A Formal Specification Model for Hardware/Software Codesign

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 this paper we present a model for specification, partitioning, and implementation of such systems. The model, called Codesign Finite State Machines (CFSMs), is based on FSMs and is particularly suited to a specific class of systems with relatively low algorithmic complexity. Pre-existing formal specification languages can be used by the designer to specify the intended behavior of the system and mapped into our model. CFSMs use a non-zero unbounded reaction delay model and hence can be indifferently implemented either in hardware or in software. The implementation only restricts the range of variation of some previously undefined delays, thus preserving formal properties of the specification across implementation refinements. The communication primitive, event broadcasting, is low-level enough to be implemented efficiently and yet general enough to allow higher-level mechanisms (such as channels) to be defined by the designer.

1 Introduction

Many different definitions of the term *"hardware-software codesign"* have been used in the literature. Some (e.g. [Tur78], [RR83], [CR78], [HJB+82], ...) consider it as the definition of the Instruction Set Architecture for a new computer, where the cost of implementing operations with hardware, microcode or subroutines, as well as the compiler technology advances required to handle highly pipelined or concurrent implementations must be taken into account. Others (e.g. [Aco92], ...) define it as the choice of the best processor/bus/memory architecture that suits a given software specification. We will use the term in a different sense (common, e.g., to [SB91], [GJM92c, GJM92b, GJM92a], [WWD92], ...), meaning the design of a special-purpose system composed of a few Application Specific Integrated Circuits cooperating with software procedures on general-purpose processors.

This restricted definition is still too wide to allow a useful formalization of generally applicable *automated* design methodologies. Embedded controllers are used in systems ranging from a portable CD player controller to the navigation control unit of a battle aircraft. Hence we will limit ourselves to relatively small real-time control systems that are composed of software on one (or few) micro-controllers and some semi-custom hardware components. We will exclude from our consideration large systems that require the coordination of many boards and tens of thousands of lines of code ([SB91], [Coc92], ...).

Available approaches to this problem can be classified in three broad groups:

1. Methods to implement software programs in hardware; for example:
   - various flavors of *CSP* ([Hoa78]), a formalism developed for correct concurrent program specification, have been translated into synchronous ([PL91], [WOB92], ...) or asynchronous ([vBKR+91], [BM87, Mar90b, Mar90a], ...) circuits.
   - languages for real-time software specification, such as *ESTEREL* or *StateCharts*, have been directly ([Ber91]) or indirectly ([NVG91]) used as hardware description languages.
2. Methods to implement hardware specifications in software (e.g., [WWD92], [GJM92c, GJM92b, GJM92a], [Str92], ...).
3. Methods to solve various particular aspects of hardware-software cooperation; for example:
   - design of interfaces between hardware and software components ([SB92], [COB92], ...).
   - formal specification of hardware-software system properties ([MKP92], ...).

None of the above mentioned approaches, though, has yet addressed, in our opinion, the problem of defining a *formal model* that is high-level enough to be used as an intermediate representation during the system design process, and yet low-level enough to be efficiently synthesized as a *mix* of hardware and software components.

In particular, methods that implement a pre-existing software language in hardware generally use syntax-directed translation methods, that result in unnecessarily cumbersome and inefficient circuits.

Moreover, the *perfect synchrony* hypothesis used by languages for real-time reactive systems like ESTEREL ([Ber91]) is not totally satisfactory for our purposes. This hypothesis requires that every component of the system has infinite computation capabilities with respect to the "typical" delays of the environment, thus ensuring that every response to an external or internal event is computed in *zero time*. The class of systems that we address in this work is strongly cost-limited, so as a general rule they require to implement as much as possible in software, leaving to hardware only the most performance critical components. This means that such systems can hardly be modeled as infinitely fast with sufficient accuracy.

Note that, unfortunately, the term "synchronous" has been used in the literature to mean at least three rather different concepts:

1. "Clocked", describing a system where all the components are synchronized by a global signal, as opposed to asynchronous systems where synchronization is a local property.
2. "Zero reaction time", describing a system where the components react instantaneously to an event ([BC84]).
3. "Without acknowledge", describing a communication protocol where the sender does not wait for the receiver to acknowledge the reception of the message ([CKN86]).

In the following, we will always use the first meaning, unless explicitly noted (for example the expression "synchronous programming language" refers to the second meaning).

Let us consider now approaches that adopted a pre-existing Hardware Description Language (HDL) as a software specification. Such methods, in a sense, did little more than performing a high-level simulation of some HDL modules in software, while the rest was mapped (and optimized) as hardware. None of these methods, to the best of our knowledge, was based on a formally defined specification language, so that formal properties of the system verified at the *specification* level could also be shown to automatically hold at the *implementation* level. This top-down refinement paradigm, advocated e.g. by Kurshan ([Kur90, Kur93]) is useful to reduce the complexity of the verification task, as checking properties of a high-level specification is often much easier than checking the same properties of an implementation. Moreover such high-level verification needs to be done only *once* for all the implementations examined during the exploration of the design space.

Even the use of the Behavioral Finite State Machine model ([WTHM92]) does not completely satisfy the formality requirement, because the composition of BFSMs is defined only by "simulation", and hence it seems hard to use in formal proofs.

Our proposal defines a *new* specification formalism, *Codesign Finite State Machines* (CFSMs), whose semantics is closely related to standard Finite State Machines. CFSMs can be used indifferently to specify hardware or software, within the limit described above to small control-dominated algorithms. The model has various desirable characteristics:

1. It is formally defined, hence it can be directly used to verify some properties that will be shared by *all implementations* (e.g. some timing independent properties, such as mutual exclusion from critical regions, absence of deadlocks, ...). The implementation in a particular style

(hardware or software) and the ensuing *optimization* steps only choose a *particular value* (or range of values) for some delay magnitudes within the model, thus ensuring correctness ([Kur90]).

2. It satisfies the requirements posed by various researchers who have worked in the area. For example:

   - Following [MKP92], it can describe the *format* of information (even though at a rather basic level), *precedence* relations between events and *functional* relations between input and output values. It can also be augmented, e.g. as in [JLHM91] or [MKP92], to describe the *constraints* that must be satisfied by the implementation and that drive the synthesis process.
   - Its underlying computational model satisfies the requirements posed by [BN92], because it allows only *finite state* and *finite interconnection*. It also reduces to a bare minimum the indeterminacy in execution times due to the real-time operating system.

The basic idea is to use a network of interacting FSMs, that communicate through a very low-level primitive: *events*. Events are *emitted* by a CFSM (or by the environment where the system operates) and can be *detected* by one or more CFSMs (thus implying a *broadcast* communication model, as opposed to point-to-point channels).

Events directly implement a *synchronous* protocol ([CKN86]) between communicating partners (i.e. a protocol without acknowledge). 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 main advantages.

1. It lends itself to a very efficient *hardware* implementation with synchronous circuits.

2. It 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 used by our proposed model implies that the sender does not "remove" the event immediately after emitting it, but only when it emits another 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 times that are 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.

Due to the reasons explained above, the notions of "time" and "event" that we use are different from the purely reactive perfectly synchronous model used, e.g., by ESTEREL. We use a *discrete* model of time, where each computing element takes a *non-zero unbounded* (at least before an implementation is chosen) time to perform its task. This model is quite realistic for synchronous systems and lends itself to efficient formal verification techniques (e.g. [Kur90], [Bur92]).

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, StateCharts, Formal Data Flow Diagrams or a subset of VHDL ([Bak93]), that will by directly translated into CFSMs. This translation task, of course, is made easier by the formal definition of the model.

1.1 Overview of the codesign methodology

As outlined above, we restrict the application of our methodology to a limited range of behaviors that we generically classify as control systems characterized by a relatively low algorithmic complexity. The most critical aspect here is the coordination between events in time. The use of an FSM-based description is therefore convenient for this class of systems, that only exchange very simple data, like pure events or small integers, and whose state space is also relatively small. What "simple" and "small" mean is obviously a practical rather than theoretical issue. The only basic limitation is to finite state systems[1].

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 CFSMs.

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 next task is *design partitioning*, i.e. the choice of software or hardware implementation for each components of the system specification. This task can be done by hand or can use automated techniques, for example based on I/O data rate requirements ([GJM92c]) or on simulation/profiling data ([EH92]).

Each CFSM can then be implemented in the chosen style. Hardware implementation is rather straightforward. 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. For a detail description about synthesis and partitioning of CFSM systems, see [CLJ+93].

The CFSM delay model plays a key role in making the software implementation possible as well. 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 then translated into a procedure that is activated at the arrival of one or more

[1] Every digital physical device, of course, has finite state. But this may not be a convenient abstraction for systems with extremely large state spaces, such as the memory of a workstation.

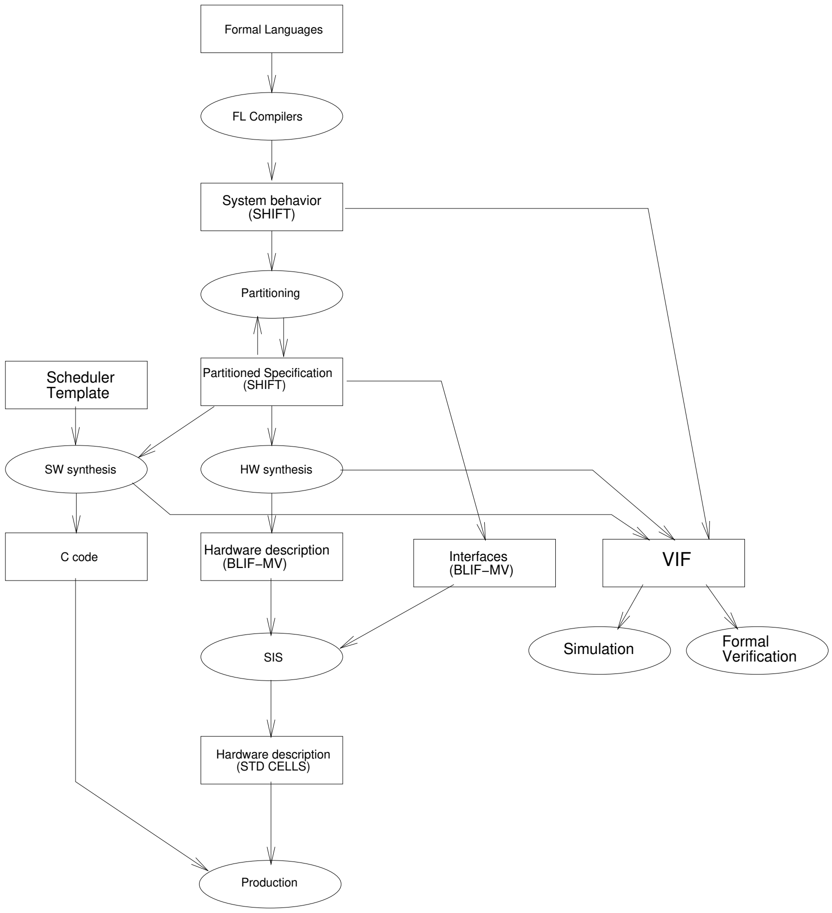

Figure 1: Codesign Framework

events the CFSM is waiting for. The procedure 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.

The rest of the paper is organized as follows. Section 2 describes both intuitively and formally the CFSM model and motivates the various design choices behind it. Section 3 describes SHIFT (Software/Hardware Intermediate FormaT), a textual format for describing CFSMs that can be used as an interchange medium between design aid tools. Section 4 shows how a network of CFSMs can be mapped into an "equivalent" network of FSMs, for synthesis and verification purposes. Section 5 draws some conclusions and outlines opportunities for future work.

2 Codesign Finite State Machines

2.1 Motivation

Finite State Machines (FSMs) are a very powerful mechanism to specify the behavior of control-oriented systems. They combine a very intuitive semantics, that is generally familiar to designers, with the availability of excellent synthesis and formal verification tools (e.g. [SSL+92, SSM+92], [Kur90, Kur93]). However, their "classical" definition has an implied synchronous hypothesis: all the FSMs used to model a system must change state and produce their outputs *simultaneously*.

A mixed hardware-software system, on the other hand, may contain components that proceed at very different speeds. Synchronous hardware modules compute their next states and outputs at each clock cycle. Software procedures, on the other hand, run sequentially on a micro-controller, and the reaction to a given condition may take hundreds of clock cycles to compute, and hundreds of clock cycles to propagate to other system components that may be waiting for it. Furthermore this delay depends on a very complex interaction of factors (such as the activity of other procedures, interrupts and so on) and hence can be almost impossible to model *deterministically*. Hardware is "always active" and computes a *function*, while an efficient implementation of real-time control software is generally "event-driven" and computes a *reaction*.

FSMs can be used to model such behavior (as we shall see later), but we believe that their use in this context is excessively cumbersome. Introducing a specialized model eases the definition of the *basic requirements* that we want to impose on the codesign system.

Existing formal models that embody the desired notion of non-deterministic timing, such as Petri nets ([Pet62], [Mur89]) or synchronous programming languages ([BC84]) are not very suitable for our purposes either.

Petri nets *per se* have *infinite state*, and there are no easy syntactic restrictions to limit the class of specifiable behaviors to *bounded* nets, which require only a finite amount of state[2]. Furthermore Petri nets directly model only *and* causality (meaning that a *conjunction* of preconditions is required for an action). Hence they are not very suitable to concisely express other causality paradigms such as, for example, *or* causality (meaning that any one of a given set of preconditions is sufficient to cause an action [Gun92]).

Synchronous programming languages require a zero reaction time that, as argued above, is not realistic enough for our purposes and that may be rather inefficient to implement in hardware, with very long combinational logic delays ([Ber91]). Our model, on the other hand, is designed for mapping onto a *clocked* architecture, hence each reaction takes a *non-zero* amount of time. This avoids both temporal paradoxes that affect synchronous programming languages[3] and the inefficiencies implicit in their direct implementation.

Hence we introduce Codesign Finite State Machines, that combine the best characteristics of

[2] Known algorithms to determine whether a given Petri net is bounded have a worst-case exponential complexity and can be very expensive to run in practice.

[3] These paradoxes in a sense are manifestations of the well-known meta-stability and oscillation physical phenomena.

FSMs, that is finite state and hence verifiability, with the best characteristics of causality-based models (such as Petri nets and synchronous programming languages), that is unbounded delays with reactive semantics. CFSMs also have the advantage that their behavior is defined using *partial orders*, and hence are amenable to verification techniques that reduce state space explosion by avoiding to construct all possible interleavings of concurrent actions ([NPW81], [BE91]).

2.2 Events and Traces

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 identified by its *name*, or in other terms by the "communication port" on which it occurs. The name is associated with a finite set of possible *values* for the event. More formally:

Definition 1 *An* event *is a triple* $e = (e_n, e_v, e_t)$*:*

- e_n *is the* name *or* type *of the event. The associated set of values will be denoted by* e_V *in the following.*
- $e_v \in e_V$ *is the* value *of the event.*
- e_t*, belonging to the set of non-negative* integers*, is the* time of occurrence *of the event.*

For example, the event with name "temperature" could occur every time a certain sensor reports a new value (e.g., in the range between 0 and 100 ^0C), or the event with name "keyboard" could occur every time the user hits a key (then the event value belongs, e.g., to the ASCII set).

Some events may not have an "interesting" value, if they denote the occurrence of a specific condition, such as a timer expiring or the user hitting the "reset" button. In this case e_V is the special symbol ϵ.

An *alphabet* describes a set of event names together with the associated values. A *timed trace* is an *ordered* finite or infinite sequence of events from some alphabet, 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).

Following [Dil88], we require sets of traces describing the behavior in time of a system to be *prefix-closed*, because every *partial* observation of a valid behavior up to a certain point in time must be considered a valid behavior as well. A timed trace T' is a prefix of a timed trace T if there exists a timed trace T'' such that $T = T'T''$. A set $\mathcal{T}$ of timed traces is prefix-closed if every finite prefix of each trace $T \in \mathcal{T}$ also belongs to $\mathcal{T}$.

More formally:

Definition 2

- *A* timed trace *is a finite or infinite sequence of events* $T = e^1, e^2, \ldots$ *such that* $i > j \Rightarrow e^i_t \geq e^j_t$ *and* $e^i_n = e^j_n \Rightarrow e^i_t \neq e^j_t$*.*
- *A* timed trace structure *is a pair* $(A, \mathcal{T})$*:*
  - $A = \{e'_n, e'_V), (e''_n, e''_V) \ldots\}$ *is the* alphabet *and*
  - $\mathcal{T}$ *is a prefix-closed set of timed traces with event names in* $\{e_n | (e_n, e_V) \in A\}$.

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:

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.*

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.

The possible sequences of events that describe the behavior of this system are represented by a timed trace structure, with time expressed in arbitrary units, e.g. seconds. Its alphabet is the set of input, output and internal events.

Consider now an arbitrary sequence of events: (`*START`, ϵ, 1), (`*KEY`, `ON`, 0), (`*END`, 5, 5), (`*END`, 10, 5)
It is not a valid timed trace because it violates the temporal ordering (between the first two events) and two events with the same name occur at time 5.

A valid timed trace of the system can also be expressed in tabular form, e.g.:

	0	1000
`*BELT`	`ON`	
`*KEY`	`ON`	`OFF`
`*ALARM`		
`*START`		
`*END`		

This trace describes a driver who fastens his seat belt while he turns on the key, drives for 1000 seconds and then turns off the key.

Another valid timed trace is:

	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.

2.3 Codesign Finite State Machines

Timed traces cannot be used in practice to *specify* the desired behavior of a system, because a timed trace structure in general is an *infinite* object. We must then define a *finite* object that *generates* a timed trace structure through its evolution in time. This object is a *network* of *Codesign Finite State Machines* (CFSMs).

A Codesign Finite State Machine is like a "classical" FSM, because both transform a set of inputs into a set of outputs by using only a finite amount of internal state. In this way the behavior of a finite state discrete system can be described as an interconnection of such interacting objects. The difference from the "classical" FSM (that will be called just FSM in the following) is that our model 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. While this difference in timing between the system components *can* indeed be modeled using FSMs (as we will show in Section 4), this model can be rather cumbersome.

Hence we choose a different approach. We first describe a formalism that satisfies our requirement of modeling objects with varying non-deterministic reaction times. Then we show how it can be represented with FSMs for formal verification purposes, since efficient FSM-based verification methods are relatively well-developed.

Another difference between CFSMs and FSMs is that inputs and outputs of an FSM are assumed to have a *value* at any point in time, which changes at discrete points in time corresponding to state changes. On the other hand, in our model we prefer to reason about *events* occurring at a point in time, rather than values, because an event-based model is well suited for reactive control systems ([BC84]).

A CFSM is basically constituted by a set of input events (each with its associated set of values), a set of output events (each with its associated set of values and possibly with an initial value), and a transition relation.

The transition relation describes how input events can cause output events. It is a *set* of *pairs of sets*. The first member of each pair is a set of *input event names and values*. The second one is a set of *output event names and values*. Each transition is *triggered* by the input events with the appropriate values and *emits* the output events with the appropriate values. The *reaction time* (i.e. the time between each input event and each output event) is *unbounded* and *non-zero*.

The CFSM transition mechanism is *non-deterministic*, due to two different reasons:

1. The reaction time is not known, hence there is a non-deterministic *delay* associated with each transition.
2. The *output events* caused by a set of input events *may* not be uniquely defined.

The latter kind of non-determinism offers a powerful mechanism for *abstraction* ([Kur90]), because we do not have to model the exact behavior of the system and its environment at all the stages of design development and verification:

- Some properties of the environment (e.g. the time when the driver fastens the seat belt) may never be known deterministically. Yet, a non-deterministic model that restricts somehow the allowed environment behavior can be necessary to verify the correctness of the design. For example, by describing the model of a driver who eventually fastens the seat belt, we can check that a system implementing the specification of Example 1 will eventually deactivate the alarm.
- On the other hand, incompletely designed system components can be described as non-deterministic CFSMs, leaving out the details for a later phase of top-down design. E.g., a redundancy code checker can initially be described as non-deterministically emitting an error status, even before the actual algorithm is defined.
- Non-determinism can also be used to simplify the formal verification task ([Kur90]), because some properties of the system may be independent, for example, of the redundancy code computation method, and their verification may be simpler if we use the more abstract model for that particular component.

The CFSM has two basic kinds of input events: *trigger* events and *pure value* events. As an example of this distinction, consider a system component that must sample temperature every minute, and react appropriately. It can be modeled as a CFSM with two input events, time and temperature. The reaction (CFSM transition) can occur only due to a time change, but it must take into account the value of the temperature event when the time change event occurs. Modeling both as events allows, for example, some other "monitor" component of the same system to react to *temperature* changes rather than time changes.

So we can say that:

1. *Trigger* events can be used *only once* to cause a transition of a given CFSM. They implement the basic *synchronization* mechanism of CFSMs. Each occurrence is *consumed* by the triggered transition, but it can, as we will see below, cause many transitions in *different* CFSMs.
2. *Pure value* events cannot directly cause a transition, but can be used to choose among different possibilities involving the same set of trigger events (and their values).

The control and data lines of a bus are another example of trigger and pure value events. Control lines must change at every bus cycle, and no new action can be taken unless they change. Data lines, on the other hand, may remain constant across cycles, for example during the fetch of a sequence of NO-OPs, and they will be used many times by the instruction decoder.

The *state* of the CFSM consists of *a set* of event types that are at the same time *input* and *output* for it. The non-zero reaction time of this feedback loop provides the "storage" capability that is required to implement the concept of state.

We want an "event-driven" model, so we impose that each transition relation element contains *at least one trigger event* and that a state event be a *pure data* event for its CFSM (but it can trigger transitions in other CFSMs).

Our definition allows the use of *more than one* state event type to allow the designer, for example, to decompose the CFSM using a *subroutine* structure. In this case, one state event type is used for the "main" procedure, that "calls" subroutines and waits for their return. Each subroutine has an event type of its own as its state. It is likely, though, that most often in practice a *single* state event type for each CFSM will be used, due to the analogy with the FSM world, in which the structure of the state space is "flat", with a single list of values.

The initial value is defined only for output events, because they are the only ones the CFSM can control. Moreover, since the value of an event becomes "defined" only when the event is specifically *emitted*, the initial value is useful only for events that are used as *pure values* by some CFSM (such as, for example, the state events for the CFSM itself). Note that an event in the initial set is defined to be *present* at time zero, so it can directly trigger transitions in other CFSMs.

We require that *at least* the events that occur both as inputs and outputs of $\mathcal{C}$ (also called "*state*" events in the following) have an *initial value*. This ensures a (possibly non-deterministic) system initialization condition.

We can now formally define:

Definition 3 *A Codesign Finite State Machine (CFSM) is a quintuple* $\mathcal{C} = (I, E, O, R, F)$*:*

- $I = \{(i'_n, i'_V), (i''_n, i''_V), \ldots\}$ *is a finite set of input event names and of the corresponding finite sets of allowed values.*
- $E \subseteq I$ *is the set of* "trigger" *input event names.*

  Events with names in $I - E$*, on the other hand, are* "pure data" *events.*
- $O = \{(o'_n, o'_V), (o''_n, o''_V), \ldots\}$ *is a finite set of output event names and of the corresponding finite sets of allowed values, such that* $E \cap O = \emptyset$*.*
- $R \subseteq \{(e_n, e_v) | (e_n, e_V) \in O, e_v \in e_V\}$ *is a set of possible initial values of (some) output events.*
- $F \subseteq \{(f^I, f^O) |\ f^I = \{(e'_n, e'_v) | (e'_n, e'_V) \in I, e'_v \in e'_V\},\ f^O = \{(e''_n, e''_v) |\ (e''_n, e''_V) \in O, e''_v \in e''_V\}\}$ *is the* transition relation[4]*.*

  For all $(f^I, f^O) \in F$ *there must exist at least one* $(i_n, i_V) \in E$*,* $i_v \in i_V$ *such that* $(i_n, i_v) \in f^I$*.*

[4]The transition relation can equivalently be defined over the Cartesian product of the sets of input and output

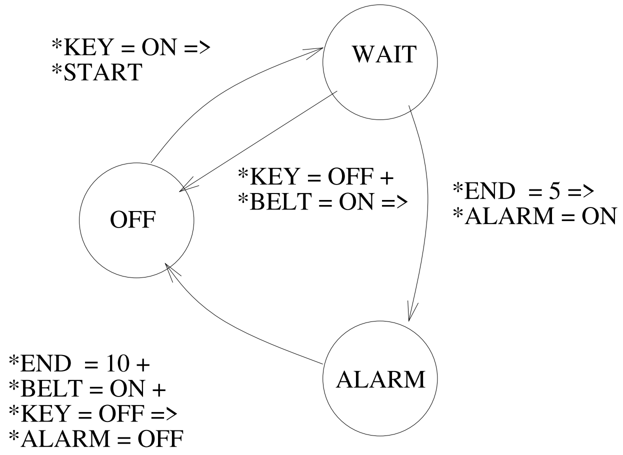

Figure 2: CFSM Specification of a simple system

The operational cycle of a CFSM goes through the following four phases (that will be described more formally below, after the definition of a network of CFSMs):

1. idle,
2. detect input events,
3. transition, according to which events are present and match a transition relation element,
4. emit output events.

Phases 1, 2 and 4 can have a duration between zero and infinity, while phase 3 takes *at least* one time unit. This is a basic difference between our model and synchronous programming languages, in which phase 1 can have a duration between zero and infinity while phases 2, 3 and 4 always have a duration of zero ([BC84]).

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 timer behavior will be described later.

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

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

values. This would lead to a more cumbersome notation when only a *subset* of the input events of a CFSM is required to trigger a transition (a very common case). As another example of this "shorthand" notation, consider the standard representation of a Boolean function using implicants rather than minterms.

- $O_1 = \{(\texttt{*START}, \{\epsilon\}), (\texttt{*ALARM}, \{\texttt{ON}, \texttt{OFF}\}), (s_1, \{\texttt{OFF}, \texttt{WAIT}, \texttt{ALARM}\})\}$.

  Note that the state event s_1 appears both as an input and as an output event. Its name is not preceded by a "*" because the state event is a pure data value for this CFSM (it does not appear in E_1).

- $R_1 = \{(s_1, \texttt{OFF})\}$.

  The only "useful" initially present event is the initial state of the CFSM. Note that, given our event-driven model, no other CFSM is allowed to use any of the other output events before they are emitted, so their initial values *do not matter* in this case.

- In the description of the transition relation we will use the symbol $\Rightarrow$ to separate the two sets of input events and of output events, for the sake of clarity.

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

Each CFSM describes a *component* of the system to be modeled. The whole system is described as a *network* of interacting CFSMs. Hierarchy can be used (as will be shown in Section 3) to bridle complexity, but the formal definition is given here, without loss of generality, in terms of a *flat* view.

Definition 4 *A* network *of CFSMs is a set of CFSMs*
$\mathcal{N} = \{\mathcal{C}_1 = (I_1, E_1, O_1, R_1, F_1), \mathcal{C}_2 = (I_2, E_2, O_2, R_2, F_2), \ldots\}$
such that no two different *CFSMs have an output event name in common (i.e. $i \neq j$ implies that $O_i \cap O_j = \emptyset$).*

Output sets must be disjoint, because we want to avoid the difficulties inherent in the implementation of the update of a single object by two concurrent agents (that would require some mutual exclusion mechanism or some resolution function).

Input sets need not be disjoint, thus implying a *broadcasting* communication mechanism (as opposed to point-to-point like in CSP).

In the seat belt example, the network of CFSMs would be composed by $\mathcal{C}_1$ as defined above plus a CFSM implementing the timer $\mathcal{C}_2 = (I_2, E_2, O_2, R_2, F_2)$ as follows:

Example 3

- $I_2 = \{(\texttt{*START}, \{\epsilon\}), (\texttt{*TICK}, \{\epsilon\}), (s_2, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\})\}$.

  Here `*TICK` represents an input event from the environment occurring once a second.

- $E_2 = \{(\texttt{*START}, \{\epsilon\}), (\texttt{*TICK}, \{\epsilon\})\}$.
- $O_2 = \{(\texttt{*END}, \{5, 10\}), (s_2, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\})\}$.
- $R_2 = \{(s_2, 0)\}$.
- $F_2 = \{(\{(\texttt{*START}, \epsilon), (\texttt{*TICK}, \epsilon), (s_2, 0)\} \Rightarrow \{(s_2, 1)\}),$
  $(\{(\texttt{*TICK}, \epsilon), (s_2, 1)\} \Rightarrow \{(s_2, 2)\}),$
  $(\{(\texttt{*START}, \epsilon), (\texttt{*TICK}, \epsilon), (s_2, 1)\} \Rightarrow \{(s_2, 0)\}),$
  $(\{(\texttt{*TICK}, \epsilon), (s_2, 2)\} \Rightarrow \{(s_2, 3)\}),$
  $(\{(\texttt{*START}, \epsilon), (\texttt{*TICK}, \epsilon), (s_2, 2)\} \Rightarrow \{(s_2, 0)\}),$
  ...
  $(\{(\texttt{*TICK}, \epsilon), (s_2, 4)\} \Rightarrow \{(s_2, 5), (\texttt{*END}, 5)\}),$
  $(\{(\texttt{*START}, \epsilon), (\texttt{*TICK}, \epsilon), (s_2, 4), \} \Rightarrow \{(s_2, 0), (\texttt{*END}, 5)\}),$
  ...
  $(\{(\texttt{*TICK}, \epsilon), (s_2, 9)\} \Rightarrow \{(s_2, 0), (\texttt{*END}, 10)\})\}$.

The formal definition of the behavior of a CFSM network as a timed trace structure uses the notion of *causality* between sets of events.

Intuitively, a set of events of a timed trace *may cause* another set of events if there exists a CFSM with a transition relation element whose *input* component matches the set of *causes* and whose *output* component matches the set of *reactions*.

Obviously, causes must strictly precede reactions in time. Moreover, no two elements of the set of causes can have the same event name, because this would imply some "state" within the system beside pure events. For example, if we allowed the *sequence* of events $(\texttt{*KEY}, \texttt{ON}, 2)$ and $(\texttt{*KEY}, \texttt{OFF}, 3)$ to cause event $(\texttt{*ALARM}, \texttt{ON}, 5)$ without using some state event to record the occurrence of the former event, this would imply that the CFSM has some other way than its state events to "remember" that `*KEY` had been `ON` at time 2 when it sees $(\texttt{*KEY}, \texttt{OFF}, 3)$. This is not coherent with our FSM-like model. Event $(\texttt{*KEY}, \texttt{ON}, 2)$ must cause a state change, and the new state value, together with event $(\texttt{*KEY}, \texttt{OFF}, 3)$, can now cause event $(\texttt{*ALARM}, \texttt{ON}, 5)$. For the same reasons, no two reactions can have the same name.

Note that elements of the transition relation with some event name occurring more than once either as input event or as output event can never be triggered. They are totally redundant, because a causality relationship involving two events with the same name is forbidden.

Definition 5 *Let* $A = \bigcup_{\mathcal{C}_i \in \mathcal{N}} (I_i \cup O_i)$ *be the set of event names and values of a CFSM network* $\mathcal{N}$. *Let* $(A, \mathcal{T})$ *denote an arbitrary timed trace structure with alphabet* A. *Let* c *and* r *be finite sets of integers indexing some elements of a timed trace* $T \in \mathcal{T}$.

The set $\{e^k = (e^k_n, e^k_v, e^k_t) | k \in c\}$ *of events indexed by* c may cause *the set* $\{e^i = (e^i_n, e^i_v, e^i_t) | i \in r\}$ *of events indexed by* r *with respect to a CFSM* $\mathcal{C} = (I, E, O, R, F)$ *of* $\mathcal{N}$ *if the following rules are satisfied:*

1. *For all* $j \in c$, $(e^j_n, e^j_V) \in I$.
   For all $i \in c$, $j \in c$, $i \neq j$ *implies* $e^i_n \neq e^j_n$.

2. *For all* $k \in r$, $(e^k_n, e^k_V) \in O$.
   For all $k \in r$, $l \in r$, $k \neq l$ *implies* $e^k_n \neq e^l_n$.

3. *For all* $k \in c$, $i \in r$, $e^k_t < e^i_t$.

4. *There exists* $(f^I, f^O) \in F$ *such that:*
   - $(e^k_n, e^k_v) \in f^I$, $(e^k_n, e^k_V) \in I \Rightarrow k \in c$,
   - $(e^j_n, e^j_v) \in f^O$, $(e^j_n, e^j_V) \in O \Leftrightarrow j \in r$,

Note that, due to Rule 4, *all* output events of a given transition must be emitted if at least *one* of them is emitted. This is necessary in order to allow the implementation of higher-level *synchronization constructs* between CFSMs. This is not possible, for example, if receiving an acknowledge from the next FIFO stage does not ensure that the current datum has been successfully passed to its successor and that the stage is ready for another datum.

We can now look at some examples of potential causality relations between sets of events. Given the timed trace (now including also state events):

	0	1	2	3	4	5	6	7	8	9	10	11
*BELT										ON		
*KEY	ON											
*ALARM								ON				OFF
*START		ϵ						ϵ				
*END							5					
*TICK	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
s_1	OFF	WAIT						ALARM			OFF	
s_2	0	0	1	2	3	4	5	6	0	1	2	3

The set of events $\{(\texttt{*START}, \epsilon, 1), (\texttt{*TICK}, \epsilon, 3), (s_2, 4, 5)\}$ *may* cause the set of events $\{(\texttt{*END}, 5, 6), (s_2, 0, 8)\}$ according to the above definition. But it should be obvious that it actually *does not*, because CFSM $\mathcal{C}_2$ makes a few other transitions in between.

On the other hand, $\{(\texttt{*START}, \epsilon, 1), (\texttt{*TICK}, \epsilon, 1), (s_2, 0, 1)\}$ seems intuitively a "more reasonable" set of causes for $\{(s_2, 1, 2)\}$.

We see that causality alone is not sufficient to define the set of legal traces produced by a CFSM network. According to Definition 5, an event with name e_n at time 0 could cause a reaction with name e'_n at time 3 even if *in the same trace* we also had an event with name e_n at time 1 causing a reaction with name e'_n at time 2. This is forbidden in our model, because we cannot have any state in the system to hold this past history, beside the last occurrence of each event type. In that example, the CFSM would need some additional information to remember the event at time 0 in order to produce its effect after reacting to the event at time 1. The only valid method to provide such memory within our model is to use *state* events as explained above. Similarly, all output events of a given transition must be emitted before the next transition can occur.

Hence we need another set of conditions, ensuring that the transitions of each CFSM not only satisfy the definition of causality given above, but are also "atomic".

This means that for each timed trace for each CFSM we can find an *ordered sequence* of pairs $(c_0, r_0), (c_1, r_1), (c_2, r_2), \ldots$ that satisfies the following conditions:

- Each (c_i, r_i) identifies events of T that are the *cause* and the *result* respectively the i-th transition of $\mathcal{C}$.
- Every output event has a cause.
- The sets of causes of transitions intersect only on "pure data" events. As we informally explained above, no "trigger" event can cause *two or more* distinct output events of the same CFSM in the same trace.

We also need the notion of *maximality* of each set causing a 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. For example, we may want to emit the alarm when the timer expires only if the driver does not fasten the belt at the same time.

Recall that trigger events can cause only one transition in each CFSM. So 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 *trigger* event and it has been used for a previous transition of the same CFSM.

Every set of causes must then be maximal, in the sense that for each chosen sequence of causality relations there must exist no other valid potential sequence for the same timed trace and CFSM that coincides up to a certain point, and then includes *more* events that have occurred *before* those that are already being considered. For example, if an event with name e'_n occurs at time 2, i.e. *later* than an event e_n occurring at time 0, then a CFSM cannot react to the event with name e'_n and ignore that with name e_n (unless, of course, the latter has been overwritten by another event with the same name).

Here by "overwriting" we informally mean the fact that some event at time t does not cause any transition in a CFSM while another event with the same name at a time later than t does cause a transition. In this case, the CFSM has *ignored* an occurrence of the event, due to our communication model where the sender does not wait for the receiver to acknowledge detection of an event. It is the designer's responsibility to make sure (by explicit handshaking between the partners, by formal verification, or by simulation) that this overwriting does not cause an incorrect system behavior.

The maximality condition is necessary to take into account the fact that an event may be detected by a CFSM and cause its transition an *unbounded* amount of time after they are emitted. So we cannot force all CFSMs to notice an event at the same time (e.g. if they are implemented in software and run on the same processor). But we can at least force each CFSM to process its input events in an ordered fashion.

Finally, we impose the obvious constraint that the set of valid timed traces must also begin with the correct values for all the output events (especially state events) whose value is specified by the designer.

The formal definition of the allowed behaviors of a CFSM network in terms of timed traces is as follows:

Definition 6 *The timed trace structure associated with a CFSM network $\mathcal{N}$ is $(A, \mathcal{T})$:*

- $A = \bigcup_{\mathcal{C}_i \in \mathcal{N}} (I_i \cup O_i)$.
- *For every CFSM* $\mathcal{C} = (I, E, O, R, F) \in \mathcal{N}$*, for every trace* $T \in \mathcal{T}$*, if* $(e_n, e_v) \in R$ *then there exists* $(e_n, e'_v) \in R$ *such that* $(e_n, e'_v, 0)$ *belongs to* T[5].
- *For each timed trace* $T \in \mathcal{T}$*, for each CFSM* $\mathcal{C} = (I, E, O, R, F) \in \mathcal{N}$ *we can find an* ordered sequence *of pairs* $\mathcal{S}_{T,\mathcal{C}} = (c_0, r_0), (c_1, r_1), (c_2, r_2), \ldots$ *of sets of integers indexing events of* T *that satisfy the following rules:*
  1. *For every* (c_i, r_i) *belonging to* $\mathcal{S}_{T,\mathcal{C}}$*,* c_i *may cause* r_i*, according to Definition 5.*
  2. *For every* (e_n^j, e_v^j, e_t^j) *such that* $(e_n^j, e_V^j) \in O$*, there exists* (c_i, r_i) *belonging to* $\mathcal{S}_{T,\mathcal{C}}$ *such that* $j \in r_i$.
  3. *For all* $i \neq j$*,* $k \in c_i \cap c_j$*, if* $(e_n^k, e_V^k) \in I$*, then* $(e_n^k, e_V^k) \notin E$.
  4. *For all* $(c_i, r_i), (c_j, r_j)$ *belonging to* $\mathcal{S}_{T,\mathcal{C}}$ *such that* $i < j$*, for all* $k \in c_i$*,* $l \in c_j$ *if* $e_n^k = e_n^l$ *then* $e_t^k < e_t^l$.
  5. *For all* $(c_i, r_i), (c_j, r_j)$ *belonging to* $\mathcal{S}_{T,\mathcal{C}}$ *such that* $i < j$*, for all* $k \in r_i$*,* $l \in r_j$*, then* $e_t^k < e_t^l$.
  6. *Every* c_i *belonging to* $\mathcal{S}_{T,\mathcal{C}}$ *is* maximal*, in the sense that for all* i *there exists no other valid* $\mathcal{S}'_{T,\mathcal{C}}$ *for the same timed trace* T *and CFSM* $\mathcal{C}$ *such that:*
     - $(c'_j, r'_j) = (c_j, r_j)$ *for all* (c'_j, r'_j) *belonging to* $\mathcal{S}'_{T,\mathcal{C}}$*,* (c_j, r_j) *belonging to* $\mathcal{S}_{T,\mathcal{C}}$ *such that* $j < i$*,*
     - $c_i \subseteq c'_i$*,*
     - *for all* $(e_n, e_v, e_t) \in (c'_i - c_i)$ *there exists* $(e'_n, e'_v, e'_t) \in c_i$ *such that* $e_t < e'_t$.

In the definition above we do not restrict the environment (i.e. the producer of events that are not output by any CFSM) to be modeled as a CFSM as well. Modeling the environment as a CFSM, though, is strongly recommended whenever possible, in order to be able to apply automated formal verification methods to the CFSM network.

The definition does not enforce *fairness* either, 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. A CFSM may even produce no reaction at any time. We only force *all* the *output* events for a transition to be emitted if one of them is emitted (due to Definition 5, Rule 4). Of course fairness will have to be imposed as an additional requirement to most meaningful implementations, either through the use of a fair software scheduler or through a formal verification that the CFSMs interact in a way that ensures no "starvation".

Note that the first two elements of F_1 in Example 2 specify a slightly different behavior from Figure 2, because they explicitly state that if *both* `*KEY` and `*BELT` are present in state `OFF`, then no reaction is needed. This is due to the fact that the *largest* possible match for a set of input

[5] I.e. if *some* initial value for an output event is specified, then that event must be initially present, with *one* of the specified values.

events is always required by Rule 6 of Definition 6. Figure 2, on the other hand, described a *non-deterministic* behavior where if both events were present a beep could be generated anyway (not quite the intended behavior assumed by the informal specification).

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.

This transition relation is still non-deterministic because if both `*BELT = ON` and `*END = 5` are present in state `WAIT`, then the CFSM can produce a beep or not depending on the resolution of non-determinism that is implicit in any software or hardware implementation. This presumably incorrect behavior should be detected during the formal verification stage, and is an example of a property of the system that is independent of delays and hence can be verified at the CFSM level.

For example, in the timed trace T described above, a valid $\mathcal{S}_{T,\mathcal{C}_1}$ is (denoting each $(c_i \Rightarrow r_i)$ pair by the corresponding events):

$$(\{(\texttt{*KEY}, \texttt{ON}, 0), (s_1, \texttt{OFF}, 0)\} \Rightarrow \{(\texttt{*START}, \epsilon, 1), (s_1, \texttt{WAIT}, 1)\})$$

$$(\{(\texttt{*END}, 5, 6), (s_1, \texttt{WAIT}, , 1)\} \Rightarrow \{(\texttt{*ALARM}, \texttt{ON}, 7), (\texttt{*START}, \epsilon, 7), (s_1, \texttt{ALARM}, 7)\})$$

$$(\{(\texttt{*BELT}, \texttt{ON}, 9), (s_1, \texttt{ALARM}, 1)\} \Rightarrow \{(s_1, \texttt{OFF}, 10), (\texttt{*ALARM}, \texttt{OFF}, 11)\})$$

Similarly, a valid $\mathcal{S}_{T,\mathcal{C}_2}$ is

$$(\{(\texttt{*TICK}, \epsilon, 0), (s_2, 0, 0)\} \Rightarrow \{(s_2, 0, 1)\})$$

$$(\{(\texttt{*TICK}, \epsilon, 1), (\texttt{*START}, \epsilon, 1), (s_2, 0, 0)\} \Rightarrow \{(s_2, 1, 2)\})$$

$$(\{(\texttt{*TICK}, \epsilon, 2), (s_2, 1, 2)\} \Rightarrow \{(s_2, 2, 3)\})$$

$$(\{(\texttt{*TICK}, \epsilon, 3), (s_2, 2, 3)\} \Rightarrow \{(s_2, 3, 4)\})$$

$$(\{(\texttt{*TICK}, \epsilon, 4), (s_2, 3, 4)\} \Rightarrow \{(s_2, 4, 5)\})$$

$$(\{(\texttt{*TICK}, \epsilon, 5), (s_2, 4, 5)\} \Rightarrow \{(s_2, 5, 6), (\texttt{*END}, 5, 6)\})$$

$$(\{(\texttt{*TICK}, \epsilon, 6), (s_2, 5, 6)\} \Rightarrow \{(s_2, 6, 7)\})$$

$$(\{(\texttt{*TICK}, \epsilon, 7), (\texttt{*START}, \epsilon, 7), (s_2, 6, 7)\} \Rightarrow \{(s_2, 0, 8)\})$$

$$(\{(\texttt{*TICK}, \epsilon, 8), (s_2, 0, 8)\} \Rightarrow \{(s_2, 1, 9)\})$$

$$(\{(\texttt{*TICK}, \epsilon, 9), (s_2, 1, 9)\} \Rightarrow \{(s_2, 2, 10)\})$$

$$(\{(\texttt{*TICK}, \epsilon, 10), (s_2, 2, 10)\} \Rightarrow \{(s_2, 3, 11)\})$$

This example, for the sake of simplicity, does not show the fact that in our model events at different times may concur to cause a specific transition, i.e. the fact that events in some c_i (or r_i) may have different "time stamps".

The "unbounded delay" feature of our model, on the other hand, appears at time 9, when $\mathcal{C}_1$ does not react immediately to the event `*BELT = ON`, but waits for two seconds. This shows that some time-dependent properties of the model, such as the fact that the alarm certainly stops after a certain amount of time *cannot* be proved at this level. For example, we must make some assumptions on the maximum time within which a certain routine implementing $\mathcal{C}_1$ will be called.

3 Software-Hardware Intermediate Format

We can describe the CFSM network defined in the previous section using a specific representation format, called *Software-Hardware Intermediate FormaT* (SHIFT), that we use to exchange CFSM descriptions in the form of files across design aid tools. This format is mainly an extension of BLIF-MV ([BCH+91]), which in turn is a multi-valued extension of the *Berkeley Logic Interchange Format* (BLIF, [SSL+92]). In SHIFT a CFSM description consists of a list of input variables, a list of output variables, and the transition relation.

A *partition* of the system, constituted by one or more CFSMs that must be completely implemented in hardware or software, is specified with the `.model` construct, which introduces its name and may specify the type of intended implementation (hardware or software) as an attribute:

```
.model <model_name> [model_attributes]
```

The input and output events of the partition are specified using the `.inputs` and `.outputs` constructs as follows:

```
.inputs <in_event_1> <in_event_2> ...
.outputs <out_event_1> <out_event_2> ...
```

As a convention, we interpret names beginning with a "*" as *event names*, while other names represent *event values*.

The set of allowed values for an event (whether input/output or internal to the partition) is given with the `.mv` construct:

```
.mv <event> <num_values> [value_0 value_1 ...]
```

Events with uninteresting values ($e_V = \{\epsilon\}$) need not be specified with `.mv` statements.

The (possible) initial value of an output event is given with the `.r` construct:

```
.r <output_event> <value>
```

Note that an event whose name appears in a `.r` statement is *present* at time 0. So it can cause an immediate reaction in some CFSM.

Each CFSM $\mathcal{C} = (I, E, O, R, F)$ is described by the `.names` construct. The symbol '`=>`' is used to separate the input event name list from the output event name list:

```
.names in_event_1 in_event_2 ... => out_event_1 out_event_2 ...
       in1_val1   in2_val1   ...    out1_val1   out2_val1   ...
       in1_val2   in2_val2   ...    out1_val2   out2_val2   ...
....
```

Input events with names prefixed by "*" correspond to *trigger* events, and the corresponding columns in the table contain the value 0 or 1 to denote the absence or the presence of the event respectively. If we need to test the *value* of an input event (whether trigger or pure data), we must add a column with the event name *without* the "*".

Each row of the table corresponds to one or more elements of the transition relation in the obvious way. The symbol "-" is used to denote a don't care value. If it appears in a column denoting an event *name*, then the event does not appear in the corresponding transition relation element. If it appears in a column denoting an event *value*, then the row corresponds to a *set* of transition relation elements, each with a possible value of the event. To reduce the size of the SHIFT files, we also introduce the "assignment" syntax to indicate that an output will take the value of the specified input. For example:

```
.mv y 3
.mv x 3
...
.names x *e  =>  y
- 1   (x)
```

is a shorthand for:

```
.names x *e  =>  y
0 1  0
1 1  1
2 1  2
```

by expanding both the don't care value construct and the assignment construct.

Models can be nested to form a hierarchy with the construct `.subckt` that has the syntax:

```
.subckt <model_name> <instance_name> [formal_name1=actual_name1
  formal_name2=actual_name2 ...]
```

This construct causes the instantiation of the model in place of the `.subckt` statement, with input and output events appropriately renamed.

Figure 3 shows how the simple set belt example described in Figure 2 and Example 2 can be specified using SHIFT.

Note how `s1` appears as both an input and an output in the `belt` CFSM. Also, note that every row of the transition relation table has at least one input event at "1". This is mandatory because every state transition must be triggered by an event. The priority between simultaneous events, as described in Section 2, is explicitly defined in the first two lines of the belt model's transition relation: if `*KEY` and `*BELT` are both present when the CFSM makes its transition, then `*KEY` is ignored. Non-determinism is hidden in lines 3 and 5 of the same transition relation. From state 1,

```
# 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
[... omitted ...]
.end
```

Figure 3: SHIFT Specification of the seat belt example

if both `*KEY = OFF` and `*END = 5` occur, a transition can be made from state 1 to state 2 emitting `*ALARM = ON`, as well as a transition to state 0 without any event emission (in fact, this is a wrong specification since in this case the alarm should not be activated).

The example also shows how a system is organized into a three-level hierarchy. At the top level, we see only `.subckt`s representing partitions as black boxes. Each partition is a `.model`. Inside each partition we can have a set of CFSMs and hierarchically defined sub-systems.

4 FSM semantics of CFSMs

Our purpose is to use CFSMs, and the associated SHIFT language, as a formal specification model for control-oriented mixed hardware and software systems. So we want to use formal verification methods to ensure the correctness of a CFSM network with respect to the properties the designer had in mind (*design verification*). We also want to implement each CFSM as a software or hardware component and show the correctness of this implementation step (*implementation verification*). For these purposes, in this section we will show how to represent the "asynchronous" behaviour of the network of CFSMs by using a "synchronous" model, i.e. a network of FSMs whose behaviour is equivalent with respect to the observable events.

We first recall some classical definitions about Finite State Machines from the literature.

Definition 7 *A* Non-deterministic Finite State Machine *(FSM) is a quintuple* $\mathcal{F} = (I, O, X, R, F)$*:*

- I *is a finite set of input symbols.*
- O *is a finite set of output symbols.*
- X *is a finite set of states.*
- $R \subseteq X$ *is the set of initial states.*
- $F \subseteq I \times X \times X \times O$ *is the transition relation.*

$\mathcal{F}$ is *completely specified* if for all $i \in I$, $x \in X$ there exists at least one $x' \in X$, $o \in O$ such that $(i, x, x', o) \in F$, i.e. if the FSM has *at least* one choice of next state/output for each input/present state combination.

$\mathcal{F}$ is *deterministic* if R is a singleton and F is a *function* $F : (I \times X) \mapsto (X \times O)$, i.e. if the FSM has *at most* one choice of next state/output for each input/present state combination.

In general an FSM that is implemented in hardware or software is *deterministic* and *completely specified*. For verification purposes we are interested in *non-deterministic, completely specified* FSMs.

Within this section we will assume that all the FSMs that we consider are synchronous, i.e., they make exactly one transition for each time unit (clock cycle).

Furthermore, inputs of each FSM will actually be n-tuples of pairs, each identifying an input *multi-valued signal* and its set of allowed values:
$I = \{(i_n^1, i_v^1) | i_v^1 \in i_V^1\} \times \{(i_n^2, i_v^2) | i_v^2 \in i_V^2\} \times \ldots$.
Each (i_n^j, i_v^j) is called a *component* of the set of input symbols.

Similarly: $O = \{(o_n^1, o_v^1) | o_v^1 \in o_V^1\} \times \{(o_n^2, o_v^2) | o_v^2 \in o_V^2\} \times \ldots$.
And:
$X = \{(x_n, x_v) | x_v \in x_V\}$.

The above assumptions allow us to define the set of timed traces of an FSM network, as follows. As usual, an FSM network is a set of FSMs such that no two of them have a component of the output symbols (i.e. an output signal) in common.

Moreover, every timed trace defines exactly one value for each signal and for the state of each FSM at each point in time. Note that this is a basic difference between *CFSM* traces and *FSM* traces: CFSMs are an *event-based* model, while FSMs are a *value-based* model (the only "event" that matters is the clock).

Each FSM starts in one of its initial states and monitors its inputs and changes state and output according to its transition relation. If the current input values match at least one element of the transition relation, the FSM non-deterministically chooses one of them and updates the value of each output and of its state accordingly. Note that we use a *Mealy* model for our FSMs, so the output is updated within *the same clock cycle*, while the state is updated at the *next clock cycle*.

More formally, let $\mathcal{N}^{\mathcal{F}} = \{\mathcal{F}_1, \mathcal{F}_2, \ldots \mathcal{F}_m\}$ be an FSM network. Let $\mathcal{I}$ be the set of components of input symbols of elements of $\mathcal{N}^{\mathcal{F}}$ (i.e. the set of input signal names and sets of values). Let $\mathcal{O}$ be the set of components of output symbols of elements of $\mathcal{N}^{\mathcal{F}}$ (i.e. the set of output signal names and sets of values). Let $\mathcal{X}$ be the Cartesian product of the sets of states of the FSMs in $\mathcal{N}^{\mathcal{F}}$.

Definition 8 *The timed trace structure describing the behavior of $\mathcal{N}^{\mathcal{F}}$ is $(A, \mathcal{T})$:*

- $A = \mathcal{I} \cup \mathcal{O} \cup \mathcal{X}$.
- *Each timed trace $T \in \mathcal{T}$ defines the value of each signal and of the state of each FSM at each time.*
- *For every $T \in \mathcal{T}$ for every $\mathcal{F} \in \mathcal{N}^{\mathcal{F}}$ there exists $(x_n, x_v) \in R^{\mathcal{F}}$ such that $(x_n, x_v, 0)$ belongs to T.*
- *Let $T_t = \{(e_n, e_v) | (e_n, e_v, t) \in T\}$ (i.e. the set of names and values of events with time t in trace T).*

  For for every $T \in \mathcal{T}$ for every $\mathcal{F} \in \mathcal{N}^{\mathcal{F}}$ for every t we have:

  - *For every component (o_n, o_V) of the output symbols of $\mathcal{F}$ there exist $(o_n, o_v) \in (o_n, o_V)$ such that $(o_n, o_v) \in T_t$ only if there exists an element of its transition relation $F^{\mathcal{F}}$ such that its current state value and all its input signal names and values appear in T_t and the corresponding output signal value for o_n is o_v.*
  - *The state at $t+1$ is (x_n, x_v) only if there exists an element of its transition relation $F^{\mathcal{F}}$ such that its current state value and all its input signal names and values appear in T_t and the corresponding next state value is x_v.*
  - *Otherwise, the output and state at time $t+1$ are the same as those at time t.*

We already examined an intuitive view of the behavior of the CFSM in terms of transitions and delays. Now 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 the two interact with the "rest of the world" in the same way. This mapping uses a set of "internal" signals that are used only for communication inside the network of FSMs. The internal signals will correspond to the external signals, except for a possible delay and the possible omission of some events (due to the "overwriting" explained in Section 2). Such internal signals of course must not be considered (i.e. they must be "hidden" from the traces [Dil88]) when we compare the behavior of the two.

In the following we will assume that time units for CFSMs and FSMs coincide.

We will also need the notion of *projection* of an n-tuple that is an input or output symbol of an FSM onto one of its components:
$((e_n^1, e_v^1), \ldots (e_n^i, e_v^i), \ldots (e_n^m, e_v^m))|_{e_n^i} = (e_n^i, e_v^i)$

Given a CFSM $\mathcal{C}$, let the corresponding FSM network $\mathcal{N}^{\mathcal{F}}$ be composed of:

1. One "main" *completely specified* FSM $\mathcal{F} = (I^{\mathcal{F}}, O^{\mathcal{F}}, X^{\mathcal{F}}, R^{\mathcal{F}}, F^{\mathcal{F}})$:
   - There are *two* input signals for each input *trigger* event of $\mathcal{C}$, excluding "state" events:
     (a) One, denoted by the "*" in the following, has values 0 and 1. The corresponding event is *present* at all the points in time when the signal has value 1, *absent* otherwise.
     (b) The other one, denoted by the same name without the "*", has the same set of allowed values as the event.
   - There is *one* input signal for each non-state *non-trigger* event of $\mathcal{C}$, carrying its value.
   - There are *two* output signals for each output event of $\mathcal{C}$, with the same convention used for trigger inputs (the state of $\mathcal{C}$ is in general visible to other CFSMs, so it must also be an output).
   - The set of states is the Cartesian product of the sets of input-output event names and values of $\mathcal{C}$ (most often in practice $\mathcal{C}$ will have only one input-output event, that is its state).
   - The set of initial states is the set of pairs of event names and initial values of the state events of $\mathcal{C}$.
   - The transition relation $F^{\mathcal{C}}$ of $\mathcal{C}$ is modified as follows into the transition relation $F^{\mathcal{F}}$ of $\mathcal{F}$, to map the fact that a CFSM does not make a transition until at least one event triggers it:
     (a) – For each element (f^I, f^O) of $F^{\mathcal{C}}$ there exists a set of elements $\{f'\}$ of $F^{\mathcal{F}}$ such that:
         * For each $(i_n, i_v) \in f^I$, $f'|_{*i_n} = (*i_n, 1)$ and $f'|_{i_n} = (i_n, i_v)$.
         * For each $(o_n, o_v) \in f^O$, $f'|_{*o_n} = (*o_n, 1)$ and $f'|_{o_n} = (o_n, o_v)$.
         * Let us suppose, without loss of generality, that $\mathcal{C}$ had a single state event (x_n, x_V). Let x_n and x'_n denote the present and next state component names of $\mathcal{F}$ respectively. If $(x_n, x_v) \in f^I$, then $f'|_{x_n} = (x_n, x_v)$. Similarly, if $(x'_n, x'_v) \in f^O$, then $f'|_{x'_n} = (x'_n, x'_v)$.

         I.e. (f^I, f^O) and each f' coincide on the set of specified events (note that events present in (f^I, f^O) are identified by the corresponding "*" signal at 1 in f').
       – All other events whose name does not appear in f^I or f^O are *not present* in each f', i.e. $f'|_{*e_n} = (*e_n, 0)$ and each combination of their values appears in some such f'.

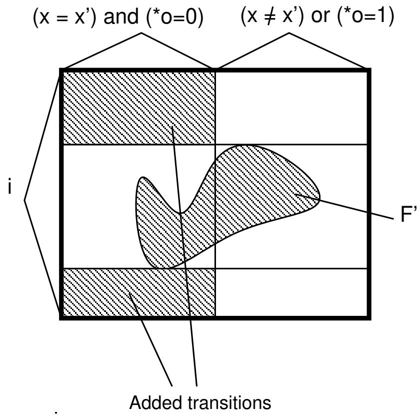

Figure 4: "Self-loop" transitions added to the FSM transition relation

Let F' denote the set of elements of $F^{\mathcal{F}}$ as defined by this mapping.

(b) The complement of the set of event values that triggers a transition causes the FSM to remain in the current state without emitting any event.
I.e. each $f'' \in (I^{\mathcal{F}} \times X^{\mathcal{F}} \times X^{\mathcal{F}} \times O^{\mathcal{F}}) - F'$ belongs to $F^{\mathcal{F}}$ if and only if:
   - For each output signal (o_n, o_V), $f''|_{*o_n} = (*o_n, 0)$.
   - Let x_n, x'_n be the present and next state component names respectively: $f''|_{x_n} = f''|_{x'_n}$.

The transformation that adds the set of f'' to $F^{\mathcal{F}}$ is also represented pictorially in Figure 4. Basically, the set of specified transitions of $F^{\mathcal{C}}$ is complemented, then projected over the set of input signals to obtain all the unspecified input combinations, and then expanded to cover all the output and next state values that did not cause a state change and did not emit an event.

It should be obvious that $\mathcal{F}$ as defined above is *completely specified*.

2. The input and output signals of $\mathcal{F}$ will be "internal", transmitted to the external world through "buffers" that mimic the unbounded detection and reaction delays of the CFSM. These buffers are represented also by FSMs.

   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 corresponding to $*i_n, i_n$. Let us denote this pair of signals by $*i'_n, i'_n$. These signals will appear in the timed traces as input signals of the FSM network representing $\mathcal{C}$ and will be shown to be "equivalent" in terms of behavior to the input events of $\mathcal{C}$.
   - Its outputs are the "internal" pair of signals $*i_n, i_n$.
   - Its set of states and *non-deterministic* transition relation are schematically shown in Figure 5, for an external signal $*i'$ with values 1 and 2 (the general case for an n-valued event is similar but more complex). The state denoted by the bold line is the initial state. Note that the FSM is non-deterministic, and memorizes the last input value received when $*i' = 1$, emitting it on $*i$, i after an *unbounded, possibly zero* amount of time.

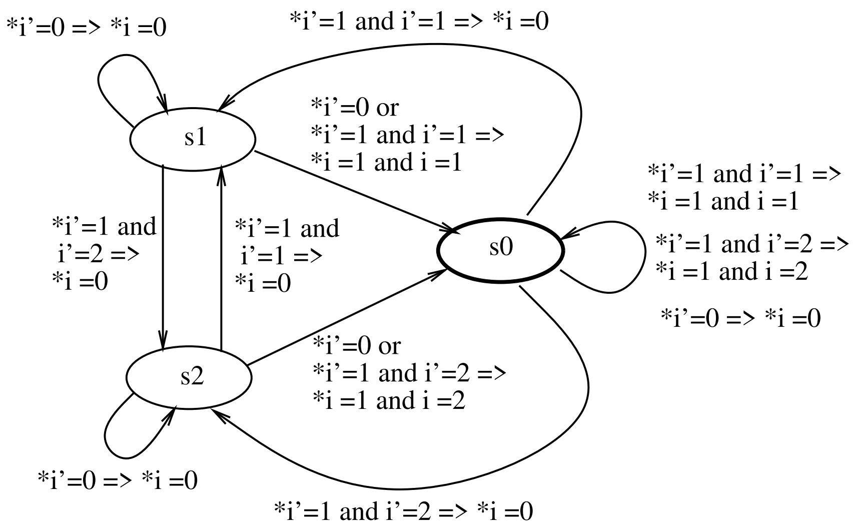

Figure 5: Input FSM structure

3. An FSM similar to those used for input signals governs the transmission of each output signal pair of $\mathcal{F}$ to the external world. Its states and transition relation are schematically shown, also for a two-valued output signal, in Figure 6. The only difference with respect to the input case is that now these FSMs have a minimum delay of *one* cycle. Moreover they *must* transmit the stored value when a new one is received, thus obeying the constraint that a CFSM transition must emit *all* its output events.

   Note that the self-loops of states s_1 and s_2 can either non-deterministically wait without transmitting the stored event or, if a new event arrives, be forced to transmit the stored one.

Let us see an example of transformation from a very simple CFSM to the associated "main" FSM. The CFSM:

```
.names *e x => x *y
       1  0    1 0
       1  1    2 0
       1  2    3 0
       1  3    0 1
```

represents a modulo-4 counter that is incremented when $*e$ occurs and that emits $*y$ when the count becomes 0.

The first part F' of the FSM transition relation is[6]:

```
.names *e e x => x'  *y y
       1  - 0    1   0  -
```

[6] Here we use SHIFT to describe both the CFSM and the FSM, with a slight abuse of notation. In practice the FSM will be represented in BLIF-MV ([BCH+91]), where `.names` tables have only a single output.

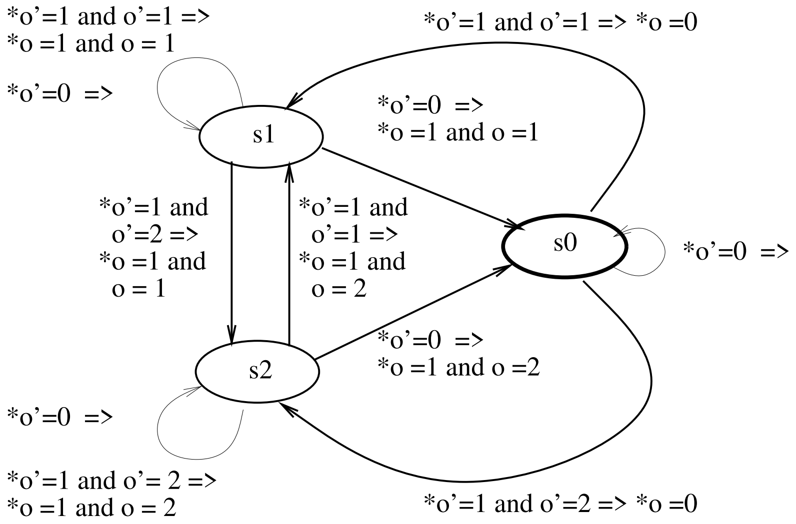

Figure 6: Output FSM structure

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

Recall that `x` and `x'` represent two distinct components, defining the present and the next state of the FSM respectively.

The complement, projected along the set of inputs, is:

```
.names *e e x => x'  *y y
        0  - -    -   -  -
```

Combining it with $*y = 0$ and $x = x'$ yields:

```
.names *e e x =>  x'  *y y
        0  - -    (x) 0  -
```

The union with the original relation gives the full transition relation for the "main" FSM.

```
.names *e e x => x'  *y y
        1  - 0    1   0  -
        1  - 1    2   0  -
        1  - 2    3   0  -
        1  - 3    0   1  1
        0  - -    (x) 0  -
```

We can now show formally the behavioral equivalence between a CFSM network and an FSM network obtained according to the rules described above:

Theorem 1 *Let* $\mathcal{N}^{\mathcal{C}} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_m\}$ *be a network of CFSMs.*

Let $\mathcal{N}^{\mathcal{F}} = \{\mathcal{F}_{11}, \mathcal{F}_{12}, \ldots \mathcal{F}_{21}, \mathcal{F}_{22}, \ldots \mathcal{F}_{m1}, \mathcal{F}_{mk}\}$ *be a network of FSMs obtained by interconnecting the FSMs obtained from each* $\mathcal{C}_i$ *as described above.*

Then:

- *For each trace* $T^{\mathcal{C}}$ *belonging to the trace structure of* $\mathcal{N}^{\mathcal{C}}$ *there exists a trace* $T^{\mathcal{F}}$ *belonging to the trace structure of* $\mathcal{N}^{\mathcal{F}}$ *such that for each event* $(e_n, e_v, t) \in T^{\mathcal{C}}$ *there exist two corresponding external signal values*[7] $(*e'_n, 1, t)$ *and* (e'_n, e_v, t) *belonging to* $T^{\mathcal{F}}$*.*
- *For each trace* $T^{\mathcal{F}}$ *belonging to the trace structure of* $\mathcal{N}^{\mathcal{F}}$ *there exists a trace* $T^{\mathcal{C}}$ *belonging to the trace structure of* $\mathcal{N}^{\mathcal{C}}$ *such that for each pair of external signal values* $(*e'_n, 1, t)$ *and* (e'_n, e_v, t) *belonging to* $T^{\mathcal{F}}$ *there exists a corresponding event* $(e_n, e_v, t) \in T^{\mathcal{C}}$*.*

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 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 $\mathcal{I}_h$ and $\mathcal{I}_s$ such that the set of timed traces of $\mathcal{N}^{\mathcal{F}}$ includes that of $\mathcal{I}_s$ and that of $\mathcal{I}_h$. However, in general the sets of timed traces of $\mathcal{I}_h$ and $\mathcal{I}_s$ are not equal.

This behavioral containment will be ensured by the fact that:

- For a hardware implementation, the input FSMs are reduced only to the s_0 state (i.e., they "transparently" transmit all events) and the output FSMs delay each event by exactly one cycle.
- 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.

[7]We ignore, for the sake of simplicity, the case of "pure value" events that are emitted by the environment. The extension to such events is trivial but would unnecessarily clutter the notation.

5 Conclusions and Future Work

This paper introduced Codesign Finite State Machines, a formal model for hardware/software codesign. The model satisfies the following fundamental requirements:

1. It uses a Finite State Machine-like paradigm that is well suited to model control-dominated circuits.
2. It is based on *events* as a basic communication primitive. Events are low-level enough to be efficiently implemented both in hardware and software, without imposing unnecessary overheads, and yet general enough to allow the construction of more powerful communications schemes (such as, for example, channels and rendez-vous).
3. It does not commit to a particular implementation choice, because the delay associated with each event/reaction pair is *non-zero* and *unbounded*.

We also described how CFSMs can be represented in a textual file format, using the SHIFT description language.

We showed how CFSMs can be mapped into a more traditional FSM-based model. The purpose of this mapping was two-fold:

- To allow formal verification of properties of the specification, by using existing FSM-based verification tools.
- To prove the correctness of the implementation in a mix of hardware or software, since the behavior of hardware can be directly modeled as a set of cooperating FSMs, and some restricted forms of software implementations can also be described in the same way.

We are currently working on an implementation of partitioning and synthesis algorithms based on this model (as outlined in Section 1).

In the future, we are planning 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 hope to be able to extend and use methods based on partial orders, such as those proposed by [NPW81] and [BE91], that avoid the state explosion problem due to the exploration of all possible interleavings of *concurrent* events. We would also like to use probabilistic timing analysis techniques to verify the satisfaction of timing constraints without the need to use expensive exhaustive verification methods based on state space exploration (e.g., [Bur92], [ACD90]).

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.

[Aco92] R. D. Acosta. Use of sdataflow specifications for software/hardware codesign. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.

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

[BC84] Gerard Berry and Laurent Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 369–448. Carnegie-Mellon Univeristy, 1984.

[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.

[BE91] E. Best and J. Esparza. Model checking of persistent Petri Nets. In *Computer Science Logic 91 (LNCS)*, 1991. Also appeared as Hildesheimer Informatik Fachbericht 11/91.

[Ber91] G. Berry. A hardware implementation of pure ESTEREL. In *Proceedings of the International Workshop on Formal Methods in VLSI Design*, January 1991.

[BM87] S. Burns and A. Martin. A synthesis method for self-timed VLSI circuits. In *Proceedings of the International Conference on Computer Design*, 1987.

[BN92] W. Baker and A. R. Newton. Synchronous parallelism and object-oriented computing for real-time software applications. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.

[Bur92] J. R. Burch. *Automatic Symbolic Verification of Real-Time Concurrent Systems.* PhD thesis, Carnegie Mellon University, August 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. Synthesis of mixed software-hardware implementations from cfsm specifications. 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.

[COB92] P. Chou, R. Ortega, and G. Borriello. Synthesis of hardware/software interface in microcontroller-based systems. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[Coc92] M. Cochran. Using the rate monotonic analysis to analyze the schedulability of ADARTS real-time software designs. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.

[CR78] P. F. Conklin and D. P. Rodgers. Advanced minicomputer designed by team evaluation of hardware-software tradeoffs. *Computer Design*, pages 129–137, April 1978.

[Dil88] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. The MIT Press, Cambridge, Mass., 1988. An ACM Distinguished Dissertation 1988.

[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.

[GJM92a] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli. Program implementation schemes for hardware-software systems. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.

[GJM92b] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proceedings of the Design Automation Conference*, June 1992.

[GJM92c] 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.

[Gun92] J. Gunawardena. Causal automata. *Theoretical Computer Science*, (101):265–288, 1992.

[HJB+82] J. Hennessy, N. Jouppi, F. Baskett, T. R. Gross, and J. Gill. Hardware-software tradeoffs for increased performance. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, March 1982.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. In *Communications of the ACM*, pages 666–677, August 1978.

[JLHM91] M. Jaffe, N. Leveson, M. Heimdahl, and B. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3), March 1991.

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

[Kur93] R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1993. To appear.

[Mar90a] A. Martin. Formal program transformations for VLSI synthesis. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs*, The UT Year of Programming Series. Addison-Wesley, 1990.

[Mar90b] A. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communications*, The UT Year of Programming Series. Addison-Wesley, 1990.

[MKP92] M.C. McFarland, T.J. Kowalski, and M.J. Peman. Language and formal semantics of the specification system CPA. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.

[Mur89] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, April 1989.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. part I. *Theoretical Computer Science*, 13:85–108, 1981.

[NVG91] S. Narayan, F. Vahid, and D. D. Gajski. System specification and synthesis with the SpecCharts language. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[Pet62] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn, Institut für Instrumentelle Mathematik, 1962. (technical report Schriften des IIM Nr. 3).

[PL91] I. Page and W. Luk. Compiling `occam` into FPGAs. In W. Moore and W. Luk, editors, *FPGAs*, pages 271–283. Abingdon EE&CS Books, 1991.

[RR83] G. S. Rao and P. L. Rosenfeld. Integration of machine organization and control program design – review and direction. *IBM Journal of Research and Development*, 27(3):247–256, May 1983.

[SB91] M. B. Srivastava and R. W. Brodersen. Rapid-prototyping of hardware and software in a unified framework. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[SB92] J. Sun and R. W. Brodersen. Design of system interface modules. In *Proceedings of the International Conference on Computer-Aided Design*, pages 478–481, November 1992.

[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.

[Str92] C. E. Stroud. Problems associated with hardware implementation of software algorithms using behavioral model synthesis. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.

[Tur78] R. Turn. Hardware-software tradeoffs in reliable software development. In *11th Annual Asilomar Conference on Circuits, Systems and Computers*, pages 282–288, 1978.

[vBKR+91] K. van Berkel, J. Kessels, M. Roncken, R. Saejis, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings of the European Design Automation Conference (EDAC)*, pages 384–389, 1991.

[WOB92] A. S. Wenban, J. W. O'Leary, and G. M. Brown. Codesign of communication protocols. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.

[WTHM92] W. Wolf, A. Takach, C.-Y. Huang, and R. Manno. The Princeton University behavioral synthesis system. In *Proceedings of the Design Automation Conference*, June 1992.

[WWD92] N. Woo, W. Wolf, and A. Dunlop. Compilation of a single specification into hardware and software. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.