Models of Computation for Embedded System Design

Luciano Lavagno
Department of Electronics
Politecnico di Torino
C. Duca degli Abruzzi 24, Torino, Italy
`lavagno@polito.it`

Alberto Sangiovanni-Vincentelli
Department of EECS
University of California
Berkeley, CA 94709 USA
`alberto@eecs.berkeley.edu`

Ellen Sentovich
Cadence Berkeley Laboratories
2001 Addison St.
Berkeley, California USA
`ellens@cadence.com`

September 28, 1998

Abstract

In the near future, most objects of common use will contain electronics to augment their functionality, performance, and safety. Hence, time-to-market, safety, low-cost, and reliability will have to be addressed by any system design methodology. A fundamental aspect of system design is the specification process. We advocate using an unambiguous formalism to represent design specifications and design choices. This facilitates tremendously efficiency of specification, formal verification, and correct design refinement, optimization, and implementation. This formalism is often called *model of computation.* There are several models of computation that have been used, but there is a lack of consensus among researchers and practitioners on the "right" models to use. To the best of our knowledge, there has also been little effort in trying to compare rigorously these models of computation. In this paper, we review current models of computation and compare them within a framework that has been recently proposed. This analysis demonstrates both the need for heterogeneity to capture the richness of the application domains, and the need for unification for optimization and verification purposes. We describe in detail our CFSM model of computation, illustrating its suitability for design of reactive embedded systems and we conclude with some general considerations about the use of models of computations in future design systems.

1 Introduction

1.1 Embedded System Design Today

An embedded system is a complex object containing a significant percentage of electronic devices (generally including at least one computer) that interacts with the real world (physical environment, human users, etc.) through sensing and actuating devices. A system is heterogeneous, i.e., is characterized by the co-existence of a large number of components of disparate type and function. For example, it may contain programmable components such as micro-processors and Digital Signal Processors, as well as analog components such as A/D and D/A converters, sensors, transmitters and receivers. In the past, the system design effort has focused on these hardware parts, leaving the software design to be done afterwards as an implementation step. However, today more than 70% of the development cost for complex systems such as automotive electronics and communication systems is attributable to software development. This percentage is increasing constantly. The challenge posed to the semiconductor industry is to provide a new generation of programmable parts and of supporting tools to help system designers develop software faster and correctly the first time.

Today much attention is devoted to the hardware-software co-design issue, i.e., to the concurrent development of Application Specific Integrated Circuits and standard hardware components, selection of programmable components, and development of the application software that will run on them. We believe that this approach in fact enters the design process too late to explore interesting design trade-offs.

1.2 Our Design Methodology Goals

The computer-aided design process should begin at a very early stage. We believe that the real key to shortening design time and coping with complexity is to start the design process *before* the hardware-software partitioning. For this reason, we believe that the key problem is not so much hardware-software co-design, but the sequence consisting of specifying what the system is intended to do with no bias towards implementation, of the initial functional design, its analysis to determine whether the functional design satisfies the specification, the mapping of this design to a candidate architecture, and the subsequent

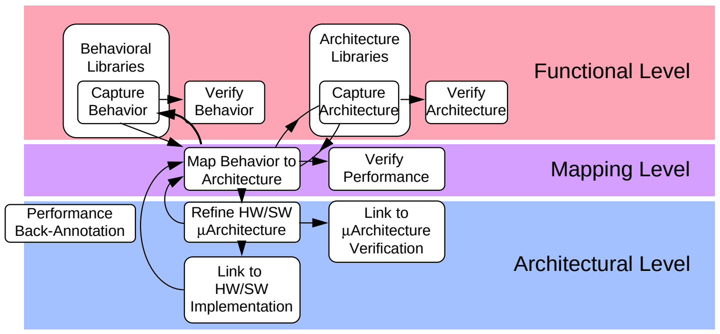


Figure 1: Proposed design strategy

performance evaluation. It is then clear that the key aspect of system design is indeed *function-architecture co-design*.

Our approach is a design methodology that is based on the use of *formal models* to describe the behavior of the system at a high level of abstraction, before a decision on its decomposition into hardware and software components is taken. Our approach also facilitates the use of existing parts. As the complexity of embedded systems increases, it is unthinkable to design new systems from scratch. Already hardware components are often standard parts that are acquired from silicon vendors, and software is often incrementally upgraded from previous versions of the same product. In the future, *design re-use* will be the key to profitability and market timing. In addition, the decreasing feature size of silicon manufacturing processes will make it possible to incorporate multiple microprocessors, complex peripherals, and even sensors and actuators on the same silicon substrate, which will force system developers and IC designers/manufacturers to deal with the problem of *exchanging Intellectual Property* in the form of designs instead of chips.

1.3 Design Strategy

The overall design strategy that we envision is depicted in Figure 1. There are, of course, other ways to design systems in common use today. The top-down nature of the design methodology that our group has advocated throughout the years may not be agreed upon by the system design community where a mixed top-down, bottom-up approach is mostly used. In our methodology, however, we believe that this approach is captured by the presence of architectural and functional libraries that could be the result of a bottom-up assembly of basic components. We strongly emphasize that no matter how the design is carried out, a *rigorous framework is necessary to reduce design iterations and to improve design quality*.

1.3.1 Design Conception to Design Description

At the functional level, a behavior for the system to be implemented is selected and analyzed against a set of specifications. The definition of specification and behavior is often the subject of hot debate. For some, there is no difference between specification and behavior. For some, specification is the I/O relation of the system to realize together with a set of constraints to satisfy and of goals to achieve, and behavior is the algorithm that realizes the function to be implemented. For others, specification is the algorithm itself. From a purist point of view, an algorithm is indeed the result of an implementation decision from a given set of specifications and we prefer to stick to this view in our design methodology. For example, if we specify the function that a system has to perform as "given a nonlinear function f over the set of reals, find x so that f(x)=0", then it is a design decision to chose the Newton-Raphson algorithm or a Gauss-Seidel nonlinear relaxation algorithm. On the other hand, for an MPEG encoder, the specification is the encoding of the compressed stream, and any implementation that creates it from a stream of images is "correct". In this second case, the first step of system design has already been decided upon and the designer has no freedom to alter the conceptual design.

1.3.2 Algorithm Design

Algorithm development is a key aspect of system design at the functional level. We believe that little has been done in this domain to help the designer to select an algorithm that satisfies the specifications. The techniques and environments for this step are often application dependent. We have experience in automotive engine control [7], where the algorithms have to have strong correctness properties due to the life critical aspects inherent in this application. In addition, the "plant" to be controlled (the combination of the engine and the drive-line) is a hybrid system consisting of continuous components (drive-line) and discrete ones (engine). To assess the properties of the algorithms, one must use control theory and sophisticated simulation techniques involving mixed differential equations-discrete event models. The understanding of the application domains yields a design methodology that integrates the application-specific view with general-purpose techniques that could be re-used in other domains of application. It is our strong belief that this step of system design carries the maximal leverage when combined with the design methodology proposed here.

1.3.3 Algorithm Analysis

The behavior of an algorithm is verified by performing a set of analysis steps. Analysis is a more general concept than simulation. For example, analysis may mean the formal proof that the algorithm selected always converges, that the computation performed satisfies a set of specifications, or that the computational complexity, measured in terms of number of operations, is bounded by a polynomial in the size of the input. In the view of design re-use, parts of the overall behavior may be taken from an existing library of algorithms. *Since it is the formal model that provides the framework for algorithm analysis, it is very important to decide which mathematical model to support in a design environment.*

1.4 Algorithm Implementation

Once the algorithm has been selected, there is an intermediate step before the selection of the architecture to support its implementation: its transformation into a set of functional components that are computationally tractable. This set of functional components have to be formally defined to ensure that the properties of the implementation of the algorithm can be assessed. To do so, the concept of models of computation is key. Most system designs use one or more of the following models of computation: Finite State Machines, Data Flow Networks, Discrete Event Systems, and Communicating Sequential Processes. A particular model of computation has mathematical properties that can be efficiently exploited to answer questions about system behavior without carrying out expensive verification tasks. An important issue here is how to compare and compose different models of computation.

Once the model(s) of computation have been selected, then we can safely proceed towards the implementation of the system by selecting the physical components (architecture) of the design.

1.5 Our Goals

The main goal of this paper is to review and compare the most important models of computations using a unifying theoretical framework introduced recently by Lee and Sangiovanni-Vincentelli [43]. We also believe that it is possible to optimize across model-of-computation boundaries to improve the performance of and reduce errors in the design at an early stage in the process.

There are many different views on how to accomplish this. There are two essential approaches: one is to develop encapsulation techniques for each pair of models that allow different models of computation to interact in a meaningful way, i.e., data produced by one object are presented to the other in a consistent way so that the object "understands" [17]. The other is to develop an encompassing framework where all the models of importance "reside" so that their combination, re-partition and communication happens in the same generic framework. While we realize that today heterogeneous models of computation are a necessity, we believe that the second approach will be possible and will provide the designer with a powerful mechanism to actually select the appropriate models of computation, (e.g., FSMs, Data-flow, Discrete-Event, that then become a lower level of abstraction with respect to the unified model) for the essential parts of the design.

At this level is also important to *orthogonalize concerns*, that is to separate different aspects of the design. In this regard, a natural dividing line is the separation between functionality and communication. That is, we view a design as composed of functional behavior (modules) and communication behavior (between modules), which are themselves further decomposed as we refine and analyze the design. It is our strong belief that communication is key in assembling systems on a chip from separate Hip's. Communication is a complex issue since the functionality of the components being interconnected must be preserved.

The separation between function and communication will be emphasized throughout the paper. In addition, a model of computation that encompasses the key aspects of Discrete Event, Data-Flow and Finite State Machine models will be presented in detail. This model, called network of Co-design Finite State Machines (CFSM), is the backbone of the POLIS system developed at the University of California at Berkeley, an environment for function-architecture co-design with particular emphasis on control-dominated applications and on software development. (See Figure 2 for a block diagram of the functionalities of the environment.) This model is also used as the basic semantic model for an industrial product of Cadence Design Systems, an environment for embedded system design including multi-media and telecommunication applications.

The paper is organized as follows. In Section 2, we present the mathematical machinery used to compare and describe the models of computation. In Section 3, we present and compare the most important models of computation. In Section 4, we introduce the CFSM model. In Section 5, we give some concluding remarks.

2 MOCs: Basic Concepts and the Tagged Signal Model

2.1 Modeling Embedded Systems with MOCs

An MOC is composed of a description mechanism (syntax) and rules for computation of the behavior given the syntax (semantics). An MOC is chosen for describing a sub-behavior of a design based on its suitability: compactness of description, fidelity to design style, ability to synthesize and optimize the behavior to an appropriate implementation. For example, some MOCs are suitable for describing complicated data transfer functions and completely unsuitable for complex control, while others are designed with complex control in mind.

There are a number of basic ideas and primitives that are commonly used in formulating models of computation. Most MOCs permit distributed system description (a collection of communicating modules), and give rules dictating how each module computes (function) and how they transfer information between them (communication). Some of the primitives include combination Boolean

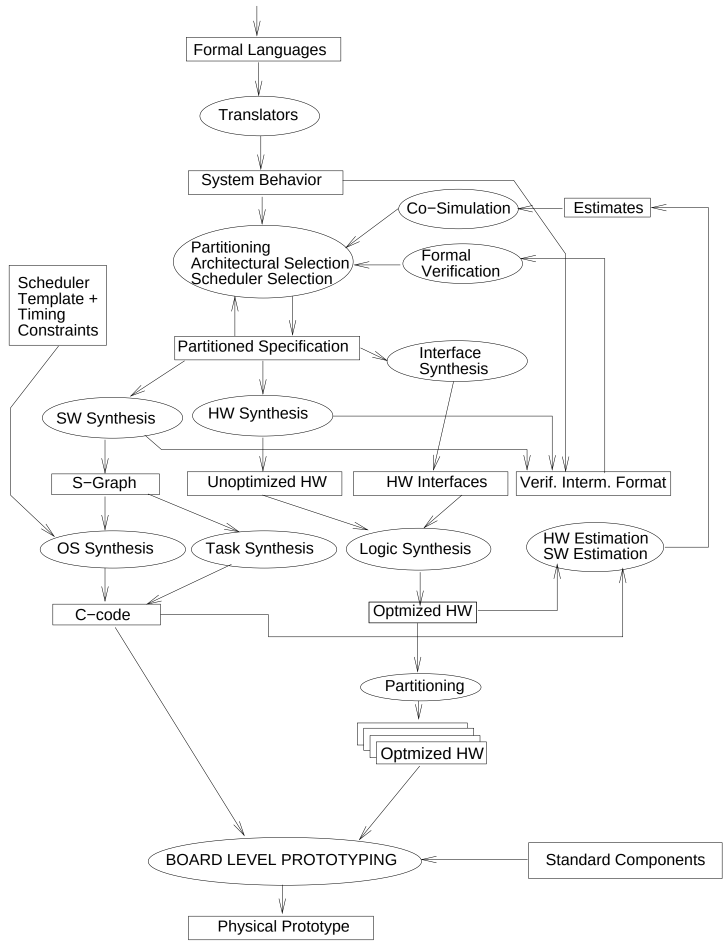


Figure 2: The POLIS design framework

functions and synchronous state machines for specifying function, and queues, buffers, schedulers for specifying communication. Function and communication are often not described completely separately for efficiency and optimization.

More precisely, MOCs are typically realized (implemented in practice) by a particular language and its semantics. We elaborate on the distinction between MOCs and languages in Section 2.2.

A *language* is a set of symbols, rules for combining them (its *syntax*), and rules for interpreting combinations of symbols (its *semantics*). Two approaches to semantics have evolved, *denotational* and *operational*. A language can have both (ideally they are consistent with one another, although in practice this can be difficult to achieve). Denotational semantics, first developed by Scott and Strachey [59], gives the meaning of the language in terms of relations. Operational semantics, which dates back to Turing machines, gives meaning of a language in terms of actions taken by some abstract machine, and is typically closer to the implementation.

Models of computation can be viewed based on the following characteristics:

- the kinds of relations that are possible in a denotational semantics
- how the abstract machine behaves in an operational semantics
- how individual behavior is specified and composed
- how hierarchy abstracts this composition
- communication style

A design (at all levels of the abstraction hierarchy from functional specification to final implementation) is generally represented as a set of components, which can be considered as isolated monolithic blocks, which interact with each other and with an environment that is not part of the design. The model of computation defines the behavior and interaction of these blocks.

We view MOCs at two levels of abstraction. At the higher level, we take the view of the tagged signal model (which we call here TSM) described in section 2.3. The TSM abstraction defines processes and their interaction using signals composed of partially ordered events, in turn composed of tags and values. We use processes to describe both functional behavior and communication behavior. This is a denotational view, though it is not associated with a particular language. We use this model to compare elements of different models of computation, styles of sequential behavior, concurrency, and communication at a high level.

At the lower level of abstraction, we take the view of general primitives for function and timing (used in the refinement of TSM processes), where each MOC constitutes a particular choice of these two. This is a more operational view. We give precise definitions for a number of terms, but these definitions will inevitably conflict with standard usage in some communities. We have discovered that, short of abandoning the use of most common terms, no terminology can be consistent with standard usage in all related communities. We attempt to avoid confusion by being precise, even at the risk of being pedantic. The basic primitive concepts are describe in Section 2.4. The primitive building blocks for specification and implementation are given in Section 2.5.

All these basic primitives and concepts are then used in Section 3 to classify and describe the main MOCs that appear in the literature.

2.2 Languages and Models of Computation

The distinction between a language and its underlying model of computation is important. The same model of computation can give rise to fairly different languages (e.g., the imperative Algol-like languages C, C++, Pascal, and FORTRAN). Some languages, such as VHDL and Verilog, support two or more models of computation[1].

The model of computation affects the *expressiveness* of a language — which behaviors can be described in the language, whereas the syntax affects compactness, modularity, and reusability. Thus, for example, object-oriented properties of imperative languages like C++ are more a matter of syntax than a model of computation.

The expressiveness of a language is an important issue. A language that is not expressive enough to specify a particular behavior is clearly unsuitable, while a language that is too expressive is often too complex for analysis and synthesis. For very expressive languages, many analysis and synthesis problems become undecidable: no algorithm will solve all problem instances in finite time.

A language in which a desired behavior cannot be represented succinctly is also problematic. The difficulty of solving analysis and synthesis problems is at least linear in the size of the problem description, and can be as bad as several times exponential, so choosing a language in which the description of the desired behavior of the system is compact can be critical.

A language may be very incomplete and/or very abstract. For example, it may specify only the interaction between computational modules, and not the computation performed by the modules. In this case, it provides an interface to a host language that specifies the computation, and is called a coordination language (examples include Linda [20], Granular Lucid [34], and Ptolemy domains [17]). Another language may specify only the causality constraints of the interactions without detailing the interactions themselves nor providing an interface to a host language. In this case, the language is used as a tool to prove properties of systems, as done, for example, in process calculi [33, 46] and Petri nets [50, 53]. In still more abstract modeling, components in the system are

[1]They directly support the Imperative model within a process, and the Discrete Event model among processes. They can also support Extended Finite State Machines under suitable restrictions known as the "synthesizable subset".

replaced with nondeterminate specifications that give constraints on the behavior, but not the behavior itself. Such abstraction provides useful simplifications that help formal verification.

2.3 The Tagged-Signal Model

At the highest level of abstraction, we adopt the tagged-signal model (TSM) proposed by Lee and Sangiovanni-Vincentelli [42]. It is a formalism for describing aspects of models of computation for embedded system specification. It is denotational in the Scott and Strachey [59] sense, and it defines a semantic framework (of signals and processes) within which models of computation can be studied and compared. It is very abstract—describing a particular model of computation involves imposing further constraints that make it more concrete.

2.3.1 Signals, tags and events

The fundamental entity in the TSM is an event: a value/tag pair. Tags are often used to denote temporal behavior. A set of events (an abstract aggregation) is a signal. Processes are relations on signals, expressed as sets of n-tuples of signals. A particular model of computation is distinguished by the order it imposes on tags and the character of processes in the model. More formally, given a set of *values* V and a set of *tags* T, an *event* is a member of $T \times V$. A *signal* s is a set of events, and thus is a subset of $T \times V$. A *functional* (or deterministic) *signal* is a (possibly partial) function from T to V. The set of all signals is denoted S. A *tuple* of n signals is denoted $\mathbf{s}$, and the set of all such tuples is denoted S^n.

The different models of time that have been used to model embedded systems can be translated into different order relations on the set of tags T in the tagged-signal model. In a *timed system* T is totally ordered, i.e., there is a binary relation $<$ on members of T such that if $t_1, t_2 \in T$ and $t_1 \neq t_2$, then either $t_1 < t_2$ or $t_2 < t_1$. In an *untimed system*, T is only partially ordered.

2.3.2 Processes

A *process* P with n signals is a subset of the set of all n-tuples of signals, S^n for some n. A particular $\mathbf{s} \in S^n$ is said to *satisfy* the process if $\mathbf{s} \in P$. An $\mathbf{s}$ that satisfies a process is called a *behavior* of the process (intuitively, it is the generalization of a "simulation trace"). Thus a *process* is a set of possible *behaviors*, or a constraint on the set of "legal" signals.

Intuitively, processes in a system operate *concurrently*, and constraints imposed on their signal tags define *communication*[2] among them. The environment in which the system operates can be modeled with a process as well.

[2]This is often called also synchronization, but we will try to avoid using the term in this sense because it is too overloaded.

For many (but not all) applications, it is natural to partition the signals associated with a process into *inputs* and *outputs*. Intuitively, the process does not determine the values of the inputs, and does determine the values of the outputs. If $n = i + o$, then (S^i, S^o) is a partition of S^n. A process with i inputs and o outputs is a subset of $S^i \times S^o$. In other words, a process defines a *relation* between input signals and output signals. A $(i + o)$-tuple $\mathbf{s} \in S^{i+o}$ is said to *satisfy* P if $\mathbf{s} \in P$. It can be written $\mathbf{s} = (\mathbf{s}_1, \mathbf{s}_2)$, where $\mathbf{s}_1 \in S^i$ is an i-tuple of *input signals* for process P and $\mathbf{s}_2 \in S^o$ is an o-tuple of *output signals* for process P. If the input signals are given by $\mathbf{s}_1 \in S^i$, then the set $I = \{(\mathbf{s}_1, \mathbf{s}_2) \mid \mathbf{s}_2 \in S^o\}$ describes the inputs, and $I \cap P$ is the set of behaviors consistent with the input $\mathbf{s}_1$.

A process F is *functional* (or deterministic) with respect to an input/output partition if it is a single-valued, possibly partial, mapping from S^i to S^o. That is, if $(\mathbf{s}_1, \mathbf{s}_2) \in F$ and $(\mathbf{s}_1, \mathbf{s}_3) \in F$, then $\mathbf{s}_2 = \mathbf{s}_3$. In this case, we can write $\mathbf{s}_2 = F(\mathbf{s}_1)$, where $F : S^i \rightarrow S^o$ is a (possibly partial) function. Given the input signals, the output signals are determined (or there is unambiguously no behavior). A process is *completely specified* if it is a total function, that is, for all inputs in the input space, there is a unique behavior.

2.3.3 Process composition

Process composition in the TSM is defined by the *intersection* of the constraints each process imposes on each signal. To facilitate its definition, we assume that all the processes that are composed are defined on the same set of signals[3]. Hence a composition of a set of processes is also a process.

In the rest of this paper, we will use processes to model both *function* and *communication*. Generally, an MOC defines a flexible mechanism for modeling function, and a rigid mechanism (signal, queue, shared variable, ...; see Section 2.5) for modeling communication. On the other hand, the TSM must compare different MOCs and hence be flexible when modeling communication as well. It is, however, useful to distinguish between the two at least conceptually, since:

1. Functional processes are mostly concerned with the *value* component of their signals, and generally do not have much to do with the *tag* component. In other terms, the constraints a functional process imposes on its input and output signals are generally complex with respect to values, but much simpler with respect to tags.

2. Communication processes are *solely* concerned with the *tag* component of their signals, while values are left untouched.

[3]This can be obtained trivially, since a process can be extended to any new signal by simply not imposing any constraint on it.

One of the most useful and important questions to ask when composing processes, is what properties of the isolated processes are *preserved by composition*. Here we focus only on two fundamental properties: functionality (unique output n-tuple for every input n-tuple) and complete specification (for every input n-tuple there exists a unique output n-tuple).

To analyze this aspect, we note that, given a formal model of the functional specifications and of the properties, three situations may arise:

1. The property is *inherent* for the model of the specification (i.e., it can be shown formally to hold for all specifications described using that model).

2. The property can be verified *syntactically* for a given specification (i.e., it can be shown to hold with a simple, usually polynomial-time, analysis of the specification).

3. The property must be verified *semantically* for a given specification (i.e., it can be shown to hold by executing, at least implicitly, the specification for all inputs that can occur).

For example, consider the functionality property. Any design described by a dataflow network (a formal model to be described later) is functional (also called deterministic or determinate in data-flow vernacular), and hence this property need not be checked for this model of computation. If the design is represented by a network of FSMs (for example, synchronous composition of Mealy Finite State Machines), even if the components are functional and completely specified, the result of the composition may be either incompletely specified (the composition has no solution) or non-functional (the composition has multiple solutions). These situations arise if and only if when a combinational feedback loop exists in the composition: with an odd number of Boolean inverters, there is no "solution" and the composition is incompletely specified, with an even number of inverters, there are multiple solutions and the composition is non-functional. A syntactical check on the composition to verify whether combinational loops exist can be carried out. If none exist, then the composition is functional and completely specified. With Petri nets, on the other hand, functionality is difficult to prove: it must be checked by exhaustive simulation.

2.3.4 Examples

Consider, as a motivating example introducing these several mechanisms to denote temporal behavior, two problems: one of analysis, modeling a time-invariant dynamical system on a computer, and one of design, the design of a two-elevator system controller.

Analysis Example The underlying mathematical model of a time-invariant dynamical system, a set of differential equations over continuous time, is not directly implementable on a digital computer, due to the double quantization of real numbers into finite bit strings, and of time into clock cycles. Hence a first translation is required, by means of an *integration rule*, from the differential equations to a set of *difference equations*, that are used to compute the values of each signal with a given tag from the values of some other signals with previous and/or current tags.

If it is possible to identify several strongly connected components in the dependency graph[4], then the system is *decoupled*. It becomes then possible to go from the total order of tags implicit in physical time to a *partial order* imposed by the depth-first ordering of the components. This partial ordering gives us some freedom in implementing the integration rule on a computer. We could, for example, play with scheduling by embedding the partial order into the total order among clock cycles. It is often convenient, for example, to evaluate a component completely, for all tags, before evaluating components that depend on it. It is also possible to spread the computation among multiple processors.

In the end, time comes back into the picture, but the *double transformation*, from total to partial order, and back to total order again, is essential to

1. *prove properties* about the implementation (such as stability of the integration method, or a bound on the maximum execution time),

2. *optimize* the implementation with respect to a given cost function (e.g., size of the buffers required to hold intermediate signals versus execution time, or satisfaction of a constraint on the maximum execution time).

Design Example. One of the key motivations for the tagged signal model was to avoid over-specification of designs. For a two-elevator system controller, a simplistic set of specifications can be expressed as follows: respond to all requests in the exact order they are received with the criterion that the maximum delay from the time a request is received and the time the elevator service is offered, is minimized. It is clear that the two elevators are concurrent subsystems and that their operation can be controlled with no need to "synchronize" their operation. It is determined by analyzing the order of events not the exact time of occurrence. However, if no assumption is made about the way requests are made, then we may end up in a dead-lock situation due to the nature of the specification. In fact, if three requests are made at the same instant of time, then the response cannot follow the specification, since there are only two elevators available. One solution to this problem is to assume that no two requests may happen at the exact same time; then the specification can be met for the two elevator system. Another solution is to arbitrarily assign priorities among requests that happen at the same time: the specification is changed to reflect these priorities instead of those implied by the order of occurrence. The most

[4] A directed graph with a node for each signal, and an edge between two signals whenever the equation for the latter depends on the former.

important aspect in design is to capture the intent of the designer by abstracting away the non-essential aspects of the system. This example illustrates that it is essential to classify MOCs by their treatment of events with the same tag. This aspect is strictly related to the notion of synchrony and asynchrony as we will see later.

Once the control algorithm has been developed, then its implementation needs to be carried out. If the algorithm is implemented in software running on a single processor, then all events processed are totally ordered and the order is determined by the intrinsic order coming from the specifications (order of occurrence of the requests) and by the existence of limited resources. Even if the partial order dictated by the algorithm exposes some potential parallelism, the presence of a single processor forces sequential execution determined by a scheduling algorithm that decides in which order the operations are executed. Hence, in the end, we need to map an abstract design into the physical world characterized by real time and limited resources that imposes a global ordering on events.

2.4 Comparing Models of Computation

A TSM process is, according to the definition given, a partial mapping from input signals to output signals. In order to consider more concrete mappings, we introduce some primitive concepts on which they are based.

System behavior, as we have previously stated, is composed of *functional behavior* and *communication behavior*, each represented by TSM processes. A process in turn is composed of *functional behavior* and *timing behavior*. Function is how things happen, or in the TSM, how events are related (how inputs are used to compute outputs) "around" a particular tag. Time is the order in which things happen, or in the TSM, the assignment of a tag to each event. The distinction between function and time is not this clean in every context. For example, a state in a finite state machine cannot be labeled as belonging exclusively to the function or time component of the behavior of the machine, but is rather based on the history of both. Nonetheless, the division between function and time, particularly at a primitive level, is useful in the conception and understanding of MOCs.

System operation can be viewed as a series of process computations, sometimes called *firings*. We will use function, time, computation (firing) to describe MOCs and their primitives.

In the sections that follow, we consider the fundamental concepts which are used to refine our processes. For functional processes, we will first consider state-less processes in which only inputs with a given tag concur to form outputs with the same tag. We then introduce the notion of state in the context of process networks. For communication processes, we then consider the primitives for concurrency and inter-process communication. Finally, we give the basic building blocks used to realize these concepts in practice. It is from these that today's most prevalent models of computation are built.

2.4.1 Process function

In the control-dominated arena, since the pioneering work of Shannon, Boolean functions have been used as a representation of both a system specification and its implementation in hardware (relay networks in Shannon's time, CMOS gates now). Several formally equivalent (but often with different levels of convenience in practice) representations for binary- and multi-valued boolean functions have been proposed, such as:

- truth table,
- Boolean network [57], which is a Directed Acyclic Graph (DAG), with a truth table associated with each node and edges carrying Boolean variable values,
- Binary Decision Diagram [15], that is also a DAG with one level of nodes for each input variable, and each node acting as a "multiplexer" between the function values associated with every variable values.

In the data-dominated arena, Data Flow actors play the role of processes and represent functions from simple state-less arithmetic operations such as addition and multiplication, to higher-level "combinational" transformations, such as Fast Fourier Transform.

2.4.2 Process State

Most models of computation include components with state, where behavior is given as a sequence of state transitions. State in a process network can always be simply implemented by means of feedback. An output and an input signal can be connected together, and thus provide a connection between process inputs and outputs beyond the tag barrier. However, we can also consider a notion of state *within* a process, since this can be useful in order to "hide" the implementation of the state information.

We can formalize this notion by considering a process F that is functional with respect to partition (S^i, S^o). Let us assume for the moment that F belongs to a timed system, in which tags are totally ordered[5]. Then for any tuple of signals $\mathbf{s}$, we can define $\mathbf{s}_{>t}$ to be a tuple of the (possibly empty) subset of the events in $\mathbf{s}$ with tags greater than t.

Two input signal tuples $\mathbf{r}, \mathbf{s} \in S^i$ are in relation E_t^F (denoted $(r^i, s^i) \in E_t^F$) if $\mathbf{r}_{>t} = \mathbf{s}_{>t}$ implies $F(\mathbf{r})_{>t} = F(\mathbf{s})_{>t}$. This definition intuitively means that process F cannot distinguish between the "histories" of $\mathbf{r}$ and $\mathbf{s}$ prior to time t. Thus, if the inputs are identical after time t, then the outputs will also be identical.

[5]A definition of state for untimed systems is also possible, but it is much more involved.

E_t^F is obviously an equivalence relation, partitioning the set of input signal tuples into equivalence classes for each t. Following a long tradition, we call these equivalence classes the *states* of F. In the hardware community, components with only one state for each t are called *combinational*, while components with more than one state for some t are called *sequential*. Note however that the term "sequential" is used in very different ways in other communities.

2.4.3 Concurrency and Communication

The sequential or combinational behavior just described is related to individual processes, and embedded systems will typically contain several coordinated concurrent processes. At the very least, such systems interact with an environment that evolves independently, at its own speed. It is also common to partition the overall model into tasks that also evolve more or less independently, occasionally (or frequently) interacting with one another. This interaction implies a need for coordinated communication.

Communication between processes can be *explicit* or *implicit*. Explicit communication implies forcing an order on the events, and this is typically realized by designating a *sender* process which informs one or more *receiver* processes about some part of its state. Implicit communication implies the sharing of tags (i.e., of a common time scale), which forces a common partial order of events, and a common notion of state. The problem with this form of communication is that it must be *physically* implemented via shared signals (e.g., a common reference clock), whose distribution may be difficult in practice.

Basic Time Time plays a larger role in embedded systems than in classical computation. In classical transformational systems, the correct result is the primary concern—when it arrives is less important (although *whether* it arrives, the termination question, *is* important). By contrast, embedded systems are usually real-time systems, where the time at which a computation takes place is very important.

As mentioned previously, different models of time become different order relations on the set of tags T in the tagged-signal model. Recall that in a *timed system* T is totally ordered, while in an *untimed system* T is only partially ordered. Implicit communication generally requires totally ordered tags, usually identified with physical time.

The tags in a *metric-time system* have the notion of a "distance" between them, much like physical time. Formally, there exists a partial function $d : T \times T \rightarrow \mathbf{R}$ mapping pairs of tags to real numbers such that $d(t_1, t_2) = 0 \Leftrightarrow t_1 = t_2$, $d(t_1, t_2) = d(t_2, t_1)$ and $d(t_1, t_2) + d(t_2, t_3) >= d(t_1, t_3)$.

Two events are *synchronous* if they have the same tag (the distance between them is 0). Two signals are synchronous if each event in one signal is synchronous with an event in the other signal and vice versa.

Treatment of Time in Systems A *discrete-event system* is a timed system where the tags in each signal are order-isomorphic with the natural numbers [42]. Intuitively, this means that any pair of ordered tags has a finite number of intervening tags. This is the basis of the underlying MOC of the Verilog and VHDL hardware description languages [62, 49].

A *synchronous system* is one in which every signal in the system is synchronous with every other signal in the system.

A *discrete-time system* is a synchronous discrete-event system.

An *asynchronous system* is a system in which no two events can have the same tag. If tags are totally ordered, the system is *asynchronous interleaved*, while if tags are partially ordered, the system is *asynchronous concurrent*. For asynchronous systems concurrency and interleaving are, to a large extent, interchangeable, since interleaving can be obtained from concurrency by embedding the partial order into a total order, and concurrency can be reconstructed from interleaving by identifying "untimed causality" [48].

Note that time is a continuous quantity. Hence real systems are asynchronous by nature. Synchronicity is only a (very) convenient abstraction, that may be expensive to implement due to the need to share tags, and hence, as discussed above, to share a reference "clock" signal.

Synchronous/reactive languages (see e.g. [28]) deserve special mention. They have an underlying synchronous model in which the set of tags in any behavior of the system implies a global "clock" for the system. However, to make this MOC synchronous in the sense of the TSM, we need to assume that every signal conceptually has an event at every tag. In some synchronous/reactive designs this may not be the case but if we define the events in the process to include a value denoting the absence of an event, then all synchronous/reactive models can be defined as synchronous in our framework. At each clock tick, each process maps input values to output values. Note that if we include the absent value for events, then discrete-event systems are also synchronous.

The main differences are:

- in the granularity of tags: intuitively, synchronous models should be used for systems in which there are fewer tags, and
- in the number of events that have the absent value at any tag: intuitively, synchronous models should be used for systems in which only few events have absent values.

Particular attention has to be devoted to events with values at the same tag and that have cyclic dependencies (*"combinational cycles"*). The existence of such dependencies implies that the input-output relation is described implicitly as the solution of an algebraic set of equations. This set of equations may have either a single solution for each input value, in which case the process is completely specified, no solution for some input value, in which case the process is functional but not completely specified, or multiple solutions for some input value, in which case the process is not functional. This is the source of endless problems in systems described by VHDL or Verilog and difficulties in synchronous/reactive languages. A possibility when facing a cyclic dependency is to leave the result unspecified, resulting in nondeterminacy or, worse, infinite computation within one tick according to the particular input values (VHDL, Verilog and some variants of StateCharts belong to this class [65]). A better approach is to use fixed-point semantics, where the behavior of the system is defined as a set of events that satisfy all processes [11]. Given this approach to the problem, there are procedures that can determine the existence of single or multiple fixed points in finite time, thus avoiding nasty inconsistencies and difficulties.

Asynchronous systems do not suffer from this problem since there cannot be cyclic dependencies at the same tag given that only one event can have a value at any given tag. Note that often asynchronous systems are confused with discrete-event systems and thus it is not infrequent to find assertions in the literature that asynchronous systems may have inconsistent or multiple solutions when indeed this is never the case!

Implementation of Concurrency and Communication Concurrency in physical implementations of systems implies a combination of *parallelism*, which employs physically distinct computational resources, and *interleaving*, which means sharing of a common physical resource. Mechanisms for achieving interleaving, generally called *schedulers*, vary widely, ranging from operating systems that manage context switches to fully-static interleaving in which multiple concurrent processes are converted (compiled) into a single process. We focus here on the mechanisms used to manage communication between concurrent processes.

Parallel physical systems naturally share a common notion of time, according to the laws of physics. The time at which an event in one subsystem occurs has a natural ordering relationship with the time at which an event occurs in another subsystem. Physically interleaved systems also share a natural common notion of time: one event happens before another and the time between them can be computed (of course, accuracy is an issue).

Logical systems, on the other hand, need a mechanism to explicitly share a notion of time (*communicate*). Consider two imperative programs interleaved on a single processor under the control of a time-sharing operating system. Interleaving creates a natural ordering between events in the two processes, but this ordering is generally unreliable, because it heavily depends on scheduling policy, system load and so on. Some explicit communication mechanism is required for the two programs to cooperate. One way of implementing this could be by forcing both to operate based on a global notion of time, which in turn forces a total order on events. This can be extremely expensive. In practice, this communication is done explicitly, where the total order is replaced by a partial order. Returning to the example of two processes running under a time-sharing operating system, we take precautions to ensure an ordering of two events only if the ordering of these two events matters. We can do this by communicating through common signals, and forcing one process to wait for a signal from the other, which forces the scheduler to interleave the processes in a particular way.

A variety of mechanisms for managing the order of events, and hence for communicating information between processes, exists. We will now examine and classify them according to the tagged-signal model, by using "special-purpose" processes to model communication. Using processes to model communication (rather than considering it as "primitives" of the tagged-signal model) makes it easier to compare different MOCs, and also allows one to consider *refining* these communication processes when going from specification to implementation [56].

Recall that the *communication* primitive in the TSM is the event, which is a two-component entity whose value is related to function and whose tag is related to time. That is, communication is implemented by two operations:

1. the transfer of values between processes (function; TSM event value),
2. the determination of the relationship in time between two processes (time; TSM event tag).

Unfortunately, often the term "communication" (or data transfer) is used for the former, and the term "synchronization" is used for the latter. We feel, however, that the two are intrinsically connected in embedded systems: both tag and value carry information *about a* communication. *Thus, communication and synchronization, as mentioned before, are terms which cannot really be distinguished in this sense.*

2.5 Basic communication primitives

In this section, we define some of the communication primitives that have been described in the literature, following the classification developed in the previous sections.

Unsynchronized In an unsynchronized communication, a producer of information and a consumer of the information are not coordinated. There is some connection between them (e.g., a buffer) but there is no guarantee that the consumer reads "valid" information produced by the producer, and no guarantee that the producer will not overwrite previously produced data before the consumer reads the data. In the tagged-signal model, the repository for the data is modeled as a process, and the reading and writing actions are modeled as events without any enforced ordering of their tags.

Read-modify-write Commonly used for accessing shared data structures in software, this strategy locks a data structure during a data access (read, write, or read-modify-write), preventing any other accesses. In other words, the actions of reading, modifying, and writing are atomic (indivisible, and thus uninterruptible). In the tagged-signal model, the repository for the data is modeled as a process where events associated with this process are totally ordered (resulting in a partially ordered model at the global level). The read-modify-write action is modeled as a single event.

Unbounded FIFO buffered This is a point-to-point communication strategy, where a producer generates (writes) a sequence of data tokens and a consumer consumes (reads) these tokens, but only after they have been generated (i.e., only if they are valid). In the tagged-signal model, this is a simple connection where the signal on the connection is constrained to have totally ordered tags. The tags model the ordering imposed by the FIFO model. If the consumer process has unbounded FIFOs on all inputs, then all inputs have a total order imposed upon them by this communication choice. This model captures essential properties of both Kahn process networks and dataflow [35].

Bounded FIFO buffered In this case (we discuss only the point-to-point case for the sake of simplicity), the data repository is modeled as a process that imposes ordering constraints on its inputs (which come from the producer) and the outputs (which go to the consumer). Each of the input and output signals are internally totally ordered, while their combination is partially ordered. The simplest case is where the size of the buffer is one, in which case the input and output events must be perfectly interleaved (i.e., that each output event lies between two input events). Larger buffers impose a maximum difference (often called *synchronic distance* [51]) between the number of input or output events occurring in succession.

Note that some implementations of this communication mechanism may not really block the writing process when the buffer is full, thus requiring some higher level of flow control to ensure that this never happens, or that it does not cause any harm.

Petri net places This is a multi-partner communication strategy, where several producers generate tokens and several consumers consume these tokens [51]. In the tagged-signal model, this is modeled as a process that keeps track of the tags of its input (from producers) and output (to consumers) signals. As in the previous case, each signal has totally ordered events, and the process makes sure that the number of input events is always greater than or equal to that of output events.

Rendezvous In the simplest form of rendezvous, which is embodied in the underlying MOC of the Occam and Lotos [64] languages, a single writing process and a single reading process must simultaneously be at the point in their control flow where the write and the read occur. It is a

	Transmitters	Receivers	Buffer Size	Blocking Reads	Blocking Writes	Single Reads
Unsynchronized	many	many	one	no	no	no
Read-Modify-Write	many	many	one	yes	yes	no
Unbounded FIFO	one	one	unbounded	yes	no	yes
Bounded FIFO	one	one	bounded	maybe	maybe	yes
Petri net place	many	many	unbounded	no	no	yes
Single Rendezvous	one	one	one	yes	yes	yes
Multiple Rendezvous	many	many	one	no	no	yes

Table 1: A comparison of concurrency and communication schemes.

convenient communication mechanism, because it has the semantics of a single assignment, in which the writer provides the right-hand side, and the reader provides the left-hand side. In the tagged-signal model, this is imposed by events with identical tags [42]. Lotos offers, in addition, multiple rendezvous, in which one among multiple possible communications is *non-deterministically* selected. Multiple rendezvous is more flexible than single rendezvous, because it allows the designer to specify more easily several "expected" communication ports at any given time, but it is very difficult and expensive to implement correctly.

Of course, various combinations of the above models are possible. For example, in a model that partially uses the unsynchronized communication scheme, a consumer of data may be required to wait until the first time a producer produces data, after which the communication is unsynchronized.

The essential features of the concurrency and communication styles described above are presented in Table 1. These are distinguished by the number of transmitters and receivers (e.g., broadcast versus point-to-point communication), the size of the communication buffer, whether the transmitting or receiving process may continue after an unsuccessful communication attempt (blocking reads and writes), and whether the result of each write can be read at most once (single reads). Note that, strictly speaking, the blocking/nonblocking read and write aspects are part of the "functional" processes, and not of the "communication" processes. However, these communication schemes also specify that aspect, and hence we chose to include in the table. A "maybe" entry means that MOCs considering both the "yes" and "no" answer have been proposed in the literature.

3 Common Models of Computation

We are now ready to use the scheme developed in the previous Section to classify and analyze several models of computation that have been used to describe embedded systems. We will consider issues such as ease of modeling, efficiency of analysis (simulation or formal verification), automated synthesizability, and optimization space versus over-specification.

We assume a background knowledge of basic, non-concurrent MOCs such as Finite Automata, Turing Machines, and Algebraic State Machines, and we focus on the timing, concurrency and communication aspects instead.

3.1 Discrete-Event

Time is an integral part of a discrete-event model of computation. Events usually carry a totally-ordered time stamp indicating the time at which the event occurs. A DE simulator usually maintains a global event queue that sorts events by time stamp.

Digital hardware is often simulated using a discrete-event approach. The Verilog language [62], for example, was designed as an input language for a discrete-event simulator. The VHDL language [49] also has an underlying discrete-event model of computation.

Discrete-event modeling can be expensive—sorting time stamps can be time-consuming. Moreover, ironically, although discrete-event is ideally suited to modeling distributed systems, it is very challenging to build a distributed discrete-event simulator. The global ordering of events requires tight coordination between parts of the simulation, rendering distributed execution difficult.

Discrete-event simulation is most efficient for large systems with large, frequently idle or autonomously operating sections. Under discrete-event simulation, only the changes in the system need to be processed, rather than the whole system. As the activity of a system increases, the discrete-event paradigm becomes less efficient because of the overhead inherent in processing time stamps.

Simultaneous events, especially those arising from zero-delay feedback loops, present a challenge for discrete-event models of computation. In such a situation, events may need to be ordered, but are not.

Consider the discrete-event system shown in Figure 3. Process B has zero delay, meaning that its output has the same time stamp as its input. If process A produces events with the same time stamp on each output, there is ambiguity about whether B or C should be invoked first, as shown in Figure 3(a).

Suppose B is invoked first, as shown in Figure 3(b). Now, depending on the simulator, C might be invoked once, observing both input events in one invocation, or it might be invoked twice, processing the events one at a time. In the latter case, there is no clear way to determine which event should be processed first.

The problem could be solved by requiring the user to provide a delay for each process, but this is not convenient in general. Hence various simulators have resorted to various heuristic techniques:

- The VHDL simulation semantics [49] uses a *synchronous* model (with unit

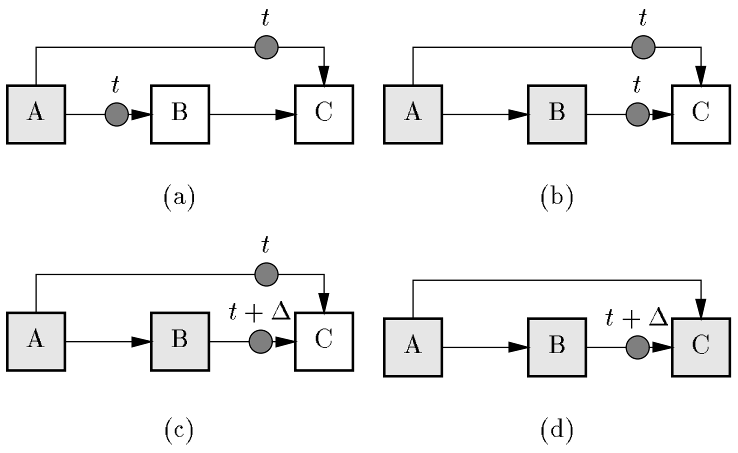


Figure 3: Simultaneous events in a discrete-event system. (a) Process A produces events with the same time stamp. Should B or C be fired next? (b) Zero-delay process B has fired. How many times should C be fired? (c) Delta-delay process B has fired; C will consume A's output next. (d) C has fired once; it will fire again to consume B's output.

delay, called "delta step") in order to provide a two-level structure of time and thus solve non-determinism within a given "real time" instant. Each instant of time (level 1) is broken into (a potentially infinite number of) totally ordered delta steps (level 2). A "zero-delay" process in this model actually has delta steps, or ordered progress towards a solution though no real time elapses. For example, if Process B contains a delta step between input and output, firing A followed by B would result in the situation in Figure 3(c). The next firing of C will see the event from A only; the firing after that will see the (delay-ordered) event from B.

- The Discrete Event domain in Ptolemy [17] uses a synchronous model, but with mostly zero delay and only enough delta steps to eliminate all zero-delay cycles.
- The BONES simulator by Cadence uses an *asynchronous* model.

Adding a feedback loop from Process C to A in Figure 3 would create a problem if events circulate through the loop without any increment in time stamp. The same problem occurs in synchronous languages, where such loops are called causality loops. No precedence analysis can resolve the ambiguity. In synchronous languages, the compiler may simply fail to compile such a program.

Discrete-event simulators attempt to identify such cases and report them to the user.

We wish to stress that delta steps do not have a meaning of time (though they are often called delta "delay"). They are just a clever mechanism to implement a fixed point computation used to compute the behavior of the system at a point in time. Fixed point iteration can also be used in the synchronous/reactive model to define its semantics and make it determinate. Hence "delta steps" can also be thought of as an "iteration index". Moreover, VHDL uses an event model that is not monotonic, and hence the fixed point may never be reached, as discussed above. On the other hand, synchronous language use a ternary logic model, in which fixed point convergence in ensured in a finite number of steps [16].

The reason why DE is a popular MOC in practice is that it has been implemented efficiently in a number of event-driven simulators, and it is quite convenient to evaluate the performance of very large and complex systems. By imposing little restriction on the modeling style, it makes simulation simple and synthesis as well as formal verification hard.

3.2 Dataflow Process Networks

In dataflow, a program is specified by a directed graph where the nodes (called *actors*) represent computations and the arcs represent totally ordered sequences (called *streams*) of events (called *tokens*). In figure 4(a), the large circles represent actors, the small circle represents a token and the lines represent streams. The graphs are often represented visually and are typically hierarchical, in that a node in a graph may represent another directed graph. The nodes in the graph can be either language primitives or subprograms specified in another language, such as C or FORTRAN. In the latter case, we are actually mixing two models of computation, where dataflow serves as the coordination language for subprograms written in an imperative host language.

Dataflow is a special case of Kahn process networks [35, 41]. In a Kahn process network, communication is by unbounded FIFO buffering, and processes are constrained to be continuous mappings from input streams to output streams. "Continuous" in this usage is a topological property that ensures that the program is determinate [35]. Intuitively, it implies a form of causality without time; specifically, a process can use partial information about its input streams to produce partial information about its output streams. Adding more tokens to the input stream will never result in having to change or remove tokens on the output stream that have already been produced. One way to ensure continuity is with blocking reads, where any access to an input stream results in suspension of the process if there are no tokens. One consequence of blocking reads is that a process cannot test an input channel for the availability of data and then branch conditionally to a point where it will read a different input.

In dataflow, each process is decomposed into a sequence of *firings*, indivisible quanta of computation. Each firing consumes and produces tokens. Dividing processes into firings avoids the multi-tasking overhead of context switching in direct implementations of Kahn process networks. In fact, in many of the signal processing environments, a major objective is to statically (at compile time) schedule the actor firings, achieving an interleaved implementation of the concurrent model of computation. The firings are organized into a list (for one processor) or set of lists (for multiple processors). Figure 4(a) shows a dataflow graph, and Figure 4(b) shows a single processor schedule for it. This schedule is a list of firings that can be repeated indefinitely. One cycle through the schedule should return the graph to its original state (here, state is defined as the number of tokens on each arc). This is not always possible, but when it is, considerable simplification results [12]. In many existing environments, what happens within a firing can only be specified in a host language with imperative semantics, such as C or C++.

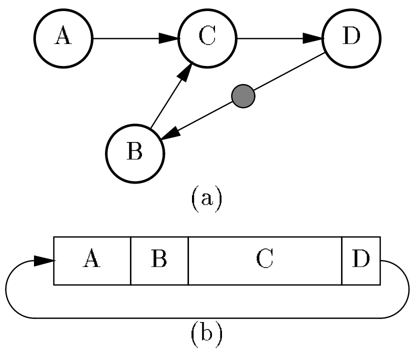


Figure 4: (a) A dataflow process network (b) A single-processor static schedule for it

A useful formal device is to constrain the operation of a firing to be functional, i.e., a simple, stateless mapping from input values to output values. Note, however, that this does not constrain the process to be stateless, since it can maintain state in a self-loop: an output that is connected back to one of its inputs. An initial token on this self-loop provides the initial value for the state.

Many possibilities have been explored for precise semantics of dataflow coordination languages, including Karp and Miller's computation graphs [37], Lee and Messerschmitt's synchronous dataflow graphs [40], Lauwereins *et al.*'s cyclo-static dataflow model [39, 13], Kaplan *et al.*'s Processing Graph Method (PGM) [36], Granular Lucid [34], and others [1, 20, 22, 60]. Many of these limit expressiveness in exchange for formal properties (e.g., provable liveness and bounded memory).

Synchronous dataflow (SDF) and cyclo-static dataflow require processes to consume and produce a fixed number of tokens for each firing. Both have the useful property that a finite static schedule can always be found that will return the graph to its original state. This admits extremely efficient implementations [12]. For more general dataflow models, it is undecidable whether such a schedule exists [18].

A looser model of dataflow is the tagged-token model, in which the partial order of tokens is explicitly carried with the tokens [3]. A significant advantage of this model is that while it logically preserves the FIFO semantics of the channels, it permits out-of-order execution.

Some examples of graphical dataflow programming environments intended for signal processing (including image processing) are Khoros [52], and Ptolemy [17].

3.3 Petri nets

Petri nets [50, 53] are, in their basic form, an infinite state model (just like dataflow) for which, however, most properties are decidable in finite time and memory. They are interesting as an uninterpreted model for several very different classes of problems, including some relevant to embedded system design (e.g., process control, asynchronous communication, and scheduling).

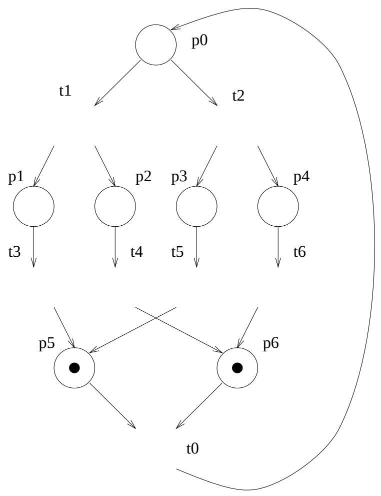
Moreover, a large user community has developed an impressive body of theoretical results and practical design aids and methods based on Petri nets. In particular, partial order-based verification methods (e.g. [63], [27], [45]) are one possible answer to the state explosion problem plaguing Finite State Machine-based verification techniques.

A Petri net (PN) is a directed bipartite graph $N = \{P, T, F\}$. P is a set of places holding the distributed state (via tokens) of the system. T is a set of transitions, denoting the activity of the system. $F \subseteq P \times T \cup T \times P$ is the flow relation, from places to transitions and vice-versa. Nodes linked by F are said to be in a predecessor/successor relationship.

Transitions are often labeled with statements in a host language, just as in the case of DF actors. The state of the PN is the *marking* of the places, that is a non-negative integer valuation ("token assignment") of each place. The dynamic evolution of the PN is determined by the firing process of transitions. A transition may fire whenever all its predecessor places are marked, and if it fires, it decrements the marking (removes a token) of each predecessor and increments the marking of each successor (adds a token).

PNs are interesting in general, and in particular in embedded system design, because they are a very general model of control, potentially with infinite state, yet very powerful analysis techniques, both exact and approximate, have been defined for them.

In particular, the firing rule of a PN bears a strong connection with linear algebra. If we represent the graph of the flow relation (given arbitrary orderings



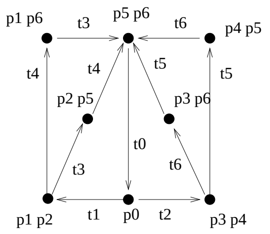


Figure 5: Example of Free Choice Petri net and its Reachability Graph

of the sets T and P) as an incidence matrix I, and if we represent the current marking as an integer vector M, we can model the effect of a sequence of transitions σ starting from M as follows. Let us denote by f^σ the "firing vector" of σ, that is a vector whose i-th position contains the number of times the i-th transition appears in σ. The marking M' reached after σ is given by

$$M' = If^\sigma + M$$

For example, consider the PN in Figure 5.(a), whose set of reachable markings is shown in Figure 5.(b). Its incidence matrix (one row for each place and one column for each transition) is:

$$\begin{vmatrix} 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 1 \end{vmatrix}$$

The first line corresponds to place $p0$, and has a 1 in position 0, because $t0$ adds 1 token to place $p0$, and -1 in positions 1 and 2 because $t1$ and $t2$ remove one token from it.

Consider now firing sequence $\sigma = t0, t1, t3$ whose firing vector (transposed) is $f^\sigma = |1101000|^t$. The marking M' reached from the initial marking $M =$

$|0000011|^t$ after firing σ is:

$$\left|\begin{array}{c}0\\0\\1\\0\\0\\1\\0\end{array}\right| = \left|\begin{array}{ccccccc}1 & -1 & -1 & 0 & 0 & 0 & 0\\0 & 1 & 0 & -1 & 0 & 0 & 0\\0 & 1 & 0 & 0 & -1 & 0 & 0\\0 & 0 & 1 & 0 & 0 & -1 & 0\\0 & 0 & 1 & 0 & 0 & 0 & -1\\-1 & 0 & 0 & 1 & 0 & 1 & 0\\-1 & 0 & 0 & 0 & 1 & 0 & 1\end{array}\right| \left|\begin{array}{c}1\\1\\0\\1\\0\\0\\0\end{array}\right| + \left|\begin{array}{c}0\\0\\0\\0\\0\\1\\1\end{array}\right|$$

that corresponds to $p2, p5$ being marked, as expected.

As another example, consider the initial marking with two tokens each in p_5, p_6 and the firing sequence $\sigma = t0, t0, t1, t3$. In that case,
$M = |0000022|^t$,
$f^\sigma = |2110000|^t$, and
$M' = |0111100|^t$.

This equation provides an interesting characterization of sequences of transitions that, when fired from a marking M, return the net to same M. These sequences, also called *T-invariants*, must be solutions to

$$0 = If^\sigma$$

This is only a *necessary* condition, of course, since the sequences must also be fireable from M (some intermediate step may yield a negative marking), but it is useful, e.g., when proving liveness conditions (e.g., showing that some transition can fire infinitely often) or schedulability properties [47].

For example, in Figure 5 firing sequence $t0, t1, t3, t4$ is a T-invariant $|1101100|^t$ that happens to be fireable from the initial marking. The reader can check that this invariant is indeed a solution of the equation shown above.

By duality (a very useful concept in Petri nets, based on exchanging the roles of places and transitions), one can also identify sets of places whose total cumulative marking cannot be changed by any firing sequence of the net. These sets, also called *S-invariants* can be used to establish the *unreachability* of a given marking, if it cannot be expressed as a linear combination of a basis of S-invariants [25]. Hence they can be very useful in proving (but not disproving) safety properties (e.g., the fact that some "dangerous" marking cannot be reached).

Invariant-based techniques become *necessary and sufficient* for a restricted (but expressive) class of PNs called *free-choice nets* [24], in which a multi-successor place must be the only predecessor of its successors. The net of Figure 5.(a) is a free-choice net, since the only multi-successor place ($p0$) has only single-predecessor successors ($t1, t2$).

In addition, reachability-based techniques for analysis, based on building the complete state space (or deciding in finite time that it is actually infinite), can also be used to prove properties of a given PN.

The basic PN model is interesting but somewhat limited in expressive power[6]. For this reasons, people have extended it in various ways, such as adding *colors* to tokens. Colored PNs are similar to Dataflow networks (with places playing the role of FIFOs and transitions playing the role of actors), but allow multiple predecessors and successors for a place/FIFO. In this way, they lose one of the most interesting properties of DF networks, *determinacy*, and of course gain something in terms of compactness and expressiveness[7].

Time can also play an explicit role in PNs. Time has been associated with transitions and places, in various combinations and forms ([55]). Generally speaking, time is associated with tokens, that carry a time stamp, and time stamps determine when transitions may fire (and thus create new tokens with new time stamps). The problem with timed PNs is, as usual with real-time MOCs, that they suffer from a particularly serious combinatorial explosion problem when reducing the originally infinite timed state space to a finite set of equivalence classes, as discussed more in detail in Section 3.8.

3.4 Synchronous/Reactive

In a synchronous model of computation, all events are synchronous, i.e., all signals have events with identical tags. The tags are totally ordered, and globally available. Unlike the discrete-event model, all signals have events at all clock ticks, simplifying the simulator by requiring no sorting. Simulators that exploit this simplification are called cycle-based or cycle-driven simulators. Processing all events at a given clock tick constitutes a cycle. Within a cycle, the order in which events are processed may be determined by data precedences, which define the delta steps. These precedences are not allowed to be cyclic, and typically impose a partial order (leaving some arbitrary ordering decisions to the scheduler). Cycle-based models are excellent for clocked synchronous circuits, and have also been applied successfully at the system level in certain signal processing applications.

A cycle-based model is inefficient for modeling systems where events do not occur at the same rate in all signals. While conceptually such systems can be modeled (using, for example, special tokens to indicate the absence of an event), the cost of processing such tokens is considerable. Fortunately, the cycle-based model is easily generalized to multirate systems. In this case, every nth event in one signal aligns with the events in another.

A multirate cycle-based model is still somewhat limited. It is an excellent model for synchronous signal processing systems where sample rates are related by constant rational multiples, but in situations where the alignment of events in different signals is irregular, it can be inefficient.

[6]It is more powerful than regular grammars, is incomparable with context-free grammars, and is less powerful than Turing machines.

[7]The formal power is the same, being that of Turing machines for both general CPN and general DF.

The more general synchronous/reactive model is embodied in the so-called synchronous languages [8]. Esterel [14] is a textual imperative language with sequential and concurrent statements that describe hierarchically-arranged processes. Lustre [29] is a textual declarative language with a dataflow flavor and a mechanism for multirate clocking. Signal [9] is a textual relational language, also with a dataflow flavor and a more powerful clocking system. Argos [44], a derivative of Harel's Statecharts [30], is a graphical language for describing hierarchical finite state machines (described more in detail in the next section). Halbwachs [28] gives a good summary of this group of languages.

The synchronous/reactive languages describe systems as a set of concurrently-executing synchronized modules. These modules communicate through signals that are either present or absent in each clock tick. The presence of a signal is called an event, and often carries a value, such as an integer.

Most of these languages are static in the sense that they cannot request additional storage nor create additional processes while running. This makes them well-suited for bounded and speed-critical embedded applications, since their behavior can be extensively analyzed at compile time. This static property makes a synchronous program finite-state, greatly facilitating formal verification.

Verifying that a synchronous program is causal (non-contradictory and deterministic) is a fundamental challenge with these languages. Since computation in these languages is delay-free and arbitrary interconnection of processes is possible, it is possible to specify a program that has either no interpretation (a contradiction where there is no consistent value for some signal) or multiple interpretations (some signal has more than one consistent value). Both situations are undesirable, and usually indicate a design error. A conservative approach that checks for causality problems structurally flags an unacceptably large number of programs as incorrect because most will manifest themselves only in unreachable program states. The alternative, to check for a causality problem in any reachable state, can be expensive since it requires an exhaustive check of the state space of the program.

In addition to the ability to translate these languages into finite-state descriptions, it is possible to compile these languages directly into hardware. Techniques for translating both Esterel [10] and Lustre [54] into hardware have been proposed. The result is a logic network consisting of gates and flip-flops that can be optimized using traditional logic synthesis tools. To execute such a system in software, the resulting network is simply simulated. The technique is also the basis to perform more efficiently causality checks, by means of implicit state space traversal techniques [58].

3.5 Communicating Synchronous Finite State Machines

Finite State Machines (FSMs) are an attractive model for embedded systems. The amount of memory required by such a model is always decidable, and is often an explicit part of its specification. Halting and performance questions are always decidable since each state can, in theory, be examined in finite time. In practice, however, this may be prohibitively expensive, and thus formal verification techniques based on interacting FSMs require various forms of (non-trivial and non-automatable) abstraction in order to be kept manageable [38, 45].

A traditional FSM consists of:

- a set of input symbols (the Cartesian product of the sets of values of the input signals),
- a set of output symbols (the Cartesian product of the sets of values of the output signals),
- a finite set of states with a distinguished initial state,
- an output function mapping input symbols and states to output symbols, and
- a next-state function mapping input symbols and states to (next) states.

The input to such a machine is a sequence of input symbols, and the output is a sequence of output symbols. The model is *synchronous* (i.e., all signals have the same tags), and hence input and output symbols are well defined (they correspond to the set of events with a given tag). It is also semantically identical to that of previous section. However, there are enough syntactic differences to warrant a separate treatment (see [28, 11] for a discussion of possible mappings between the two).

Traditional FSMs are good for modeling sequential behavior, but are problematic for modeling system with concurrency or large memories, because of the state explosion problem. Every global state of a concurrent system must be represented individually, even when interleaving of independent actions may give rise to an exponential number of states. Similarly, a memory has as many states as the number of values that can be stored at each location *raised to the power* of the number of locations. The number of states alone is not always a good indication of complexity, but it often has a strong correlation.

Harel advocated the use of three major mechanisms that reduce the size (and hence the visual complexity) of finite automata for modeling practical systems [31]. The first one is hierarchy, in which a state can represent an enclosed state machine. That is, being in a particular state a has the interpretation that the state machine enclosed by a is active. Equivalently, being in state a means that the machine is in one of the states enclosed by a. Under the latter interpretation, the states of a are called “or states.” Or states can exponentially reduce the complexity (the number of states) required to represent a system. They compactly describe the notion of *preemption* (a high-priority event suspending or “killing” a lower priority task), that is fundamental in embedded control applications.

The second mechanism is concurrency. Two or more state machines are viewed as being simultaneously active. Since the system is in one state of each parallel state machine simultaneously, these are sometimes called "and states." They also provide a potential exponential reduction in the size of the system representation.

The third mechanism is non-determinism. While often non-determinism is simply the result of an imprecise (maybe erroneous) specification, it can be an extremely powerful mechanism to reduce the complexity of a system model by *abstraction*. This abstraction can either be due to the fact that the exact functionality must still be defined, or that it is irrelevant to the properties currently considered of interest. E.g., during verification of a given system component, other components can be modeled as non-deterministic entities to compactly constrain the overall behavior. A system component can also be described non-deterministically to permit some optimization during the implementation phase. Non-determinism can also provide an exponential reduction in complexity. Note that non-determinism can be divided into and-non-determinism and or-non-determinism. In the first, the resolution of the non-determinism executes all possibilities, while in the second, resolution chooses just one. And-non-determinism is equivalent to hierarchy.

These three mechanisms have been shown in [26] to cooperate synergistically and orthogonally, to provide a potential triple exponential reduction in the size of the representation with respect to a single, flat deterministic FSM[8].

Harel's Statecharts model uses a synchronous concurrency model (also called synchronous composition). The set of tags is a totally ordered countable set that denotes a global "clock" for the system. The events on signals are either produced by state transitions or inputs. Events at a tick of the clock can trigger state transitions in other parallel state machines at the same clock. Unfortunately, Harel left open some questions about the semantics of causality loops and chains of instantaneous (same tick) events, triggering a flurry of activity in the community that has resulted in at least twenty variants of Statecharts [65].

A model that is closely related to FSMs is Finite Automata. FAs emphasize the acceptance or rejection of a sequence of inputs rather than the sequence of output symbols produced in response to a sequence of input symbols. Most notions, such as composition and so on, can be naturally extended from one model to the other. FAs without accepting conditions are also called Labeled Transition Systems in the literature.

[8]The exact claim in [26] was that and-non-determinism (in which all non-deterministic choices must be successful), rather than hierarchical states, was the third source of exponential reduction together with "or" type non-determinism and concurrency. Hierarchical states, on the other hand, were shown in that paper to be able to simulate "and" non-determinism with only a polynomial increase in size.

3.6 Process algebrae

Synchronous FSMs, as described above, have a clear and deterministic composition mechanism that makes them relatively easy to understand, synthesize and verify. Of course, there is also a significant drawback: deciding when composition is well-defined (loosely speaking, there are no combinational loops) has a high computational complexity.

Moreover, for many applications, the tight coordination implied by the synchronous model is inappropriate. In particular, it is very difficult to keep a tight synchronization between heterogeneous components of an embedded system, since the pace of a synchronous system is dictated by its slowest component. In response to this, a number of more loosely coupled asynchronous FSM models have evolved, including CSP [33], CCS [46], behavioral FSMs [61], SDL process networks [61], and codesign FSMs [21].

In this section we focus on process algebraic models that constitute the semantical foundation of the Occam and Lotos [64] languages[9]: Communicating Sequential Processes [33] and the related Calculus of Communicating Systems [46]. In the following we discuss only the control aspect of CSP and CCS, and ignore the fact that their processes can also manipulate data via assignments, tests and so on. We also do not consider recursion, that can be defined in the process algebra but has limited interest (except for tail recursion, that defines looping) in the context of embedded systems.

The behavior of each process is modeled by a Labeled Transition System (only finite LTSs are of interest in embedded system design, for obvious reasons). Arcs in the transition system are labeled with signal names, and the state transition activity imposes a total order on the signals of each process. Communication is based on rendezvous. That is, two LTSs may share a signal, thus imposing that all the events of that signal must occur in both processes ("at the same time", if we interpret tags as time). Finally, process algebrae generally imply a completely *interleaved* view of concurrent actions, meaning that *no two events may have the same tag*. Concurrent (i.e., independent) events occur in all possible interleaving in the LTS.

No two events may have have the same tag, and hence process algebrae are an inherently asynchronous model. Note that a single LTS is an *interleaved* asynchronous model, while multiple LTSs communicating via rendezvous (and, equivalently, Petri nets in which at most one token can reside in each place in each reachable marking) are a *partially ordered* asynchronous model. As mentioned above, the rich theory of *regions* [48, 23] can be used to freely move between the two classes of models.

The result of process *composition* using this communication mechanism is another LTS, thus resulting in a hierarchical compositional model[10]. Composi-

[9]Ada also uses rendezvous, although the implementation is stylistically quite different, using remote procedure calls rather than more elementary communication primitives.

[10]Compositionality means that two or more communicating processes can be viewed as a

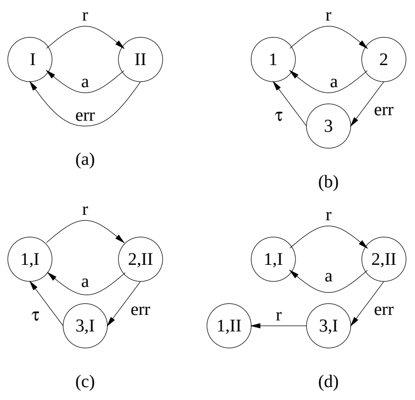


Figure 6: Example of Labeled Transition Systems and rendezvous communication

tionality is very important for proving properties of the system in a hierarchical fashion. This property is also true of communicating *synchronous* Finite State Machines, but not of dataflow networks (i.e., a dataflow network is different from an actor).

Let us consider a simple case of an interface with error detection. The LTSs specifying the protocols followed by the two partners are shown in Figure 6.(a-b).

1. The sender has two states, first sending a *request* on signal R, then waiting for either an *acknowledgment* of correct reception on signal A, or an *error* indication on signal E.

2. The receiver has a similar behavior, but in case of error, it requires one *internal action* (labeled τ) to resynchronize, and hence it has a third state.

The composed LTS using the rendezvous mechanism is shown in Figure 6.(c). Note how the state space of the composition is the product of the two state spaces, and the two LTSs synchronize on common edge labels.

For the sake of comparison, Figure 6.(d) shows the *synchronous* composition of the same two LTSs. Note how in case of error, the receiver waits for one clock tick, and hence becomes de-synchronized with the transmitter, thus leading to a deadlock[11].

Rendezvous-based models of computation are sometimes called *synchronous* in the literature. However, by the definition we have given, they are not syn-

single process, that can in turn be used as a unit and composed with others.

[11] Of course, the fact that synchronous composition deadlocks while asynchronous composition does not is just a coincidence. It is easy to construct an example where the converse can happen.

chronous. Events are partially ordered, not totally ordered, with rendezvous points imposing the partial ordering constraints.

3.7 SDL process networks

SDL [61] is a language for specification, simulation and design of telecommunication protocols. Its underlying semantical model[12] is based on a process network. Each process is an FSM, and communication is via one unbounded FIFO queue per process. If we ignore the ability of a process to manipulate its input queue, the MOC is roughly equivalent to DE, with the restriction that the FSM can only read one event at a time.

SDL networks have a basic implementability problem, since both the size of the queues and the topology of the network can change at run time. (Processes can be created on the fly, and signals can be routed dynamically based on process identifiers.) Hence they either require a software implementation based on a Real-Time Operating System with dynamic memory allocation and task instantiation, or require the designer to pre-size queues and pre-instantiate all processes.

3.8 Timed Automata

Synchronous and asynchronous Finite State Machines cannot reason easily about time, since in the best case (the synchronous one) time must be represented by counting clock ticks. This may cause a state explosion, and has been proven to be an inadequate abstraction of reality unless special care is taken [19].

For this reason, Alur and Dill [2] have proposed explicitly introducing time as a continuous quantity in the Timed Automata MOC. A Timed Automaton (TA) is a special case of hybrid systems, which are described in the next section. It is sufficiently restricted, so that most properties become decidable. A TA is a Finite Automaton (FA) plus a set of *clocks*. The state of the TA is the state of the FA together with a real valuation of the clocks. A transition of the TA is labeled with a symbol (from the FA alphabet) and a Boolean formula over atomic propositions comparing clocks with integers. The transition can also reset some clocks to zero.

While the state space of a TA is clearly infinite, a key result by Alur and Dill shows that it admits a finite state representation, by means of a partition into equivalence classes. Basically, [2] showed that the exact value of a clock does not matter after it grows beyond the largest constant with which it can be compared in any transition label. This imposes an equivalence relation on those portions of the state space that grow towards infinity. Moreover, since comparisons involve only integers, one can also partition the remaining part of

[12] As usual, we focus on the control and communication aspects over the data computations, which are commonly specified with an imperative host language, in addition to a more formal and less practical treatment based on Abstract Data Types.

the space into a *finite set of equivalence classes* (called regions), that admit a normal form representation (computed via an all-pair shortest path algorithm).

This is a very significant contribution, however it has shown only limited practical applicability so far because the state explosion problem is even more severe than in the communicating FSM case. Good generally applicable abstraction techniques are only beginning to be developed for TAs.

3.9 Hybrid systems

A hybrid system is a Finite Automaton in which each state is associated with a set of differential equations, and transitions occur when inequalities over the continuous variables of the differential equations are satisfied. Hybrid systems are a powerful mechanism for modeling non-linear dynamic systems, and thus are becoming an essential tool in control theory. However, they are clearly Turing-equivalent, and hence too powerful, in almost all of their incarnations, with the notable exceptions of Timed Automata described above. It is likely that they will play an ever increasing role in embedded system design due to the growing need to raise the level of abstraction, but it is difficult to give them a complete and fair treatment in this brief overview, and we refer the interested reader to [32].

In the TSM, there are two possible views of hybrid systems (and hence of TAs).

1. A hybrid system (FA plus differential equations) can be modeled as a single TSM process. This provides an easy mechanism for composing hybrid systems. Signal tags in this case are order-isomorphic with the real numbers, but tags in which a transition of the automaton can occur can be only discrete.
2. A hybrid system can be modeled as a set of TSM processes. In this case we have two components for each hybrid system:
   - one process, whose signal tags can only be discrete, represents the automaton, and *multiplexes* the hybrid system outputs between
   - a set of processes, each behaving as a set of differential equations.

4 Codesign Finite State Machines

Codesign Finite State Machines (CFSMs) are the underlying MOC of the POLIS embedded system design environment [21, 6]. We describe them at length because, as we will argue later, they combine interesting aspects from several other MOCs, while preserving both formality and efficiency in implementation. As we pointed out above, one of the most important properties of an MOC is synchronicity or asynchronicity. We wish to summarize our views on this topic to motivate the introduction of yet another MOC.

4.1 Synchrony and asynchrony

Synchrony and asynchrony represent two fundamentally different views of time. That is, synchrony uses the notions of zero and infinite time, while asynchrony uses non-zero finite (and typically bounded) time. Both synchrony and asynchrony have appeared a number of times in our previous descriptions of various models of computation. In this section, we summarize our previous presentations of synchrony and asynchrony, and consider the differences in the behaviors produced under each model.

As usual, we consider a system of processes interacting through events.

4.1.1 Synchrony

Basic operation: At each clock tick (i.e., tag of its signals), each module reads inputs, computes, and produces outputs simultaneously. That is, all the synchronous events (both inputs and outputs) happen simultaneously, implying zero-delay calculations. In between clock ticks, an "infinite" time passes. Of course, no calculations happen in zero time in practice, nor does one wait an infinite amount of time between ticks (it is normally finite but unspecified). In practice, the computation times are much smaller than the clock rate, and thus can be considered to be zero with respect to the reaction times of the environment. The very desirable feature of designs implemented as synchronous systems with no cyclic dependencies among values of events with the same tags, is that the behavior of the implementation is not dependent upon the timing of the signals, thus simplifying tremendously the verification task.

Triggering and Ordering: All modules are triggered to compute at every clock tick. At a tick, there is no ordering of reading of inputs, computations, or writing of outputs. However, an ordering can be imposed in addition with the concept of delta steps (delays). A delta step (delay), as previously mentioned, is the (zero) time that passes between events at the same clock tick and which serves simply to order the events.

System solution: The system solution is the output reaction to a set of inputs. A well-designed synchronous system will have a unique solution (assignment to all signals) at each clock tick, though the corresponding models of computation, as well as many synchronous languages or specification methods, allow the designer to specify systems that do not have this property (see, e.g, [65]). We recall that the presence of cyclic dependencies among values of events with the same tag are responsible for this difficulty. It is the domain of the language and its semantical interpretation to verify whether a unique solution exists. Synchronous systems that have a unique solution, have a "single" finite state machine equivalent even though they consist of several interconnected components, and thus can be analyzed and verified with efficient techniques.

Implementation cost: Adherence to the synchronous assumption, that is, a process computes in negligible time compared to its environment, is a property that must be verified or enforced on the final design, and which may be expensive to implement. The assumption is checked on the final implementation. For hardware, one must ensure that the clock period is higher than the maximum possible computation time for a synchronous block; this may imply a clock rate that is much slower than might otherwise be achieved. For software, one must ensure that an invoked process is allowed to complete before another process or the operating system changes its inputs.

4.1.2 Asynchrony

Basic operation: Asynchronous events always have a non-zero amount of time between them: it is impossible to specify that two events happen simultaneously in a truly asynchronous system (as in real life ...). An individual process can run whenever it has a change on its inputs, and it may take an arbitrary time (that is typically bounded) to complete its computation.

Triggering and Ordering: A module is only triggered to run (and always triggered to run) when it has inputs that have changed. However, among the triggered modules, there is no a priori ordering of processes. One may later be imposed by a scheduling algorithm, but this is part of the implementation choice.

System solution: There is strong dependency of the solution from input signals and their timing. Thus, asynchronous systems are much more difficult to analyze. In addition, in a practical implementation or a model thereof, some events may *appear* to happen simultaneously. In practice it may be difficult and expensive to maintain the total ordering. If the actual order of these seemingly simultaneous events is not preserved, any order may be used possibly resulting in multiple behaviors. This is no longer an asynchronous model but a discrete-event model that has no guarantee of uniqueness of the solution because of the possible cyclic dependency of values of events with the same tags. It is this practical aspect that has misled many when assessing the properties of asynchronous systems, loading asynchronous systems with problems that are typical of discrete-event systems.

Implementation cost: Asynchronous implementations are usually chosen when the cost, particularly in terms of computation time, is too high for a synchronous solution. The flexibility provided by an asynchronous implementation implies that different parts of the same system (or the same system under different inputs) can operate at quite different rates, only communicating at particular check-points in the computation. For system design, it is usually imperative to have an asynchronous model at the highest level of communication. On the other hand, analysis of the behavior of designs implemented as asynchronous systems has to take into consideration the timing of the signals and, hence, is much more complex than the analysis of synchronous systems. This is the reason why much research on asynchronous system has been devoted to implementations that are more or less insensitive to “internal” delays, thus retaining the most desirable property of synchronous systems without paying the full penalty implied in a synchronous implementation.

4.1.3 Combining Synchrony and Asynchrony

An ideal MOC for system design should combine the advantages of verifiability in synchrony and flexibility in asynchrony in a globally asynchronous, locally synchronous (GALS) model. It is important to be explicit about where the boundary is between synchrony and asynchrony, because the behavior of the two, clearly, is very different. The differences can be illustrated simply in terms of event buffering and timing of event reading/writing.

In an asynchronous implementation, there is typically a need for an explicit buffering mechanism for the events, since it is not known when a module will run and hence read its inputs, and since different modules will run at different times and use the same input at different times. For synchrony, inputs are all read once at the beginning of a computation, so one global copy of each event value suffices and this one copy is cleared at the end of each tick. Thus, a synchronous communication transmits all events simultaneously and in zero time with no buffering; every module is guaranteed to see the same set of events at each clock tick. An asynchronous communication transmits events when ready and through buffers; each module sees its own stream of inputs which depends on the global scheduling.

Many programs will behave the same for an asynchronous or synchronous implementation, and such systems are typically more tolerant to implementation fluctuations. One can program in a style that is more robust with respect to these differences, by, for example

- Never assuming or waiting for simultaneous (synchronous) events. Since simultaneity is nearly impossible to guarantee, it is more robust to wait for the occurrence of two events rather than the *simultaneous* occurrence of them.

- Never programming with a global timing, e.g. a global clock tick, in mind. Synchronous languages have mechanisms for allowing a clock tick to pass, and thus for counting clock ticks and waiting for a certain amount of this artificial time to pass. Asynchronous systems of course do not have such a specific notion of time, so the same style of programming with an asynchronous model interpretation (in which a clock tick usually forces an ordering rather than referring to a time) will produce different behavior.

One may certainly use these programming techniques within a synchronous portion of the design. At the system level, however, a time-intolerant style of programming and thinking about the behavior of a design should be employed.

Our CFSM model reflects these views and was strongly motivated by the need of combining synchronous and asynchronous behavior where it made most sense.

4.2 CFSM Overview

Each CFSM is an extended FSM, where the extensions add support for data handling and asynchronous communication. In particular, a CFSM has

- a *finite state machine part* that contains a set of inputs, outputs, and states, a transition relation, and an output relation.
- a *data computation part* in the form of references in the transition relation to external, instantaneous (combinational) functions.
- a *locally synchronous behavior*: each CFSM executes a transition by producing a single output reaction based on a single, snap-shot input assignment in zero time. This is synchronous from *its own perspective*.
- a *globally asynchronous behavior*: each CFSM reads inputs, executes a transition, and produces outputs in an unbounded but finite amount of time as seen by the rest of the system. This is asynchronous interaction from the *system perspective*.

This semantics, along with a scheduling mechanism to coordinate the CFSMs, provides a GALS communication model: Globally (at the system level) Asynchronous and Locally (at the CFSM level) Synchronous.

4.3 Communication Primitives

4.3.1 Signals

CFSMs, as TSM processes, communicate through *signals*, which carry information in the form of *events*. They may function as *inputs*, *outputs*, or *state signals*.

A signal is communicated between two CFSMs via a *connection* (single-input, single-output communication process) that has an associated *input buffer* (or 1-place buffer), which contains one memory element for the event (event buffer) and one for the data (data buffer).

The event is *emitted* (produced) by a *sender* CFSM setting the event buffer to 1. It may be *detected* and *consumed* by a *receiver* CFSM. It is detected by reading the event buffer; it is consumed by setting the buffer to 0.

A signal is therefore *present* if it has been emitted and not yet consumed. In the tagged signal model, this means that the input signal of the connection has had an event with a tag larger than the largest tag of the output signal.

The data may be *written* by a sender and *read* by a receiver. Reading and writing is done on the data buffer on the connection between the sender and the receiver.

A control signal carries only event information, i.e., it may only be emitted and detected/consumed and its value is irrelevant. A data signal carries only data information, i.e., it may only be read and written.

An input signal can only be detected/consumed and read (depending on its status as a signal, control signal, or data signal). A (possibly incomplete) set of values for the input signals of a CFSM is termed *input assignment*, a set of input values read by a CFSM at a particular time is termed *captured input assignment*, and an input assignment with at least one present event is termed *input stimulus.*

An output signal can only be emitted and written. A (possibly incomplete) set of values for the outputs of a CFSM is termed *output reaction.*

A state signal is an internal input/output data signal; it may be written and subsequently read by its CFSM. A *state* is a set of values for the state signals. A set of states may be given by a subset of state values. States are implicitly represented by the state signals and hence may be encoded or symbolic. The state signals could be considered part of the input and output signal sets, and it is only for the exposition that they are separated: discussion of scheduling and runnable CFSMs is facilitated by identifying an input assignment that triggers the CFSM separately from its state.

Where the type is unimportant, we may refer to any of the basic signal types (signal, control signal, data signal, input signal, output signal, state signal) simply as signal.

As will be seen in the behavior sections to follow, CFSMs *initiate communication through events.* The input events of a CFSM determine when it may react. That is, the model forbids a CFSM to react unless it has at least one input event present (except for the initial reaction, described in the functional behavior section). Without this restriction, a global clock would be required to execute the CFSMs at regular intervals, and this clock would in fact be a triggering input for all CFSMs. This would clearly imply a more costly implementation. A CFSM can trigger itself by emitting an output and detecting that same signal in the next execution. A CFSM with at least one present input event is termed *runnable.*

For CFSM A to send a signal S to B, A writes the data of S and then emits its event. This order ensures that the data is ready when the event is communicated. B is scheduled, sees the event (which is its stimulus), reads the corresponding data, and reacts. Pure data signals will only be read and written by a CFSM that has already been triggered by the presence of another input event.

4.3.2 CFSM Networks

A *net* is a set of connections on the same output signal, i.e., it is associated with a single sender and at least one receiver (in the TSM, it is a set of connection processes with the same input). There is an input buffer (TSM connection) for each receiver on a net, hence the communication mechanism is *multi-cast*: a sender communicates a signal to several receivers with a single emission, and each receiver has a private copy of the communicated signal. Each CFSM can thus independently detect/consume and read its inputs.

A *network* is a set of CFSMs and nets. The behavior of the network (and even of a single CFSM) depends on both the individual behavior, and that of the global system. In the mathematical model, the system is composed of CFSMs and a scheduling mechanism coordinating them. It can be implemented as:

- a set of CFSMs in software (e.g., C), a compiler, an operating system, and a microprocessor (the *software domain*),
- a set of CFSMs in hardware (e.g., gates mapped to an FPGA), a hardware initialization scheme, and a clocking scheme (the *hardware domain*), and
- the *interface* between them (e.g., a polling or interrupt scheme to pass events from hardware CFSMs to software ones via the RTOS, a memory-mapped scheme to pass events from software to hardware).

Thus the scheduling mechanism in the model may take several forms in the implementation: a simple RTOS scheduler for software on a single processor and concurrent execution for hardware, or a set of RTOSs on a heterogeneous multi-processor for software and a set of scheduling FSMs for hardware.

The CFSM model does not require any coordination between these schedulers in order to guarantee correct behavior, apart from an implementation of the event delivery mechanism (the interface). Explicit or implicit coordination is required only in order to satisfy timing constraints, which in turn may guarantee an ordering of events and/or a particular functional behavior.

4.4 Timing Behavior

In the CFSM model, a global "scheduler" controls the interaction of the CFSMs, and invokes each appropriately during execution of the design. The system output will depend on the functional and timing behavior of the individual CFSMs, and the functional and timing behavior of their ensemble.

The scheduler operates by continually deciding which CFSMs can be run, and calling them to be executed. Each CFSM is either *idle* (waiting for input events or waiting to be run by the scheduler), or *executing* (generating a single reaction). During an *execution*, a CFSM reads its inputs, performs a computation, and possibly changes state and writes its outputs.

The mathematical model places few restrictions on the timing of an execution. Each CFSM execution can be associated with a single *transition point*, t_i, in time. The model dictates that it is at this point that the CFSM begins reacting: reading inputs, computing, changing state, and writing outputs. Since the reaction time is unbounded, one cannot say exactly at which time a particular input (event or data) is read, at which time that input had previously been written, or at which time a particular output is written. There are, however, some restrictions. For each execution, each input signal is read at most once, each input event is cleared at every execution, and there is a partial order on the reading and writing of signals. Since the data value of a signal (with an event and data part) only has meaning when that signal is present, the model dictates that the event is read before the data. Similarly for the outputs, the data is written before the event, so that it is valid at the time the event is emitted.

This means that for transition point t_i,

- an input may be read at any time between t_i and t_{i+1} (but not later, because that would correspond to transition point t_{i+1}),
- the event that is read may have occurred at any time between t_{i-1} and t_{i+1},
- the data that is read may have been written at any time between t_0 and t_{i+1}, and
- the outputs are written at some time between t_i and t_{i+1}.

After reading an input, its value may be changed by the sender before t_{i+1}, but the receiver reacts to the captured input and the new value is not read until the next reaction.

This flexibility in timing can have non-intuitive behavior.

Example: Event/data separation. Suppose a sender S writes data value v1 at t_1 for signal X and emits it at t_2. Let this emission be e1, so the pair is (e1, v1). This is illustrated in Figure 7. A receiver R at t_i sometime later reads the event, but takes longer than expected to read its corresponding value v1. S communicates X again: value v2 then emission of X, for pair (e2, v2). R now reads the value for X, and reads v2. The captured input for the R is thus the pair (e1, v2), which was not the pair intended by S. Furthermore, data v1 has been lost forever, even though it was sent with an event to signal its presence.

Problems such as this can easily be resolved by requiring the appropriate level of *atomicity* in the model and in the implementation, i.e., by restricting some parts of the communication to take place simultaneously and instantaneously – as a single entity. In the CFSM model, the requirement is simply that the input events are read atomically. At each t_i, a CFSM reads its input events without interruption, and without those events being overwritten by the sending CFSMs. This is easily implemented in software by reading a bit-vector of input events in one instruction, and in hardware by clocking all

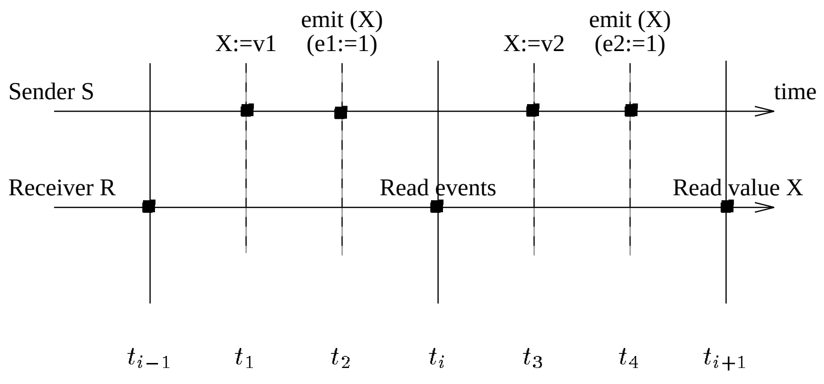


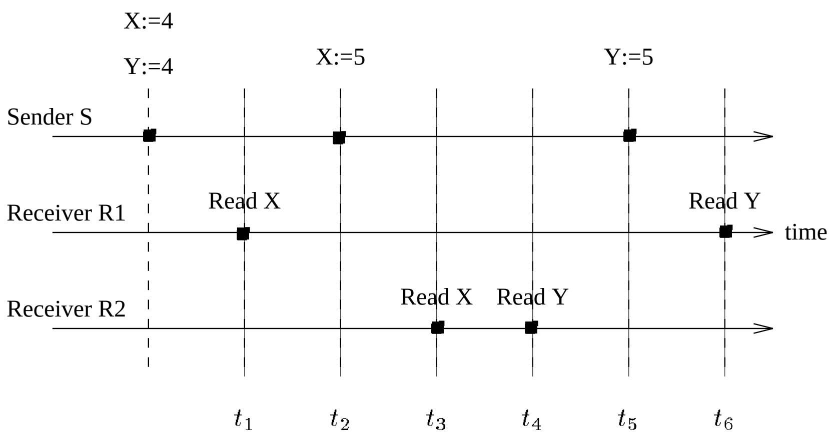
Figure 7: Event/data separation.

CFSMs together, with a separate read phase and a compute/write phase (per clock cycle).

Atomicity of input event reading implies an implementation that retains much of the flexibility required for efficiency, while mitigating the worst of the synchronicity problems. It should be clear that allowing input event and data reading and output event and data writing to happen at completely arbitrary times leads to behavior with very difficult to predict and prescribe results. Given the event-based communication of CFSMs, atomicity of input event reading is a natural means of ensuring some predictability: a receiver CFSM is guaranteed to see a snapshot of input events that are simultaneously present at some point in real time. Additional constraints, if necessary, can be imposed to ensure that the values subsequently read are meaningful. These constraints will vary considerably depending on the implementation chosen and the design constraints.

Example: no atomicity of data reading. Consider a sender S and two receivers R1 and R2, as illustrated in figure 8. S is sending the value of signals X and Y. Both X and Y are currently 4 and are changing to 5. R1 reads X at t_1 and Y at t_6. R2 reads X at t_3 and Y at t_4. S changes X at t_2 and Y at t_5. R1 therefore captures $X = 4$ and $Y = 5$ while R2 captures $X = 5$ and $Y = 4$. Not only do they capture different input assignments, but R1 captures a set of values that never occurs simultaneously. Note that if X and Y were (control/data) signals, they would be sent with events as well, and they would both be changed to 5 before the events were emitted and hence before the receivers can read them. However, those new values can be overwritten by the sender if the receiver doesn't read fast enough, leading to the separation of the event/data pair as illustrated in the previous example.

Example: atomicity of event reading. Now consider a system with a



Receiver R1: X = 4, Y = 5

Receiver R2: X = 5, Y = 4

Figure 8: No atomicity of data value reading.

sender S and three receivers R1, R2, and R3 as illustrated in Figure 9. S emits control signals X and Y at times t_2 and t_4. R1 reads at t_1 and sees both absent. R2 reads at t_3 and sees only X present. R3 reads at t_5 and sees both X and Y present. Though each has a different captured input assignment, each sees an input assignment that occurs at some point in real time.

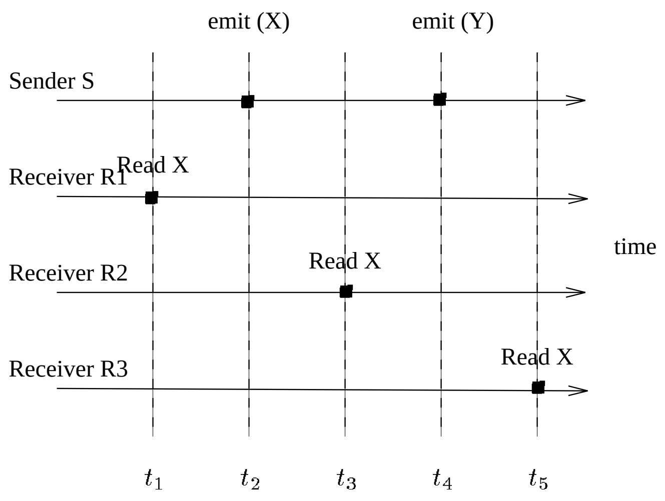
4.5 Functional Behavior

The functional behavior of a CFSM at each execution is determined by the specified *transition relation* (TR). This relation is a set of tuples (**input_set**, **previous_state**, **output_set**, **next_state**) where **input_set** is an input assignment, **previous_state** is a state, **output_set** is an output assignment, and **next_state** the next state. Each tuple of the TR represents a specified *transition* of the machine, and the set of tuples is the specified behavior of the machine. A transition in which the **input_set** includes an input stimulus is termed a *valid transition*.

At each execution, a CFSM

1. Reads an input assignment.

2. Looks for a transition **Transition** = (**input_set**, **previous_state**, **output_set next_state**) such that the read input includes **input_set** and the present state of the CFSM matches **present_state** (hence the absence of an input from **input_set** means "don't care about that input").



Receiver R1 : X, Y absent
Receiver R2 : X present, Y absent
Receiver R3 : X, Y present

Figure 9: Atomicity of event reading.

3. If **Transition** is found, it is executed by
   (a) consuming the inputs (setting input event buffers to 0)
   (b) making the state transition to **next_state**
   (c) writing the new output events in **output_set** (hence absence from **output_set** means "don't emit/don't modify").
4. If **Transition** is not found, the CFSM consumes no inputs, makes no state change, and writes no outputs.

The last case, in which no matching transition is found, is known as the *empty execution*. If this can happen for some input stimulus, the transition relation is *incomplete*; otherwise it is *complete*. For software, this is precisely the same behavior that would be produced if this CFSM had not been scheduled by the RTOS. If several transitions match, the CFSM is non-deterministic and the execution can perform any of the matching transitions. For the implementation, all CFSMs must be deterministic in order to simulate and synthesize the behavior. Non-determinism can be used at the initial stages of a design in order to model partial specifications, later to be refined into deterministic CFSMs.

A *trivial transition* is one in which no output events are emitted, no output values are changed, and no state change is effected, but inputs are consumed. It effectively discards the current input assignment and waits for a new one. Trivial transitions are specified in the TR like any other transition, with **output_set** empty (or leaving state variables unchanged).

Each state variable may have a designated set of *reset values* (or initial values) that are specified with the transition relation. A set of reset values, one for each state variable, is a *reset state* (or initial state). If there are several reset values for a state variable, there are several reset states. This represents a non-deterministic starting condition which must be resolved before synthesis or simulation can be performed.

The *initial transition(s)* is a special transition(s) where the present state part is equal to the reset state. Moreover, this transition is allowed to not have any input events present in the input assignment. (Recall that for all other transitions, at least one input event is required to trigger the CFSM.) The initial transition(s) may be specified for a CFSM, but is not required. There may be several possible initial transitions depending on the initial state(s) and the values of the corresponding input assignments. If there is non-determinism, again, it must be resolved before synthesis and simulation.

4.6 CFSMs and process networks

We can now classify CFSMs along the same lines that were used for the other formal models.

CFSMs are an asynchronous Extended FSM model, that is different from CSP and CCS because communication is not via rendezvous but via bounded (1-deep) non-blocking buffers, and different from SDL since queues are bounded and the process network topology is fixed.

Moreover, each CFSM can be modeled with an LTS in which each edge label can involve presence and absence tests of several signals, while in CSP, CCS, and SDL each label consists of a single symbol.

Signals are distinguished among inputs and outputs. Transitions, unlike dataflow networks, can be conditioned to the *absence* of an event over a signal. Hence CFSMs are not continuous in Kahn's sense [35] (arrival of two events in different orders may change the behavior).

The semantics of a CFSM network is defined based on a *global* explicit notion of time (imposing a total ordering of events). Thus CFSMs can be formally considered as *synchronous* with *relaxed timing*. I.e., while a global consistent state of the signals is required in order to perform a transition, no relationship is required between the tags of input events involved in a given transition, nor between those of its output events. There is only a partial order relationship between input and output events of the transition (inputs must have tags smaller than outputs).

Finite buffering without blocking write implies, as mentioned above, that events can be *overwritten*, if the sending end is faster than the receiving end.

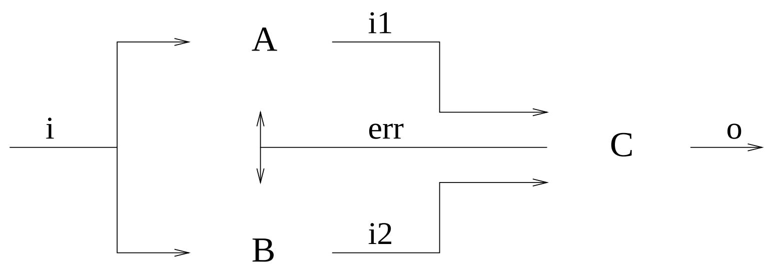


Figure 10: Example of CFSM network.

This sort of "deadline violation" in the CFSM context may or may not be a problem, depending both on the application and on the type of event.

The designer can make sure that "critical" events are never lost:

- either by providing an explicit handshaking mechanism, built by means of pairs or sets of signals, between the CFSMs,
- or by using scheduling techniques that ensure that no such loss can ever occur [5].

4.7 Examples of CFSM behaviors

In this sections we provide a few examples of what it means to specify a "behavior" with a relaxed timing model such as CFSMs, and we discuss the notion of *behavior equivalence classes*.

Consider a simple case of three CFSMs as shown in figure 10. These CFSMs specify an "almost dataflow" behavior. CFSMs A and B take the same input stream i, and perform two different kinds of (unimportant) processing on it, by producing an output event for every input event. CFSM C takes an event from each input $i1$ and $i2$ and produces

- either an event on its o output if there is no error (e.g., its inputs are within a specified range)
- or an event on its err output, if some problem in the input stream or the CFSM state is detected.

The err signal causes A and B to perform some recovery action (e.g., realign their state variables).

The intuitive behavior specified by these three CFSMs is, in the designer's eyes, the same regardless of the scheduling in time of CFSM transitions, as long as:

1. no events are lost (they are all "critical" in this case), and
2. (possibly) some latency constraint is satisfied (e.g., o may be needed earlier than the next external input arrival).

This means that, given the choice of possible timed executions of these CFSMs, they are partitioned into a set of *equivalence classes*. Let us consider, for the sake of simplicity, only reaction times and no other scheduling constraints (each CFSM is allocated its own hardware resource or processor). Let us assume that every CFSM has the same reaction time of n_r time units if there are no errors, and of $2n_r$ in case of errors (C when it emits *err*, A and B when they detect *err*). Let us also assume that input events arrive at a regular rate of n_i time units, and that there are only "no-missed-event" constraints and no other no latency constraints.

In this case, we can consider the following equivalence classes with respect to the above mentioned intuitive behavior:

1. Zero-delay executions: $n_r = 0$. These are logically inconsistent (non-causal in the Esterel terminology [11]), since if C detects an error, A and B should instantaneously react and produce different outputs (conceivably without the error conditions). This is clearly absurd.
2. Executions in which the execution delay of A, B and C is larger than the inter-arrival time of inputs: $n_i < n_r$. These executions clearly do not satisfy the intuitive behavior requirements listed above.
3. Executions in which the delay of A, B and C is smaller than the inter-arrival times of external inputs, but larger than half of that: $n_i/2 < n_r \leq n_i$. These executions handle correctly the *normal* flow of data, but "miss" an input in the case of error. This happens because the execution time of C is too long and causes it to miss the first input events after the error. Also A and B are too slow and miss an input event when recovering.
4. Executions in which the delay of A, B and C is smaller than half of the inter-arrival times of external inputs: $n_r \leq n_i/2$. In this case, no event is lost.

If errors are infrequent enough, the designer may want to consider the two last classes to be equally good, and accept the cheapest one. On the other hand, if errors are frequent, or if every single input really matters (there is zero redundancy in the input stream), only the last, conceivably most expensive equivalence class is acceptable.

Note how the notion of equivalence classes can be applied to analyze also executions in which a scheduler coordinates CFSMs by enforcing mutual exclusion constraints (this is an appropriate model, e.g., for a single-processor implementation). In that case event loss can occur due to

- Timing constraints. E.g., $n_i < 3n_r$ would imply that an execution falls into the second class above, and misses deadlines due to excessive processor occupation.
- "Incorrect" scheduling. E.g. if $3n_r \leq n_i < 4n_r$ and the scheduler activates the CFSMs in the fixed order ACBC. In that case, the first activation of C is valid, since it has an input stimulus, but redundant, since it always results in an empty execution (assumed with the same n_r delay), and causes the system to miss a deadline even during "normal operation". On the other hand, another valid schedule ABC will not miss deadlines in normal operation.

Obviously this definition of "equivalence classes" between behaviors of a CFSM network is very application-dependent, and as such difficult to formalize. Here we can suggest only a few criteria that could be used for this purpose, such as:

1. Equality between streams of values produced at some output by two different timed behaviors of the same network, given the same stream of values on the inputs ("dataflow" equivalence).
2. Compatibility with a given partial ordering between events ("Petri net" equivalence).
3. No missed critical events, possibly qualified as, e.g., "no missed events except for the first n events after abnormal event x" ("quasi-dataflow" equivalence).
4. Equality of input-output sequences, possibly modulo reordering of "concurrent events", with respect to a completely deterministic reference specification ("golden model" equivalence).
5. Equality of input-output sequences modulo *filtering* by some testbed entities that model the external, physical system constraints ("filtered" equivalence).

While we are still far from a formalization of these criteria, we believe that the richness of the CFSM model stems, among other factors, from the ability to exploit all these sorts of equivalence while, for example, dataflow networks can exploit only one (dataflow equivalence). We will further elaborate on this in the next section.

5 Conclusions

The relative advantages and disadvantages of the various MOCs have been described in the previous sections. We are still far from having a single agreed-upon standard MOC that is suitable for all types of embedded system designs.

Some authors ([17]) advocate heterogeneity at the MOC level as an essential requirement of embedded systems. However, based on the discussions above, we can identify a new MOC that is expressive enough to capture most practical embedded systems, and formal to permit efficient verification and synthesis of some special cases.

This model is that of CFSMs with *initially unbounded FIFO buffers*. Bounds on buffers (essential for implementability) are imposed by *refinement*, exactly as timing information is refined in the original CFSM model. The motivations for this proposal are as follows.

1. Local synchrony: concentrating the control inside relatively large atomic synchronous entities helps the designer to better understand the overall coordination. Models such as Colored Petri Nets, which view control at a finer level of granularity, are difficult to use for large realistic designs.

2. Global asynchrony: breaking synchronicity helps resolving composition problems and mapping to a heterogeneous architecture. Synchronous models, as argued previously, cannot handle multi-rate efficiently, especially when the rates of different signals are totally uncorrelated. This is essentially due to the need to always consider all signals, including those that are not present in most clock cycles.

3. Unbounded buffers: leaving buffers unbounded at the outset offers opportunities to perform *static and quasi-static scheduling* whenever possible ([40, 18, 47]). As a result, some buffers become statically sized, as determined by the static schedule of reader and writer processes. The designer sizes the remaining buffers to ensure implementability, and to use simulation, formal verification or Real-time scheduling [4, 5] to validate the design in presence of finite FIFO buffers.

The use of *lossy buffers* (i.e., with non-blocking write) is somewhat arguable, because

- there are cases in which loss is essential (in general, any time there are tight timing constraints, that make some of the signals irrelevant under some conditions, e.g., an emergency), and

- there are cases in which loss is problematic (in general, any time one would like to model dataflow computations or any other blocking communication mechanism, such as, e.g., Remote Procedure Call).

Our choice is to *keep buffers lossy in the formal model*, and give the designer tools to *verify a priori if loss can occur*, as well as to *enforce no loss for some buffers in the implementation*. In general this can be enforced at an acceptable cost only under some specific conditions, e.g., that the lossless buffer must be local to a processor, or that the communicating processes must be statically scheduled with respect to each other.

The resulting model combines interesting properties of the main MOCs seen above, while still keeping a strong link to verifiability and implementability. In particular:

- At the initial, untimed level it describes a partial ordering between signal tags, and hence captures a whole class of possible implementations on a variety of architectural options. These options include software on multiple processors, pipelined hardware, and so on.
- It keeps computation (in the FSM), communication (in the buffers) and timing (in the architectural mapping) as separate as possible.
- After architectural mapping, it becomes essentially a Discrete Event model, and thus lends itself to performance and power consumption analysis, in order to evaluate architectural trade-offs.
- A subset, such that CFSMs have a deterministic behavior (i.e., behave as SDF actors) can be statically scheduled as SDF [40]. A larger subset can also be quasi-statically scheduled (thus performing static buffer sizing) by means of a mapping to Petri Nets [47].

While the opportunities for system-level optimization offered by this choice still need to be fully explored, we can already envision a design flow in which the designer uses multiple languages, depending on the domain of application and other requirements (e.g., tool availability, company policy or personal preference), that all have a semantics in terms of CFSMs. Then multiple scheduling, allocation, partitioning, hardware and software synthesis algorithms can be applied on the CFSM network, possibly depending on the identification of special cases that admit an especially efficient implementation. Formal verification and simulation can be used throughout the design process, thanks to the refinement-based design applied to a formal model. Refinement occurs both at the functional level, implementing CFSMs ([6]), and at the communication level, implementing communication ([56]) and scheduling ([6, 47]). This scheme has been adopted in the POLIS system and has been also followed by a commercial product of the Alta Group of Cadence Design Systems, Inc.

Acknowledgments We thank Prof. Ed Lee for the work that led to the development of the Tagged Signal Model that has been used throughout this paper, and for his many contributions to the field of system design for many years. Part of this paper has been adapted from an earlier version co-authored with Prof. Lee and Dr. Stephen Edwards. We wish to thank the Polis team and the Felix team of Cadence Design Systems, Inc., who shared the discoveries and developments which lead to this work. In particular, we wish to thank Dr. Jim Rowson for the key role he has played in putting together the design methodology and the architecture in the Cadence work, and Dr. Felice Balarin for introducing the notion of equivalence classes between CFSM behaviors. We are indebted to our colleagues of the University of California at Berkeley, of the Politecnico di Torino, of Cadence Berkeley and European Labs and of PARADES for the many discussions and their contributions. This work has been partially sponsored by grants of CNR and Cadence Design Systems.

References

[1] W. B. Ackerman. Data flow languages. *Computer*, 15(2), 1982.

[2] R. Alur and D. Dill. Automata for Modeling Real-Time Systems. In *Automata, Languages and Programming: 17th Annual Colloquium*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335, 1990. Warwick University, July 16-20.

[3] Arvind and K. P. Gostelow. The U-Interpreter. *Computer*, 15(2), 1982.

[4] F. Balarin, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal verification of embedded systems based on CFSM networks. In *Proceedings of the Design Automation Conference*, 1996.

[5] F. Balarin and A. Sangiovanni-Vincentelli. Schedule validation for embedded reactive real-time systems. In *Proceedings of the Design Automation Conference*, 1997.

[6] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems – The POLIS approach.* Kluwer Academic Publishers, 1997.

[7] A. Balluchi, M. Di Benedetto, C. Pinello, C. Rossi, and A. Sangiovanni-Vincentelli. Hybrid control for automotive engine management: The cut-off case. In *First International Workshop on Hybrid Systems: Computation and Control.* LNCS - Springer-Verlag, April 1997.

[8] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[9] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions on Automatic Control*, 35(5):525–546, May 1990.

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

[11] G. Berry. The foundations of esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[12] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Press, Norwood, Mass, 1996.

[13] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Static scheduling of multi-rate and cyclo-static DSP applications. In *Proc. 1994 Workshop on VLSI Signal Processing*. IEEE Press, 1994.

[14] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9), 1991.

[15] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[16] J.A. Brzozowski and C-J. Seger. Advances in Asynchronous Circuit Theory – Part I: Gate and Unbounded Inertial Delay Models. *Bulletin of the European Association of Theoretical Computer Science*, October 1990.

[17] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *Interntional Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[18] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, U.C. Berkeley, 1993. UCB/ERL Memo M93/69.

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

[20] N. Carriero and D. Gelernter. Linda in context. *Comm. of the ACM*, 32(4):444–458, April 1989.

[21] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Hardware/software codesign of embedded systems. *IEEE Micro*, 14(4):26–36, August 1994.

[22] F. Commoner and A. W. Holt. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[23] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri nets from finite transition systems. *IEEE Transactions on Computers*, 47(8):859–882, August 1998.

[24] J. Desel and J. Esparza. *Free choice Petri nets*. Cambridge University Press, New York, 1995.

[25] J. Desel, K.-P. Neuendorf, and M.-D. Radola. Proving non-reachability by modulo-invariants. *Theorectical Computer Science*, 153(1-2):49–64, 1996.

[26] D. Drusinski and D. Harel. On the power of bounded concurrency. I. Finite automata. *Journal of the Association for Computing Machinery*, 41(3):517–539, May 1994.

[27] P. Godefroid. Using partial orders to improve automatic verification methods. In E.M Clarke and R.P. Kurshan, editors, *Proceedings of the Computer Aided Verification Workshop*, 1990. DIMACS Series in Discrete Mathematica and Theoretical Computer Science, 1991, pages 321-340.

[28] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[29] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1319, 1991.

[30] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, 1987.

[31] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, 16(4), April 1990.

[32] T.A. Henzinger. The theory of hybrid automata. Technical Report UCB/ERL M96/28, University of California, Berkeley, 1996.

[33] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

[34] R. Jagannathan. Parallel execution of GLU programs. In *2nd International Workshop on Dataflow Computing*, Hamilton Island, Queensland, Australia, May 1992.

[35] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[36] D. J. Kaplan et al. Processing graph method specification version 1.0. The Naval Research Laboratory, Washington D.C., December 1987.

[37] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal*, 14:1390–1411, November 1966.

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

[39] R. Lauwereins, P. Wauters, M. Adé, and J. A. Peperstraete. Geometric parallelism and cyclostatic dataflow in GRAPE-II. In *Proc. 5th Int. Workshop on Rapid System Prototyping*, Grenoble, France, June 1994.

[40] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *IEEE Proceedings*, September 1987.

[41] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, May 1995.

[42] E. A. Lee and A. Sangiovanni-Vincentelli. The tagged signal model - a preliminary version of a denotational framework for comparing models of computation. Technical report, Electronics Research Laboratory, University of California, Berkeley, CA 94720, May 1996.

[43] E.A. Lee and A. Sangiovanni-Vincentelli. Comparing models of computation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 234–241, 1996.

[44] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *Proc. of the IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.

[45] K. McMillan. *Symbolic model checking.* Kluwer Academic, 1993.

[46] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[47] M.Sgroi, L.Lavagno, and A.Sangiovanni-Vincentelli. Quasi-static scheduling of free-choice petri nets. Technical report, UC Berkeley. `http://www-cad.eecs.berkeley.edu/~sgroi/qss.ps`, 1997.

[48] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96:3–33, 1992.

[49] Institute of Electrical and Electronics Engineers. *IEEE standard VHDL language reference manual.* IEEE, 1994.

[50] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1981.

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

[52] J. Rasure and C. S. Williams. An integrated visual language and software development environment. *Journal of Visual Languages and Computing*, 2:217–246, 1991.

[53] W. Reisig. *Petri Nets: An Introduction.* Springer-Verlag, 1985.

[54] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits: A hardware implementation of LUSTRE. In *Real-Time, Theory in Practice, REX Workshop Proceedings*, volume 600 of *LNCS*, pages 195–208, Mook, Netherlands, June 1992. Springer-Verlag.

[55] T. G. Rokicki. *Representing and Modeling Digital Circuits.* PhD thesis, Stanford University, 1993.

[56] J. Rowson and A. Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the Design Automation Conference*, pages 178–183, 1997.

[57] E.M. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proc of the ICCD*, pages 328–333, October 1992.

[58] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proceedings of the European Design and Test Conference*, March 1996.

[59] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, MA, 1977.

[60] P. A. Suhler, J. Biswas, K. M. Korner, and J. C. Browne. Tdfl: A task-level dataflow language. *J. on Parallel and Distributed Systems*, 9(2), June 1990.

[61] W. Takach and A. Wolf. An automaton model for scheduling constraints in synchronous machines. *IEEE Tr. on Computers*, 44(1):1–12, January 1995.

[62] D.E. Thomas and P. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[63] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[64] P.H.J. van Eijk, C. A. Vissers, and M. Diaz, editors. *The Formal description technique Lotos.* North-Holland, 1989.

[65] M. von der Beeck. A comparison of statecharts variants. In *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *LNCS*, pages 128–148. Springer-Verlag, 1994.