef enum { s5, s10 } type_2_t;
typedef enum { OFF, WAIT, ALARM } type_3_t;

static type_1_t _v_KEY   = OFF;
static type_1_t _v_BELT  = OFF;
static type_1_t _v_ALARM = OFF;
static type_2_t _v_END   = 0;
```

```
static type_2_t _v_s1    = 0;

/* processes from SHIFT spec */

void
_t_belt_n1(e)
int e; /* process' inp_event buffer */
{

   if (_v_s1 == OFF) {
      if (occurred(_c_belt_n1_c1,e)) {
         if (_v_KEY == ON) {
            _v_s1 = WAIT;
            emit(_e_START);
         }
      }
      else
      if (occurred(_c_belt_n1_c2,e)) {
         if (_v_BELT == ON) {
            _v_s1 = OFF;
         }
      }
   }

   else
   if (_v_s1 == WAIT) {
      if (occurred(_c_belt_n1_c3,e)) {
         if (_v_KEY == OFF) {
            _v_s1 = OFF;
         }
      }
      else
      if (occurred(_c_belt_n1_c4,e)) {
         if (_v_BELT == ON) {
            _v_s1 = OFF;
         }
      }
      else
      if (occurred(_c_belt_n1_c5,e)) {
         if (_v_END == s5) {
            _v_s1 = ALARM;
            _v_ALARM = ON;
            emit(_e_ALARM);
         }
      }
   }
```

```
   else
   if (_v_s1 == ALARM) {
      if (occurred(_c_belt_n1_c6,e)) {
         if (_v_KEY == ON) {
            _v_s1 = OFF;
            _v_ALARM = OFF;
            emit(_e_ALARM);
         }
      }
      else
      if (occurred(_c_belt_n1_c7,e)) {
         if (_v_END == s10) {
            _v_s1 = OFF;
            _v_ALARM = OFF;
            emit(_e_ALARM);
         }
      }
      else
      if (occurred(_c_belt_n1_c8,e)) {
         if (_v_BELT == ON) {
            _v_s1 = OFF;
            _v_ALARM = OFF;
            emit(_e_ALARM);
         }
      }
   }
}


/* Initialize processes */
void
init_procs()
{
   int p;

   for (p=0; p<N_PROCS; p++) {
      inp_event[p] = 0x0;
   }

   proc[_p_belt_n1]    = *_t_belt_n1;
}

/* ------------------------------------------------------------------- */

/*main scheduler*/
main()
{
```

```
    int p;

    init_procs();
    for(;;) {
       assign_events_to_proc_buffs();
       for(p=0; p<N_PROCS; p++) {
          if(inp_event[p]!=0) {
             (proc[p])(inp_event[p]);
             inp_event[p] = 0;
          }
       }
    }
}
```

In the example, the following conventions are applied:

- Event values are prefixed by `_v_`, like in

  `static type_1_t _v_BELT = OFF;`

- Process identifiers are prefixed by `_p_`, like in

  `#define _p_belt_1 0`

- The actual code of software procedures are prefixed by `_t_`, like in

  `void _t_belt_n1(e) int e;`

  `_n1` stands for task 1. In general we have a task for each `.names` table (but some further optimization can also be applied, as explained in section 7).

- Event identifiers are prefixed by `_e_`, like in

  `#define _e_KEY 1`

- Event pattern names are prefixed by `_c_`, like in

  `int _c_belt_n1_c1[2] = { 0x02, 0x01 };`

  which represents the first pattern (c1) of the first task (n1) of model `belt`. The pair of values above indicates which events must NOT be present (bits at 0) and which events must be present (bits at 1) for the pattern to be detected. The line above represents the pattern `------01`, or `*KEY=1` and `*BELT=0`. The constants `_c_<name>_n<i>_c<j>` are generated by the software synthesis program. If no optimization is performed, each of those constant pairs represents a pattern of events corresponding to a line in a `.names` table of a SHIFT specification.

- The tables `event_io_port` and `event_io_bit` store the information regarding which port and which port bit need to be polled by the OS to detect the event. The table `event_sens_tabl` is a global sensitivity list that, for each event, lists the tasks that can sense it and how the event is encoded in the task's input buffer. Notice that the same input event does not have to, and in general cannot, be mapped in the same bit of each task's input buffer.

An interesting example of synthesized software for a multiple-task application is shown in section 9.

As far as the coding style is concerned, we do not claim this software is easily readable or elegant. Style is not first priority in real-time software. Besides, this code is automatically generated from the SHIFT format and is not meant to be read by the user. Optimization is done on the SHIFT format; validation is done on the VIF format.

This scheme lends itself both to polling and interrupt implementations of event detection, as an active task can be suspended and resumed at any time.

An interrupt-based implementation is similar to the polling-based one showed in the above example. The interrupt handler that receives input events from other partitions updates the buffers of the relevant tasks exactly as in the polling case. Note that there may be a problem due to an excessive number of interrupts preventing any task from being executed. This problem must be handled by the system designer and it is an interesting application for formal verification.

It is important to understand the timing behavior of such a software implementation, and how it fits in our model of delay. The following table gives a *qualitative* demonstration of a particular possible path of behavior of this software implementation (only some signals are shown). By comparing it with the similar table given for hardware we notice that state changes do not happen right after the event that triggers them, and the output events are emitted some cycles after the state has been set. Also the value of an event is set before the event is actually emitted.

*KEY	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.	.
KEY	.	.	.	ON	.	.	.	.	.	.	.	.	.	.	.	.
*START_TIME	.	.	.	.	.	.	.	1	.	.	.	.	.	.	.	.
STATE	OFF	OFF	OFF	OFF	OFF	WAIT	WAIT	WAIT	WAIT	WAIT	WAIT	ALARM	ALARM	ALARM	ALARM	ALARM
*END	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.
END	.	.	.	.	.	.	.	.	.	5	.	.	.	.	.	.
*ALARM	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.	.
ALARM	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	ON

The model of delay for an interrupt-based task is identical to one for the polling case. The sensing of the current interrupt is delayed because a task detects its input events some cycles after the interrupt occurs.

5 Coping with Algorithmic Complexity: Pre-built Blocks

Although general in terms of expressiveness, SHIFT is not specifically designed for specifying arithmetics or complex algorithms, but only for control-dominated applications. The idea is that a SHIFT function specifies the reactive part of the behavior, whereas the details of the algorithms associated with the actions invoked can be specified by standard pre-built functions. Libraries are made available that provide standard components (like adders, counters, etc.) already optimized and mapped in SHIFT, BLIF-MV, and C code. In our environment, a library is a directory in which SHIFT (.shift) files, BLIF-MV (.blifmv) files, and C (.c) source files are stored.

For example, suppose we are synthesizing the belt example as hardware. The model `INC_4` is specified in `.subckt` statement but it is missing from the input SHIFT file. In this case, the hardware synthesis process looks it up in a library. Since a SHIFT specification is organized into a three-layer hierarchy if a missing model is called from a level of layer 0 or 1, the hardware synthesis program looks in the specified library for a SHIFT model description which is included and synthesized. If it is called from a level of layer 2, the hardware synthesis program looks for a BLIF-MV model description which is copied to the output file as is. In the library there is also a INC_4.c file. This is used in a similar way in the case of software synthesis. This is quite similar to using library files in software design.

Figure 5 shows a library BLIF-MV description of a 4-bit incrementer. Note that in this example multi-valued variables are explicitly bit-encoded. This corresponds to the definition of an "interface" block between the symbolic world of BLIF_MV and the binary world where addition is defined. In principle, a multi-valued mapping associating each symbolic value with its successor could have been defined, thus implying the use of a symbol encoding program (such as, e.g., NOVA [Vil86], MUSTANG [DMNSV88] or JEDI [LN89]). Using encoding/decoding and a binary adder correspond to an optimized hand implementation of the very same behavior. In "life-size" design, it is unpractical to list explicitly encodings in a BLIF-MV file. For this reason, we reserve the names `_enc*` and `_dec*` where `*` replaces an actual suffix, for predefined encoding methods. That is, when the BLIF-MV to BLIF translator program runs into a `.model _enc_foo`, it calls an encoding procedure identified by the name `foo`, and disregards the actual content of the model.

```
.model INC_4
.inputs a
.outputs i c
.mv a 16
.mv i 16
.subckt _enc_int e0 x=a y0=y0 y1=y1 y2=y2 y3=y3
.subckt INC_4_BIT i0 a0=y0 a1=y1 a2=y2 a3=y3 i0=i0 i1=i1 i2=i2 i3=i3 c=c
.subckt _dec_int d0 x0=i0 x1=i1 x2=i2 x3=i3 y=i
.end

.model INC_4_BIT
.inputs a0 a1 a2 a3
.outputs i0 i1 i2 i3 c
.names cc
```

```
1
.subckt INC_1_BIT i0 a=a0 c0=cc i=i0 c1=c0
.subckt INC_1_BIT i1 a=a1 c0=c0 i=i1 c1=c1
.subckt INC_1_BIT i2 a=a2 c0=c1 i=i2 c1=c2
.subckt INC_1_BIT i3 a=a3 c0=c2 i=i3 c1=c
.end

.model INC_1_BIT
.inputs a c0
.outputs i c1
.names a c0 i
0 0 0
0 1 1
1 0 1
1 1 0
.names a c0 c1
0 0 0
0 1 0
1 0 0
1 1 1
.end
```

Library modules for both software and hardware implementation are likely to use heavily "elementary" blocks like the adder and incrementer described above. In order to relieve as much as possible the designer from the burden of keeping track of delays, such blocks are defined as "combinational" logic blocks. They are distinguished from standard reactive modules by a values-only interface, without events. So the CFSM event-based delay model obviously cannot be meaningfully applied to them. The time a library module consumes will be accounted for by the reactive transformation of a CFSM that uses it.

In the hardware case, combinational modules do not contain feedback or `.latch` constructs, and the clock period is chosen long enough to allow propagation through them.

In the software case, functional descriptions consist of fragments of code (macros of functions) in which only combinational operations are allowed, i.e. no explicit assignments are used. For example, the software module corresponding to the hardware component in Figure 5 is just

```
#define INC_4(x) (int4_t) x+1;
```

Such combinational software modules are inserted in the appropriate `if then else` branch of the procedure implementing the calling module, and the software timing analysis tool can handle them just like state change statements or `emit` procedure calls.

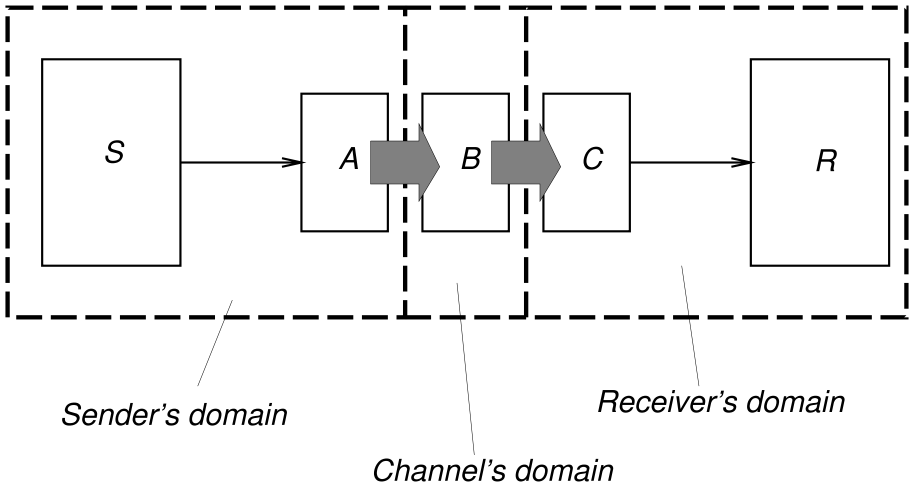

Figure 7: Interface between heterogeneous domains

6 Interfacing Implementation Domains

The communication among partitions is based on discrete event exchange. Since event emission and detection are implemented differently in each implementation domain, an interfacing mechanism is needed. We concentrate on the synchronizing aspect of the event communication mechanism, since the event value is generally easy to store (e.g. a latch or a memory location).

We want partitions to be unaware of what their neighbors are like. The way each partition will emit and detect events is fixed and depends only on the partition's implementation. Interfaces are automatically synthesized in order to translate the sender partition's representation of an event into the receiver partition's representation of it. The representation of events in each implementation domain is defined by the types of implementation we synthesize defined in section 4:

- In hardware an event type is represented as a wire. A hardware partition detects the occurrence of an input event whenever an input wire is found high; an output event is emitted by setting an output wire for a single clock tick.
- In software, a task detects the occurrence of an input event whenever it polls its input buffer and finds it non-zero (`occurred(event,InpBuff) != 0`); an output event is emitted by writing a value to virtual port (performed by `emit(event)`).

Because of the different nature of implementation domains, conceptually we can think of an interface mechanism as a three-layer block (see figure 7.) This idea is expressed more formally by Borriello in [Bor88]. Within the sender's domain, block A translates the event into the representation of the channel's domain. Block B within the channel domain actually gets the event across. Block C transforms the event into the sender's representation. In practice, interfaces can come in the form of cooperating mixed hardware-software solutions. The hardware part of an interface is put at the border between partitions just as another piece of hardware. The software part is embedded in the RT/OS and the interrupt handler.

The specific interfaces we are presenting here have been defined under the assumption that the hardware process is always able to sense the incoming event whereas the software process may not, with the exception of events carried by interrupt lines. Therefore, an event directed to hardware is simply sent, whereas an event sent to software must be saved until it is captured.

These interfacing mechanisms have been chosen for their simplicity so that as little as possible is added to the pure behavioral specification of the system as it is provided by the designer. As explained in section 1, they do not guarantee that all events will be correctly sent and received.

We do not claim these interfaces are optimal or minimal. However, they are simple and consistent with the model of behavior defined in our theory. In our approach, we only need to deal with the event exchange primitive. Any other more sophisticated mechanism can be built on top, and in terms, of this primitive.

Here we are considering two types of domain: synchronous hardware and software embedded in a micro-controller. Considering that more than one software process can share a CPU, and that an event can be fed to an interrupt line of a micro-controller, seven types of interfaces must be considered:

1. **Hardware to hardware.** Events are represented as signals. Since the hardware we synthesize is perfectly synchronous, no delay is involved. Thus, a simple wire is good enough.
2. **Software to hardware.** We need to generate a 1-clock signal. A mixed hardware software mechanism is required. First, a bit of a virtual port is set by `emit(e)`. Second, the sender's scheduler sends a pulse over an output port and resets the virtual port Third, the pulse is transformed into a 1-clock pulse by an hardware circuit that outputs it to the hardware receiver.
3. **Hardware to non-interrupt software.** We need to transform a 1-clock pulse into a value of a bit of an input port of a processor. The presence of the event must be saved until it is copied into the receiver's input buffer. A mixed hardware software mechanism can do the job. The pulse from the hardware sender is stored by an interface sequential circuit that keeps it until the receiver's scheduler, after reading it, resets it. Then, the receiver's scheduler sets the input buffers of the tasks which are sensitive to this event.
4. **Software to non-interrupt software on separate processors.** We need to set the value of an input buffer of another processor. As above, the presence of the event must be saved until it is copied into the receiver's input buffer. We want to keep the two processes, receiver and sender, decoupled. That is, the sender should not wait for the receiver to acknowledge. A mixed hardware software mechanism can do the job. First, a bit of a virtual port is set by `emit(e)`. Second, the sender's scheduler sends a pulse over an output port and resets the virtual port. The hardware part and the reception software behave just like type 3.
5. **Software to non-interrupt software on the same processor.** We need to set the value of input buffers within the same processor. A pure software solution is required. First, the value of a virtual port is set by `emit(e)`. Second, the scheduler sets the input buffers of the tasks sensitive to this event and resets the bit of the virtual port.
6. **Software to interrupt software on separate processors.** We need to send a pulse to an output port. As far as sending the signal over the line is concerned, this case is subsumed

by type 2. Here, though, the interrupt handler is awakened by the incoming edge, sets the input buffers of the receiver tasks, and runs until no input buffer is found set.

7. **Hardware to interrupt software.** We need to get a pulse across. The hardware sender needs no special handling. The software receiver behaves as in type 6.

These interfacing mechanisms can be modeled as synchronous FSMs. Figure 8 shows the FSMs corresponding to the hardware parts described above, along with circuits implementing them and timing diagrams that demonstrate how events propagate through the circuit.

The hardware part of interfaces is synthesized in the form of BLIF-MV code. Each interface is a sub-circuits inserted properly to the output of the hardware synthesis procedure.

An important point is that these interfaces are modeled as depth-1 buffers. We do not consider generalizing this approach to depth-n FIFO buffers. Instead we think that such a solution can be specified by the designer at the high level description of the system. In other words, we do not want to add queues or other specific features to a system specification unless explicitly specified by the designer.

In general, we may have a situation in which the number of interface lines exceeds the number of I/O ports of the micro-controller. In that case, the designer can explicitly model reactive multiplexers and demultiplexers in the initial design (i.e. in the SHIFT specification) and assign them to the hardware domain when we do the partitioning. This task can also be automated.

An alternative technique is that all input events are carried by an interrupt line of the micro-controller, and the type of event is encoded by a n-bit data that is input to a set of input ports. This simplifies the event detection mechanism but requires some circuitry overhead to multiplex and route to the input port the value associated to the last occurred event.

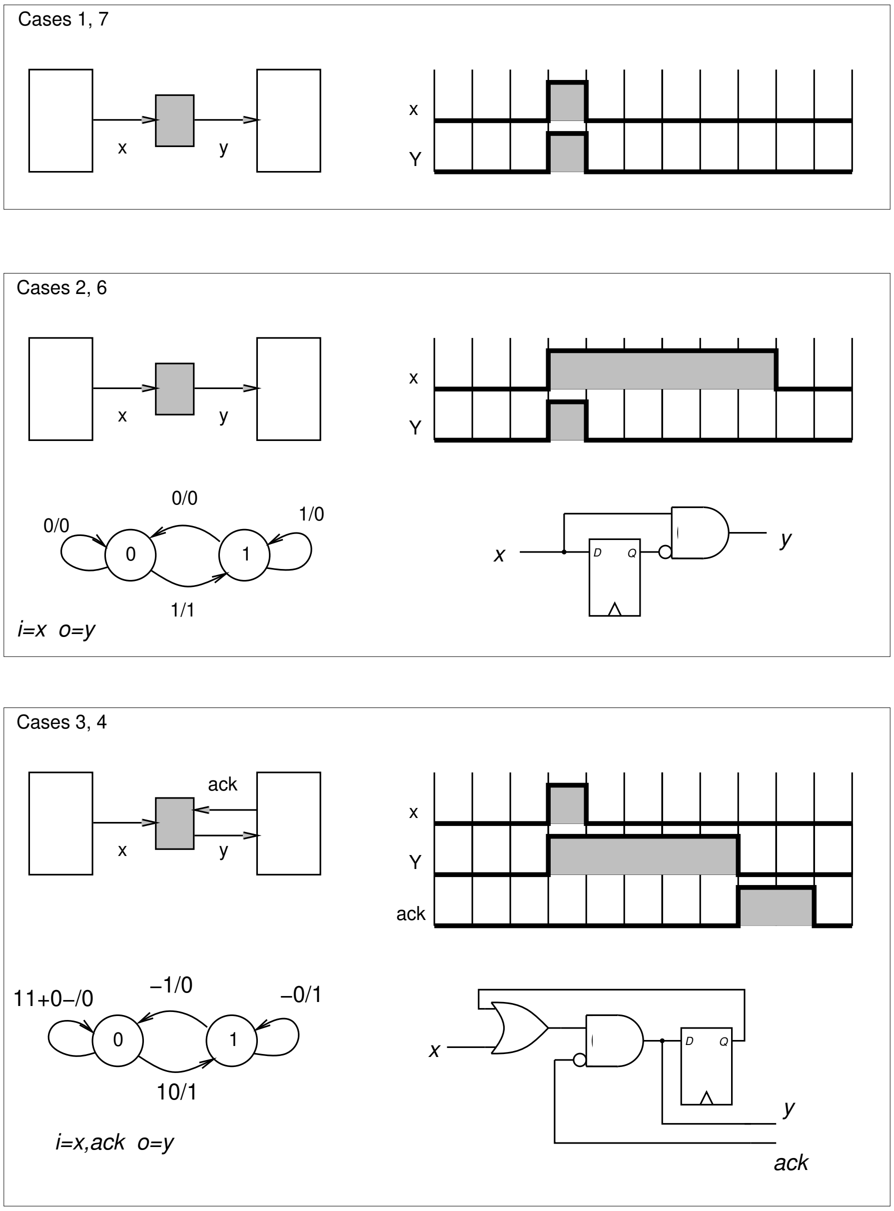

Figure 8: Interface types

7 Software Optimization

A class of useful optimizations for software implementations is the *collapsing* of multiple `.names` tables into a single one as well as the *checking* of input *don't cares*.
In SHIFT each `.model` may be made up of a set of `.names` tables. Each `.names` table is synthesized as a software task. This technique represents a technology mapping step from the specification level (SHIFT) to a particular implementation style (C code). However, as in logic synthesis, an optimization step can be useful to improve some parameters of the target implementation. As far as software is concerned, the goal is to reduce code size and to speed up execution time. It is worthwhile noting that the optimization step is not always necessary as in real-time systems the goal is not to design something which is as fast as possible but something which meets the timing constraints. Same considerations can apply to code size reduction.
Let us suppose we are given a particular application in which code size is a strong constraint. In this case, synthesizing each `.names` table separately may not be convenient because we cannot benefit from statements which are shared by different tasks. Let us look at a very simple example:

```
.model foo
.inputs X
.outputs Y Z
.mv Y 2 1 2
.names X => Y
 0    1
 1    2
.end
.names X => Z
 0    1
.end
```

We can directly apply the technology mapping step and implement each `.names` table as a task. The first `.names` table is implemented by the following task (for the sake of clarity a pseudo-C code is used):

```
task1(X) {
 if(X==0) Y := 1;
 if(X==1) Y := 2;
}
```

The second `.names` table is implemented as follows:

```
task2(X) {
 if(X==0) Z := 1;
}
```

However, if we collapse the two tasks into a single task we can exploit the common statement `if(X==0)` and get a smaller code size:

```
task(X)
{
 if(X==0)
 {
   Y := 1; Z := 1;
 }
 if(X==1) Y := 2;
}
```

Besides collapsing of different `.names` tables there is another class of optimizations based on the detection of *input don't cares*. For example, suppose to have a model as follows:

```
.model foo
.inputs X Y Z
.outputs K
.names X Y Z => K
 0 1 1   1
 1 1 1   1
.end
```

where `X` is binary variable. Obviously, the value of `K` does not depend on the value of `X`, so the table, after the optimization step, can be implemented by the following task:

```
task(Y,Z)
{
 if(Y==1 and Z==1) K := 1;
}
```

The input part of a row of a `.names` table is a cube. In fact, it's an *and* of (in general) multi-valued variables. Hence, the basic idea to reduce the code size is to benefit from cubes which are common to different `.names` tables as well as cubes which are contained in other ones in the same `.names` table. As it will be explained later in this chapter, the optimization step is equivalent to the minimization of a set of multi-valued boolean functions. We will use Espresso-mv([RSV87]) to do such minimization.

7.1 Software Primitives and Size of a Task

Since the goal of our technique is to reduce the code size of the synthesized software we define both the objects the technology mapping is based on and the way to measure the size of the synthesized software with respect to those objects. We use four software objects: the test statement, implemented by the `if(predicate)`, the `data value assignment`, the `emit(event)` statement which implements `event emission`, and `occurred(pattern,event)` which is used in the `if` predicate for checking if the event `event` is contained in the pattern of events `pattern`. In general, the predicate

of the `if` statement is in the `sum-of-product` form.
The measure of code size of a software implementation is an estimate because we cannot make any strong assumption about the results produced by the optimized compiler used to produce the assembly code from C code. Moreover, we are not concerning about any particular compiler. Since the structure of the software implementation is very simple and regular (a task is a sequence of `if` statements, assignments and `event emission` statements) we compute the size of a task by adding three contributions:

- the number of `if` statements
- the number of `emit` statements
- the number of `data value` assignments

Then, given different tasks the total code size is given by the sum of the task sizes.
As the optimization step can reduce both the number of `if` statements and the number of `data value` assignments and the number of `event emission` statements, this is a meaningful measure of code size reduction. In the first example, the size of `task1` is 4, the size of `task2` is 2, the total size is 6, and the size of the *reduced* task `task` is 5.

7.2 Basic Assumptions

We implement a task by using `if` statements. For example, given two tasks:

```
task1(X,Y)
{
 if(X==1 and Y==0) Z := 1;
}
```

```
task2(X)
{
 if(X==1) W := 1;
}
```

We can combine them in a single one:

```
task(X,Y)
{
 if(X==1 and Y==0) Z := 1;
 if(X==1) W := 1;
}
```

This is a correct composition of the two tasks because the input-output behavior of the task `task` is exactly the same as that of `task1` and `task2`. In order to make sure that we synthesize correct implementation we require `.names` tables to be deterministic. Non-determinism which is allowed in our methodology at the specification level [CLJ+93] must be resolved by the designer before the synthesis step.
A `.names` table is non-deterministic if given two rows with the same input part, there exists at least a variable (either trigger event or data value) such that the correspondent values are different. More formally:

Definition 1 *Let* $C = (I, E, O, R, F)$ *be a CFSM specified by the* `.names` *table* $N = (I, O, S)$ *where* I *is the set of input event variables,* O *is the set of output event variables,* S *is the set of rows of values of the table. Then,* N *is non-deterministic if there exist two rows* $r_k = (i_k, o_k)$, $r_j = (i_j, o_j)$, $r_k, r_j \in S$, $k \neq j$, *where* i_k, i_j *are cubes (possibly in the multi-valued logic sense) such that* $i_i \cap i_j \neq \emptyset$ *,* o_k, o_j *are set of output values and there exists at least an output value* $o_e \in o_i \cap o_j$ *for the same output event variable* $o \in O$*.*

For example, the table below is non-deterministic because the cube `XYZ` is contained in the cube `YZ`, and `H` takes values 2 and 1 for the same input pattern `X=1, Y=1, Z=1`:

```
.names X Y Z => H
1 1 1  2
- 1 1  1
.end
```

This `.names` table is implemented by the following task:

```
task(X,Y,Z)
{
 if(X==1 and Y==1 and Z==1) H := 2;
 if(Y==1 and Z==1) H := 1;
}
```

However, there's a problem because `H` is assigned two different values for the same input pattern. Depending on the order in which the `.names` table rows are implemented `H` will be assigned either 1 or 2. Given a deterministic `.names` table this situation will never happen.
Yet, having deterministic `.names` table is not enough when composing tasks. However, since in the CFSM model we don't allow two different CFSMs to have two outputs in common ([CLJ+93]) any composed task will be correct. In fact, it's never the case the same variable is assigned two different values for the same input pattern.

7.3 Software Optimization Technique

The problem of reducing the code size is equivalent to the problem of minimizing a set of multi-valued boolean functions. We use Espresso-mv for the minimization. The technique is based on four steps:

- *collapsing* the `.names` tables in a single one
- *mapping* the resulting `.names` table into a Espresso-mv input table
- run Espresso-mv on the input table
- synthesize the software implementation from the minimized table

In the first step, we create two new sets of input event variables and output event variables and then a new table by composing the original ones. We use a new symbol U which represents the `unknown action`. Formally:

Definition 2 *Given two CFSMs $C_1 = (I_1, E_1, O_1, R_1, F_1)$ and $C_2 = (I_2, E_2, O_2, R_1, F_1)$ specified by two* `.names` *tables $N_1 = (I_1, O_1, S_1)$ and $N_2 = (I_2, O_2, S_2)$ such that $|S_1| = m$ and $|S_2| = n$ then, the collapsed* `.names` *table $N = (I, O, S)$ is defined by:*

- $I = I_1 \cup I_2$, $O = O_1 \cup O_2$
- *S such that $|S| = m + n$ is a set of rows $r_i = (i_i, o_i)$ where:*
  $i_i = (v', v'')$ where v', v'' are rows of input event values such that if v' is a row of input event values of S_1 then each value in v'' will be $'-'$ and if v'' is a row of input event values of S_2 then each value in v' will be $'-'$;
  $o_i = (w', w'')$ where w', w'' are rows of output event values such that if w' is a row of input event values of S_1 then each value in w'' will be U and if w'' is a row of input event values of S_2 then each value in w' will be U.

Let us see an example: we are given two `.names` tables $N_1 = (I_1, O_1, S_1)$ and $N_2 = (I_2, O_2, S_2)$ where $I_1 = \{X, Y, Z\}$, $I_2 = \{Y, Z\}$, $O_1 = \{H, K\}$, $O_2 = \{G, J\}$, $|S_1| = |S_2| = 2$. The new table is $N = (I, O, S)$ where $I = \{X, Y, Z\}$, $O = \{G, H, J, K\}$, and $|S| = |S_1| + |S_2| = 4$.

```
.names X Y Z => H K
5 1 0  1 1
4 1 0  1 1
.end

.names Y Z => G J
1 0  1 6
0 1  7 2
.end
```

By collapsing the two tables we obtain the following table:

```
.names X Y Z => H K G J
5 1 0   1 1 U U
4 1 0   1 1 U U
```

```
- 1 0   U U 1 6
- 0 1   U U 7 2
.end
```

For example, in the third row $r_3 = (i_3, o_3)$, $i_3 = (v', v'')$ where v'=- and v''=10 because v'' is the first row of input events of S_2, $o_3 = (w', w'')$ where w'=UU and w''=16 because w'' is the first row of input events of S_2. Notice that we used a new symbol U which represents the `unknown action`. We don't put either a zero or a don't care because we don't want the resulting table to be non-deterministic. For example, in the third row (which comes from the second table) H and K variables take value U. In fact, for input 510 both the first row and the third row are satisfied: if H in the third row were equal to 0 then for the same input the output H would be both equal to 1 and 0. Since Espresso needs a deterministic input we will map the U symbol in the appropriate way.
As a second step we map the collapsed table into an Espresso input table according to the following rules:

- any input multi-valued variable is an input multi-valued variable in 1-hot-encoding
- any input binary variable is an input binary variable
- all the output variables are mapped to a sigle 1-hot-encoding output multi-valued variable which takes values on the cartesian product of the values of the former multi-valued variables
- any `U` symbol is an empty multi-valued literal (a sequence of all 0's).

There are several observations about these rules:

- since we want to express `and` of input event variables we need as many variables as the input variables in the single SHIFT table
- as for the outputs, the interpretation is different: in fact, each value represents the fact that either an event is emitted or not or a pure data value is assigned to a variable, so we only need one multi-valued variable

Let us see how the collapsed table is mapped into an Espresso input table: let the range of X be {1,2,3,4,5}, the range of G be {1,2,3,4,5,6,7}, and the range of J be {1,2,3,4,5,6}; as a consequence, we need 5 bits to encode X, 7 bits to encode G, and 6 bits to encode J. Y, Z, H, and K are binary variables. As for the input part, no encoding is needed for the binary variables. As for the output, we encode each binary variable by using two bits.

```
.mv 4 2 5 17
.ib y z
.label var=2 x1 x2 x3 x4 x5
.label var=3 h0 h1 k0 k1 g1 g2 g3 g4 g5 g6 g7 j1 j2 j3 j4 j5 j6
1 0 00001 | 01 01 0000000 000000
```

```
1 0 00010 | 01 01 0000000 000000
1 0 11111 | 00 00 1000000 000001
0 1 11111 | 00 00 0000001 010000
.end
```

The table above shows the way the collapsed table is mapped to a set of multi-valued boolean functions. The set is defined by the multi-valued output variable: each column defines a different multi-valued boolean function, 1's denoting the minterms in on-set.
To see the third step let us go back to our first example:

```
.model foo
.inputs X
.outputs Y Z
.mv Y 2 1 2
.names X => Y
 0    1
 1    2
.end
.names X => Z
 0    1
.end
```

First, we have to collapse the two `.names` tables. The resulting table is as follows:

```
.names X => Y Z
0 1 U
1 2 U
0 U 1
.end
```

Then, we map the table to an Espresso input table: let the range of X be {0,1}, the range of Y be {1,2}, the range of Z be {0,1}.

```
.mv 2 1 4
.ib x
.label var=1 y1 y2 z0 z1
0 1000
1 0100
0 0001
.end
```

Then, we run Espresso which returns the following result:

```
.ib x
.i 1
.o 4
.ob y1 y2 z0 z1
.p 2
1 0100
0 1001
.e
```

which can be implemented as a single task:

```
task(X)
{
 if(X==1) Y := 2;
 if(X==0)
 {
   Y := 1; Z = 1;
 }
}
```

7.4 Example

In conclusion, the following example shows the effectiveness of our technique. We are given three `.names` tables. The first one represents the next-state function of an automaton, the other ones specify the output functions. Notice that if we implemented each `.names` table separately we would get three tasks, 42 `if` statements, and 43 output statements (either *event emission* or *data value* assignment). By using our technique we get a task with 14 `if` statements and 23 `output` statements.

```
.model joystick
.inputs DIRECTION *DIRECTION BUTTON RESET
.outputs UP DOWN LEFT RIGHT FIRE STOP
.mv DIRECTION    0 1 2 3
.mv STATE  2 3 4
.r state 2
.names STATE DIRECTION *DIRECTION *BUTTON *RESET => STATE
2  -  -  1  1  2
2  0  1  0  1  2
2  1  1  0  1  2
2  2  1  0  1  2
2  3  1  0  1  2
2  -  1  0  1  2
2  -  0  0  1  3
2  -  0  1  0  4
2  0  1  0  0  4
```

```
2  1  1  0  0  4
2  2  1  0  0  4
2  3  1  0  0  4
2  -  1  0  0  4
3  -  -  1  -  2
3  0  1  0  -  2
3  1  1  0  -  2
3  2  1  0  -  2
3  3  1  0  -  2
3  -  1  0  -  2
4  -  -  -  1  2


.names STATE DIRECTION *DIRECTION *BUTTON *RESET => *UP *DOWN *LEFT
2  0  1  0  1    1  0  0
2  1  1  0  1    0  1  0
2  2  1  0  1    0  0  1
2  0  1  0  0    1  0  0
2  1  1  0  0    0  1  0
2  2  1  0  0    0  0  1
3  0  1  0  -    1  0  0
3  1  1  0  -    0  1  0
3  2  1  0  -    0  0  1

.names STATE DIRECTION *DIRECTION *BUTTON *RESET => *RIGHT *FIRE *STOP
2  -  -  1  1    0  1  1
2  0  1  0  1    0  0  1
2  1  1  0  1    0  0  1
2  2  1  0  1    0  0  1
2  3  1  0  1    1  0  1
2  -  1  0  1    0  0  1
2  -  0  0  1    0  0  1
2  -  0  1  0    0  1  0
2  3  1  0  0    1  0  0
3  -  -  1  -    0  1  0
3  3  1  0  -    1  0  0
4  -  -  -  1    0  0  1
.end
```

Here is the single collapsed `.names` table:

```
.mv 6 3 3 4 15
.ib direction button reset
.label var=3  st2 st3 st4
.label var=4  direction0 direction1 direction2 direction3
.label var=5  st2 st3 st4 up0 up1 down0 down1 left0 left1 right0 right1
```

```
fire0 fire1 stop0 stop1
 - 1 1 100 1111 | 100 00 00 00 00 00 00
 1 0 1 100 1000 | 100 00 00 00 00 00 00
 1 0 1 100 0100 | 100 00 00 00 00 00 00
 1 0 1 100 0010 | 100 00 00 00 00 00 00
 1 0 1 100 0001 | 100 00 00 00 00 00 00
 1 0 1 100 1111 | 100 00 00 00 00 00 00
 0 0 1 100 1111 | 010 00 00 00 00 00 00
 0 1 0 100 1111 | 001 00 00 00 00 00 00
 1 0 0 100 1000 | 001 00 00 00 00 00 00
 1 0 0 100 0100 | 001 00 00 00 00 00 00
 1 0 0 100 0010 | 001 00 00 00 00 00 00
 1 0 0 100 0001 | 001 00 00 00 00 00 00
 1 0 0 100 1111 | 001 00 00 00 00 00 00
 - 1 - 010 1111 | 100 00 00 00 00 00 00
 1 0 - 010 1000 | 100 00 00 00 00 00 00
 1 0 - 010 0100 | 100 00 00 00 00 00 00
 1 0 - 010 0010 | 100 00 00 00 00 00 00
 1 0 - 010 0001 | 100 00 00 00 00 00 00
 1 0 - 010 1111 | 100 00 00 00 00 00 00
 - - 1 001 1111 | 100 00 00 00 00 00 00
 1 0 1 100 1000 | 000 01 00 00 00 00 00
 1 0 1 100 0100 | 000 10 01 10 00 00 00
 1 0 1 100 0010 | 000 10 10 01 00 00 00
 1 0 0 100 1000 | 000 01 10 10 00 00 00
 1 0 0 100 0100 | 000 10 01 10 00 00 00
 1 0 0 100 0010 | 000 10 10 01 00 00 00
 1 0 - 010 1000 | 000 01 10 10 00 00 00
 1 0 - 010 0100 | 000 10 01 10 00 00 00
 1 0 - 010 0010 | 000 10 10 01 00 00 00
 - 1 1 100 1111 | 000 00 00 00 10 01 01
 1 0 1 100 1000 | 000 00 00 00 10 10 01
 1 0 1 100 0100 | 000 00 00 00 10 10 01
 1 0 1 100 0010 | 000 00 00 00 10 10 01
 1 0 1 100 0001 | 000 00 00 00 01 10 01
 1 0 1 100 1111 | 000 00 00 00 10 10 01
 0 0 1 100 1111 | 000 00 00 00 10 10 01
 0 1 0 100 1111 | 000 00 00 00 10 01 10
 1 0 0 100 0001 | 000 00 00 00 01 10 10
 - 1 - 010 1111 | 000 00 00 00 10 01 10
 1 0 - 010 0001 | 000 00 00 00 01 10 10
 - - 1 001 1111 | 000 00 00 00 10 10 01
 .end
```

Espresso returns:

```
.ib direction button reset
.mv 6 3 3 4 15
.ob st2 st3 st4 up0 up1 down0 down1 left0 left1 right0 right1 fire0 fire1 stop0 stop1
.label var=3 st2 st3 st4
.label var=4 direction0 direction1 direction2 direction3
.p 16
100 100 1000 | 000 00 10 10 00 00 00
100 100 0001 | 000 00 00 00 00 10 10
10- 100 1000 | 000 01 00 00 00 00 00
10- 100 0001 | 000 00 00 00 01 00 00
10- 010 1000 | 100 01 10 10 00 00 00
10- 110 0100 | 000 10 01 10 00 00 00
10- 110 0010 | 000 10 10 01 00 00 00
10- 010 0001 | 100 00 00 00 01 10 10
010 100 1111 | 001 00 00 00 10 01 10
1-- 010 0110 | 100 00 00 00 00 00 00
100 100 1111 | 001 00 00 00 00 00 00
001 100 1111 | 010 00 00 00 10 10 01
-11 100 1111 | 100 00 00 00 10 01 01
101 100 1111 | 100 00 00 00 10 10 01
--1 001 1111 | 100 00 00 00 10 10 01
-1- 010 1111 | 100 00 00 00 10 01 10
.e
```

Each row represents an `if` statement. As far as input multi-valued variables are concerned, the don't care is represented by `1111`. In that case, we don't include those variables in the predicate of the `if` statement. As for outputs, we emit the event if the corresponding variable has value 1. The first row is not synthesized because the 1-hot-encoding value of the output variable corresponds to **down=0, *left=0*. Same considerations apply to the second row. In the remaining rows, there is at least an event to emit or a data value to assign so we synthesize them.

8 System Validation

System validation is needed to ensure 1) correctness of an implementation with respect to specification and 2) correctness with respect to some user-defined property. The former, sometime called implementation verification, is accomplished by construction in [CLJ+93] for implementations of CFSM specifications. The latter can be divided into specification verification and design verification. Specification verification is done when the property to be validated is implementation-independent. Design verification is done when implementation-dependent parameters are used. These tasks are performed mostly by prototyping and simulation. Prototyping is clearly expensive in turn-around time, and, in addition, cannot be performed until most of the detailed design is completed. Simulation is expensive in terms of CPU time. For complex systems, only relatively few input patterns can be tried, thus reducing power of simulation in exposing errors in design. Formal verification is a technique that allows for proving mathematically that a certain specification is met by an implementation or that some formally specified properties are true for the design. This technique is obviously very powerful; however, its computational complexity is very high. We believe that formal verification can be a useful tool for early error detection. In fact, our formal model for HW/SW systems has been driven partially by the desire of carrying out validation using formal verification techniques.

8.1 Formal Verification

Several approaches have been proposed for formal verification. The one that, in our opinion, is most suited for the task at hand is based on finite state machines. If the properties are described by automata, then formal verification corresponds to the language containment problem. If the properties are expressed as statements in a logic language, such as Computation Tree Logic (CTL)[EC82], then the verification problem is called model checking.

The first step in formal verification is to define a time model of the system. In [CLJ+93], the behavior in time of a SHIFT specification is formally defined. A CFSM C is represented by the corresponding FSM network $\mathcal{N}^{\mathcal{F}}$ which is composed of a "main" completely specified FSM $\mathcal{F}$, input FSMs, and output FSMs. The behavior specified by $\mathcal{N}^{\mathcal{F}}$ is an abstracted specification of the system's real behavior in the sense of Kurshan [Kur90]. The non-determinism in the input and output FSMs reflects unbounded delay. For the network of CFSMs $\mathcal{N}^{\mathcal{C}}$, there is a corresponding model M of network of $\mathcal{N}^{\mathcal{F}}$ Any implementation I of $\mathcal{N}^{\mathcal{C}}$ is contained in M, or $I \subseteq M$. An example of verification is to verify that a given sequence of transitions σ, an undesirable behavior, is not consistent with M. If $\sigma \not\subseteq M$, then $\sigma \not\subseteq I$, since $I \subseteq M$.

We can use CTL formula [BCMD90] to express properties. To cast this model checking problem into a language containment problem, we will only need to convert the CTL formula into a task automata. With CTL, the property "Alarm will not be on forever" in our seat belt example, can be expressed as

$$AG(ALARM\_ON \rightarrow AF(ALARM\_OFF))$$

where, given a predicate f

- AGf means for every path, at every node on the path f holds
- AFf means for every path, there exists a state on the path at which f holds.
- $\rightarrow$ is the implication.

Path is defined as a sequence of states and is used to represent possible behavior or computations of the transition system[CBG+92]. This CTL formula literally describes the behavior, "It is always true that ALARM_ON is true implies that for all path there exists a state where ALARM_OFF is true."

Note that the property is expressed in terms of paths and states of an FSM. Most successful verification systems visit the states of the system using Binary Decision Diagram (BDD) based implicit techniques.

A "real" example of the use of formal verification technique is as follows: In a digital dashboard designed at a company in the automotive industry, an oil level probe device controlled by software could malfunction if kept powered for over 20 minutes. The enabling signal comes in the form of a square wave of 50ms period. The controlling algorithm can be abruptly stopped when the engine is started. In that case, the probe should be disabled. As it turned out, the designer overlooked that bit of information, and the probe was not explicitly disabled. The result is that the probe either remains enabled or disabled depends on the phase of the wave. The prototype of the system passed all tests. When the malfunction was discovered, it took an entire day of attempts to reproduce the faulty behavior. In our methodology, the following formula

$AG(KEY\_ON \rightarrow (AG\,\neg PROBE\_ON))$

where $AG, \rightarrow$ is the same as defined before, could have been used to describe the desired property. A qualitative model checker (i.e. one that does not handle timing explicitly) could have exposed the error early in the design stage. However, we must stress that formal verification is still a research topic and not widely used in practical cases. This is due to two main causes: 1) Result is related to the way in which properties are expressed and system described. 2) Computational complexity. In fact, the possibility of an explosion in the size of the internal representation in a formal verification tool such as COSPAN [HK87] is a major obstacle to the extensive application of formal verification techniques to real industrial designs. Reduction techniques are being proposed to cope with this problem. Automatic techniques have been proposed to reduce the complexity of CTL model checking [CSSVB92, BSV92]. Non-automatic techniques that make use of abstraction, like the one proposed by Kurshan [Kur90] to verify ω-regular properties and the one proposed by Grumberg et al [DLG92] to perform model checking using the restricted temporal logic CTL*, seem powerful. However, their effectiveness depends heavily on the designer's ingenuity. For a complete example of qualitative real-time formal verification within our methodology, see [CSV92].

Implementation-dependent design verification is very important if all or parts of the system have been through the entire design process. It should be clear that the behavior of M is an abstraction of the real time behavior of a system. Between this and the behavior of a given implementation I, there is a number of intermediate FSMs M', $M \supseteq M' \supseteq I$, which can be used for system verification. Such an M' can be obtained by composing M with more precise and less abstracted *implementation-specific* components such as *Timing Descriptors* T which capture information about what the delays may be in a given implementation. That is, using transition relations, we can build a $M' = M \cdot T$ where T restricts the non-determinism of M [Kur90]. The FSM specified by M' can be used as an input to a formal verification algorithm. The composition with timing descriptors is not different from what Alur et al. proposed [ACD90].

A non-trivial problem is how to derive the timing descriptor of a transformation. The information needed must be provided by the synthesis process. This is trivial for hardware, in which there is always a 1-clock delay. For software, it may be desirable to be able to compute the maximum run time, the minimum, the average, or give a probability distribution.

For our seat belt example, we want to figure out if, according to our description, the seat belt will never start buzzing from 15 seconds after the key has been turned on. In TCTL [ACH+92] notation, this becomes:

$$AG(KEY\_ON \rightarrow AF_{(>15)s}(AG\ \neg ALARM\_ON))$$

where $AF_{(>15)s}f$ means for every path, there exists a state on the path at which f holds after 15 seconds.

When precise timing information has to be taken into account, the complexity of formal verification grows. For example, let's assume that the seat belt alarm be implemented with a hardware timer and with software for control. For the hardware timer, we can replace the main, input and output FSMs with a verification timer/counter abstraction. The system has a total of 9 verification timers (5 for software input, 3 for software output, 1 for hardware). The verification timers associated with input and output FSMs are used to restrict non-determinism as previously discussed. It is a consequence of the composition of $\mathcal{N}^{\mathcal{F}}$ with the implementation-dependent timing descriptor. There are also over 2000 system states (i.e. cartesian product of the state spaces of the FSMs). Moreover, the based clock of the system will be in the range of microseconds for typical microcontroller, while for the hardware timer we need to count 5 and 10 seconds. Immediately we see a state-explosion problem.

As an experiment, we formulate this seat belt design into a network of $\mathcal{N}^{\mathcal{F}}$ as in[CLJ+93]. We then use iterative method for quantitative real-time verification as proposed in [BSV92] to verify the property "Alarm will not be on forever" as described earlier. The verification is allowed to run overnight on a DEC Alpha machine and only intermediate result is obtained. The size of the representation simply grows too large for the verification algorithm to reach its conclusion. Before the algorithm is terminated, BDD node count have grown in excess of 180,000. Other reduction techniques are being investigated to reduce the complexity of real-time model checking, as discussed in [Kur90, CDHWT92, HNSY92].

8.2 Simulation

Simulation should be used whenever the complexity of the system makes formal verification infeasible. Since the system is inherently discrete, the number of distinct traces of the system with bounded delay given by timing descriptors will be finite. It is then possible to simulate all non-deterministic paths. In general, only a few "corner" cases chosen by designer need to be simulated.

It is interesting to see how formal verification and simulation can be used together in a complementary fashion. In an early phase of the design, a more abstracted model of the system, or even just models of single components, can be used to formally verify whether an unwanted situation can occur. If formal verification shows that this can happen (for example, a "bad" state can indeed be reached in the abstracted model), the designer will then turn to simulation. Simulation will be done on a more detailed model of the system. Those paths that were involved in the failed formal verification will be tried. The reason for simulating the failed path is to make sure the failure is real. Since our model for verification is quite abstracted, it contains many behavior that is not part of the implementation behavior. Simulation done on a more detailed model can identify these "false path".

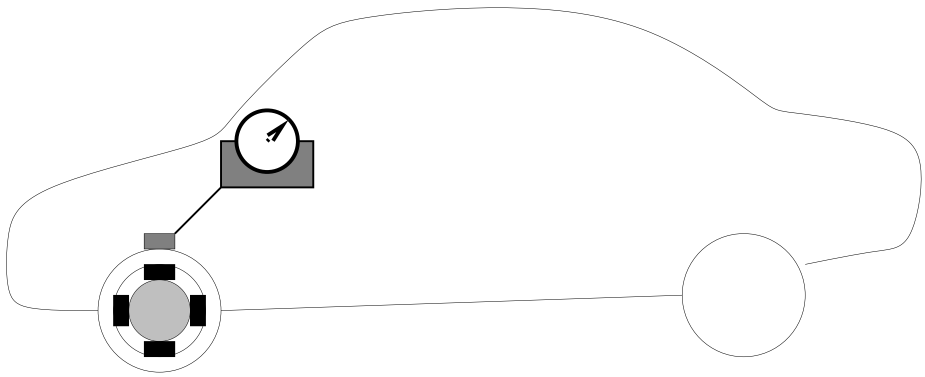

Figure 9: A car Dashboard

9 An Example: a Car Dashboard

The example we propose is a subsystem of a car dashboard. The functions considered here are the odometer and the speedometer. The system is structured as follows: a proximity sensor placed near the wheel shaft detects the passing of an indentation. At each passage a pulse is sent to the dashboard. There are usually four indentations on the shaft, so that each pulse represents 1/4 of a wheel round. The dashboard senses the pulses from the wheel and displays the current speed and the total amount of miles run. Such a system, shown in figure 9, has to:

1. Measure the instantaneous speed by counting wheel pulses in a given time interval. The length of the time interval is chosen so that the number of pulses counted in one interval gives the speed in the desired unit, like km/h.
2. Filter the speed value to improve resolution. The instantaneous speed is added to the previous average speed and the total is divided by two.
3. Drive a PWM signal proportionally to the value of the filtered speed. The angular position of the speed gauge is proportional to the saturation of the PWM period. For example, a 1/2 duty cycle puts the gauge at half scale.
4. Accumulate speed pulses. Every K pulses refresh the odometer display.

Let v be the speed, and f the frequency of the wheel pulses. v is given by

$$v[km/h] = K \times f[s^{-1}]$$

where K is a unit conversion constant. To avoid computing $K \times f$, we want to measure a frequency g in unit such that the value of g equals the corresponding speed in km/h. If I is a constant that gives the pulse/meter ratio of the wheel, we can measure the speed in Km/h by counting the pulses within an time interval of $t = I/3.6$.

Figure 10 depicts a data-flow diagram of the system. *Structured Analysis* [PM85] notation is used. Dashed arrows represent pure events. Solid lines represent data flows. Rectangles represent memories. Ellipses represent transformations.

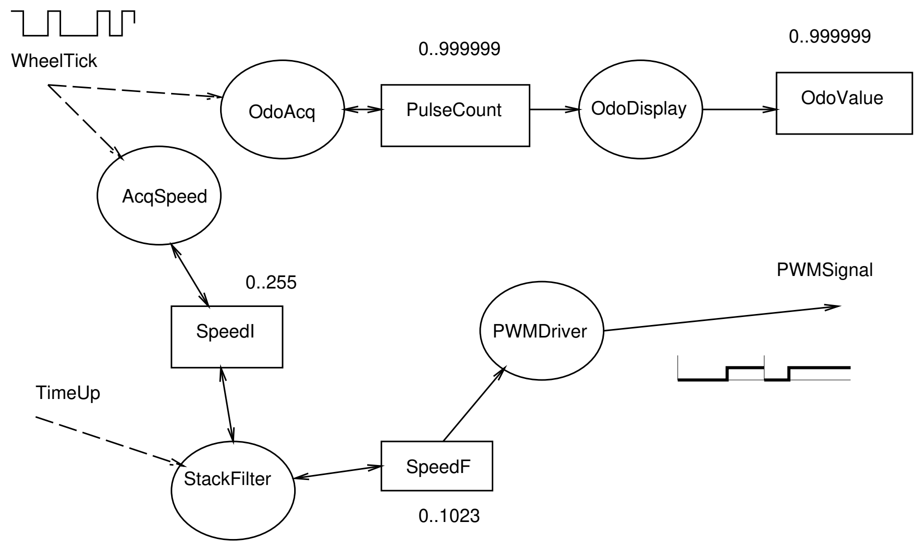

Figure 10: Dashboard flow-chart

A SHIFT specification of the system is as follows:

```
.model dashboard
.inputs *reset *WheelTick *TimeUp *tick
.outputs PWMSignal OdoValue
.mv OdoValue 999999
.mv PWMSignal 2 L H
.mv PulseCount 999999
.mv SpeedI 256
.mv SpeedF 256
.subckt AcqSpeed 1    *reset=*reset *e=*WheelTick  y=SpeedI
.subckt StackFilter 2 *reset=*reset *e=*TimeUp    x=SpeedI  y=SpeedF
.names  PWMDriver 3   *reset=*reset *e=*tick      x=SpeedF  y=PWMSignal
.subckt OdoAcq 4      *reset=*reset *e=*WheelTick y=PulseCount
.subckt OdoDisplay 5  *reset=*reset x=PulseCount  y=OdoValue
.end

.model AcqSpeed
.inputs *reset *e
.outputs y
.mv y 256
.names *reset *e y_INC => y
1 - - 0
0 1 - (y_INC)
.subckt INC_8 1 a=y i=y_INC c=cc
.end

.model StackFilter
.inputs *reset *e x
```

```
.outputs y
.mv x 256
.mv y 512
.mv x_INTERNAL 256
.mv x_y_FILTER 256
.mv x_PLUS_y 512
.names *reset *e x x_y_FILTER => x_INTERNAL y
1 - - - (x) 0
0 1 - - 0   (x_y_FILTER)
.subckt DIV2_9_8 1 a=x_PLUS_y d=x_y_FILTER
.subckt ADDER_8_9  2 a=x_INTERNAL b=y s=x_PLUS_y c=cc
.end

.model OdoAcq
.inputs *reset *e
.outputs y
.mv y 999999
.names *reset *e y_INC => y
1 - - 0
0 1 - (y_INC)
.subckt INC_17 1 a=y i=y_INC
.end

.model OdoDisplay
.inputs *reset x
.outputs y
.mv x 999999
.mv y 999999
.mv x_y_DIFF_OK 2 Y N
.mv x_DIFF_y 999999
.mv CONST 999999
.names const
3000
.names *reset *always x x_y_DIFF_OK => y
1 - - -  0
0 1 - Y  (x)
.names *reset *always => *always
1 -  1
- 1  1
.subckt SUB_17 1 a=x b=y d=x_DIFF_y
.subckt GTE_12  2 a=x_DIFF_y b=const g=x_y_DIFF_OK
.end

.model PWMDriver
.inputs *reset *e x
.outputs y
.mv y 2 L H
```

```
.mv x 256
.mv n 256
.mv k 256
.names n x => k
0 -  (x)
.names *tick n n_EQ_k => y
1 0 0   L
1 - 1   H
.names *reset *e n_INC => n
1 - - 0
0 1 - (n_INC)
.subckt EQ_8  1 a=n b=k e=n_EQ_k
.subckt INC_8 2 a=n b=n_INC c=cc
.end
```

A hardware implementation follows. The library models used are not listed.

```
.model dashboard
.inputs $reset $WheelTick $TimeUp $tick
.outputs PWMSignal OdoValue
.mv OdoValue 999999
.mv PWMSignal 2 L H
.mv PulseCount 999999
.mv SpeedI 256
.mv SpeedF 256
.subckt AcqSpeed 1    $reset=$reset $e=$WheelTick y=SpeedI
.subckt StackFilter 2 $reset=$reset $e=$TimeUp    x=SpeedI  y=SpeedF
.names  PWMDriver 3   $reset=$reset $e=$tick      x=SpeedF  y=PWMSignal
.subckt OdoAcq 4      $reset=$reset $e=$WheelTick y=PulseCount
.subckt OdoDisplay 5  $reset=$reset x=PulseCount  y=OdoValue
.end

.model AcqSpeed
.inputs $reset $e
.outputs y
.mv y 256
.mv _y 256
.names y $reset $e y_INC _y
- 1 - -   0
- 0 1 -   (y_INC)
- 0 0 -   (y)
.subckt INC_8 1 a=y i=y_INC c=cc
.latch _y y
.end

.model StackFilter
```

```
.inputs $reset $e x
.outputs y
.mv x 256
.mv y 512
.mv _y 512
.mv x_INTERNAL 256
.mv _x_INTERNAL 256
.mv x_y_FILTER 256
.mv x_PLUS_y 512
.names x_INTERNAL $reset $e x _x_INTERNAL
- 1 - -   (x)
- 0 1 -   0
- 0 0 -   (x_INTERNAL)
.names y $reset $e x_y_FILTER _y
- 1 - -   0
- 0 1 -   (x_y_FILTER)
- 0 0 -   (y)
.subckt DIV2_9_8 1 a=x_PLUS_y d=x_y_FILTER
.subckt ADDER_8_9  2 a=x_INTERNAL b=y s=x_PLUS_y c=cc
.latch _x_INTERNAL x_INTERNAL
.latch _y y
.end

.model OdoAcq
.inputs $reset $e
.outputs y
.mv y 999999
.mv _y 999999
.names y $reset $e y_INC _y
- 1 - -   0
- 0 1 -   (y_INC)
- 0 0 -   (y)
.subckt INC_17 1 a=y i=y_INC
.latch _y y
.end

.model OdoDisplay
.inputs $reset x
.outputs y
.mv x 999999
.mv y 999999
.mv _y 999999
.mv x_y_DIFF_OK 2 Y N
.mv x_DIFF_y 999999
.mv CONST 999999
.names const
3000
```

```
.names y $reset $always x x_y_DIFF_OK _y
- 1 - - -  0
- 0 1 - Y  (x)
- 0 1 - N  (y)
- 0 0 - -  (y)
.names $reset $always _$always
1 -  1
- 1  1
0 0  0
.subckt SUB_17 1 a=x b=y d=x_DIFF_y
.subckt GTE_12  2 a=x_DIFF_y b=const g=x_y_DIFF_OK
.latch _y y
.latch _$always $always
.end

.model PWMDriver
.inputs $reset $e x
.outputs y
.mv y 2 L H
.mv _y 2 L H
.mv x 256
.mv n 256
.mv _n 256
.mv k 256
.names n x k
0 -  (x)
.names y $tick n n_EQ_k _y
- 1 0 0   L
- 1 - 1   H
- 1 1 0   (y)
- 0 - -   (y)
.names n $reset $e n_INC _n
- 1 - - 0
- 0 1 - (n_INC)
- 0 0 - (n)
.subckt EQ_8  1 a=n b=k e=n_EQ_k
.subckt INC_8 2 a=n b=n_INC c=cc
.latch _y y
.latch _n n
.end
```

A software implementation is shown below.

```
/* version 1.0 (6 apr 93) */

file : ``SWLIB.h''
```

```
/*******************************************************************/
#define INC_8(x) (int8)x+1;
#define INC_32(x) (int32)x+1;
/*******************************************************************/

file : ``qdb.c''
/*******************************************************************/
/* version 1.0 (apr/93)*/

#include SWLIB.h

#define ADDR_A <io port addr. >
#define ADDR_B <io port addr. >
#define ADDR_C <io port addr. >

#define IN_PORT_A ADDR_A
#define IN_PORT_B ADDR_B
#define IN_PORT_C ADDR_C

typedef char  int8;
typedef short int16;
typedef long  int32;

typedef void (*foop_t)();   /*foop_t = pointer to a function returning void*/

#define N_PROCS 5           /* number of processes */

/* process names */
#define _p_AcqSpeed_1    0
#define _p_StackFilter_1 1
#define _p_PWMDriver_1   2
#define _p_OdoAcq_1      3
#define _p_OdoDisplay_1  4

foop_t proc[N_PROCS];       /* array of pointers to functions */
int inp_event[N_PROCS];     /* array of function input buffers (in order)*/

/* event id's */
#define _e_reset     0
#define _e_WheelTick 1
#define _e_TimeUp    2
#define _e_tick      3

/* io_port[event_id]   : addr where event is to be read */
/* io_bit [event_id]   : bit of IO_port carries the event
/* event_sens_tabl[n]  : event, process that is sensitive to event,
                         n is the index of the event/proc relation element */
```

```
/* each "column" specifies INP_port and bit in the port for an inp event */
/*                        reset       WheelTick   TimeUp      tick        */
int *event_io_port[] = { IN_PORT_A,  IN_PORT_A,  IN_PORT_B,  IN_PORT_C };
int  event_io_bit[]  = { 0x01,       0x02,       0x04,       0x08      };

#define N_E_P_REL 9   /* size of event/proc relation = rows/3 of event_sens_tabl*/

int event_sens_tabl[] = { _e_reset, _p_AcqSpeed_1,    0x01,
                          _e_reset, _p_StackFilter_1, 0x01,
                          _e_reset, _p_PWMDriver_1,   0x01,
                          _e_reset, _p_OdoAcq_1,      0x04,
                          _e_reset, _p_OdoDisplay_1,  0x01,

                          _e_WheelTick, _p_AcqSpeed_1, 0x02,
                          _e_WheelTick, _p_OdoAcq_1,   0x08,

                          _e_TimeUp, _p_AcqSpeed_1,    0x04,
                          _e_TimeUp, _p_StackFilter_1, 0x02,

                          _e_tick, _p_PWMDriver_1, 0x02,
        };

/* constants that define which bit of a process' inp buff carry an event */
/* the ...._1 means event must be 1, ....._0 event must be 0             */
/* used by "occurred"        zero  one                                   */
int _c_AcqSpeed_n1_c1[2] = { 0x00, 0x01 };   /* reset */
int _c_AcqSpeed_n1_c2[2] = { 0x01, 0x04 };   /* !reset * TimeUp */
int _c_AcqSpeed_n1_c3[2] = { 0x05, 0x02 };   /* !reset * !TimeUp * WheelTick */
int _c_OdoAcq_n1_c1[2]   = { 0x00, 0x04 };   /* reset */
int _c_OdoAcq_n1_c2[2]   = { 0x04, 0x08 };   /* !reset * WheelTick */

/* returns != 0 if event is found in io_port */
#define got_inp_event(event) (event_io_bit[event] & *event_io_port[event])

/* set a bit in proc's inp_event buffer */
#define put_inp_event(pos) (inp_event[event_sens_tabl[pos+1]] = \
     inp_event[event_sens_tabl[pos+1]] | event_sens_tabl[pos+2])

/* Reset latched events : see section "interfaces" */
void
acknowledge_events() {
  ..........
};

/* check io_ports to find events */
void
```

```
assign_events_to_proc_buffs()
{
   int n, pos, event;

   for (n=0; n<N_E_P_REL; n++) {
      pos = n * 3;
      event = event_sens_tabl[pos];
      if (got_inp_event(event)) {
         put_inp_event(pos);
      }
   }
   (void) acknowledge_events();
}

/* check inp_event buffer */
#define ZEROES  0
#define ONES    1
#define occurred(cond, e) (((cond[ZEROES] & e) == 0x0) \
      && (cond[ONES] == (cond[ONES] & e)))

/* set io_port or internal port */
#define emit(event) (*event_io_port[event] = \
      *event_io_port[event] | event_io_bit[event])


/* ---------------------------------------------------------------------- */

/*Initialize processes' vars*/
   static int8  _v_SpeedI      = 0;
   static int8  _v_s           = 0;
   static int32 _v_PulseCount  = 0;

/* processes from SHIFT spec */

void
_t_AcqSpeed_1(e)
int e; /* process' inp_event buffer */
{
   if (1) {
      if (occurred(_c_AcqSpeed_n1_c1,e)) {
         _v_s = 0;
         _v_SpeedI = 0;
      }
      else if (occurred(_c_AcqSpeed_n1_c2,e)) {
         _v_SpeedI = _v_s;  /* problem!!! how to decide the order ? */
         _v_s = 0;
      }
```

```
      else if (occurred(_c_AcqSpeed_n1_c3,e)) {
         _v_s = INC_8(_v_s);
         _v_SpeedI = _v_SpeedI;
      }
   }
}

void
_t_StackFilter_1(e)
int e; /* process' inp_event buffer */
{
 ........
}

void
_t_PWMDriver_1(e)
int e; /* process' inp_event buffer */
{
 .......
}

void
_t_OdoDisplay_1(e)
int e; /* process' inp_event buffer */
{
  ..........
}

void
_t_OdoAcq_1(e)
int e; /* process' inp_event buffer */
{
   if (1) {
      if (occurred(_c_OdoAcq_n1_c1,e)) {
         _v_PulseCount = 0;
      }
      else if (_c_OdoAcq_n1_c2,e) {
         _v_PulseCount = INC_32(_v_PulseCount);
      }
   }
}


/* Initialize processes */
void
init_procs()
{
```

```
   int p;

   for (p=0; p<N_PROCS; p++) {
      inp_event[p] = 0x0;
   }

   proc[_p_AcqSpeed_1]    = *_t_AcqSpeed_1;
   proc[_p_StackFilter_1] = *_t_StackFilter_1;
   proc[_p_PWMDriver_1]   = *_t_PWMDriver_1;
   proc[_p_OdoAcq_1]      = *_t_OdoAcq_1;
   proc[_p_OdoDisplay_1]  = *_t_OdoDisplay_1;
}

/* ------------------------------------------------------------------ */

/*main scheduler*/
main()
{
   int p;

   init_procs();
   for(;;) {
      assign_events_to_proc_buffs();
      for(p=0; p<N_PROCS; p++) {
         if(inp_event[p]!=0) {
            (proc[p])(inp_event[p]);
            inp_event[p] = 0;
         }
      }
   }
}
```

10 Conclusions and Future Work

This paper introduced a methodology for partitioning and synthesis of hardware-software systems specified with CFSMs. This methodology satisfies the fundamental requirements outlined in Section 1:

1. *Natural specification.* Our methodology utilizes a formal specification model called CFSM, that is well suited to model control-dominated systems[CLJ+93].
2. *Easy partitioning.* Since both hardware and software have the same CFSM representation at the specification level, we do not have to commit ourselves to a particular mix of software-hardware implementation. In Section 3, we defined the interfaces that are necessary for communication between partitions. These interfaces make flexible interactive partitioning possible, and constitute the framework for automatic partitioning algorithm.
3. *Software and hardware synthesis.* We have demonstrated hardware synthesis for a particular hardware structure, namely synchronous hardware with latched output. We also outlined a method for software synthesis of CFSMs using simple if-then-else construct. A simple "fair" scheduler is also discussed.
4. *Optimization.* We have shown a software optimization technique for reducing the code size.
5. *Validation.* The FSM model derived from CFSM[CLJ+93] is compatible with input format of many formal verification algorithms. We have shown how a simple hardware-software system can be modeled and how a simple property may be verified with a particular verification algorithm.

In the future, we are planning to investigate the possibility of an automatic constraint driven partitioning algorithm for hardware-software systems. We also will look at the implementation of an interrupt based scheduler. As far as the optimization part is concerned, we will investigate the issue of reducing software execution time. On the validation front, we need to find a suitable level of abstraction for implemented systems that is simple enough for existing verification algorithms, yet detailed enough to reflect the true behavior of the implemented system. We also want to explore the possibility to adopt formal verification methods for a CFSM specification that do not require a cumbersome and expensive translation into equivalent FSMs. We will also concentrate our effort to assess the potential of the method on some "life-size" example from the automotive industry.

References

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-Checking for Real-Time Systems. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425, June 1990.

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of timed transition systems. *Proceedings of CONCUR*, 1992.

[Bak93] W. Baker. Application of the synchronous/reactive model to the VHDL language. Technical report, U.C. Berkeley, 1993.

[BAPM83] M. Ben-Ari, A. Pneuli, and Z. Manna. The temporal logic of branching time. *Acta Inf.*, (20):207–226, 1983.

[BCG91] G. Berry, P. Corunné, and G. Gonthier. The synchronous approach to reactive and real-time systems. *IEEE Proceedings*, 79, September 1991.

[BCH+91] 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, and H.Y. Wang. BLIF-MV:an interchange format for design verification and synthesis. Technical Report UCB/ERL M91/97, U.C. Berkeley, November 1991.

[BCMD90] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential circuit verification using symbolic model checking. *Proceedings of the Design Automation Conference*, pages 46–51, 1990.

[Bor88] G. Borriello. *A New Interface Specification Methodology and its Application to Transducer Synthesis*. PhD thesis, U.C. Berkeley, May 1988. (technical report UCB/CSD 88/430).

[BSV92] F. Balarin and A. Sangiovanni-Vincentelli. A verification strategy for timing-constrained systems. *Proceedings of the Fourth Workshop on Computer-Aided Verification*, pages 148–163, 1992.

[Bur92] J. R. Burch. *Automatic Symbolic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon University, August 1992.

[CBG+92] E. Clarke, J. Burch, O. Grumberg, D. Long, and K. McMillan. Automatic verification of sequentila circuit designs. *Phil. Transaction, Royal Society of London.*, A(339):105–120, 1992.

[CDHWT92] C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. *IEEE Real-Time Systems Symposium*, 1992.

[CKN86] D. Del Corso, H. Kirkman, and J. D. Nicoud. *Microcomputer buses and links*. Academic Press, London, 1986.

[CLJ+93] M. Chiodo, L. Lavagno, A. Jurecska, P. Giusto, H. Hsieh, and A. Sangiovanni-Vincentelli. A formal model and methodology for hardware/software co-design. Technical report, U.C. Berkeley, 1993.

[CLM91] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proceedings of the IEEE*, 79(9), September 1991.

[CSSVB92] M. Chiodo, T. Shiple, A. Sangiovanni-Vincentelli, and R. Brayton. Automatic reduction in CTL compositional model checking. *Proceedings of the Fourth Workshop on Computer-Aided Verification*, pages 225–238, 1992.

[CSV92] M. Chiodo and A. Sangiovanni-Vincentelli. Design methods for reactive real-time system co-design. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.

[DH89] D. Druzinski and D. Hare. Using statecharts for hardware description and synthesis. *IEEE Transacions on Computer-Aided Design*, 8(7), July 1989.

[DLG92] H. De-Leon and Orna Grumberg. Modular abstractions for verifying real-time distributed systems. *Proceedings of the Fourth Workshop on Computer-Aided Verification*, pages 3–17, 1992.

[DMNSV88] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State Assignment of Finite State Machines Targeting Multi-Level Logic Implementations. In *IEEE Transactions on Computer-Aided Design*, pages 1290–1300, December 1988.

[EC82] E. A. Emerson and E. M. Clarke. Using branching-time temporal logic to syntehsisze synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[EH92] R. Ernst and J. Henkel. Hardware-software codesign of embedded controllers based on hardware extraction. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.

[FG*et al.*86] A. Fuggetta, C. Ghezzi, and *et al.*. Formal data flow diagrams. *IEEE Transaction of Software Engineering*, 1986.

[GJM92] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli. System-level synthesis using re-programmable components. In *Proceedings of the European Design Automation Conference (EDAC)*, pages 2–7, March 1992.

[HK87] Z. Har'El and R. Kurshan. Cospan user's guide. *AT&T*, October 1987.

[HNSY92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Technical Report, Cornell University*, 1992.

[Kur90] R. P. Kurshan. Analysis of discrete event coordination. In *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[LN89] B. Lin and A. R. Newton. Synthesis of multiple level logic from symbolic high-level description languages. In *Proceedings of the International Conference on Very Large Scale Integration*, 1989.

[PM85] P.T.Ward and Mellor. *Structured development for Real-Time Systems*. Yourdon Press, 1985.

[RSV87] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, September 1987.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, U.C. Berkeley, May 1992.

[SSM+92] E.M. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, October 1992.

[Vil86] T. Villa. Constrained encoding in hypercubes: Applications to state assignment. In *U. C. Berkeley, ERL Memo 86/44*, May 1986.