October 19, 1998. Revised from Technical Memorandum UCB/ERL M97/57, Aug. 17, 1997.

HIERARCHICAL FINITE STATE MACHINES WITH MULTIPLE CONCURRENCY MODELS

Alain Girault, Bilung Lee, and Edward A. Lee

University of California at Berkeley and INRIA
Alain.Girault@inrialpes.fr, bilung@eecs.berkeley.edu, eal@eecs.berkeley.edu

Abstract

This paper studies the semantics of hierarchical finite-state machines that are composed using various concurrency models, particularly dataflow, discrete-events, and synchronous/reactive modeling. It is argued that all three combinations are useful, and that the concurrency model can be selected independently of the decision to use hierarchical FSMs. In contrast, most formalisms that combine FSMs with concurrency models, such as Statecharts (and its variants) and hybrid systems, tightly integrate the FSM semantics with the concurrency semantics. An implementation that supports three combinations is described.

1.0 INTRODUCTION

Manna and Pnueli [35] argue that concurrency is the essential feature of reactive systems, a class that includes all embedded systems, real-time systems, and many software systems. In concurrent systems, modules consist of relatively autonomous agents that interact through messaging of some sort. The rules of interaction of the agents, the semantics of the composition, is what we call the *model of computation*.

Models of computation that support concurrency are numerous. A popular one today is *threads*, where a set of sequential processes operate on the same data. More sophisticated concurrent models of computation include CSP (communicating sequential processes) [25], the pi calculus [38], dataflow [19], process networks [28], discrete events [14], and the synchronous/reactive model [5]. These models are more sophisticated in the sense that complex concurrent systems can be more easily designed, and the designs yield better to analysis. The block diagram languages used in signal processing, for example, almost all have some variant of dataflow semantics, and often yield to deadlock analysis, static scheduling, and reasonably efficient synthesis of embedded software or hardware.

While concurrency is a major source of complexity, it is not the only one. Increasingly intricate sequential control logic also adds difficulty to design, particularly when errors in the control sequence can have fatal consequences for the user, as is the case in many embedded systems. Finite state machines (FSMs) have long been used to describe and analyze intricate control sequences. Because of their finite nature, FSMs yield better to analysis and synthesis than alternative control models, such as sequential programs with if-then-else and goto. For example, with an FSM, a designer can enumerate the set of reachable states to ascertain that a particularly dangerous state cannot be reached. The same question may be undecidable in a richer language.

Most modern electronic systems have both intricate control requirements and concurrency. Thus, combining FSMs with concurrent models of computation is an attractive and increasingly popular approach to design. Since Harel introduced that Statecharts model [23] in 1987, a number of variations have been explored [44]. The Argos language [36][37], for example, combines FSMs with a synchronous/reactive concurrency model [5]. Jourdan *et al.* [27] combine the synchronous language Lustre [22] and Argos.

Many researchers have combined FSMs with concurrent models of computation that are significantly different from that of Statecharts. SDL combines process networks with FSMs [4]. The codesign finite state machine (CFSM) model [16] combines FSMs with a discrete-event concurrency model. Pankert *et al.* combine synchronous dataflow [31] with FSMs [40][36]. Program-state machines (PSM) combine imperative semantics with FSMs [39][43]. Hybrid systems [1][24] mix concurrent continuous-time systems (usually given as differential equations) with finite automata. Simulink, from The MathWorks, Inc., provides a simulation environment for such combinations. All of these examples, however, tightly intertwine the concurrency model with the automata semantics. Except for Simulink and Statecharts (and some of its variants), they also have limited compositionality in that they permit automata only in the leaf cells of the hierarchy (as in SDL), or only permit automata at the top of the hierarchy (as in hybrid systems).

With Statecharts, Harel dramatically increased the usability of FSMs through two innovations [23]. First, FSMs can be hierarchically combined. A single state *a* at one level of the hierarchy is interpreted as being in one of several states, e.g. *b*, *c*, or *d*, at a lower level of the hierarchy. These are often called “or states” because being in state *a* is interpreted as being in state *b*, *c*, *or d*. Second, FSMs can be concurrently combined. An FSM with states *a* and *b* can be composed with an FSM with states *c* and *d*, resulting in an FSM that is in state *ac*, *bc*, *ad*, or *bd*. These are sometimes called “and states” because the FSM can be in both state *a* and *c*, for example. Both innovations allow state machines to be represented compactly and intuitively.

While the static interpretation of “and states” is clear, their dynamics are far less clear. Given two concurrent FSMs, when do they make state transitions, relative to one

This research was conducted as part of the Ptolemy project, which is sponsored by the Defense Advanced Research Project Agency (DARPA), the State of California MICRO program, Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, NEC, and Philips. Girault was supported by INRIA.

another? How should they communicate their state and/or transitions? These questions greatly complicate the FSM model of computation, and indeed were not completely resolved by Harel initially. This is part of the reason for the proliferation of variations of concurrent hierarchical FSM models of computation [44].

Harel loosely defined state transitions in concurrent FSMs to be simultaneous. A state transition could broadcast an event, visible immediately to all other FSMs. The other FSMs could then make state transitions immediately, and also broadcast events. As long as there is no circular logic (circular dependencies among transitions), this notion of simultaneous transitions is well-defined. Real circular dependencies can lead to genuine paradoxes and/or to underdetermined behavior. However, apparent circular dependencies prove to be common in practical systems, primarily because of the use of hierarchy, so the model had to be refined. The Argos language [37] and others refine the model by applying the synchronous/reactive (SR) principle [5], which resolves apparent circular dependencies by seeking at each instant a *least fixed point*, a globally consistent behavior. The SR principle, first developed by Berry in the Esterel language [8], gives a well-defined and determinate semantics to simultaneous concurrent actions. But there is no reason to restrict concurrent FSMs to SR semantics.

Indeed, all known high-level concurrency models have their strengths and weaknesses. SR models are good at describing tightly coordinated control, but overspecify systems that do not need such tight synchronization. Dataflow and process networks models are much more loosely synchronized, but poorly model control logic and resource management. Discrete-event models are excellent for describing hardware or other physically disjoint agents, but their physical notion of time is awkward for more conceptual or abstract concurrency.

This paper advocates decoupling the concurrency model from the hierarchical FSM semantics. We describe a family of models of computation, called **charts* (pronounced "starcharts"). Unlike Statecharts and other concurrent hierarchical FSMs, *charts do not define a concurrency model, but rather show how to embed hierarchical FSMs within a variety of concurrency models. Thus, the concurrency model can be chosen to match the problem at hand. Is tight synchronization possible? Desirable? If not, then an SR model is inappropriate, and perhaps a dataflow or process network model would be a better choice. Is there a globally consistent notion of time? If not, then a discrete-event model will be inappropriate, and perhaps a CSP model would be a better choice. The same hierarchical FSM language works with all of these concurrency models.

The hierarchy in *charts is arbitrarily deep, and concurrency models and FSMs can be placed anywhere within it. An FSM can be nested within a module in a concurrency model, with the interpretation that the FSM describes the behavior of the module. Conversely, a subsystem in some concurrency model can be nested within a state of an FSM, with the meaning that the subsystem is active if and only if the FSM is in that state. The latter is particularly well suited to describing *modal systems*, where modes of operation are modeled as states of an FSM.

More interestingly, once we have decoupled FSM semantics from concurrency semantics, heterogeneous combinations using multiple concurrency models become possible. Systems can truly be built up from modular components that are separately designed, and each subsystem can be designed using the models of computation best suited to it.

The main objective of this paper is to give a scalable approach to design. By "scalable" we mean that subsystems can be designed, analyzed, verified, and synthesized relatively independently of one another, and can then be composed in a way that the composition can be analyzed, verified, and synthesized. To achieve these objectives, our models of computation must satisfy two objectives. First, they must be *compositional*. This means that composite modules can be treated as primitive modules. Second, they must support *heterogeneity*. This means that composite modules can be embedded within a foreign model of computation. To preserve analyzability, this embedding should be done with a maximum amount of information hiding.

A side effect of supporting heterogeneity is that more specialized models of computation become more useful. They do not need to solve all problems because alternatives are available. They only need to solve some problems well. Thus, it becomes practical to use specialized models of computation, such as finite-state machines and synchronous dataflow, which have strong formal properties, excellent paths to synthesis, and natural and intuitive syntaxes.

We begin by adapting a standard notation for FSMs, which is compact and efficient when considering an FSM in isolation, to get a notation more suitable for studying compositions of FSMs. To do this, we have to put more emphasis than usual on the interaction between an FSM and its environment. We then consider combining FSMs with three popular concurrent models of computation: dataflow, discrete events, and the synchronous/reactive model. In the case of dataflow, we introduce a new subset of dataflow called heterochronous dataflow that combines particularly well with FSMs. We then briefly describe an experimental implementation in the Ptolemy environment [13], where hierarchical FSMs can be combined with dataflow, discrete-event, and synchronous/reactive concurrency models.

2.0 FINITE STATE MACHINES

2.1 The Basic FSM

An FSM is a five-tuple [26]

$$(Q, \Sigma, \Delta, \sigma, q_0) \tag{1}$$

where

1. Q is a finite set of symbols denoting states.
2. Σ is a set of symbols denoting the possible inputs.
3. Δ is a set of symbols denoting the possible outputs.
4. σ is a transition function mapping $Q \times \Sigma$ to $Q \times \Delta$.
5. $q_0 \in Q$ is the initial state.

In one *reaction*, an FSM maps a current state $p \in Q$ and an input symbol $a \in \Sigma$ to a next state $q \in Q$ and an output symbol $b \in \Delta$, where $\sigma(p, a) = (q, b)$. Given an input *word*, or sequence of symbols from the input alphabet Σ, and an initial state, a sequence of reactions will produce a sequence of states and an output word, or sequence of symbols from the output alphabet Δ. All sequences are potentially infinite.

A directed graph, called a *state transition diagram*, is popular for describing an FSM. As shown in figure 1, each elliptic node represents a state and each arc represents a transition. Each transition is labeled by "*guard*/*action*", where *guard* $\in \Sigma$ represents the input symbol that triggers the transition, and *action* $\in \Delta$ represents the output symbol when the transition is triggered. The arc without a source state points to the initial state, i.e. state α. During one reaction of the FSM, one transition is triggered, chosen from the set of enabled transitions. An enabled transition is an outgoing transition from the current state where the guard matches the current input symbol. The FSM goes to the destination state of the triggered transition and produces the output symbol indicated by the action of the triggered transition.

In this paper, we focus on *deterministic* and *reactive* FSMs. An FSM is deterministic if from any state there exists at most one enabled transition for each input symbol. An FSM is reactive if from any state there exists *at least* one enabled transition for each input symbol. To simplify notation and ensure that all our FSMs are reactive, every state is assumed to have an implicit self transition, i.e. going back to the same state, for each input symbol that is not a guard of an explicit outgoing transition. Each such self transition has as its action some default output symbol, denoted by ε, which has to be an element of Δ. Sometimes, this default symbol is interpreted to mean "empty" and is omitted from the output word [26].

For example (see figure 1), suppose $Q = \{\alpha, \beta\}$, $\Sigma = \{a, b\}$, $\Delta = \{\varepsilon, u, v\}$, $q_0 = \alpha$ and σ: $Q \times \Sigma \to Q \times \Delta$ is such that $\sigma(\alpha, b) = (\beta, v)$ and $\sigma(\beta, a) = (\alpha, u)$, then we also must have the implicit self transitions $\sigma(\alpha, a) = (\alpha, \varepsilon)$ and $\sigma(\beta, b) = (\beta, \varepsilon)$. A possible *trace*, or sequence of reactions, is shown in figure 2.

2.2 Multiple Inputs and Outputs

An FSM is embedded in an environment. The environment may in fact be part of the overall system under design, or may be out of the control of the designer. In either case, it provides a sequence of input symbols, and the FSM reacts by providing a sequence of output symbols, meanwhile tracing a sequence of states.

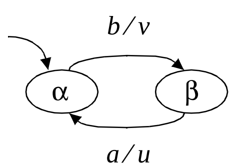

Figure 1. A basic FSM.

Current State	α	α	β	β	…
Input Symbol	a	b	b	a	…
Next State	α	β	β	α	…
Output Symbol	ε	v	ε	u	…

Figure 2. A possible trace for the basic FSM in figure 1.

Frequently, the interaction with the environment needs to be modeled in more detail. It may not be convenient, for example, to consider the FSM to have only a single input symbol. Multiple inputs and multiple outputs may be a more natural model. To handle this, the input alphabet can be factored and expressed as a cartesian product $\Sigma = \Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_M$. Here, the input to the FSM consists of *M signals*, where the i^{th} signal is a sequence of *events* represented by symbols from the *signal alphabet* Σ_i. The FSM reacts to a set of M simultaneous symbols from the M signals. The output alphabet can be similarly factored. Reactions emit events on signals.

2.3 Pure and Valued FSMs

A common special case, called a *pure FSM*, is where the size of the input symbol set is a power of two, $|\Sigma| = 2^M$, and each signal alphabet has size two, $|\Sigma_i| = 2$ for $1 \le i \le M$. We interpret this to mean that at a reaction, each signal consists of an *event* that is either *present* or *absent* (hence $|\Sigma_i| = 2$). A common notation in this scenario assigns a name to each signal, such as "a", and denotes the alphabet corresponding to that signal by $\Sigma_i = \{a, \bar{a}\}$, interpreted as {$a$ is *present*, a is *absent*}. Thus for example, consider an FSM with two input signals $I = \{a, b\}$ and two output signals $O = \{u, v\}$. The input alphabet is written $\Sigma = \{ab, a\bar{b}, \bar{a}b, \overline{ab}\}$ and the output alphabet is written $\Delta = \{uv, u\bar{v}, \bar{u}v, \overline{uv}\}$, where $\varepsilon = \overline{uv}$ is the default symbol.

In a *valued FSM*, the input and output alphabets are again factored into signal alphabets, but at least one of these signal alphabets has size greater than two (it might even be infinite). We again interpret one element of such an alphabet to denote absence of an event, while the remaining elements denote presence of an event and a value for the event. Valued FSMs are often used to augment automata with arithmetic operations, which are awkward to specify directly using pure FSMs. In our scenario, this augmentation is not fundamentally needed because arithmetic operations can be specified instead in a foreign model of computation better suited to them, such as dataflow. Nonetheless, valued FSMs may provide a more convenient syntax, even if they add nothing fundamental in expressiveness, so we will briefly discuss their ramifications.

In a pure FSM, the size of the input alphabet grows exponentially with the number of input signals. Thus, it can become quite inconvenient to define a reactive FSM by explicitly specifying outgoing transitions from every state for every input symbol. This may be a very large number of transitions. To avoid this problem, a single transition may bear as a guard a subset of Σ rather than a single symbol. It would thus represent an ensemble of transitions compactly. An arbitrary subset of Σ can be defined by a boolean expression in the input signals. For example, if $\Sigma = \{ab, a\bar{b}, \bar{a}b, \overline{ab}\}$, the boolean expression "$\neg a \vee b$" (**not** a **or** b) represents the subset $\{ab, \bar{a}b, \overline{ab}\}$. Thus, for pure FSMs, guards will be represented as boolean expressions of the input signals.

Consider the example in figure 3 with states $Q = \{\alpha, \beta\}$, input signal alphabet $I = \{a, b\}$ and output signal alphabet $O = \{u, v\}$. The guard "$\neg a \vee b$" of the transition from α to β is enabled by any input in $\{ab, \bar{a}b, \overline{ab}\}$. The guard "$a$" of the transition from β to β is enabled by any input in $\{a\bar{b}, ab\}$.

For valued FSMs, more complicated boolean-valued expressions can be used for the guards. For example, suppose that in a valued FSM, the i^{th} signal is named "a" and has the alphabet $\Sigma_i = \Re$, the set of real numbers. Then a guard may contain comparison operators and real numbers, for example "a < 10". This compactly represents an uncountably infinite number of transitions. However, it also makes it much more difficult to reason about the FSM. In fact, with a sufficiently rich expression language, it will make some questions undecidable. Thus, valued FSMs carry a high cost: loss of analyzability.

Fortunately, because our model is heterogeneous, valued FSMs add no fundamental expressiveness. Instead of a guard "a < 10" we could specify a guard "b" and externally, say in a dataflow model, compute the function

$$b = \begin{cases} true; & a < 10 \\ false; & \text{otherwise} \end{cases} . \qquad (2)$$

By supporting heterogeneous combinations of models of computation, *charts permits us to keep FSMs pure, simplifying formal analysis, while not compromising on expressiveness. Of course, the combined model is no more analyzable than the valued FSM, but at least we are able to analyze the pure FSM.

For pure FSMs, actions are specified with a reasonably compact notation. The default output for each signal is assumed to denote an absent event. An action thus only lists output signals that are to have present events in the current reaction. In other words, all output events that are not explicitly emitted in an action are absent. For example, in figure 3, the action "u" of the transition from β to β implies output symbol $u\bar{v}$, i.e. output signal u is present and output signal v is absent event when the transition is triggered. The absence of v is implicit, not explicit, a fact that will become important for hierarchical FSMs. If all events in an action are implicitly absent, the action is omitted altogether. For example, in figure 3, the label of the transition from β to α consists of just the guard "$\neg a \wedge b$", and when this transition is triggered, both output signals u and v are implicitly absent.

For valued FSMs, an action denotes any output value that is different from the default. The default is again implicitly emitted, and may denote absence of an event. Since the number of transitions in an FSM is finite, the possible emitted values form a finite set, and thus could be represented by a finite number of boolean signals. Thus, again, a pure FSM could be used without fundamental loss of expressiveness. Externally, in some other model of computation, these boolean signals could be translated into values. Hence, although valued FSMs may provide a more convenient syntax, they are not fundamentally required for expressiveness.

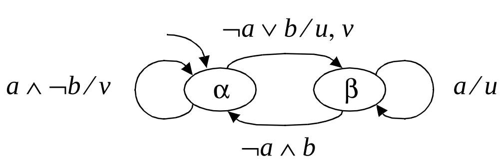

Figure 3. A pure FSM.

A possible trace for the FSM of figure 3 is shown in figure 4. Note that in state β, when both inputs a and b are absent, an implicit self-transition is taken and both outputs u and v are absent.

2.4 Hierarchy

The basic FSM, which is flat and sequential, has a major weakness; most practical systems have a very large number of states and transitions. Representation and analysis become difficult. One of Harel's solutions to this problem is *hierarchy.* In a hierarchical FSM, a state may be further refined into another FSM. We will call the inside FSM the *slave* and the outside FSM the *master* in such a composition. For example, we can let the state β in figure 3 refined into another FSM but let the state α not be refined, as illustrated in figure 5.

At a fundamental level, hierarchy adds nothing to the model of computation. Nor does it reduce the number of states. But it can significantly reduce the number of transitions and make the FSM more intuitive and easy to understand. The transition from β to α in figure 5 is simply a compact notation for transitions from γ to α and δ to α. The state space of the equivalent flat FSM is simply $Q = \{\alpha, \gamma, \delta\}$.

The input alphabet for the slave FSM is a subset of the input alphabet of its master FSM. In a pure or valued FSM,

Current State	α	α	β	β	β	…
a	*present*	*absent*	*present*	*absent*	*absent*	…
b	*absent*	*absent*	*present*	*absent*	*present*	…
Next State	α	β	β	β	α	…
u	*absent*	*present*	*present*	*absent*	*absent*	…
v	*present*	*present*	*absent*	*absent*	*absent*	…

Figure 4. A possible trace for the embedded FSM in figure 3.

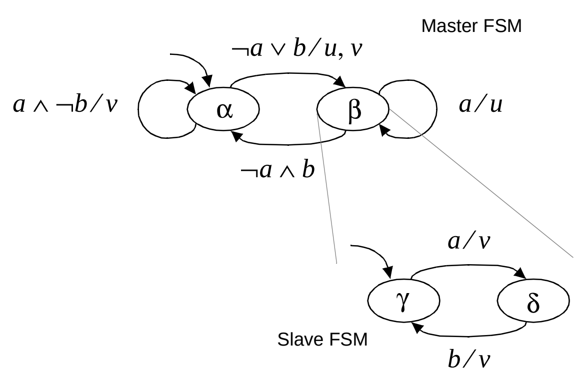

Figure 5. A hierarchical FSM.

the input signals for the slave FSM are a subset of the input signals for the master. Similarly, the output signals from the slave FSM are a subset of the output signals from its master.

The hierarchy semantics define how the slave FSMs reacts relative to the reaction of its master FSM. A reasonable semantics defines one reaction of the hierarchical FSM as follows: if the current state is not refined, the hierarchical FSM behaves just like a basic FSM. If the current state is refined, then first the corresponding slave FSM reacts and then the master FSM reacts. Thus, two transitions are triggered, so two actions are taken. These two actions must be somehow merged into one.

In the case of pure FSMs, it is easy to merge the actions and avoid conflicting definitions of the output between the slave and the master. We take an output event to be present if the action of the master or any slave FSM below it emits an event on that output. Since an action does not explicitly emit the symbol for absence of an event, no conflict is possible in this syntax. For example, if figure 5 is in state β and substate γ and input signal a is present, the triggered action of the slave FSM is “v” and the triggered action of the master FSM is “u”. Thus, the output of the hierarchical FSM is uv (both output signals u and v are present). A possible trace for the hierarchical FSM is shown in figure 6.

For a valued FSM, we adopt the convention again that an action makes no explicit mention of an absent event. However, since two actions can both emit an event with different values, the syntax permits conflicting definitions of the output. In Esterel, a function can be specified to combine the conflicting the definitions [8]. For example, for two reals, the values might be added. We prefer to consider this an error condition, because the values can be more conveniently and flexibly combined externally, in a model of computation better suited to numerical computation. Thus, for a valued (determinate) FSM, no two triggered transitions should emit the same output signal.

In the example of figure 5, the hierarchical FSM has only two levels. However, the slave FSM can actually be another hierarchical FSM, so the depth of hierarchy is arbitrary. The semantics generalizes trivially.

3.0 HETEROGENEITY — MIXING FSMS WITH CONCURRENCY MODELS

Hierarchical FSMs are not by themselves adequate for describing most complex systems. For one thing, numerical computations are extremely awkward to express within this model. For practical application to complex systems, the FSM model of computation has to be combined with others.

Current State	α	α	β,γ	β,δ	α	…
a	*present*	*absent*	*present*	*absent*	*absent*	…
b	*absent*	*absent*	*absent*	*present*	*absent*	…
Next State	α	β,γ	β,δ	α	β,γ	…
u	*absent*	*present*	*present*	*absent*	*present*	…
v	*present*	*present*	*present*	*present*	*present*	…

Figure 6. A trace for the hierarchical FSM in figure 5.

One commonly used solution is to generalize the activity associated with an action. For instance, in the Stateflow tool from the MathWorks, Inc., an action can invoke a function or assign a value to a variable. Moreover, FSMs in this tool can be embedded within a block diagram system called Simulink, allowing for numerical computations outside the FSM.

Function calls and variable assignments, by themselves, are still quite limited. They provide, for example, no concurrency. One could work around this limitation by using procedures rather than functions, and permitting them to operate on global state outside the FSM. If this is done in an undisciplined way, however, it would provide a very chaotic and poorly characterized programming model. Imposing some discipline on this model seems essential. It needs a model of computation. In the most recent version of Simulink (version 2.2), actions can invoke Simulink block diagrams, allowing a multi-level hierarchy as in Statecharts. The computation invoked by an action is specified using the Simulink concurrency model.

Unfortunately, the very richness of possibilities makes it difficult to decide *a priori* which models of computation should be used. Each has its strengths and weaknesses. We advocate leaving that choice up to the application designer, rather than building it into the language. Thus, the language should support *heterogeneity.* A convenient way to support heterogeneity is the black box approach. For a system consisting of a set of interconnected modules, each module can be treated as a black box. Some model of computation is chosen to govern the interaction between boxes, but the contents of boxes need not be governed by this same model of computation. The only requirement is that the interfaces of boxes must conform to a standard accepted by the outer model of computation. Thus, a box may encapsulate a subsystem specified by one model of computation within a system specified by another. In other words, heterogeneity allows different models of computation to be systematically and modularly combined together.

Our hierarchical FSM model of computation is easily extended to support heterogeneity. A state or transition may be refined to a black box that reacts to some subset of the input signals by emitting events on some subset of the output signals. Internally, this black box need not be an FSM. It could be, for example, a Turing machine (that halts), a C procedure (that eventually returns), a dataflow graph, etc.

In the reverse scenario, the FSM model of computation can be used to describe a module inside some other model of computation, as long as that model of computation provides a way to unambiguously determine the input symbols and when a reaction should occur. For example, in figure 7, three FSMs are embedded inside the blocks of a “block diagram language”. The exact semantics of this embedding (the *interaction semantics*) needs to be defined in terms of both the semantics of the block diagram language and the FSM. Most interestingly, however, if the block diagram language has concurrent semantics (e.g. dataflow), then the slave FSMs are concurrent FSMs.

In this section, we explore the interaction semantics of FSMs with various concurrent models of computation, namely *dataflow* (DF), *discrete-event* (DE), and *synchronous/reactive* (SR). Our objective is to develop semantics that supports arbitrary nestings of these concurrent models with FSMs. We wish for an FSM to be able to define a module in a concurrent system and for a state to be able to be refined to a concurrent subsystem. The depth and order of the nesting is arbitrary. As shown in figure 8, we adopt the notation that square boxes indicate modules in a concurrent model of computation and ellipses indicate states in an FSM.

3.1 Termination

In general, the systems of interest may not terminate. Concurrent models of computation are usually defined with this in mind [17]. The reaction of an FSM, however, will usually need to take finite time. This means that if a state refines to a concurrent subsystem, that subsystem must react in finite time to the inputs, possibly emitting output events as a result. This implies a finiteness of computation that is not intrinsic to many concurrent models of computation.

For some models of computation, there is a simple solution [15]. The execution of a non-terminating system can often be divided into a set of *iterations*. Each iteration can be associated with a reaction of the master FSM. We require, therefore, that any concurrent model of computation that can refine a state of an FSM have a well-defined finite iteration. We will explore the implications of this requirement in terms of specific examples below.

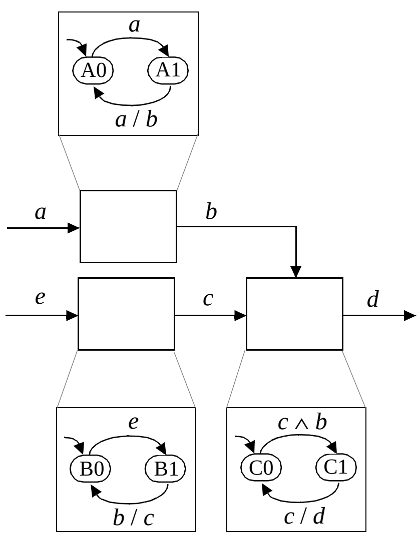

Figure 7. Three FSMs are embedded inside the blocks of a block diagram language.

The reaction of an FSM is discrete and for most applications will be required to take finite time. The sequence of reactions, however, may not be finite if the input sequence is not finite. Thus, a model of computation that can include modules refined to an FSM must be capable of supplying an infinite sequence of inputs and requesting an infinite sequence of discrete reactions. This is not a problem for any of the concurrent models of computation being considered.

3.2 Dataflow with FSM

The dataflow model of computation, originally introduced by Dennis [19], can be thought of as a special case of the *process networks* (PN) model, originally introduced by Kahn [28]. Lucid is an early language with dataflow semantics [3][45]. In PN, a network of concurrent processes communicate through unbounded FIFO queues. Formally, a process in a PN network is a prefix-monotonic function that maps a set of potentially infinite input sequences into a set of potentially infinite output sequences [28].

In the DF special case, a process consists of a sequence of discrete, atomic units of computation called *firings*[19]. In DF, a process is often called an *actor*. A denotational formal semantics for Dennis dataflow is given in [29]. Our description here is informal and operational. The DF special case is better suited to our purposes since the discrete firings map naturally into reactions of a slave FSM playing the role of a dataflow actor.

Both DF and PN, however, can easily describe applications that do not terminate, meeting our objective in this regard. For DF (but perhaps not for PN), we can invent a natural definition of an iteration. Specifically, we will define an iteration of a DF graph to be the minimum set of actor firings (greater than zero) that return the FIFO queues to the same size that they were at the beginning of the iteration. Unfortunately, for general dataflow graphs, it is undecidable whether a finite iteration exists [12]. Moreover, there may not be a unique minimum set of actor firings.

To get around these problems, we specialize further to a subclass of dataflow called *synchronous dataflow* (SDF) [31], reviewed below, for which these problems disappear. In

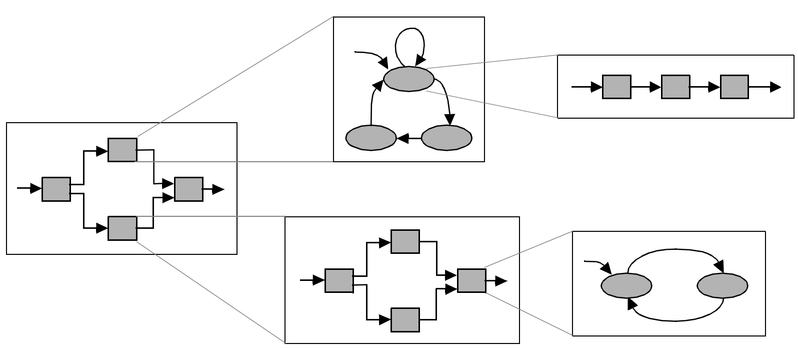

Figure 8. Hierarchical nesting of FSMs with concurrency models.

SDF, each time a dataflow actor fires, it consumes and produces a fixed number of tokens on its input and output FIFO queues. Even for this more restricted model, some interesting fundamental issues arise. We advocate a semantics for combining SDF with FSM that is much more expressive than either SDF or FSM alone, but falls short of the full expressive power of general dataflow. In exchange for this loss in expressiveness, we can ensure that intrinsic properties of a design, like deadlock or bounded memory execution, remain decidable, a very desirable property for embedded systems.

3.2.1 Synchronous Dataflow

Under the SDF model of computation, a system consists of a set of blocks interconnected by directed arcs. The blocks represent functions that map input data into output data. The data are divided into *tokens*, which are treated as atomic (indivisible) units. An arc represents a potentially infinite sequence, or *stream* of tokens. Streams are carried by conceptually unbounded first-in-first-out (FIFO) queues. A *firing* of a block is an atomic computation that *consumes* a fixed number of tokens from each input arc and *produces* a fixed number of tokens on each output arc. The number of tokens consumed and produced on each input and output can be viewed as part of the *type signature* of the actor, along with, of course, the data type of the tokens. These numbers can be used to unambiguously define an *iteration*, or minimal set of firings that return the queues to their original size, as we will now explain [31]. This is done by writing for each arc a *balance equation*,

$$r_i\, p_i = r_j\, c_j\,, \tag{3}$$

where the arc here is assumed to go from actor i to actor j, and on this arc, actor i produces p_i tokens and actor j consumes c_j tokens. The variables r_i and r_j are defined to be the number of firings of actors i and j, respectively. The balance equations tell us that the number of tokens produced by the source actor equals the number of tokens consumed by the destination actor.

Although in general r_i and r_j may be infinite, in which case (3) is trivially satisfied, the balance equations may also be satisfied by a finite number of firings. Indeed, to implement an SDF systems, we seek a finite solution to the balance equations, and then construct a finite schedule where actor i is fired r_i times and data precedences are respected. Such a finite schedule produces and consumes the same number of tokens on each arc, and thus can be iterated indefinitely with bounded resources. Thus, we take the variables r_i to be unknown and attempt to solve the balance equations to find the smallest number (greater than zero) of firings of each actor such that the balance equations are satisfied.

Assuming there are M arcs and N actors, then there will be M equations in N unknowns, r_i, $1 \le i \le N$. It can be shown that for a connected graph, there is either a unique smallest positive solution for the unknowns, called the *minimal solution*, or the only solution is $r_i = 0$, $1 \le i \le N$. When the minimal solution exists, we define an iteration to consist of exactly r_i firings of each actor i. When there is no solution, the SDF graph is considered defective and an error is reported (analogous to a type error a strongly typed language). Thus, for SDF, it is decidable whether an iteration exists. If it does, it is unique, and the firing schedule for an iteration can be determined at compile time.

The simplest SDF graphs are *homogeneous*, defined to mean that every actor produces and consumes a single token on each input and output arc. For such graphs, an iteration always consists of exactly one firing of each actor, $r_i = 1$, $1 \le i \le N$. The schedule of such firings must obey the data precedences (a token must be in a queue before it can be consumed). Thus, to avoid deadlock, all directed cycles in a homogeneous SDF graph must have at least one initial token (often called a *delay*) on at least one arc in the cycle. Arbitrary SDF may require more than one initial token on some arcs, but unlike general dataflow, it is decidable whether a given set of initial tokens is sufficient to prevent deadlock [31].

3.2.2 FSM inside SDF

When an FSM subsystem is a slave to an SDF actor, it must externally obey SDF semantics. Thus it must consume and produce a fixed number of tokens on every input and every output. In the simplest case, the FSM subsystem refines a homogeneous SDF actor. Each input to the SDF actor provides a single data token, which takes on values from some alphabet. The cross product of these signal alphabets forms the input alphabet for the FSM, perfectly matching our FSM model in section 2.2. The actions of the FSM will be able to emit events on each output signal, representing each event by a symbol from the corresponding signal. Any outputs that are not emitted by the FSM in an action will be assigned the default element of the alphabet, as usual.

The only subtlety in this approach is that an “absent” event appears explicitly as a token in the SDF graph, where the value of this token encodes the “absent” interpretation using the default symbol. A simple approach would be to encode presence and absence using boolean-valued tokens. In the other concurrent models of computation, absence of an event will correspond to absence of a token. A key property of dataflow, however, is that absence of a token is not a well-defined, testable condition, so the absence of an event must be encoded in a (present) token.

Consider the example in figure 9, where there are two pure FSMs refining homogeneous SDF actors. An iteration of the SDF graph consists of a single firing of each actor.

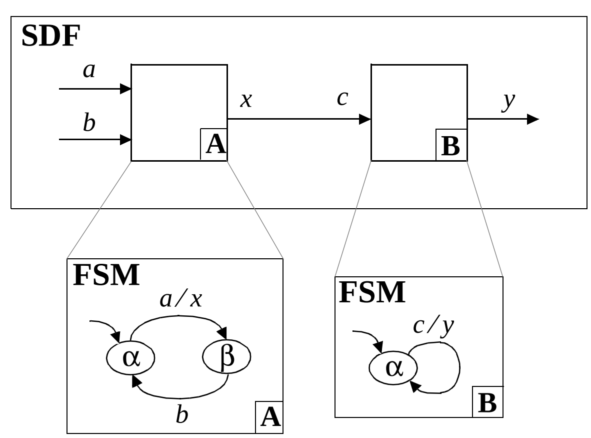

Figure 9. Two FSMs, refining homogeneous SDF blocks, are embedded in an SDF system.

Since there is no initial token on the arc between them, actor **A** fires before actor **B** in the iteration. The names on the arcs ("*a*", "*b*", "*x*", etc.) indicate the names of the nearest input or output of a dataflow actor. Suppose that in some iteration the input tokens have values indicating that *a* is present and *b* is absent, and that both **A** and **B** are in state α. The SDF system reacts as follows:

- fire **A**: Since *a* is present, make the transition from α to β, and let the output *x* be assigned the value indicating it is present.
- fire **B**: Since $c = x$ is present, make the transition back to state α, and let the output *y* be present.

Although this simple example may not look like a concurrent FSM, it is one, in fact. Within an iteration, **A** and **B** must fire sequentially. Across iterations, however, they can fire concurrently. The i^{th} firing of **A** may be concurrent with the $(i - m)^{th}$ firing of **B** for any $m > 0$ such that $i - m > 0$. Of course, with more complicated SDF graphs, there can be much more concurrency, even within an iteration.

We can easily devise a syntax that permits an FSM to refine a non-homogeneous SDF actor. For a non-homogeneous actor (i.e., an actor where more than one token of each input/output can be consumed or produced), we syntactically differentiate each token of a given input or output by concatenating its occurrence to its name. Borrowing notation from the Signal language [6], "*a*" denotes the most recent (last) token consumed from input *a*, "*a*\$1" denotes the next most recent token consumed, and "*a*\$2" the next most recent. Consider the example in figure 10, focusing for now on levels (d) and (e). The numbers in parentheses at level (d) indicate the number of tokens consumed or produced by the corresponding actor. The guard on the arc from α to β in **A** on level (e) is $a \wedge a\$1$, which means that both tokens consumed from the *a* input must have the value representing a present event. In **B**, the action *y* means that the first (oldest) output token on output *y* will have a value representing an absent event (because *y*\$1 is not mentioned), while the second (newest) token on output *y* will have a value representing a present event (because *y* is mentioned).

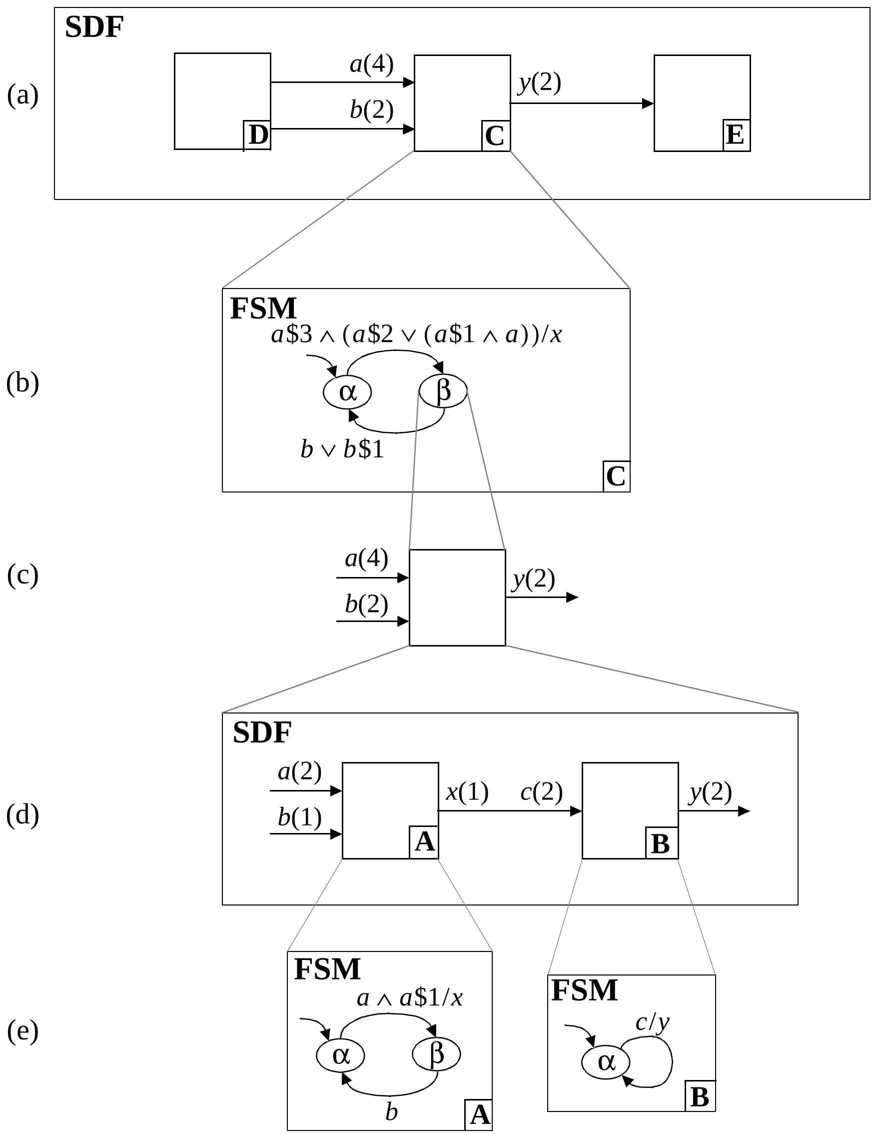

Figure 10. Two FSMs, behaving like multirate blocks, are embedded in an SDF system.

By default, state transitions occur whenever a dataflow actor that refines to an FSM fires. Sometimes, however, we will prefer for transitions to occur only between iterations of the dataflow graph. This will prove important below where states of the FSM may themselves be refined. In this case, there are two types of firings of the dataflow actor that refines to FSM. In type A, no transition is taken and no action is performed, but if the current state is refined, the refinement subsystem is fired. In type B, the refinement system is fired, a transition is taken, and the corresponding action is performed. Type B firings will always be the last of an iteration.

Consider again the example of figure 10, focusing on levels (a) and (b). Suppose that the schedule for the top-level SDF system is {**D**,**C**,**C**,**C**,**E**}. The first two firings of actor **C** are type A firings, where only the subsystem refining the current state (either α or β) is fired. The third firing of **C** is a type B firing, where the refinement system is fired, a transition is taken according to the values of the inputs a and b, and the corresponding action is performed.

The notation described here has an obvious extension to valued FSMs. We leave the details to the reader.

3.2.3 SDF inside FSM

If an SDF graph refines a state of an FSM, when that state is the current state, the next reaction will consist of one iteration of the SDF graph followed by a reaction of the FSM. If the slave SDF graph is homogeneous (consumes a single token from each input and produces a single token on each output), then it fits the FSM model naturally. At each reaction, each input has a symbol from the corresponding signal alphabet. Even if this symbol is interpreted as denoting an absent event, it nonetheless provides a token for the SDF graph to consume.

If the slave SDF graph is not homogeneous, the semantics becomes more subtle. Suppose for example that the SDF subsystem of figure 10(d) is to be used as a slave within another FSM, say the one at level (b). Solving the single balance equation for the subsystem at level (d) (there is only one arc entirely inside the subsystem, and hence only one balance equation) indicates that one iteration will consist of two firings of **A** and one firing of **B**. Thus, as shown at level (c), the type signature for the subsystem indicates that *four* tokens will be consumed from input *a* and *two* from input *b*, and two tokens will be produced at output *y*, in one iteration of the subsystem. The semantics we choose is that the resulting composite SDF type signature becomes the type signature of the FSM subsystem itself. Thus, whatever system the FSM at level (b) is embedded in must treat the FSM like an SDF actor with the given type signature.

There are a number of potential complications. First, composing synchronous dataflow actors to create a new synchronous dataflow actor is not always possible. An example is shown in figure 11. There, if actors A and B are combined to form a synchronous dataflow actor C, the behavior changes. If for example actor C is connected as shown, then the system deadlocks with actor C, but not with actors A and B. This problem can be resolved for general dataflow, which can be made compositional, but not for synchronous dataflow [29]. For the purposes of this paper, we assume that only valid aggregations are specified.

A second complication is that the FSM at level (b) in figure 10 might not be embedded within an SDF environment. Suppose for example that it is embedded within a discrete-event environment. In this case, the semantics must be that of SDF embedded within DE, which is covered in [15]. The key, therefore, is that an FSM that contains slave SDF graphs must itself be treated as an SDF actor with the type signature determined by the slave SDF graphs.

A third complication is that the type signature may not be the same in different states. In this case, the FSM system cannot be treated as an SDF actor because the number of tokens it produces and consumes is dependent on its state. This possibility is extremely interesting, and represents a major increment in expressive power, if it can be handled cleanly. We deal with it in the next subsection.

3.2.4 Heterochronous Dataflow

When an FSM system has more than one state refined to an SDF graph, the simplest case is where the type signatures of the SDF graphs are identical. Then the FSM system itself is treated as an SDF actor with this type signature. Consider however the situation where the type signatures are different. For example, in figure 12, one of the SDF graphs consumes three tokens and produces one, while the other consumes one and produces two. In this case, there are two possible type signatures for the FSM subsystem, and hence it cannot be embedded within an SDF graph.

One option is to embed the FSM system within a dynamic dataflow (DDF) or boolean dataflow (BDF) graph [12]. In DDF and BDF, the number of tokens consumed and produced need not be constant for each actor. However, the price we pay for this approach is high. In DDF and BDF, many questions about the system are undecidable, such as whether it will deadlock and whether the memory required by the FIFO queues is bounded [12]. More importantly, synthesis becomes more difficult and implementations more expensive. Moreover, it seems that this choice of semantics provides more generality than we really need for this application. So we invent a new model of computation that we call *heterochronous dataflow* (HDF).

In HDF, an actor has a finite number of type signatures, where each type signature specifies the number of tokens consumed and produced. When such an actor fires, a well-

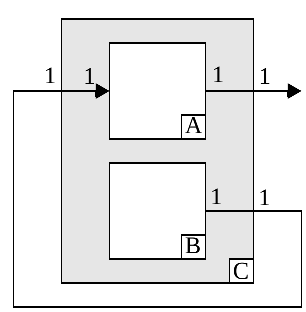

Figure 11. Actor C is not a valid SDF composition of A and B. The actors produce and/or consume a single token on each firing, as suggested by the annotations.

defined type signature is in effect. But type signatures are allowed to change between firings.

This model of computation is related to *cyclo-static dataflow* (CSDF) [11]. In CSDF, an actor cycles through a finite list of type signatures. It is easy to generalize the balance equations so that all such actors complete an integer number of cycles in an iteration of the overall system. Thus, once again, an iteration is finite, and static scheduling is possible. In HDF, however, the order in which type signatures are used is not cyclic, nor even predictable.

If we allow the type signature of an actor to change between any two firings, then it is easy to show that this model of computation has the full expressive power of BDF and DDF, and hence of Turing machines. A more modest generalization is possible by restricting the changes in type signature to occur at more controlled points in the execution.

When an HDF system starts execution, there is an initial type signature in effect for each actor. These type signatures can be used to solve the balance equations, finding an iteration. The semantics we choose for HDF is that each type signature must remain constant for the duration of the corresponding iteration. To ensure this, the FSM components do not change state until their last firing in an iteration. At the completion of the iteration, a new set of type signatures is in effect, so the balance equations must be solved anew to redefine an iteration.

In the example in figure 12, the top of the hierarchy is an HDF system. The middle actor in this system has two possible type signatures, consuming three and producing one or consuming one and producing two. Since this is the only actor refined into an FSM, there are two sets of solutions to the balance equations. Two corresponding sequential schedules are {**A**,**A**,**A**,**B**,**B**,**C**,**C**} and {**A**,**B**,**B**,**C**,**C**,**C**,**C**}. Since state α is the initial state of the FSM, the HDF system starts by executing the first schedule. After the second firing of **B**, the FSM is allowed to change state based on observations of the inputs. At that point, if the two most recent consumed tokens (in the iteration) indicated "present," then the state changes to β. After completion of the HDF iteration, instead of repeating the α schedule, the β schedule is invoked.

There are a number of alternatives for implementing HDF. If the number of possible type signature combinations is small, as for the example in figure 12, it is probably best to precompute (at compile time) all balance equation solutions, and all iteration schedules. Unlike DDF or BDF, it is always theoretically possible to precompute all schedules for all possible iterations. In general, however, the number of type signature combinations is exponential in the number of HDF nodes, so this approach can become impractical. Fortunately, the balance equations can be solved in time that is only linear in the number of arcs plus the number of actors, and a schedule can be found in time that is linear in the number of firings and the number of edges [10], so it may not be impractical to compute schedules dynamically between iterations. We are currently exploring these implementation alternatives.

Although the number of type signature combinations can be exponential in the number of actors, it is finite. For each combination, all key questions are decidable (deadlock, bounded memory), and schedules can be statically constructed. Thus, we have retained the key advantage of SDF (decidability), but have dramatically increased its expres-

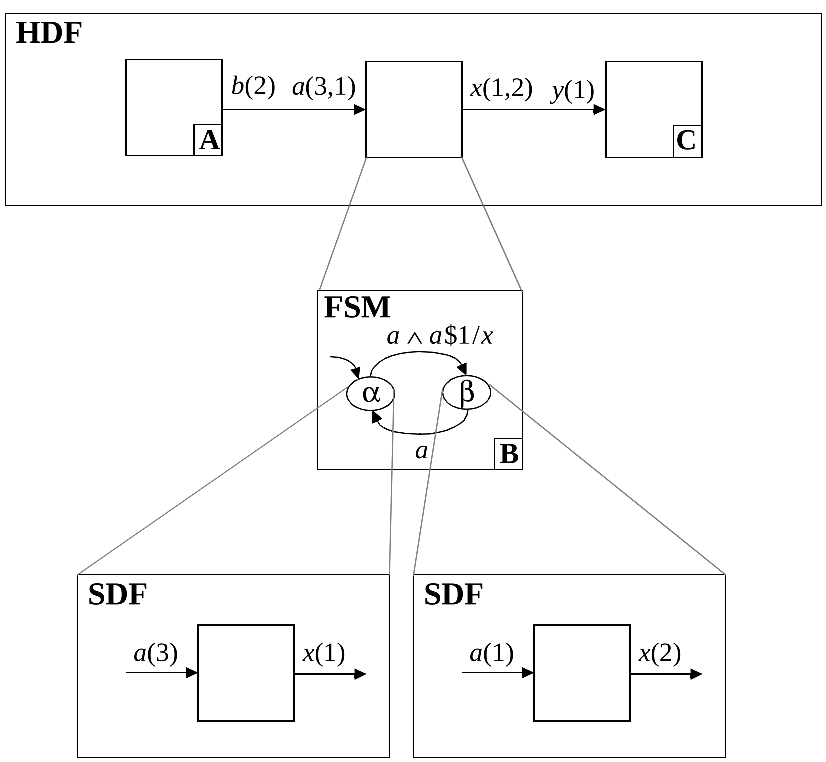

Figure 12. An FSM with states that refine to SDF subsystems with different type signatures.

siveness. However, we can construct designs where not all combinations are reachable. Obviously, we need not worry about scheduling such combinations. But if the language for expressing guards is rich enough, then which combinations are reachable will not be decidable.

HDF has one significant disadvantage. When a state transition occurs depends on a global solution the balance equations, rather than a local definition. This could make using it harder, as it compromises the modularity of a design.

Note that in figure 12, in addition to the type signatures implied by the SDF refinements of the states, there are type signatures implied by the guards on the transitions. The guard $a \wedge a\$1$ implies that there are at least two tokens consumed from input a in one iteration (and that this input is pure). A compiler will have to check that these constraints on the type signature are consistent with the type signature of the refinement of the state from which the arc containing the guard emanates.

3.2.5 Dynamic Dataflow

The dynamic dataflow (DDF) and Boolean dataflow (BDF) models of computation permit actors to consume and produce a variable number of tokens on each firing. This enhancement by itself is sufficient to make the models Turing complete (they can implement a universal Turing machine) [12]. At a fundamental level, these models are therefore much more expressive than SDF or HDF. The price we pay is that deadlock and bounded memory become undecidable and schedules can no longer (always) be constructed at compile time.

To combine FSMs with DDF and BDF, we use the concept of firing rules, formalized in [29]. For the purposes of this paper, these firing rules simply imply that any dataflow actor must assert, prior to any firing, how many tokens it needs on each input[1]. So if an FSM refines a DDF actor, then in each state of the FSM, we must determine how many tokens need to be consumed on each input for the next firing (the next reaction of the FSM).

The semantics we adopt is simple; at least one token is consumed on every input signal mentioned in the guard of any outgoing transition from the current state. If multiple tokens are mentioned for a single signal, using the notation "$a\$i$" for any positive integer i, then for each such signal, we find the largest index i mentioned, and consume that many tokens plus one.

Thus, in each state, we know how many tokens will be consumed at each input in the next reaction. These numbers become the firing rules for the DDF actor refined by the FSM, specifying the number of tokens that must be present on the inputs for the next firing to occur.

The inverse scenario is a bit more complicated. If a DDF graph refines a state of an FSM, then the firing rules of the DDF graph are exported to the environment of the FSM. That is, when the FSM is in the state so refined, the entire FSM becomes a DDF actor that will only be invoked when the firing rules of the DDF subsystem, *treated as an actor,* are met. This seems simple enough, but in fact, most realizations of DDF semantics are not compositional, meaning that a DDF subsystem *cannot be treated as an actor,* and hence cannot have well-defined firing rules. Techniques for making DDF compositional, and for determining the resulting firing rules, are covered in [29], and are beyond the scope of this paper. It is sufficient for our purposes here to know that it can be done.

1. In [29], an actor may also assert what the token values must be. It is a simple exercise to show that omitting this capability does not compromise Turing completeness. Moreover, for reactive FSMs, adding this capability would not increase expressiveness. Thus, we omit it.

3.3 Discrete Events with FSM

Dataflow is a loosely synchronized concurrency model, where events are partially ordered according to their data precedences. Because of this partial ordering of events, many realizations of a dataflow system are possible, so systems are not overspecified. Moreover, it implies a great deal of concurrency, which can be exploited through parallel implementations. However, the resulting loose synchronization is also a key weakness of dataflow. Because of it, dataflow is not well suited for explicitly modeling resource sharing and resource usage. We study, therefore, two popular concurrency models that are more tightly synchronized, DE and SR. The formal relationship among all of these models of computation is studied in [33].

The *discrete-event* (DE) model of computation [14] is particularly useful for modeling distributed or parallel hardware or software and their communication infrastructure. It carries a notion of *global time*, a value, usually a real number, that is known simultaneously throughout the system. An event in a signal occurs at a point in time. In a simulation of such a system, each event carries both a value and a *time stamp* that indicates the time at which the event occurs. The time stamp of an event is typically generated by the actor that produces the event, and is determined by the time stamp of input events and the latency of the block. The DE simulator needs to maintain a global event queue that sorts the events by their time stamps, and chronologically processes each event by sending it to the appropriate actor, which reacts to the event (*fires*).

A formal semantics for DE is given in [30], which also references other formal treatments. The semantics is based on constructing a metric space using the so-called Cantor metric, and defining signals to be elements of this metric space. Causality turns out to be a key property of operators on signals, and can be characterized in terms of contraction mappings in the metric space. Determinacy is ensured if feedback loops contain a contraction mapping.

3.3.1 FSM inside DE

Since the DE model of computation, like dataflow, has well-defined firings, embedding FSM within DE is straightforward from a control perspective. An FSM that refines a DE actor reacts when the DE actor fires, which occurs when there is an event at one of its inputs, and that event has the smallest time stamp of all events in the event queue[2]. If that event has a value, then that value is made available to the FSM for testing by the guards. If the other input signals do not also have events with the same time stamp available for this reaction, then those signals are assigned an input symbol indicating the absence of event. Unlike dataflow, absence of an event is represented in DE with absence of a token.

In a reaction, an FSM that refines a DE actor may emit output events. Those output events translate directly into events in the DE domain. However, in DE, they must be assigned a time stamp, something that the FSM semantics does not provide for. We choose semantics where the FSM system appears to the DE system as a *zero-delay* actor. If an output is generated in a reaction, it is assigned the same time stamp as the input that triggered that reaction.

Consider the example shown in figure 13. Suppose that an event for *a* with a time stamp *t* is the next to be processed in the global event queue, and both FSM **A** and **B** are in state α. Then, the DE system reacts as follow:

- Fire **A**: Since there exists an event for *a*, **A** makes the transition from α to β, and emits the pure event *x*. In DE, this event will have time stamp *t* and thus will be the next to be processed.
- fire **B**: Since there exists an event for *c*, **B** makes the transition back to state α, and emits *y*. In DE, the event on *y* will have time stamps *t*.

Since DE semantics is event-driven, an actor does not fire if there are no events at its inputs. This leads to some subtleties with guards. Consider the example in figure 14, and suppose that the FSM is in state α. The guard on the only outgoing transition indicates that *a* must be absent for the transition to trigger. Implicitly, however, *b* must be present, or the FSM would not react (there would be no event to trigger a firing). Thus, it would be clearer to give the guard as $\neg a \wedge b$. If the guard were given instead as $\neg a \wedge \neg b$, the transition would never fire, since *a* and *b* are the only two inputs and the actor will not fire when both are absent.

3.3.2 DE inside FSM

Much as we did with dataflow, if a state in an FSM refines to a DE subsystem, then the properties of that subsystem are exported to the environment of the FSM. If that environment is not DE, but something else, such as dataflow, then the semantics of DE within dataflow apply [15]. If more than one state of the FSM refines, then all must refine to a DE subsystem, but the semantics imposes no other consistency constraint, as we had to do with SDF[1]

If the environment of an FSM is DE, the semantics is simple. The FSM will react when any of the inputs is present. The input that triggers the firing will have as its time stamp the *current time* of the environment. If the current state refines to a DE subsystem, then that subsystem will be simulated until its current time matches that of the environment. In the meantime, it may emit events, which become outputs to the environment with time stamps equal to the current time (or later).

As is always the case with DE modeling where zero-delay actors are permitted, there can be semantic problems with directed cycles that have zero delay [33]. Consider the example in figure 15. When **A** reacts to an event on *a*, it starts a process by which an event will circulate through the cycle forever with no advance of time. There are a number of solutions to problem, but all of them are intrinsic to DE and not to the DE/FSM combination, and hence are beyond the scope of this paper.

3.4 Synchronous/Reactive Systems with FSM

Even though time is a real number in a DE system, for any well-behaved DE simulation, time in fact advances in discrete steps. Recognizing that, we could instead use a model of computation where only the discrete steps are

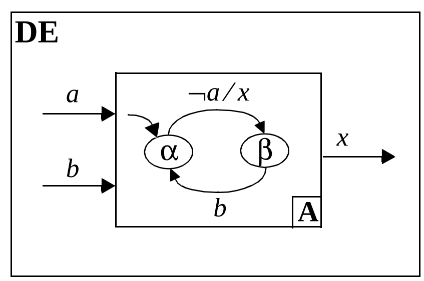

Figure 14. The guard on the upper transition is incomplete, in that event *b* must be present if *a* is absent and the FSM is reacting.

2. There is some ambiguity when there is more than one event in the event queue with the same smallest time stamp. Various DE simulators deal with situation differently. See [15] for a discussion of this issue. For the purposed of this paper, it makes no difference what technique is used.

1. A particular programming environment may impose constraints on the data types of the tokens, but that is not an issue being addressed in this paper.

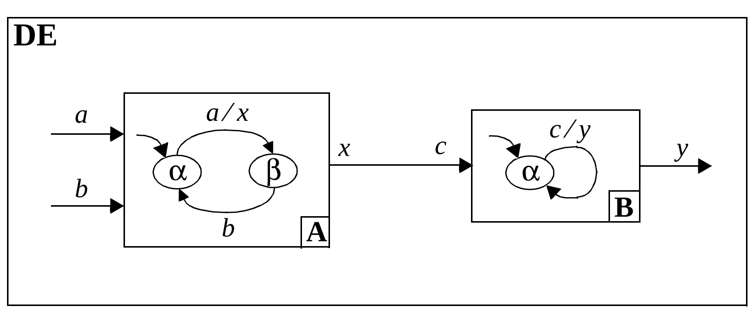

Figure 13. Two FSMs that refine DE actors.

modeled, and not the time continuum. In addition, we can resolve the problem highlighted above with zero-delay feedback loops by adopting a *fixed-point semantics*. With these two innovations, we get the *synchronous/reactive* (SR) model of computation [5]. SR is synchronous in the same sense as synchronous digital circuits. Time delays in computations become irrelevant, so a useful conceptual gimmick is to assume that computations take zero time. SR has a major advantage over DE in that an SR model can be compiled into either sequential code or parallel circuits. DE, in contrast, is difficult to implement efficiently in sequential code, although it is used routinely to specify circuits, which are intrinsically parallel (via the VHDL and Verilog languages).

Execution of an SR system occurs at a sequence of global, discrete, instants called *ticks* (as in ticks of a clock). At each tick, each signal either has no event (is absent) or has an event (is present, possibly with a value). At each tick, signals are related by functions that have signals as arguments and define signals. In general, directed cycles are permitted. I.e., for signals a and b, and functions f and g, we might have

$$a = f(b)$$
$$b = g(a) \tag{4}$$

Thus, at each tick, signals are defined by a set of simultaneous equations using these functions. A solution is called a fixed point, and the task of a compiler is to generate code that will find such a fixed point.

To ensure that the system is deterministic, that the implementation always finds the same solution given the same inputs, each function is required to be *monotonic* in a very particular sense. Suppose that a function f has input signal a with signal alphabet $\{\varepsilon, a_1, a_2, ...\}$. We augment the alphabet with a special symbol $\perp$, pronounced "bottom," that we interpret to mean "unknown." The function must be defined for input $\perp$ (the output will often, but not always be $\perp$). We then define a "flat" partial order on the augmented set, $\{\perp, \varepsilon, a_1, a_2, ...\}$, as shown in figure 16(a). In this diagram, $\perp$ is below ("less than") everything else in the set, and no two other elements in the set are comparable (neither can be less than the other). The function f is monotonic if

$$a \le a' \Rightarrow f(a) \le f(a'), \tag{5}$$

where the symbol "≤" is interpreted with respect to this partial order. The partial order and the notion of a monotonic function is easily generalized to allow functions with multiple arguments. It is then possible to use a fixed point theorem based on the Knaster-Tarski fixed-point theorem to show that any network of such monotonic functions has a least fixed point, where "least" is with respect to this partial order [18]. The least fixed point is taken to be the semantics of the network of functions. This basic approach was pioneered by Scott [42], Manna [34], and others. Many practical implementations of the SR model have been constructed, starting with the Esterel language [8].

Finding the fixed point is straightforward, in principle. The functions are simply evaluated in any order until we converge to a fixed point. Choosing a good order for evaluating the functions can greatly impact performance, obviously. In [20], Edwards proposes and compares several algorithms for choosing a good order of evaluation.

Functions are allowed to change between ticks. Thus, a module in SR has two distinct behaviors that we call *produce* and *transition*. In the produce phase, the current function is evaluated to determine outputs given the current information about the inputs. In the transition phase, the function is changed in preparation for the next tick.

Most familiar functions are *strict*, meaning that all arguments must be known before the function output is defined. Strict functions are always monotonic. A directed loop of strict functions has the solution $\perp$ (unknown) for all signals.

It is not uncommon, however, to have functions where the output can be determined even if some of the inputs are not known. The use of non-strict functions allows directed loops with less trivial solutions. We will see that FSMs can be described as non-strict functions that map input events into output events in each reaction.

3.4.1 Simple FSM Inside SR

Embedding an FSM as an SR module seems straightforward in the following sense. If at a tick the inputs to the FSM are known, then the FSM can react to them and possibly assert output events. Any output events that are not asserted would then be known to be absent. However, there are two difficulties with SR. First, the current state of the FSM may refine to an SR or non-SR subsystem. Second, the inputs may not be completely known. In particular, if the SR system includes a directed loop, then the inputs cannot be known at the start of the tick for all the modules in the loop.

In this subsection, assume the states of the FSM are not refined. Consider the example in figure 17, where there are two FSMs, **A** and **B**, embedded in an SR system and

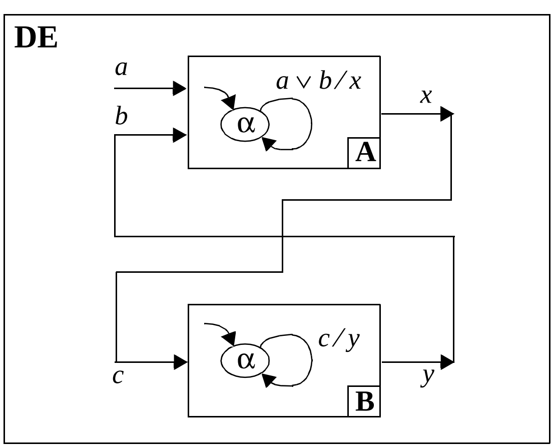

Figure 15. As with all DE modeling, zero-delay loops can

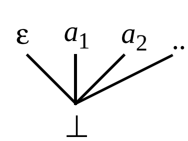

Figure 16. Partial orders used to define SR functions.

enclosed in a directed loop. In **A**, the function mapping the inputs *a*, *b* into the output *x* in state α is

$$f_x^{\alpha}(a, b) = (a \wedge b) \vee (a \wedge \neg b) = a \ . \quad \textbf{(6)}$$

This function does not depend on *b*, so if the FSM is in state α and *a* is observed to be present or absent, then we specify whether *x* will be present or absent without observing *b*. Thus, in state α, the SR function defined by this FSM is not strict. It only needs to observe *a*, not *b*.

The above analysis can be automated to get a simplified function for each output at each state using standard techniques from digital logic design. These simplified functions will indicate for each state what inputs need to be known to define an output.

We then define two phases of execution of an FSM within SR, also called *produce* and *transition*. To complement firing types A and B used for FSM within dataflow, we might call these firing types C and D, respectively. In the produce phase, a type C firing, the FSM observes the inputs and determines whether any output function can be evaluated. If so, it is evaluated so that the output is defined. If not, it indicates that the outputs are still unknown. The produce phase may be invoked any number of times in a single tick, as long as the output functions are monotonic. The transition phase (a type D firing) makes whatever state transition is enabled by the current inputs, but ignores the action associated with that transition.

Thus, a run-time scheduler can sequence through function evaluations, iterating until a fixed point is found. The scheduler executes in three phases (cf. [20]):

1. First, invoke the produce phase for each FSM (and other SR blocks) however many times is needed for it to either define the outputs or reach a fixed point. An algorithm for ordering these invocations is given by Edwards [20].
2. If any signals remain undefined, signal a causality loop error.
3. Invoke the transition function of every FSM in the SR system.

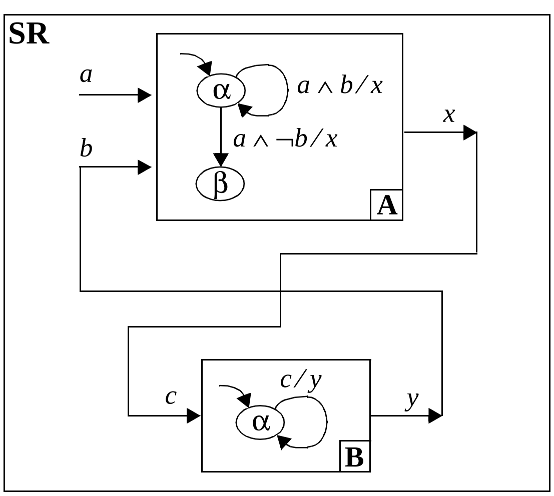

Figure 17. Two FSMs are embedded in an SR system.

The iterative procedure in step (1) may seem costly at first glance, but experience indicates that with intelligent scheduling, convergence to a fixed point is very fast [20]. Moreover, the iterative procedure is amenable to embedding in compiled code, so it does not imply an interpreted execution style. However, causality loops are only detected at run time, and hence can only be reported at run time. This can be a serious impediment to using such a scheme in embedded systems.

3.4.2 Refined FSMs Inside SR

We consider two cases. If the current state of an FSM refines to an SR subsystem, then the produce phase of the FSM should invoke the produce phase of the SR subsystem. No other change is needed. If the FSM refines to non-SR subsystem, then we have to be more cautious. In that case, we assume that the non-SR subsystem defines a strict function, and modify the SR scheduling as follows:

1. Same as above.
2. Look at all FSMs in the SR system where the current state refines to a non-SR subsystem and that subsystem has not fired. If there are non, continue to step 3. Otherwise, if all of these have undefined inputs, then signal a causality loop error. Otherwise, fire all refinements that have all inputs defined and repeat steps 1 and 2.
3. If any signals remain undefined, signal a causality loop error.
4. Invoke the transition function of every FSM in the SR system.

We do not have enough experience with this doubly iterative procedure to know how costly it is. This is future work.

3.4.3 SR inside FSM

Embedding SR systems within FSM is straightforward. If the current state of an FSM refines to an SR subsystem, then the semantics of SR are simply exported to the boundary of the FSM.

4.0 VERIFICATION AND SYNTHESIS

Synthesis of hardware or software from FSMs is standard practice, and has been supported for many years in widely used CAD packages. Synthesis of hardware (e.g. [46]) and software (e.g. [10]) from SDF graphs has been demonstrated. Synthesis of hardware (e.g. [9][41]) and software (e.g. [8]) from SR has also been demonstrated. Given our simple composition semantics, it is not hard to come up with ways to combine independently synthesized components. Although still somewhat limited, such combinations have been demonstrated for embedded software by Edwards [20]. DE is used more for modeling than synthesis, so synthesis is not much of an issue.

Verification of FSMs (reachability analysis and model checking) is well studied. Verification of SDF graphs includes liveness analysis (or conversely, deadlock detection) [31]. Independent analysis is not compromised by our approach. However, verification of an SDF/FSM combination, in general, becomes much more difficult. It is probable that because of fundamental decidability questions, simulation will remain the main validation method for most aspects of the combined system.

One of the advantages of our approach is that it permits the use of established and reasonably mature synthesis and verification technologies within each model of computation, and provides a simple and determinate mechanism for combining the results. The determinacy of the combination ensures that validation of the combination by simulation is practical.

5.0 IMPLEMENTATION

An experimental implementation of several of the combinations discussed here has been implemented in the Ptolemy software environment [13]. The SDF, DE, and SR models were already present in the software, and minimal modifications were required to interface them to FSM. The only significant complication encountered was that, in order to support arbitrary hierarchical combinations of all four models, all four had to have hooks supporting the produce and transition phases of execution required for partial evaluation in SR. For SDF and DE, the "produce" phase does nothing, and the "transition" phase implements a standard firing. Thus, SDF and DE have strict behavior. To get a modular software architecture, the object-oriented principle of polymorphism is used, where the default behavior of a model of computation is strict, but specific models can override this behavior.

6.0 EXAMPLES

6.1 The Reflex Game

A commonly used example for control-intensive software environments is the "reflex game" [7]. Our version of the reflex game is a two-player game (to introduce more concurrency).

6.1.1 Description of the Game

The inputs to the system are *coin*, *ready*, *go*, *stop* and *time*. All but the last are user inputs, while the last simply counts off time. The outputs are *blueLt*, *yellowLt*, *greenLt*, *redLt* and *flashTilt*, used to control a user interface. Normal play proceeds as follows:

1. Either player may assert *coin* to start the game. A status light turns blue.
2. When player 1 is ready, he presses *ready*, and the status light turns yellow.
3. When player 2 presses *go*, the status light turns green and player 1 presses *stop* as fast as he can.
4. The game ends, and the status light turns red.

The game measures the reflexes of player 1 by reporting the time between *greenLt* and *redLt*. There are some situations where the game ends abnormally, and a "tilt" light flashes. These are:

1. After *coin* is asserted, player 1 does not press *ready* within *L* time units.
2. Player 1 presses *stop* before or at the same instant that player 2 presses *go*.
3. After player 2 presses *go*, player 1 does not press *stop* within *L* time units.

One additional rule is that if player 2 does not press *go* within *L* time units after player 1 presses *ready*, then *go* is asserted by the system, and the game advances to wait for player 1 to press *stop*.

6.1.2 Heterogeneous Realization of the Game

Our realization of the game is shown in figure 18, To simulate the real-time behavior of the game, we use DE as the topmost level (a), modeling the environment of the game (including the players). The DE model contains a **clock** to generate time ticks, models of the two players, a **reflex** block modeling the implementation of the game, and a **display** block. It also contains a **merge** block because either player can assert *coin*.

At the next level of the hierarchy (b), inside the **reflex** block, we have a two state FSM. The states are **game off** and **game on**. Inside the **game on** state, at level (c), we use an SR model consisting of the two players. These are interconnected with a zero-delay feedback loop, so we exploit the fixed-point semantics of SR.

At level (d), the two players are refined into concurrent FSMs. Player 1 starts in the **idle** state, and when *ready* is asserted, emits a *start* event and transitions to the **wait go** state. This causes player 2 to transition to the **wait** state and emit a *yellowLt* event. The rest of the behavior at this level should now be evident from the figure.

In several states, we need to count ticks from the clock to watch for time outs. This counting is a simple arithmetic computation that can be performed using the dataflow graph shown at level (e). This graph simply counts ticks, compares the count against a constant, and emits a *timeout* event when the threshold is exceeded.

6.1.3 Esterel Realization

Figure 19 shows an Esterel realization of the two-player reflex game. The description is concise, taking slightly less space than the one in figure 18. This application is a good match for the concurrent semantics of Esterel, which is synchronous/reactive. However, this Esterel module does not include a model of the environment. Esterel programs generally specify modules that are intended to reside within some foreign realization of the environment, such as a C program. There is no support for discrete-event modeling.

The computational aspects of the reflex game, which involve only trivially simple arithmetic, are also a good match for Esterel. For more sophisticated computations, such as signal processing, it is common for Esterel programs to fall back on modules written in C for their implementation. By contrast, in the *charts model, a designer could use dataflow models of the sophisticated computations, which are somewhat higher level (more abstract) than C programs.

Which description, Esterel or *charts, is more readable or understandable will depend heavily on the familiarity of the reader with the languages involved. We believe that the version in figure 18 will be more easily understood in general.

6.1.4 VHDL and C Realizations

Esterel has proven paths to synthesis of both hardware and software [8][9]. Code generation from dataflow graphs for both hardware and software targets has also been demonstrated [10][40], and has appeared in a number of commercial products, such as SPW from Cadence and COSSAP from Synopsys. Synthesis of embedded software for a version of synchronous/reactive semantics that admits heterogeneity has been demonstrated [20]. Synthesis of hardware from FSM models is routine in CAD software, and synthesis of embedded software from FSM models has appeared in commercial products, such as Stateflow from The MathWorks. Thus, all of the elements are in place for synthesis from the *charts heterogeneous model. Nonetheless, we have not yet completed a synthesis tool that

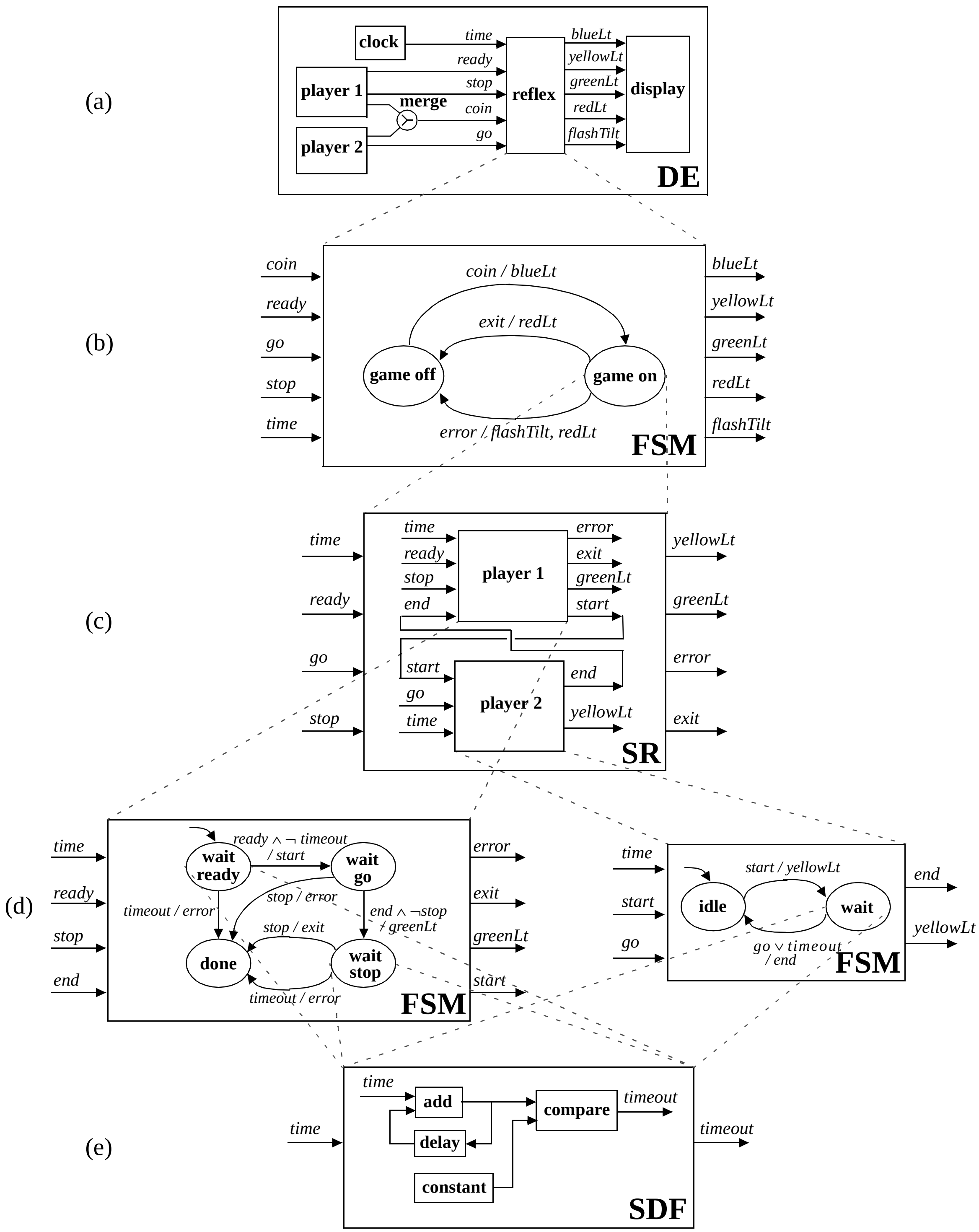

Figure 18. The system of the reflex game can be hierarchically decomposed into five levels of subsystems.

performs the entire task, and we do not wish to imply that this is a trivial task.

Although we have not implemented automatic synthesis for the *charts model, we manually crafted implementations of the two-player reflex game in both (synthesizable) VHDL and C. This code is written in a style similar to what would be generated by a synthesis program. The complete VHDL implementation is shown in figure 20, although not in a readable font. VHDL is a relatively verbose language, and this description, which includes almost no comments, occupies

Figure 20. VHDL description of the two-player Reflex game, with the segment corresponding to level (b) in figure 18 shown in a readable font at the upper right. At the lower right is the C version.

more than five pages, and like the Esterel program, does not model the environment. The C description is somewhat shorter, occupying less than four pages. In figure 20 at the right we show the VHDL and C descriptions of the level (b) FSM from figure 18. The FSM is implemented very directly as if-then-else clauses in both cases. From these code segments, we hope the reader is convinced that the translation from the syntax in figure 18 to this syntax is relatively straightforward. Our conclusion is that C and VHDL should be back-end languages, synthesized from higher level descriptions for the purpose of interfacing to lower-level synthesis tools (compilers and logic synthesis), and that *charts provides a reasonable higher-level description.

```
module REFLEX :

constant L = 10 : integer;

input TIME, READY, STOP, GO, COIN;

output DISPLAY: integer;
output BLUELT, YELLOWLT, GREENLT, REDLT, FLASHTILT;

emit REDLT;
emit DISPLAY(0);

loop
  await COIN;
  emit BLUELT;
  signal START, EXIT, END in
    trap T1 in
      [ % player 1
      var X := 0 : integer in
      do
        await READY
      watching L TIME timeout exit T1 end;
      emit START;
      await
        % the STOP transition has the higher priority
        case STOP do
          exit T1
        case END do
          emit GREENLT
      end await;
        trap T2 in
          [
          do
            await STOP
          watching L TIME timeout exit T1 end;
          exit T2;
          ]
        ||
          [
          every TIME do
            X := X+1
          end every
          ]
        handle T2 do
          emit EXIT;
          emit DISPLAY(X)
        end trap
      end var
      ] % end player 1
    ||
      [ % player 2
        await START;
        emit YELLOWLT;
        await
          case GO do
            emit END
          case L TIME do
            emit END
        end await
      ] % end player 2
    handle T1 do
      emit FLASHTILT;
      emit REDLT
    end trap
  end signal
end loop

end module
```

Figure 19. Esterel realization of the two-player reflex game.

6.2 Digital Cellular Telephone

The reflex game example is rich enough to encompass multiple models of computation, but simple enough to be summarized on a page. A more practical application that exhibits many of the same features is a digital cellular telephone. It includes intensive numerical signal processing (a good match for SDF), in both the speech coder and radio modem. It may also include features such as speech recognition for hands-free dialing. These signal processing components are each quite sophisticated, and may involve modal models that would be appropriately constructed by combining SDF with FSM. For example, equalization of a fading radio channel may involve the use of distinct algorithms during the establishment of a connection vs. steady state. Also, power conservation dictates the use of simpler algorithms when the channel is benign, suggesting that mode changes would be driven by channel estimators.

A cellular phone also includes a substantial amount of embedded control logic for call processing and multiple-access protocols. Time-division multiple access (TDMA), such as that used in GSM phones, requires accurate real-time state transitions. Such protocols can get quite intricate, so the ability to systematically verify correctness of FSM models may become valuable. Even without formal verification, if an FSM model is more easily understood than a C program, then a design constructed in terms of FSMs is more likely to be correct.

Modeling a cellular phone requires modeling its environment, which can itself be quite complex. Multiple-access scenarios, with varying numbers of other users, should be part of the model. Multipath fading should also be modeled, although the level of detail of the model will depend on what question is being asked about the design (a transfer-function-level model would be used to verify the design of the radio modem, while a drop-out-event model would be used to verify the robustness of the protocol implementations). Many of the features of the environment can be conveniently modeled using DE. Detailed modeling of multipath fading is well-suited to SDF.

A cellular telephone also contains analog RF circuitry, adding a further element of heterogeneity beyond any we have discussed in this paper. Mixed-signal models that include FSMs, DE, and SDF are an active area of research. Commercial systems have already appeared that support subsets of these, such as Saber from Analogy, which models DE systems together with continuous-time systems, and HP Ptolemy, which models SDF systems together with continuous-time systems.

Finally, a cellular phone development project is a multi-team effort, and coordination of the diverse tasks of the teams is a major challenge. In practice, cellular telephone design efforts use a heterogeneous set of tools and methodologies. Different techniques are used for each of the embedded DSP software, the embedded microcontroller software, the custom digital hardware, and the analog and RF hardware. We believe that the *charts model provides a good framework for coordination of such efforts.

7.0 CONCLUSIONS

We have described the combination of finite-state machines with three different concurrency models, dataflow, synchronous/reactive (SR) systems, and discrete-event (DE) systems. These three concurrency models have different strengths and weaknesses, and are thus applicable in different situations. Dataflow (Section 3.2) is well-suited to numerical computation, such as signal processing, but poorly suited to resource management and control logic. SR (Section 3.3) is well-suited to resource management and control logic, but overspecifies numerical computational systems by imposing synchrony. DE (Section 3.4) is well-suited to modeling hardware systems, but poorly suited to more abstract specifications because of its physical notion of time. Finite-state machines complement all three of these with sequential control that is easily analyzed and synthesized. We have given semantics for each concurrent model of computation combined with FSM.

An example is described that uses all four models of computation. The resulting combination is easily understood by anyone familiar with all four models of computation, but obviously would be obtuse to someone familiar with only a subset. This particular example was chosen precisely to illustrate our claim for heterogeneity and for multiples models of computation. However, most designs of similar complexity would only require a subset of the four models of computation.

There are many issues that are not discussed in this paper. These include enhancements that are possible in FSM, for example to support preemptive transitions, where the refinement of a state is not fired prior to taking the transition. Another issue that is not dealt with is what should be done with the state of a refinement of a state of an FSM. It is possible to support a "history entry," where entering a state starts the refinement subsystem in whatever state it was last in.

8.0 REFERENCES

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The Algorithmic Analysis of Hybrid Systems," *Theoretical Computer Science*, Vol. 138, No. 1, pp. 3-34, February 1995.

[2] C. André and M.-A. Péraldi, "Effective Implementation of Esterel Programs," in *Proc. of the Fifth Euromicro Workshop on Real-Time Systems*, Oulu, Finland, June1993.

[3] E. A. Ashcroft, "Proving Assertions about Parallel Programs," *J. of Computer and Systems Science*, Vol. 10, No. 1, pp. 110-135, 1975.

[4] F. Belina, D. Hogrefe, A. Sarma, *SDL with Applications from Protocol Specification*, Prentice Hall International (UK), Hemel Hempstead, 1991.

[5] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1270-1282.

[6] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Tr. on Automatic Control*, Vol. 35, No. 5, pp. 525-546, May 1990.

[7] R. Bernhard, G Berry, F. Boussinot, R. de Simone, G. Gonthier, A. Ressouche, J. P. Rigault, J. M. Tanzi, "Programming a Reflex Game in Esterel V3," Rapport de Recherche No. 07/91, INRIA, Sophia-Antipolis, France, June, 1991.

[8] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, 19(2):87-152, 1992.

[9] G. Berry, "A hardware implementation of pure Esterel," *Sadhana, Academy Proceedings in Engineering Sciences*, Indian Academy of Sciences, 1992, vol. 17, no 1, pp. 95-130.

[10] S. S. Bhattacharyya, P. K. Murthy and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, Norwell, Mass, 1996.

[11] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete, "Cyclo-Static Dataflow," *IEEE Transactions on Signal Processing*, 44(2):397-408, February 1996.

[12] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Tech. Report UCB/ERL 93/69, **Ph.D. Dissertation**, Dept. of EECS, University of California, Berkeley, CA 94720, 1993.

[13] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int. Journal of Computer Simulation*, special issue on "Simulation Software Development," vol. 4, pp. 155-182, April, 1994. (http://ptolemy.eecs.berkeley.edu/papers/JEurSim).

[14] C. Cassandras, *Discrete Event Systems, Modeling and Performance Analysis*, Irwin, Homewood IL, 1993.

[15] W.-T. Chang, S.-H. Ha, and E. A. Lee, "Heterogeneous Simulation — Mixing Discrete-Event Models with Dataflow," invited paper, to appear, RASSP special issue of the *Journal on VLSI Signal Processing*, January, 1997. (http://ptolemy.eecs.berkeley.edu/papers/96/heterogeneity)

[16] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, "Hardware-Software Codesign of Embedded Systems," *IEEE Micro*, August 1994, pp.26-36.

[17] R. Cleveland, S. A. Smalka, et al., "Strategic Directions in Concurrency Research," *ACM Computing Surveys*, Vol. 28, No. 4, December 1996.

[18] B. A. Davey and H. A. Priestly, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[19] J. B. Dennis, "First Version Data Flow Procedure Language", Technical Memo MAC TM61, May, 1975, MIT Laboratory for Computer Science.

[20] S. A. Edwards, *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*, Ph.D. dissertation, UCB/ERL M97/31, Department of EECS, University of California at Berkeley, Berkeley, California, May 1997.

[21] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993.

[22] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1305-1319.

[23] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program.*, vol 8, pp. 231-274, 1987.

[24] T. A. Henzinger, "The Theory of Hybrid Automata," *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pp. 278-292.

[25] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[26] J. Hopcroft and J. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison-Wesley Publishing Company, 1979.

[27] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond, "A Multiparadigm Language for Reactive Systems," *Proceedings of the 1994 IEEE International Conference on Computer Languages (ICCL '94)*, pp. 211-218, Toulouse, France, May 16-19, 1994.

[28] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

[29] E. A. Lee, "A Denotational Semantics for Dataflow with Firing," Memorandum UCB/ERL M97/3, Electronics Research Laboratory, U. C. Berkeley, January 1997.

[30] E. A. Lee, "Modeling Concurrent Real-time Processes Using Discrete Events," Invited paper to *Annals of Software Engineering*, Special Volume on Real-Time Software Engineering, to appear, 1998. Also UCB/ERL Memorandum M98/7, March 4th 1998.

[31] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, January, 1987.

[32] E. A. Lee and T. M. Parks, "Dataflow Process Networks,", *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-801, May, 1995. (http://ptolemy.eecs.berkeley.edu/papers/processNets)

[33] E. A. Lee and A. Sangiovanni-Vincentelli, "A Denotational Framework for Comparing Models of Computation," ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, January 30, 1997. (http://ptolemy.eecs.berkeley.edu/papers/97/denotational/).

[34] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill, Inc., 1974.

[35] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.

[36] 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, Oct. 1991.

[37] F. Maraninchi, "Operational and compositional semantics of synchronous automaton compositions," In *CONCUR '92, Third International Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 550-564, Stony Brook, NY, August 1992. Springer-Verlag.

[38] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes, I," *Information and Computation*, Vol. 100, No. 1, September 1992.

[39] S. Narayan, F. Vahid, D. D. Gajski, "SpecCharts: A Language for System Level Specification and Synthesis", Proc. of Int. Symp. on Computer Hardware Description Languages, Marseille, April 1991.

[40] M. Pankert, O. Mauss, S. Ritz, and H. Meyr, "Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis," *Proceedings of the 1994 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Vol. 2, pp. 449-452, Adelaide, Australia, April 19-22, 1994.

[41] F. Rocheteau and N. Halbwachs, "Implementing reactive programs on circuits: A hardware implementation of LUSTRE," In *Real-Time, Theory in Practice*, REX Workshop Proceedings, Mook, Netherlands, June 3-7, 1991, volume 600 of *Lecture Notes in Computer Science*, pages 195-208, Springer-Verlag, 1992.

[42] D. Scott, "Outline of a mathematical theory of computation", *Proc. of the 4th annual Princeton conf. on Information sciences and systems*, 1970, 169-176.

[43] F. Vahid, S. Narayan, D. D. Gajski, "SpecCharts: A VHDL Front-End for Embedded Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 6, pp. 694-706, June 1995.

[44] M. von der Beeck, "A Comparison of Statecharts Variants," in Proc. of Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863, pp. 128-148, Springer-Verlag, Berlin, 1994.

[45] W. W. Wadge and E. A. Ashcroft, *Lucid, the dataflow programming language*, London Academic Press, 1985.

[46] P. Zepter and T. Grötker, "Abstract Multirate Dynamic Data-Flow Graph Specification for High Throughput Communication Link ASICs", submitted to the *IEEE VLSI DSP Workshop*, The Netherlands, 1993.

Response to review

The paper needs another round of revisions. The big picture is seems to be that there are already a lot of models around. To be truly valuable, a greater effort needs to be made to demonstrate, through examples, that this model is better than others.

We have augmented the detail in both key examples of the paper, the reflex game, in response to reviewer #1's comments. However, we do not believe that the above assessment accurately summarizes the situation. In fact, this paper presents a systematic way to amalgamate existing models, thus it is presenting a solution to the problem that "there are a lot of models around." It takes as a premise that these models are around because they are useful.

Only one reviewer remains to be satisfied. Please address his or her comments very carefully in the next revision.

We have done that.

Review Number 1.

You have improved the paper considerably - kudos! However, I've recommended to the associate editor that more concreteness be imparted to the reflex-game example and/or the cellular phone example to make the value of the paper significantly go up.

In particular,

- add more meat to the cellular phone example (a page more perhaps)

- work out the reflex-game example to some level of detail.

This example does help tie together many of the ideas presented to that point - but does not "deliver" in terms of showing the advantage of the method in any of the axes (design description, verification, or synthesis)

So how about if you tell us what could potentially happen to the reflex-game example after having described it. Show how one might generate more concrete VHDL descriptions for portions of it. Show how one might generate C or reactive-software code for it. Actually carry the exercise down to nuts and bolts in your lab, but report only the high-level gist of your experiments.

Seeing concretely how two boxes described at two different - but compatible in your framework - representations turn into actual C-code/HDL-code/circuits and how they interface concretely would be a detail worth learning.

Taking a page or two more to detail the reflex-game example thus will be a valuable addition to your work. (If you have done all this already, simply report it!)

We have elaborated on the cell phone example, and provided much more detail on the reflex game example. We are more convinced than ever that the reflex game is an ideal example for our purposes. We have included an Esterel description of the game, which is concise and elegant, but, we believe, considerably harder to understand than the *charts model. We have also constructed implementations of the game in both VHDL and C, and we show a single module of each of these (see figure 20). These code segments hopefully also make concrete what the path to synthesis is. Although we do not (yet) have automated synthesis from the *charts model, it does not take much imagination to see how these code segments could be generated automatically.

We agree, by the way, that it is a major improvement to the paper to include these details.

One minor request: if it is possible to receive the next revision of the paper in single-space and possibly also in two-column, it will be far easier for me to read it.

OK.

Specific comments:

Page 4: first developed...: I'd prefer replacing with "developed" - CS ideas tend to be discovered and re-discovered...

We are not convinced, but have yielded anyway. We do believe the credit for this innovation goes to Berry. Even if the germ of the idea had already appeared (which we doubt), the Esterel language was certainly the first full-scale embodiment of the idea. Nonetheless, we have changed the wording as you request.

Page 9: Add I.e. before |\Sigma|

Done.

Page 22: I'm sorry but the descriptions of type A and B are still not clear. Can you provide (or refer to, if already there) an example?

We have added an example:

"Consider again the example of figure 10, focusing on levels (a) and (b). Suppose that the schedule for the top-level SDF system is {DCCCE}. The first two firings of actor C are type A firings, where only the subsystem refining the current state (either \alpha\ or \beta) is fired. The third firing of C is a type B firing, where the refinement system is fired, a transition is taken according to the values of the inputs a and b, and the corresponding action is performed."

> *Page 35: So far as I know, Manna [33] is not a pioneering work in the sense of discovery - but surely in terms of pedagogy. So retain Scott, and cite Strachey if you need a second (more deserving) name (thus representing either side of the Atlantic as is regarded as polite!)*

This is correct. We have modified the reference accordingly.

> *Edward gives technique for choosing [19]. Very awkward.*

Fixed.

> *Page 39: Synthesis of hardware or software from FSMs is easy.*
>
> *Very casual (and meaningless) sentence. Replace "easy" by something more precise.*

We now say: "Synthesis of hardware or software from FSMs is standard practice, and has been supported for many years in widely used CAD packages." Figure 20 also shows what synthesized hardware and software might look like, in the form of synthesizable VHDL and C code.

> *Page 40: last line. reporting the time between ready and stop.*
>
> *Must it not be go and stop? See clause 3 on page 40*

Correct. Fixed.

> *Page 43: Cellular phone section peters out. Was this more fully*
>
> *worked out elsewhere? What a great example it would be!*

We agree, and have developed it further. However, it still remains speculative. A full-scale development would require tens or hundreds of person years. Cell phones are surprisingly sophisticated devices.

> *Page 43: after last line. So what's recommended? (Don't end in a negative or "hopeless" note.)*

We have modified this paragraph to read:

"An example is described that uses all four models of computation. The resulting combination is easily understood by anyone familiar with all four models of computation, but obviously would be obtuse to someone familiar with only a subset. This particular example was chosen precisely to illustrate our claim for heterogeneity and for multiples models of computation. However, most designs of similar complexity would only require a subset of the four models of computation."