SPECIFICATION AND DESIGN OF REACTIVE SYSTEMS

by

Bilung Lee

Memorandum No. UCB/ERL M00/29

15 May 2000

SPECIFICATION AND DESIGN OF REACTIVE SYSTEMS

by

Bilung Lee

Memorandum No. UCB/ERL M00/29

15 May 2000

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Specification and Design of Reactive Systems

by

Bilung Lee

Doctor of Philosophy in Engineering – Electrical Engineering
and Computer Sciences

University of California, Berkeley

Professor Edward A. Lee, Chair

Reactive systems maintain an ongoing interaction with their environment, and respond to inputs from the environment by possibly sending outputs to it. Most importantly, in most reactive systems, the functionality performed in response to inputs is no longer limited to data manipulation and numeric computation. Instead, a sizable portion of effort is often required to focus on control logic that manages the sequencing of operating tasks and the switching among various modes in consecutive interaction. For example, a digital cellular phone may contain a signal processing part including speech compression and decompression. In addition, it also needs to include a substantial amount of control logic for call processing and multiple-access protocols.

Finite state machines (FSMs) have long been used to describe and analyze intricate control logic. Due to their finite nature, FSMs yield better to analysis and synthesis than alternative control models, such as sequential programs with if-then-else and switch-case constructs. For example, with an FSM, a designer can enumerate the set of reachable

states to assure that a safety property is satisfied such that a particular set of dangerous states will never be reached. However, in large systems, the control functionality can become so complex that the flat, sequential FSM model becomes impractical because a very large number of states and transitions are required.

Hierarchical concurrent FSMs (HCFSMs) dramatically increase the usefulness of FSMs by extending them with structuring and communicating mechanisms. Hierarchy allows a state of an FSM to be refined into another FSM, i.e. a set of sub-states. Concurrency allows multiple simultaneously active states, each refined as an FSM, to communicate through messaging of some sort. However, most models that support HCFSMs tightly integrate one concurrency semantics with the FSM semantics. This prevents the designers from choosing among various concurrency semantics the one that is best applicable to the problem at hand. Moreover, like the basic FSM model, HCFSMs are good for describing control logic, but not for intensive data computation. Hence, they are not enough by themselves for the complete design of a system with both sophisticated control and intensive computation.

In fact, we observe that the FSM and the concurrency semantics in HCFSMs can be orthogonal properties. Moreover, the concurrent states of HCFSMs are actually the syntactic shorthand for interconnection of FSMs in a concurrency model. Therefore, in this thesis, we advocate decoupling the concurrency semantics from the FSM semantics. After equipping the basic FSM with hierarchy and heterogeneity, a hierarchical combination of FSMs with various concurrency models becomes feasible. We call this heterogeneous combination *charts (pronounced "starcharts"). *charts do not define a single concurrency semantics but rather show how FSMs interact with various concurrency models without ambiguities. This enables selection of the most appropriate concurrency model for the problem at hand. In particular, computation-oriented models, such as dataflow models, can be included to complement the FSMs. Using *charts, systems can truly be built up from

modular components that are separately designed, and each subsystem can be designed using the best suited model to it.

Professor Edward A. Lee, Chair Date

謹獻給我的父母

To my parents

Contents

Acknowledgements vi

1 Introduction 1

1.1 Reactive Systems 1
1.1.1 Concurrency 2
1.1.2 Control Logic 2
1.2 Finite State Machines 2
1.3 Hierarchical Concurrent Finite State Machines 3
1.4 Our Scheme: *charts 6
1.5 Related Work 9
1.5.1 Statecharts 9
1.5.2 Argos 10
1.5.3 Mini-Statecharts 11
1.5.4 SpecCharts 11
1.5.5 Specification and Description Language 12
1.6 Summary 12

2 Finite State Machines (FSMs) 14

2.1 The Basic FSM 14
2.2 Multiple Inputs and Multiple Outputs 16
2.3 The Pure FSM 17
2.4 The Valued FSM 19
2.5 Hierarchy 25
2.6 Heterogeneity 27
2.7 Hierarchical Entries and Exits 29
2.8 Shared Slaves 31
2.9 Local Events 32
2.10 Initial Transitions and Conditional Initial States 35
2.11 Instantaneous Transitions and States 37
2.12 Simulation Algorithm 38

3 Integration with Concurrency Models 40

3.1 Hierarchical Combination 40
3.2 Discrete Events 42
3.2.1 FSM inside DE 44
3.2.2 DE inside FSM 47
3.3 Synchronous/Reactive Models 50
3.3.1 FSM inside SR 53
3.3.2 SR inside FSM 57
3.4 Synchronous Dataflow 57
3.4.1 FSM inside SDF 60
3.4.2 SDF inside FSM 63

3.5 Dynamic Dataflow 65
3.5.1 FSM inside DDF 65
3.5.2 DDF inside FSM 66
3.6 Communicating Sequential Processes 67
3.6.1 FSM inside CSP 70
3.6.2 CSP inside FSM 71

4 Applications 72

4.1 Embedded Systems 72
4.1.1 Example: Digital Watches 73
4.1.1.1 Problem Description 73
4.1.1.2 *charts Realization 75
4.1.2 Example: Railroad Controllers 80
4.1.2.1 Problem Description 80
4.1.2.2 *charts Realization 81
4.2 Image Processing 85
4.2.1 Example: Run-Length Coding 86
4.2.1.1 Problem Description 86
4.2.1.2 *charts Realization 86
4.3 Communication Protocols 88
4.3.1 Example: Two-Phase Commit Protocol 90
4.3.1.1 Problem Description 90
4.3.1.2 *charts Realization 92
4.3.2 Example: Alternating Bit Protocol 96
4.3.2.1 Problem Description 96
4.3.2.2 *charts Realization 97
4.4 Linear Hybrid Systems 99
4.4.1 Example: Water-Level Monitors 100
4.4.1.1 Problem Description 100
4.4.1.2 *charts Realization 101
4.4.1.3 Simulation with Fixed Time Increments 103
4.4.1.4 Simulation with Varied Time Increments 104
4.5 Comparison 106
4.5.1 Problem Description 107
4.5.2 *charts Realization 108
4.5.3 Esterel Realization 108
4.5.4 VHDL and C Realizations 111

5 Conclusion 113

5.1 Open Issues 115
5.1.1 Software/Hardware Synthesis 115
5.1.2 Formal Semantics 116

Bibliography 117

Acknowledgements

I am greatly appreciative and thankful to my advisor, Professor Edward Lee, for his continuous support and guidance of this work. His experience and vision in both theoretical and practical research has been a constant source of inspiration to me.

I would also like to thank Professor David Messerschmitt and Professor Ilan Adler for serving on my dissertation committee. Their careful review and useful suggestions are highly appreciated.

Many colleagues share with me not only their technical knowledge but also their personal experience. Our stimulating discussions in the office and interesting talks at lunch get-togethers enrich my life at Berkeley. I would like to gratefully acknowledge them, including Wan-teh Chang, John Davis, Bart Kienhuis, Yuhong Xiong, Jie Liu, and Stephen Neuendorffer.

Finally, I want to express my deepest gratitude to my loving family, brothers Chang and Ming, sister-in-law Mei-Rong, and most importantly, my parents. I am indebted to them for all their support, encouragement, patience, and belief in me. This work is dedicated to them.

1

Introduction

1.1 Reactive Systems

Complex systems can be generally categorized into three classes: *transformational*, *interactive*, and *reactive* systems [46][16]. Transformational systems operate on inputs available at the beginning and stop after delivering outputs at the end. Most traditional computing programs fall into this category. In contrast, interactive and reactive systems typically do not terminate (unless they fail). These systems maintain an ongoing series of interactions with their environment. Interactive systems constantly operate on inputs when the systems are ready, and deliver outputs when the systems are willing to. Examples of interactive systems include operating systems and multimedia network applications. This thesis is concerned with reactive systems, such as embedded systems, real-time systems, and many software systems. Unlike interactive systems that interact at their own speed, reactive systems follow a pace dictated by the environment. The environment determines when the systems must react and provides inputs. The systems respond to the inputs by possibly sending outputs to the environment.

1.1.1 Concurrency

An essential feature of reactive systems is concurrency [67]. Most reactive systems contain multiple simultaneous components and modules that interact with one another through messaging of some sort. Concurrency greatly complicates the design of reactive systems. Fortunately, many models that support concurrency have been proposed [29] and can be utilized to facilitate the system design. Examples of concurrency models include communicating sequential processes [48], dataflow [34], discrete events [7], Petri nets [74], process networks [54], and the synchronous/reactive model [10]. These models conceptually define the rules of interaction among concurrent components and modules of a system. Therefore, the designers can validate the correctness of intended concurrent behaviors for a system specified and simulated with the concurrency models.

1.1.2 Control Logic

For most reactive systems, there is a clear distinction between control logic and data computation [17]. In other words, the functionality performed in response to inputs is no longer limited to data manipulation and numerical calculation. Instead, a sizable portion of effort is often required to focus on control logic that manages the sequencing of operating tasks and the switching among various modes in consecutive interaction. For instance, a digital cellular phone may contain a signal processing part including speech compression and decompression. In addition, it also needs to include a substantial amount of control logic for call processing and multiple-access protocols.

1.2 Finite State Machines

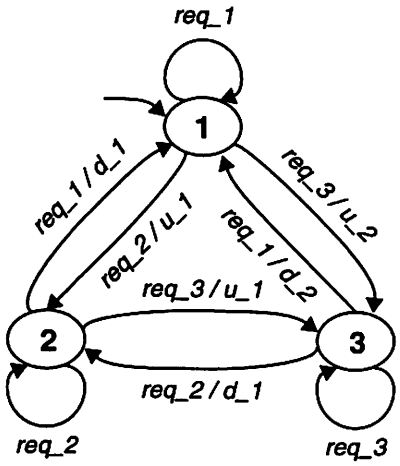
Finite state machines (FSMs) have long been used to describe and analyze sequential control logic. Visually, FSMs are represented in the form of nodes denoting states and arcs

denoting transitions. In figure 1.1(a), an FSM is used to specify the controller of an elevator in a three-story building. Compared with the specification using plain English (see figure 1.1(b)) and C code (see figure 1.1(c)), the FSM is more effective in conveying information about the control behavior of the system. Furthermore, due to their finite nature, FSMs yield better to analysis and synthesis than alternative control flow models, such as flowcharts [32][81]. For example, with an FSM, a designer can enumerate the set of reachable states to assure that a safety property is satisfied such that a particular set of dangerous states will never be reached. However, in large systems, the control functionality can become so complex that the flat, sequential FSM model becomes impractical because a very large number of states and transitions are required.

1.3 Hierarchical Concurrent Finite State Machines

Hierarchical concurrent FSMs (HCFSMs) dramatically increase the usefulness of FSMs by extending them with structuring and communicating mechanisms. Hierarchy allows a state of an FSM to be refined into another FSM, i.e. a set of sub-states. Concurrency allows multiple active states, each refined as an FSM, to operate simultaneously and to communicate with one another. Figure 1.2(a) illustrates a HCFSM that specifies a three-bit counter with initialization and interruption mechanisms. In this figure, the **Counting** state is decomposed into three concurrent states, **A**, **B**, and **C**, each of them further refined into two states. Compared with the specification in a flat FSM model (see figure 1.2(b)), the HCFSM model can reduce the complexity of state and transition spaces because of its concurrency and hierarchy.

A popular and seminal representative of the HCFSM model was introduced as the Statecharts formalism [43]. Since then, a number of variants have been explored [86], including the Argos language [69]. The Statecharts formalism and most of its variants



(a) Finite state machine

If the elevator is on the floor 1 and the floor requested is the floor 1, then the elevator remains on the floor 1.
If the elevator is on the floor 1 and the floor requested is the floor 2, then the elevator is raised up 1 floor.
If the elevator is on the floor 1 and the floor requested is the floor 3, then the elevator is raised up 2 floors.

If the elevator is on the floor 2 and the floor requested is the floor 1, then the elevator is lowered down 1 floor.
If the elevator is on the floor 2 and the floor requested is the floor 2, then the elevator remains on the floor 2.
If the elevator is on the floor 2 and the floor requested is the floor 3, then the elevator is raised up 1 floor.

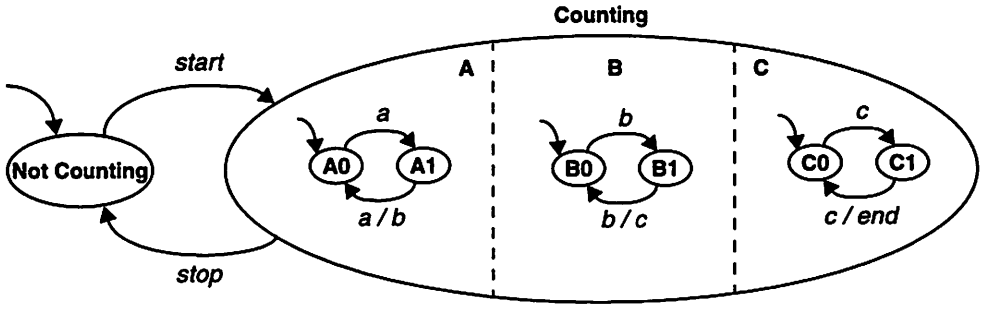
If the elevator is on the floor 3 and the floor requested is the floor 1, then the elevator is lowered down 2 floors.
If the elevator is on the floor 3 and the floor requested is the floor 2, then the elevator is lowered down 1 floor.
If the elevator is on the floor 3 and the floor requested is the floor 3, then the elevator remains on the floor 3.

(b) Plain English

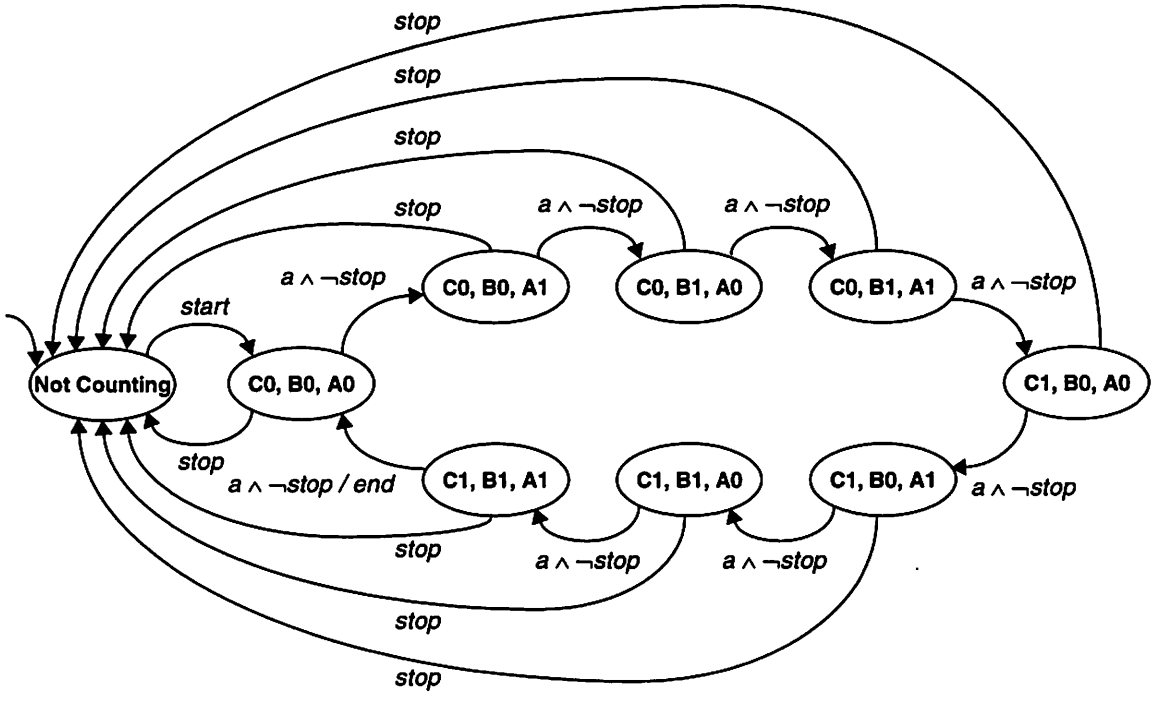
```
while (1) {
  switch (cur) {
    case 1:
      if (req_1) {
      u_1=0; d_1=0; u_2=0; d_2=0; cur=1;
      } else if (req_2) {
      u_1=1; d_1=0; u_2=0; d_2=0; cur=2;
      } else if (req_3) {
      u_1=0; d_1=0; u_2=1; d_2=0; cur=3;
      }
      break;
    case 2:
      if (req_1) {
      u_1=0; d_1=1; u_2=0; d_2=0; cur=1;
      } else if (req_2) {
      u_1=0; d_1=0; u_2=0; d_2=0; cur=2;
      } else if (req_3) {
      u_1=1; d_1=0; u_2=0; d_2=0; cur=3;
      }
      break;
    case 3:
      if (req_1) {
      u_1=0; d_1=0; u_2=0; d_2=1; cur=1;
      } else if (req==2) {
      u_1=0; d_1=1; u_2=0; d_2=0; cur=2;
      } else if (req==3) {
      u_1=0; d_1=0; u_2=0; d_2=0; cur=3;
      }
      break;
  }
}
```

(c) C code

Figure 1.1 Specification of an elevator controller by three different approaches.



(a) HCFSM specification



(b) FSM specification

Figure 1.2 Specification of a three-bit counter with initialization and interruption mechanisms by the HCFSM and the FSM models.

have concurrency semantics based on the synchronous/reactive model. Nevertheless, many schemes have evolved to adopt concurrency models that are significantly different from that of Statecharts. Examples include codesign finite state machines (CFSMs) [28][6] with a discrete-event model, the process coordination calculus (PCC) [39][73] with a dataflow model, the SpecCharts language [70][71] with a discrete-event model, and the specification and description language (SDL) [83][9] with a process network model.

In general, HCFSMs have two major weaknesses. First, the concurrency semantics within the HCFSM model is tightly integrated with the FSM semantics. As a consequence, most formalisms that support HCFSMs only provide a build-in and fixed concurrency semantics. However, while the von Neumann model has introduced a widely accepted semantics for sequential behaviors, no universal model has yet emerged for concurrent behaviors [59]. Thus, using those formalisms will prevent the designers from choosing among various concurrency semantics the one that is best applicable to the problem at hand.

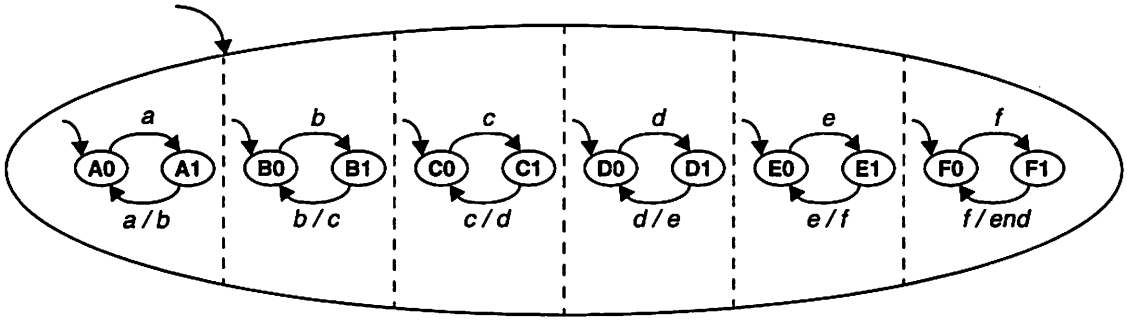
Furthermore, similar to basic FSMs, HCFSMs are awkward for expressing data computation, even though they are well-suited for control logic. For example, a HCFSM in figure 1.3(a) describes a simple six-bit counter. However, the same system behavior can be more elegantly specified by a dataflow graph depicted in figure 1.3(b). Hence, the HCFSM model is not enough by itself to effectively complete the design of a system with both sophisticated control and intensive computation.

1.4 Our Scheme: *charts

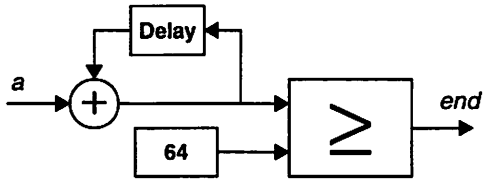
In fact, we observe that FSM, concurrency and hierarchy can be orthogonal semantic properties in the HCFSM model. The FSM semantics specifies the sequential behavior of a system in the form of states and transitions. The concurrency semantics specifies the

interaction between multiple simultaneous states, each refined as an FSM. The hierarchy semantics specifies the interaction between a state and the refining FSM in that state. Moreover, the concurrent states in a HCFSM (see figure 1.4(a)) are actually syntactic shorthand for interconnection of FSMs communicating within a concurrency model (see figure 1.4(b)). This means that the HCFSM model can be considered as the hierarchical combination of the FSM and the concurrency models.

Therefore, we advocate decoupling the concurrency semantics from the FSM semantics. After equipping the basic FSM with hierarchy and heterogeneity, the hierarchical combination of FSMs with various concurrency models becomes feasible. We call this



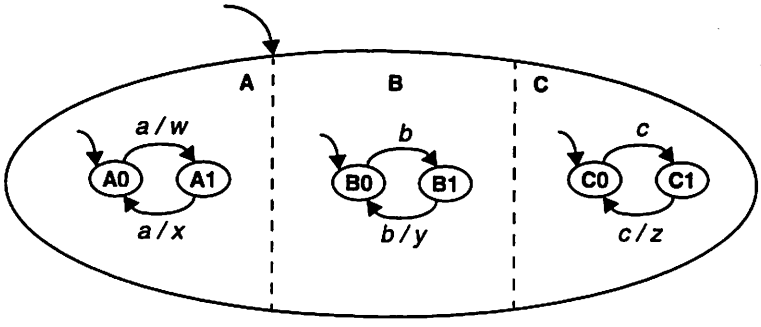
(a) HCFSM specification



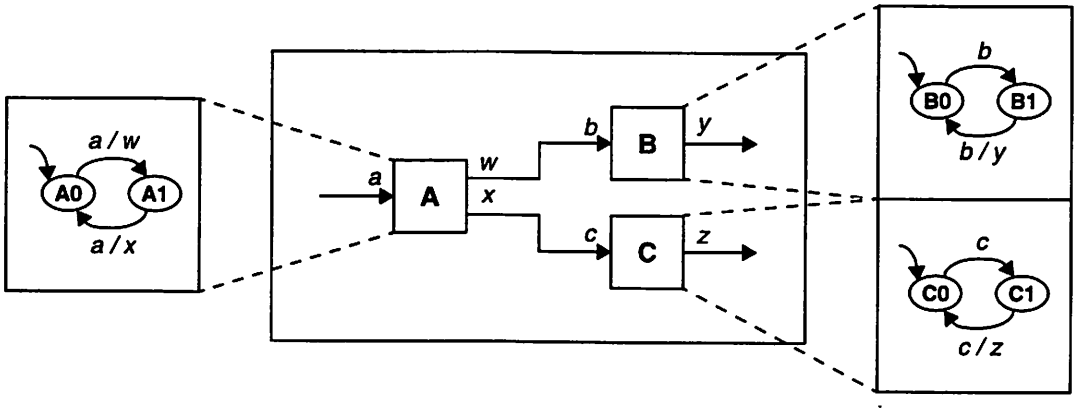
(b) Dataflow specification

Figure 1.3 Specification of a six-bit counter by the HCFSM and the dataflow models.

heterogeneous combination *charts[1] (pronounced "starcharts"). In *charts, we do not define a single concurrency semantics but rather show how FSMs interact with various concurrency models without ambiguities. This enables selection of the most appropriate concurrency model for the problem at hand. In particular, computation-oriented models,



(a) Concurrent states in a HCFSM



(b) Interconnection of FSMs in a concurrency model

Figure 1.4 The concurrent states in a HCFSM are actually syntactic shorthand for interconnection of FSMs in a concurrency model.

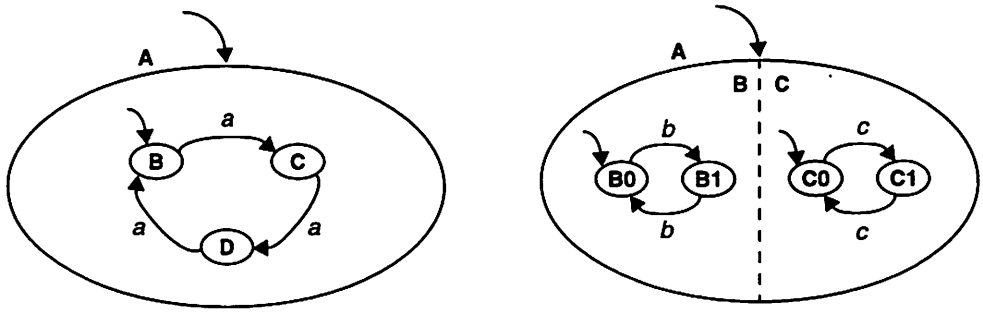
1. The asterisk is meant to suggest a wildcard, which stands for an arbitrary number of concurrency models that can be integrated with the FSM model.

such as dataflow models, can be included to complement the FSMs. Using *charts, systems can truly be built up from modular components that are separately designed, and each subsystem can be designed using the best suited model to it.

1.5 Related Work

1.5.1 Statecharts

The Statecharts [43] formalism was developed mainly for specification of control-dominated reactive systems. In Statecharts, FSMs can be hierarchically and concurrently combined through two major innovations. First, a state of an FSM can be decomposed into a set of sequential substates, i.e. another FSM. For example, figure 1.5(a) depicts a state **A** refined into three sequential substates **B**, **C**, and **D**. The state **A** is called an OR state, since being in state **A** has the interpretation of being in one of its substates **B**, **C**, or **D**. Second, a state of an FSM can be decomposed into a set of concurrent substates, each of them further refined as an FSM. For example, in figure 1.5(b), a state **A** is refined into two concurrent substates **B** and **C**, each of them further refined into two substates. The state **A** is



(a) OR state (b) AND state

Figure 1.5 Examples of an OR state and an AND state in Statecharts.

called an AND state, since being in state **A** is interpreted as being in both of its immediate substates **B** and **C** simultaneously.

The Statecharts formalism has become very successful due to its compact and intuitive representation of FSMs. However, in Statecharts, there exist problems for some syntactic constructs, such as the inter-level transitions that are considered by many as violating modularity in hierarchical design. In addition, there are questions left open about some semantics, such as the behavior of parallel transitions in cyclic dependency. Hence, those issues concerning syntax and semantics give rise to the proliferation of at least twenty-one variants of Statecharts [86].

1.5.2 Argos

The Argos language [69][68] is perhaps the best known one among those Statecharts variants that attempt to solve the syntactic and semantic problems. It basically follows the approaches pioneered by Statecharts to graphically represent HCFSMs. Nonetheless, Argos improves the syntax and semantics of Statecharts in several ways. In Argos, to allow the modular design of complex systems, transitions between states are confined within the same level of hierarchy. Moreover, the Statecharts formalism loosely defines that concurrent states communicate through broadcasting mechanism, where a transition can emit an event that is immediately broadcast to the entire system. The Argos language further applies fixed point semantics formally to concurrency, and thus resolves the causality problems resulting from cyclic dependency among concurrent states. In addition, local events can be used in Argos to reduce the scope of broadcast to a specific subsystem instead of the whole system.

1.5.3 Mini-Statecharts

Another interesting Statecharts variant is the Mini-Statecharts formalism [76][77]. Using the approach adopted by Argos, Mini-Statecharts removes some syntactic constructs, such as inter-level transitions, from Statecharts in order to enable system design in a fully modular way. However, unlike Argos, in which communication among concurrent states is always interpreted with fixed point semantics, Mini-Statecharts supports three types of feedback operators, each of them providing a different communication mechanism. Under an instantaneous feedback, an event emitted by a transition is broadcast at the same time instant as the event that causes the transition, and the execution of parallel transitions involves finding a fixed point for all events at the given instant of time. A microstep feedback also allows an emitted event to be broadcast instantaneously, but the execution of parallel transitions follows a natural causal order. Finally, an emitted event using a delayed feedback will be broadcast at the next time instant instead of the same time instant.

1.5.4 SpecCharts

The SpecCharts language [70][71] is based on the program-state machine model [36][84], where imperative program constructs are allowed to reside in the leaf states of HCFSMs. The HCFSM description supported by SpecCharts has a concurrency semantics that is fundamentally different from that common in the Statecharts family. The SpecCharts language is created as an extension of VHDL [5], and thus exhibits the same concurrency semantics as that of VHDL, which is essentially a discrete-event model. Another feature in SpecCharts is that two types of transition arcs are differentiated for exiting from hierarchical states. A transition-on-completion arc will be taken only when the subsystem of the source state has completed its specified behaviors. A transition-immediately arc will

be taken immediately regardless of whether the specified behaviors in the subsystem of the source state have been finished or not.

1.5.5 Specification and Description Language

The specification and description language (SDL) [83] was standardized by CCITT (International Telegraph and Telephone Consultative Committee), and is mainly known for its applications in protocol specification [9]. A system described in SDL consists of a number of FSMs communicating in a process network. In contrast to other schemes that support HCFSMs, SDL permits FSMs only in the leaf components of the hierarchy, and thus has limited compositionality. Moreover, in an SDL specification, each FSM is treated as a process attached with an infinite-size buffer, and the network of concurrent processes communicate through those buffers with blocking read, non-blocking write mechanism.

1.6 Summary

Most modern reactive systems have both intricate control and sophisticated concurrency requirements. Thus, combining FSMs with concurrency models is an attractive and increasingly popular approach to design. Since the Statecharts formalism was introduced, a number of variants have been explored. The Argos language, for example, combines FSMs with a synchronous/reactive model. Many researchers have combined FSMs with concurrent models that are significantly different from that of Statecharts. The SpecCharts language combines a discrete-event model with FSMs. The specification and description language combines a process network model with FSMs. Codesign finite state machines combine a discrete-event model with FSMs. The process coordination calculus combines a dataflow model with FSMs. All of these examples, however, tightly intertwine the concurrency model with FSMs. In addition, most of them focus on specification of intricate control logic and thus are not good for specification of intensive data computation.

In this thesis, we advocate decoupling the concurrency model from the FSM model. We describe a family of models, called *charts. Unlike other hierarchical concurrent FSM models, *charts does not define a single concurrency model, but rather shows how to embed FSMs within a variety of concurrency models. Thus, designers can choose the best suited concurrency model for the problem at hand. Moreover, each model has its own strengths and weakness, our *charts formalism allows the designers to mix and match various models such that different models can complement one another. For example, if dataflow models are included, they can be used to specify computation-intensive parts of the system and thus can complement the weakness of FSMs.

In the chapters that follow, we will present the details of our *charts formalism, which hierarchically nests FSMs with various concurrency models. In chapter 2, 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. In chapter 3, we consider combining FSMs with five popular concurrency models: discrete events (DE), the synchronous/reactive model (SR), synchronous dataflow (SDF), dynamic dataflow (DDF), and communicating sequential processes (CSP). In chapter 4, we demonstrate how to apply *charts in practical applications. Finally, in chapter 5, we summarize the results and discuss open areas for future work.

2

Finite State Machines (FSMs)

2.1 The Basic FSM

An FSM is a five tuple [51]

$$(Q, \Sigma, \Delta, \sigma, q_0)$$

where

- Q is a finite set of states.
- Σ is an input alphabet, consisting of a set of input symbols.
- Δ is an output alphabet, consisting of a set of output symbols.
- σ is a transition function, mapping $Q \times \Sigma$ to $Q \times \Delta$.
- $q_0 \in Q$ denotes the initial state.

In one *reaction*, the FSM maps a current state $\alpha \in Q$ and an input symbol $u \in \Sigma$ to a next state $\beta \in Q$ and an output symbol $v \in \Delta$, where $\sigma(\alpha, u) = (\beta, v)$. Given an initial state and a sequence of input symbols, a *trace* (a sequence of reactions) will produce a sequence of states and a sequence of output symbols. All traces are potentially infinite.

A directed graph, called a *state transition diagram*, is popular for describing an FSM. As shown in figure 2.1, each elliptic node represents a state and each arc represents a tran-

sition. Each transition is labeled by "*guard / action*", where *guard* $\in \Sigma$ and *action* $\in \Delta$. The arc without a source state points to the initial state, i.e. state α. During one reaction of the FSM, one transition is taken, 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 taken transition and produces the output symbol indicated by the action of the taken transition.

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 ensure all our FSMs to be reactive but not to complicate the notation, every state is assumed to have an *implicit self transition* (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.

For example, figure 2.1 describes an FSM with $Q = \{\alpha, \beta\}, \Sigma = \{u, v\}, \Delta = \{\varepsilon, x, y\}, q_0 = \alpha$ and σ: $Q \times \Sigma \rightarrow Q \times \Delta$. Moreover, in addition to the two explicit transitions $\sigma(\alpha, v) = (\beta, y)$ and $\sigma(\beta, u) = (\alpha, x)$, we also must have the implicit self transitions $\sigma(\alpha, u) = (\alpha, \varepsilon)$ and $\sigma(\beta, v) = (\beta, \varepsilon)$. A possible trace of this FSM is shown in figure 2.2.

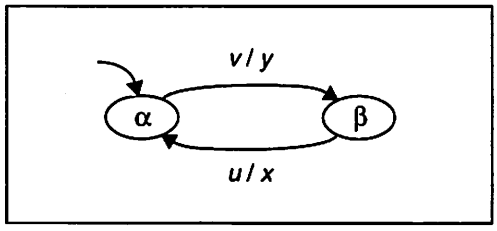


Figure 2.1 A basic FSM.

2.2 Multiple Inputs and Multiple Outputs

An FSM is embedded in an environment. The environment may be part of an overall system under design, or may be out of the designer's control. In either case, the environment provides a sequence of input symbols, and the FSM reacts by producing a sequence of output symbols, meanwhile tracing a sequence of states. At each reaction, the FSM responds to a single input symbol from the environment and produces a single output symbol to the environment. However, the interaction of the FSM with the environment frequently needs to be modeled in more detail. It may not be convenient to consider the FSM to have only a single input and a single output, as shown in figure 2.3(a). Instead, multiple inputs and multiple outputs may be a more natural model.

Current State	α	α	β	β	...
Input Symbol	*u*	*v*	*v*	*u*	...
Next State	α	β	β	α	...
Output Symbol	ε	*y*	ε	*x*	...

Figure 2.2 A possible trace for the basic FSM in figure 2.1.

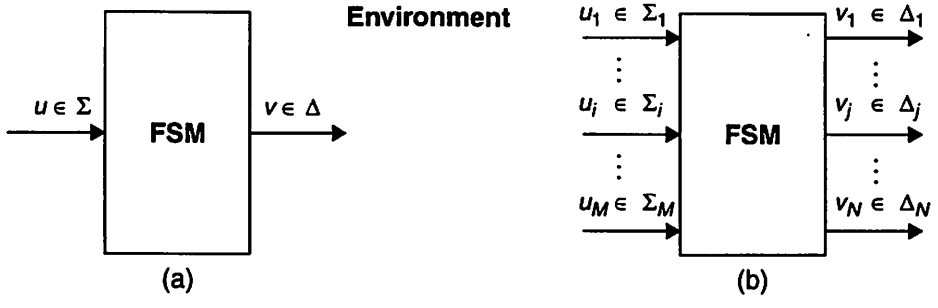


Figure 2.3 Interaction of the FSM with the environment: (a) single input and single output, and (b) multiple inputs and multiple outputs.

To handle this, the input alphabet Σ can be factored and expressed as a cartesian product $\Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_M$. Hence, at a reaction, the input to the FSM consists of M *events*, where the ith event carries a symbol from the *event alphabet* Σ_i. The output alphabet can be similarly factored and expressed as $\Delta = \Delta_1 \times \Delta_2 \times \ldots \times \Delta_N$. Hence, at a reaction, the output from the FSM consists of N events. At each reaction, the FSM responds to a set of M events from the environment, and produces a set of N events to the environment, as shown in figure 2.3(b).

2.3 The Pure FSM

A common special case is the *pure FSM*, where both the input and the output alphabets have size power of two (i.e. size(Σ) = 2^M and size(Δ) = 2^N), and the size of each event alphabet is two (i.e. size(Σ_i) = 2 for $1 \le i \le M$, and size(Δ_j) = 2 for $1 \le j \le N$). This is interpreted to mean that at a reaction, each event is either *absent* or *present* (hence, size(Σ_i) = 2 and size(Δ_j) = 2).

Instead of the input alphabet Σ and the output alphabet Δ, a pure FSM is specified with the input events I and the output events O, where each event is denoted by a name. Nevertheless, the input and the output alphabets of the pure FSM can still be deduced from the given input and output events, respectively. For example, consider a pure FSM with two input events $I = \{u, v\}$ and two output events $O = \{x, y\}$. The event alphabet for u is denoted by $\Sigma_1 = \{\bar{u}, u\}$, which means that $\{u$ is absent, u is present$\}$, and the event alphabet for v is denoted by $\Sigma_2 = \{\bar{v}, v\}$. Thus, the input alphabet Σ is equal to $\Sigma_1 \times \Sigma_2$, i.e.

$\{\overline{uv}, \bar{u}v, u\bar{v}, uv\}$[1]. The output alphabet Δ can be similarly deduced and is $\{\overline{xy}, \bar{x}y, x\bar{y}, xy\}$, where $\overline{xy}$ (i.e. absent for both events) is considered the default output symbol ε.

In a pure FSM, the size of the input alphabet grows exponentially with the number of input events. Hence, for describing a pure FSM, it can easily require a very large number of transitions if from every state an explicit outgoing transition is specified for every input symbol. To alleviate this problem, the guard of a transition can be a subset of the input alphabet instead of a single input symbol. This allows us to compactly represent as a single transition an ensemble of transitions that have the same action. Usually, a subset of the input alphabet can be more briefly denoted by a boolean expression in the input events. For example, for an input alphabet $\Sigma = \{\overline{uv}, \bar{u}v, u\bar{v}, uv\}$, the boolean expression "$\neg u \vee v$" (**not** u **or** v) represents the subset $\{\overline{uv}, \bar{u}v, uv\}$. Therefore, for the pure FSM, the guards will be represented as boolean expressions in the input events. Figure 2.4 illustrates a pure FSM with states $Q = \{\alpha, \beta\}$ and input events $I = \{u, v\}$. The guard "$\neg u \vee v$" of the transition from state α to state β is enabled by any input symbol in $\{\overline{uv}, \bar{u}v, uv\}$. The guard "$u$" of the transition from state β to state β is enabled by any input symbol in $\{u\bar{v}, uv\}$.

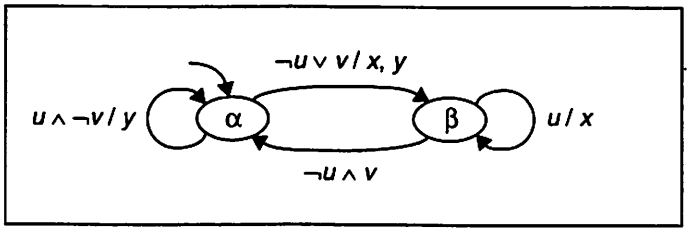


Figure 2.4 A pure FSM.

1. The standard notation for an element of the cartesian product is a tuple, such as $(a, b) \in A \times B$ where $a \in A$ and $b \in B$. Here we use a more compact notation to denote the element, i.e. ab instead of (a, b). This notation is highly compact when we want to form a complex cartesian product in a later section.

The action of the transition in a pure FSM is also specified in a compact notation. It only lists output events that are present at the reaction in which this transition is taken. All other output events that are not explicitly emitted in the action are implied absent at the reaction. Consider the example of figure 2.4 with output events $O = \{x, y\}$. The action "x" of the transition from state β to state β implies the output symbol $x\bar{y}$. Thus, the output event x is present and the output event y is absent when this transition is taken. If all output events are implicitly absent for a transition, the action is omitted together with the slash before it. For the example in figure 2.4, the label of the transition from state β to state α is just the guard "$\neg u \wedge v$" (**not** u **and** v), and when this transition is taken, both output events x and y are implicitly absent.

Figure 2.5 shows a possible trace for the pure FSM of figure 2.4. Note that in state β, when both input events u and v are absent, an implicit self transition is taken and thus both output events x and y are absent.

2.4 The Valued FSM

The *valued FSM* is an augmentation of the pure FSM with arithmetic handling. In a valued FSM, the input and the output alphabets are again factored into event alphabets.

Current State	α	α	β	β	β	...
u	*present*	*absent*	*present*	*absent*	*absent*	...
v	*absent*	*absent*	*present*	*absent*	*present*	...
Next State	α	β	β	β	α	...
x	*absent*	*present*	*present*	*absent*	*absent*	...
y	*present*	*present*	*absent*	*absent*	*absent*	...

Figure 2.5 A possible trace for the pure FSM in figure 2.4.

Some event alphabets may have size two like in the pure FSM. Most importantly, however, each of the remaining event alphabets is further factored and expressed as a cartesian product of two alphabets, a *status alphabet* and a *value alphabet*. The status alphabets of these event alphabets all have size two. This is again interpreted to mean that at a reaction, each of these events is either absent or present. The sizes of the value alphabets may be different in different event alphabets, and they might even be infinite. Each element of the value alphabet is interpreted to represent

- a value that the event carries, when the event is present at a reaction,
- a default value, when the event is absent at the first reaction,
- or a value that is required to be the same as at the previous reaction, when the event is absent at a reaction.

An event is called a *pure event* if it corresponds to an event alphabet with size two. Otherwise, it is called a *valued event* if it corresponds to an event alphabet that is further factored. Note that the pure FSM is simply a degenerate case of the valued FSM, consisting of only pure events.

For example, consider a valued FSM with two input events $I = \{u, v\}$. Suppose that u is a valued event with a value alphabet denoted by $\{(0), (1)\}$[1] and v is a pure event. Thus, the event alphabet for u is $\Sigma_1 = \{\bar{u}(0), \bar{u}(1), u(0), u(1)\}$, which means that {$u$ is absent but with the previous value 0, u is absent but with the previous value 1, u is present with a value 0, u is present with a value 1}, and the event alphabet for v is simply $\Sigma_2 = \{\bar{v}, v\}$. As a result, the input alphabet of the valued FSM is $\Sigma = \{\bar{u}(0)\bar{v}, \bar{u}(0)v, \bar{u}(1)\bar{v}, \bar{u}(1)v, u(0)\bar{v}, u(0)v, u(1)\bar{v}, u(1)v\}$.

1. The parentheses around each value serve a special purpose: They can clearly distinguish the status and the value for an event in the resulting input and output alphabets denoted in our compact notation.

While the pure event carries only a *status* (either absent or present), the valued event carries both a status and a value. This may cause ambiguity when the guard of a transition is specified as a boolean expression that contains valued events, since it is unclear whether the boolean expression operates on the statuses or the values of valued events. For example, if $\Sigma_1 = \{\bar{u}(0), \bar{u}(1), u(0), u(1)\}$, $\Sigma_2 = \{\bar{v}(0), \bar{v}(1), v(0), v(1)\}$ and $\Sigma = \Sigma_1 \times \Sigma_2$, a guard "$\neg u \wedge v$" may represent the subset $\{\bar{u}(0)v(0), \bar{u}(0)v(1), \bar{u}(1)v(0), \bar{u}(1)v(1)\}$ based on the statuses of the events or the subset $\{\bar{u}(0)\bar{v}(1), \bar{u}(0)v(1), u(0)\bar{v}(1), u(0)v(1)\}$ based on the values of the events. One solution is to introduce new operators that explicitly refer to the status and the value of an event, respectively. In Esterel [18], for example, "present u" refers to the status of the event u, and "?u" refers to the value.

However, we prefer another solution that does not introduce new operators to complicate the notation. Similar to Statemate [45], we further refine the guard of the transition into two parts and denote the guard as "*trigger* [*condition*]", where *trigger* and *condition* are the boolean expressions in the input events. To distinguish references to the status and the value of an event, we define that an event is referred to by status if it appears in the trigger of a guard or by value if it appears in the condition of a guard. Note that a pure event must not appear in the condition of a guard since it does not carry a value. The subset of Σ represented by the guard is defined as the intersection of the two subsets of Σ represented by the trigger and the condition. If a transition does not depend on the values of the input events at all, the condition of the guard is omitted together with the brackets around it. In this case, the subset of Σ represented by the guard is simply the subset of Σ represented by the trigger. Similarly, the trigger of a guard can be omitted. However, we consider omission of the trigger for a transition to serve a special purpose, which will be described in section 2.11.

Consider a valued FSM depicted in figure 2.6, which has states $Q = \{\alpha, \beta\}$ and input events $I = \{u, v\}$. Suppose that $\Sigma_1 = \{\bar{u}(0), \bar{u}(1), u(0), u(1)\}$ and $\Sigma_2 = \{\bar{v}(0), \bar{v}(1), v(0),$

$v(1)\}$. The trigger "$\neg u \wedge v$" of the guard for the transition from state β to state α represents the subset $\{\bar{u}(0)v(0), \bar{u}(0)v(1), \bar{u}(1)v(0), \bar{u}(1)v(1)\}$. The condition "$u == 1 \wedge v != 1$"[1] (*u* **equal to** 1 **and** *v* **not equal to** 1) of the same guard represents the subset $\{\bar{u}(1)\bar{v}(0), \bar{u}(1)v(0), u(1)\bar{v}(0), u(1)v(0)\}$. Thus, the guard "$\neg u \wedge v$ [$u == 1 \wedge v != 1$]" is enabled by the input symbol in $\{\bar{u}(1)v(0)\}$. The guard of the transition from state α to state α is just the trigger "$u \wedge \neg v$". Thus, this guard is enabled by any input symbol in $\{u(0)\bar{v}(0), u(0)\bar{v}(1), u(1)\bar{v}(0), u(1)\bar{v}(1)\}$, which is the subset of Σ represented by the trigger.

In a valued FSM, in addition to listing the output events to be emitted, the action of the transition is augmented with value assignment, which allows each emitted event to be assigned a value. Note that a pure event must not be assigned a value since it does not carry a value. In the valued FSM of figure 2.6, suppose that $O = \{x, y\}$, $\Delta_1 = \{\bar{x}(0), \bar{x}(1), x(0), x(1)\}$, and $\Delta_2 = \{\bar{y}(0), \bar{y}(1), y(0), y(1)\}$. The action "$x(1), y(0)$" of the transition from state β to state β represents the output symbol $x(1)y(0)$. Thus, the output events x and y are present with the values 1 and 0, respectively, when this transition is taken. The value assigned to the emitted event of an action is not limited to a constant. It can be an expression in the input events that are referred to by value. Thus, a number of transitions may be compactly represented by a single transition that has an action with expression assign-

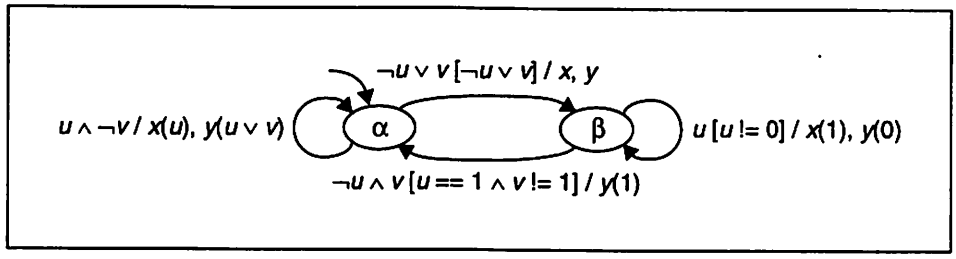


Figure 2.6 A valued FSM.

1. We adopt the same operator precedence as in the C++ programming language. Thus, the operator == and != have higher precedence than the operator ∧.

ment. Consider the example in figure 2.6. The action "$x(u)$, $y(u \vee v)$" of the transition from state α to state α may represent the output symbol $x(0)y(0)$, $x(0)y(1)$, or $x(1)y(1)$ depending on the current values of the input events u and v. If the valued event emitted in an action is not assigned a value, it is considered to have a default value, chosen from the corresponding value alphabet. In figure 2.6, the action "x, y" of the transition from state α to state β represents the output symbol $x(0)y(0)$, assuming that the default values for the output events x and y both are the value 0. Same as in the pure FSM, the output events that are not explicitly emitted are implied absent. However, one subtlety is that the resulting output symbol depends on the previous values of those absent events. For the example in figure 2.6, the action "$y(1)$" of the transition from state β to state α may represent the output symbol $\bar{x}(0)y(1)$ or $\bar{x}(1)y(1)$ depending on the previous value of output event x.

A possible trace for the valued FSM of figure 2.6 is shown in figure 2.7. Note that if an input or output event is absent at a reaction, its value must be the same as at the previous reaction, as required by the interpretation of the value alphabet.

Current State	α	β	β	α	α	...
u (status)	*present*	*present*	*absent*	*present*	*absent*	...
u (value)	0	1	1	0	0	...
v (status)	*present*	*absent*	*present*	*absent*	*absent*	...
v (value)	1	1	0	0	0	...
Next State	β	β	α	α	β	...
x (status)	*present*	*present*	*absent*	*present*	*present*	...
x (value)	0	1	1	0	0	...
y (status)	*present*	*present*	*present*	*present*	*present*	...
y (value)	0	0	1	0	0	...

Figure 2.7 A possible trace for the valued FSM in figure 2.6.

As mentioned earlier, the size of the value alphabet for a valued event can be infinite. This is one key feature for the valued FSM to be more expressive than the pure FSM. For example, in a valued FSM, suppose that the value alphabet for an input event u is $\aleph$ (the set of natural numbers). Thus, a transition with the guard "u [u != 0]" compactly represents an infinite number of transitions that have the same action. Moreover, suppose that an output event x also has $\aleph$ as its value alphabet. Thus, a transition with the action "$x(u)$" compactly represents an infinite number of transitions that have the same guard. We cannot have the same expressiveness as shown in these two scenarios by just using the pure FSM.

However, one of our goals is to provide heterogeneity, which will be discussed in section 2.6. This allows the pure FSM to be used without fundamental loss of expressiveness if it is used in conjunction with a foreign model that supports arithmetic handling. For example, instead of a guard "u [u != 0]" in a valued FSM, we could specify a guard "$u \wedge w$" in a pure FSM and externally compute the function

$$w = \begin{cases} \text{present;} & u \text{ != } 0 \\ \text{absent;} & \text{otherwise} \end{cases}$$

in the foreign model.

Nevertheless, the valued FSM does provide a more convenient syntax for arithmetic operations directly within the FSM. For example, consider a valued FSM with input events $I = \{u, v\}$ and output events $O = \{x, y\}$. Suppose that the value alphabet is $\aleph$ for every input and every output events. Then, the arithmetic expression can be contained in the condition of a guard, such as "$(u + v) * u == 8$". In addition, it can be included by the expression assigned to the emitted event in an action, such as "$x(u + v)$, $y(u * v)$".

2.5 Hierarchy

FSMs, which are flat and sequential, have a major weakness: Most practical systems have a very large number of transitions and states. To alleviate this problem, *hierarchy* allows a state of the FSM to be refined into another FSM. For example, figure 2.8 shows a *hierarchical FSM* in which state β is refined. With respect to the inner FSM called the *slave*, the outer FSM is called the *master*. Moreover, if a state is refined, it is called a *hierarchical state*, such as state β; otherwise, the state is called an *atomic state*, such as state α. The input events for the slave are a subset of the input events for the master. Similarly, the output events from the slave are a subset of the output events from the master.

The hierarchy semantics define how the slave reacts relative to the reaction of its master. A reasonable semantics defines one reaction of the hierarchical FSM as follows: If the current state is an atomic state, the hierarchical FSM behaves just like an FSM without hierarchy. If the current state is a hierarchical state, first the corresponding slave reacts, and then the master reacts. Thus, two transitions are taken, and their actions must be somehow merged into one.

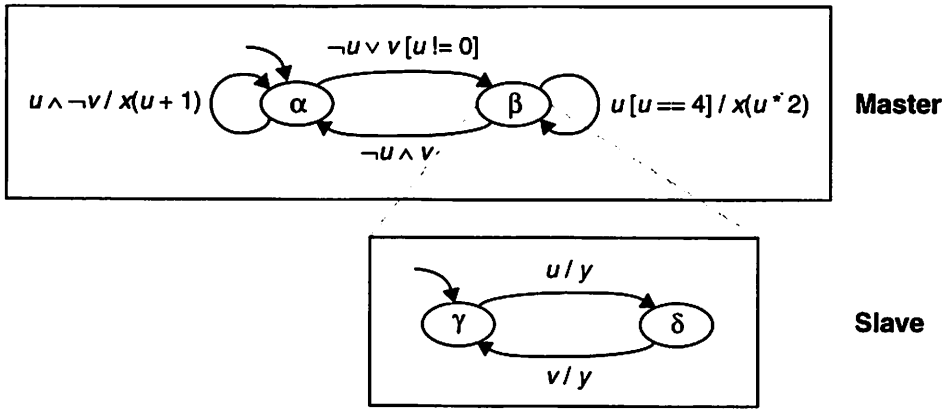


Figure 2.8 A hierarchical FSM.

In the case that the output event is pure, it is easy to avoid conflicting definitions of the event between the actions of the slave and the master. We take an output event to be present if the action of the master or any slave below it emits that event. Since an action does not explicitly set an output event absent, no conflict is possible for the pure event in this syntax.

Now we consider that the output event is valued, and distinguish three cases.

- When an output event is emitted by only one action in either the master or any slave, it is present with a value assigned by the action that emits it.
- When an output event is not emitted by any action, it is absent but with a value the same as the most recent value retained by the master. Even if any slave has a different previous value for the output event, this value should not be used since it must be older than the one retained by the master (otherwise the master should have known and retained it).
- When an output event is emitted by more than one action: If these actions emit the event with the same value, the event is present with that value. However, conflicting definitions of the output event occur if these actions emit the event with different values. In Esterel [18], a function can be specified to combine the conflicting definitions. For example, for two natural numbers, the values might be added. We prefer to consider this an error situation, because the values can be more conveniently and flexibly combined externally in a foreign model that is better suited to numerical computation. Thus, at a reaction, no more than one transition should emit the same output event with different values.

Consider the example of figure 2.8 with input events $I = \{u, v\}$ and output events $O = \{x, y\}$. Suppose that the input event u and the output event x are the valued events with $\aleph$

as their value alphabets, and the events *v* and *y* are the pure events. A possible trace for this hierarchical FSM is shown in figure 2.9. In this example, the hierarchical FSM has only two levels. However, the slave can actually be another hierarchical FSM, so the depth of hierarchy is arbitrary. The semantics generalizes trivially.

At a fundamental level, hierarchy adds nothing to expressiveness. Neither does it reduce the number of states. However, it can significantly reduce the number of transitions and make the FSM more intuitive and easy to understand. The transition from state β to state α in figure 2.8 is simply a compact notation for transitions from state γ to state α and state δ to state α.

2.6 Heterogeneity

An FSM (even with hierarchy) is not by itself adequate for describing most complex systems. For one thing, this model is extremely awkward for expressing intensive numerical computations. Therefore, for practical application to complex systems, the FSM has to be combined with other models, i.e. we have to add *heterogeneity* to the FSM.

Current State	α	α	β, γ	β, δ	β, γ	...
u (status)	*present*	*absent*	*present*	*present*	*absent*	...
u (value)	2	2	4	6	6	...
v (status)	*absent*	*absent*	*absent*	*present*	*present*	...
Next State	α	β, γ	β, δ	β, γ	α	...
x (status)	*present*	*absent*	*present*	*absent*	*absent*	...
x (value)	3	3	8	8	8	...
y (status)	*absent*	*absent*	*present*	*present*	*absent*	...

Figure 2.9 A possible trace for the hierarchical FSM in figure 2.8.

One 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 is chosen to govern the interaction between boxes, but the contents of boxes need not be governed by this same model. The only requirement is that the interfaces of boxes must conform to a standard accepted by this model. Thus, a box may encapsulate a subsystem specified by one model within a system specified by another. In other words, heterogeneity allows different models to be systematically and modularly combined together.

Our hierarchical FSM is easily extended to support heterogeneity. The slave of a hierarchical state need not be an FSM. The key principle is that the slave must have a well-defined terminating computation that reacts to input events by possibly asserting output events. Therefore, the slave could be, for example, a Turing machine (that halts), a C procedure (that eventually returns), a dataflow graph (with a well-defined iteration), etc. It could even be concurrent.

The hierarchy semantics is similarly defined as in section 2.5 with one subtle modification: If the current state is a hierarchical state, first the corresponding slave is invoked, and then the master reacts. When the slave is invoked, it performs a determinate and finite operation, called a *step* of the slave, which reacts to input events and may assert output events. One step of a slave FSM is one reaction of the FSM.

In the reverse scenario, we need for an FSM to be able to describe a module inside some other model. This can be done as long as that model provides a way to determine the input events and when a reaction should occur for each FSM. For example, in figure 2.10, an outer model is schematically illustrated with rectangular boxes representing modules. Two FSMs are embedded inside the modules. Most interestingly, they are concurrent FSMs if the outer model has concurrent semantics.

2.7 Hierarchical Entries and Exits

When a slave of a hierarchical state is invoked for the first time, unambiguously it will start from its initial conditions (e.g. the initial state for an FSM). When it is subsequently invoked, we may wish to reinitialize it or allow it to continue from the last known conditions. Thus, as in Statecharts [43], we support a transition entering a hierarchical state to be either *initial entry* or *history entry*. Initial entry starts the slave from the initial conditions like the first invocation. History entry permits the slave to resume computation from the final conditions of the last invocation.

As shown in figure 2.11, we illustrate an initial entry by additionally drawing a small circle with an **I** in it at the end of an arc. For example, the transition from state α to state β is an initial entry. When this transition is taken in a reaction, the slave inside the next state β will be invoked from its initial state γ in the next reaction. The other transitions without additional drawing at the end of arcs are considered the history entries. For example, the transition from state α to state α is a history entry. When this transition is taken in a reaction, the slave inside the next state α will be invoked from its last state in the next reaction. In figure 2.11, the slaves may actually be specified by other models as discussed in section

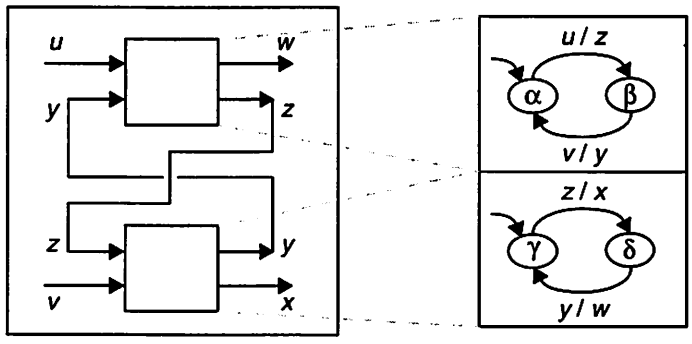


Figure 2.10 Two FSMs are embedded inside the modules of another model.

2.6. Hence, each inner model must also provide a way to start from its initial conditions or to continue from the last known conditions in response to an initial entry or a history entry.

Under normal circumstances of a hierarchical FSM, if the current state is a hierarchical state, the corresponding slave is invoked prior to taking the transition. However, we may need to immediately interrupt before the slave is invoked in some situations. Thus, we support a transition exiting from a hierarchical state to be either *preemptive* or *non-preemptive* [86]. If a preemptive transition is taken, the slave of the current state will not be invoked. Otherwise, for a non-preemptive transition, the slave is invoked normally.

As shown in figure 2.11, we depict a preemptive transition by additionally drawing a small circle at the beginning of an arc. For example, the transition from state β to state α is preemptive. This preemptive transition is taken when the hierarchical FSM is in state β and substate δ and the input events *u* and *v* are absent and present, respectively. Under this circumstance, the output event *y* is absent (instead of present normally) since the slave inside state β will not be invoked. The other transitions without additional drawing at the beginning of arcs are considered non-preemptive, such as the transition from state α to state β.

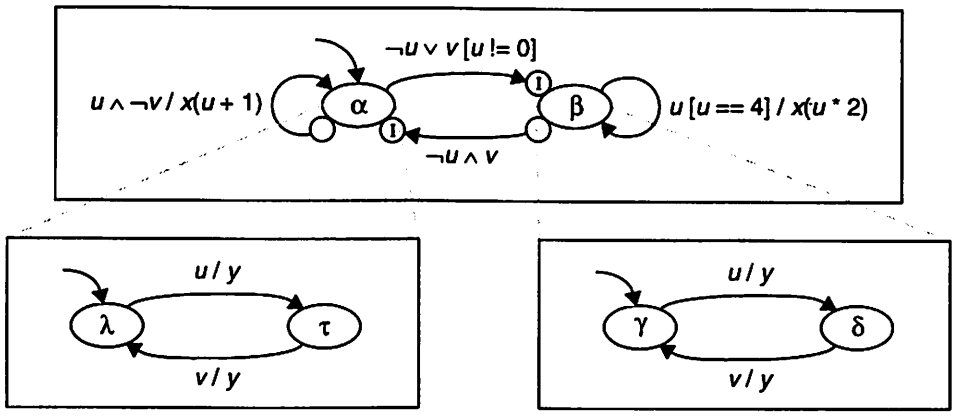


Figure 2.11 A hierarchical FSM with hierarchical entries and exits.

Consider the example of figure 2.11 with input events $I = \{u, v\}$ and output events $O = \{x, y\}$. Suppose that the input event u and the output event x are valued events with $\aleph$ as their value alphabets, and the events v and y are pure events. A possible trace for this hierarchical FSM is shown in figure 2.12.

2.8 Shared Slaves

When supporting the hierarchy within an FSM, one direct approach is to simply have an individual slave attached to each hierarchical state. In contrast, however, we first let a collection of slaves be attached to the FSM and then associate each hierarchical state with a slave in the collection. This approach naturally enables multiple states to share a same slave.

One advantage of this sharing mechanism is to save redundant slaves. In the previous example (see figure 2.11), two slaves are exactly the same (except for the names of states), and either slave is reinitialized whenever the master switches to the corresponding state from the other state. Hence, it is not necessary to have two separate slaves. In other words, the two states of the master may share a same slave, as shown in figure 2.13. The traces of

Current State	α, λ	α, λ	β, γ	β, δ	α, λ	...
u (status)	*present*	*absent*	*present*	*absent*	*present*	...
u (value)	2	2	4	6	8	...
v (status)	*absent*	*absent*	*absent*	*present*	*present*	...
Next State	α, λ	β, γ	β, δ	α, λ	β, γ	...
x (status)	*present*	*absent*	*present*	*absent*	*absent*	...
x (value)	3	3	8	8	8	...
y (status)	*absent*	*absent*	*present*	*absent*	*present*	...

Figure 2.12 A possible trace for the hierarchical FSM in figure 2.11.

this hierarchical FSM with a shared slave are exactly the same as those with two separate slaves (with substitution of λ and τ by γ and δ, respectively, in the traces).

Another advantage is that the local conditions of a shared slave (e.g. the current state of an FSM) may be carried and accessed even across the reactions in which the master switches among different states. One typical application is the *hybrid system* [1][2], which can be considered a hierarchical FSM that has a shared slave as the dynamic laws of the system. This application will be further demonstrated in section 4.4.

2.9 Local Events

Similar to the Argos language [69], we allow the FSM to have local events. In general, the local events can be utilized in the following three scenarios.

- As local variables of the FSM: The local events can be used to retain information across the reactions of the FSM. In this case, they serve the same purpose as the local variables in an *extended FSM* [20][75][27]: They may reduce the number of visible states in the FSM. For example, figure 2.14 shows an FSM that emits an output event x

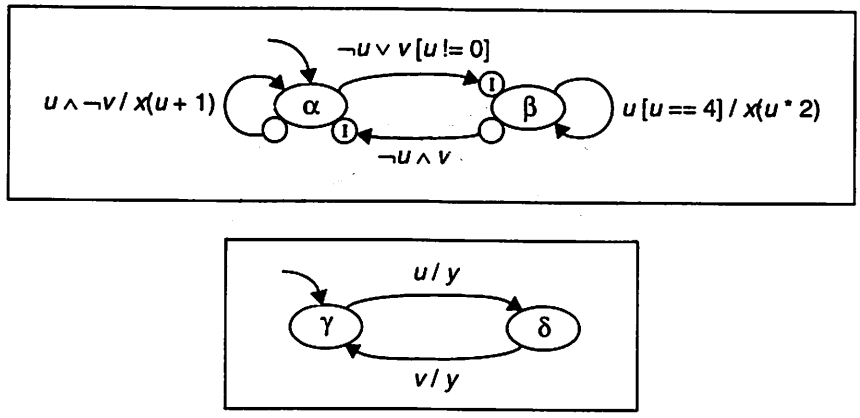


Figure 2.13 A hierarchical FSM with a shared slave.

whenever receiving an input event *u* eight times. At the upper right corner, a rectangular box labeled with *r* declares a local event *r*, which is used to count the occurrence of input event *u*. Without this local event, the FSM would have required a state for each occurrence of input event *u* (up to the eighth time). Therefore, the local event *r* hides an additional six states by its values.

- As output events of the slave: The local events can be used to pass information from the slave to the master FSM. For example, in figure 2.15, the master has a local event *r*, which is also the output event of the slave..Suppose that the master is in state α and the

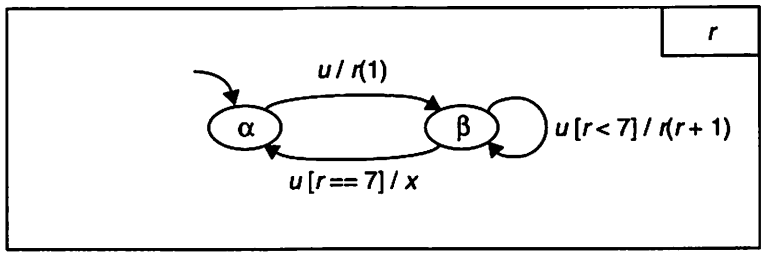


Figure 2.14 An FSM with a local event as a local variable.

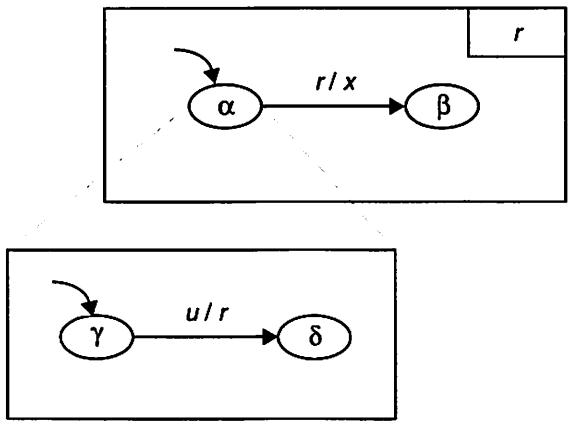


Figure 2.15 A local event of the master serves as an output event of the slave.

slave is in state γ. When the input event *u* occurs, the slave takes the transition to state δ and emits the event *r*, which in turn let the master take the transition to state β. Moreover, the mechanism shown in figure 2.15 can replace an outgoing inter-level transition [86], which is considered by many a violation of modular design. In other words, the two transitions from state γ to state δ and then state α to state β have the same effect as a transition across hierarchy boundary from state γ to state β with a label "*u* / *x*".

- As input events of the slave: The local events can be used to pass information from the master FSM to the slave. As shown in figure 2.16, the event *r* is a local event of the master and an input event of the slave. The master starts in state α, and when an input event *u* occurs in a reaction, emits the event *r* and takes the transition to state β. This causes the slave to take the transition to state δ in the next reaction. Note that since these two taken transitions must spread over two reactions of the hierarchical FSM, the local event itself is not enough to replace the incoming inter-level transition [86]. This problem will be resolved in section 2.10.

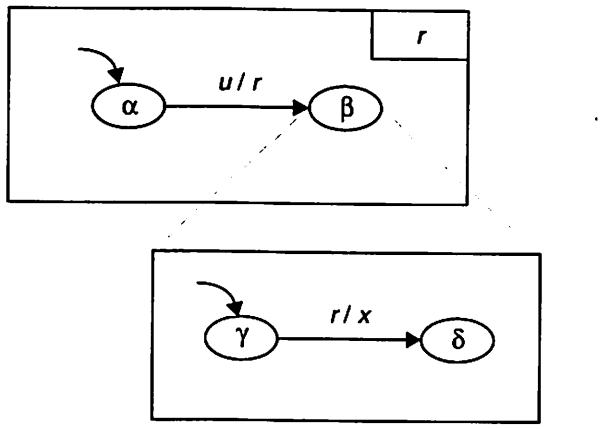


Figure 2.16 A local event of the master serves as an input event of the slave.

2.10 Initial Transitions and Conditional Initial States

In our FSM, we extend an arc without a source state to be a fully functional transition in the sense that it may have a guard and an action. This transition is called an *initial transition* since it leads to the initial state. When an initial transition does not have a label, just like all we have seen so far, it is considered without a guard and an action, and is defined to be always enabled. One main purpose of the initial transition is to allow the FSM to perform initialization, such as assigning initial values of local events, before the FSM starts from its initial state either in the first reaction or due to initial entry. Consider figure 2.17 for example. The initial transition is labeled by "/ $r(0)$". This transition is always enabled because it does not have a guard. Hence, before the FSM starts from the initial state α in the first reaction, it takes the initial transition and thus assigns the local event r a value 0.

Furthermore, we generalize the notion of an initial state to be a set of possible initial states. Each of these possible initial states is called a *conditional initial state*, and must be pointed to by an initial transition. When an FSM begins in the first reaction or is entered by an initial entry, one initial transition is taken, chosen from the set of enabled initial transitions. Then, the FSM starts from the conditional initial state which the taken initial transition points to. To maintain determinism and reactivity of the FSM, it is required that there exists exactly one enabled initial transition. For example, in figure 2.18, an FSM

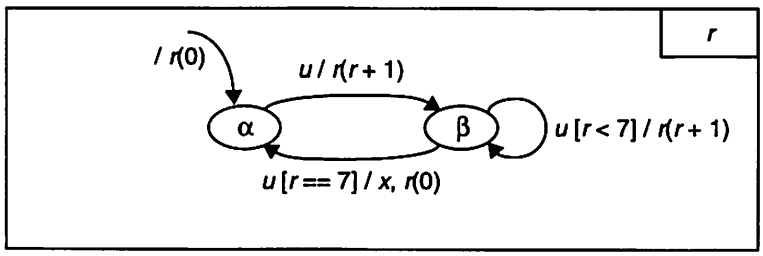


Figure 2.17 An FSM with an initial transition to initialize a local event.

starts from either state α or state β depending on whether input event *u* is present or absent.

As shown in figure 2.19, now we can replace the incoming inter-level transition by a transition with initial entry entering a conditional initial state. The master starts in state α, and when an input event *u* occurs in a reaction, emits the event *r* and takes the transition to state β. In the next reaction, as a result, the slave takes the initial transition to conditional initial state γ and emits the output event *x*. Subsequently, the slave emits the output event *y* and takes a regular transition from state γ to state γ. The first two transitions from state α to

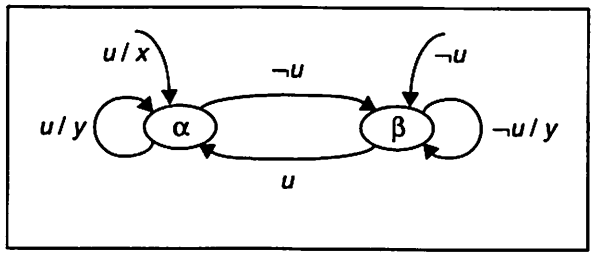


Figure 2.18 An FSM with conditional initial states.

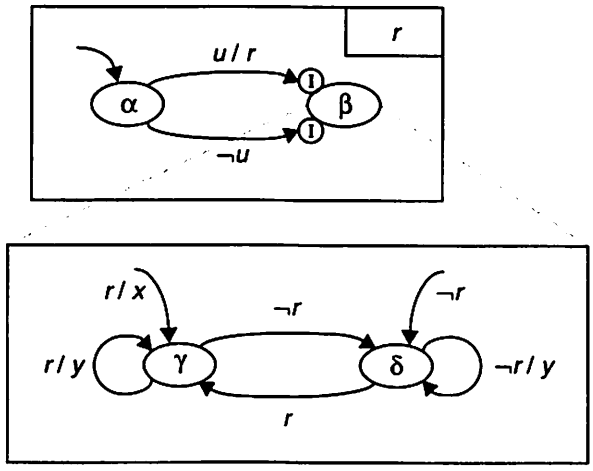


Figure 2.19 The incoming inter-level transition can be replaced by a transition with initial entry entering a conditional initial state.

state β and then to state γ have the same effect as a transition across the hierarchy boundary from state α to state γ with a label "*u* / *x*", since the initial transition does not prevent the third transition from being taken in the second reaction.

2.11 Instantaneous Transitions and States

When the current state of an FSM is an atomic state and an explicit outgoing transition of that state does not have a trigger in its guard, the FSM may enable this transition right away without waiting for any input events to occur. Hence, in an atomic state, any explicit outgoing transition that does not have a trigger in its guard is defined as an *instantaneous transition*, which may be taken right away even after other transitions have been taken in the same reaction. For example, the FSM depicted in figure 2.20 starts in state α with local event *r* initialized to a value 0. When the input event *u* occurs in a reaction, the FSM increases the value of local event *r* by 1 and takes the transition to state β. At this moment, since the instantaneous transition labeled by "[$r < 8$]" is enabled, the FSM takes the transition to state α right away in the same reaction.

An *instantaneous state* is an atomic state in which all explicit outgoing transitions are instantaneous transitions and are *completely specified*. The explicit outgoing transitions of a state are said to be completely specified if the implicit self transition is always not

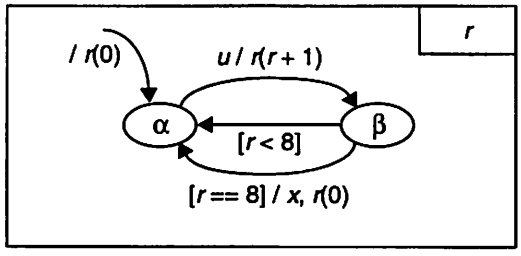


Figure 2.20 An FSM with instantaneous transitions and an instantaneous state.

enabled in that state for any given input events. Therefore, the instantaneous state will never remain across reactions of the FSM. For example, the state β in figure 2.20 is an instantaneous state.

2.12 Simulation Algorithm

To accommodate all the features discussed for the FSM, we come up with the following algorithm for simulating one reaction of the FSM:

1. Clear the statuses of all output events. I.e. let them all be absent.
2. If the FSM is not in the first reaction and is not entered by an initial entry, go to (7).
3. Clear the statuses of all local events.
4. Check all initial transitions. If more than one is enabled, flag a non-deterministic error and go to (21). If none is enabled, flag a non-reactive error and go to (21).
5. Emit the events in the action of the enabled initial transition.
6. Enter the conditional initial state which the enabled initial transition points to. I.e. let it become the current state for the following reaction.
7. If the current state is not a hierarchical state, go to (9).
8. For each input event of the slave, get the status and the value from the corresponding input or local event of this FSM.
9. If the FSM is not in the first reaction and is not entered by an initial entry, clear the statuses of all local events.
10. Check all preemptive transitions of the current state. If more than one is enabled, flag a non-deterministic error and go to (21). If exactly one is enabled, go to (15).
11. If the current state is not a hierarchical state, go to (14).

12. Perform one invocation of the slave. According to the entry type of the enabled transition in previous reaction, the slave either starts from the initial conditions or resumes from the final conditions of the last invocation.

13. For each emitted output event of the slave, update the status and the value to the corresponding output or local event of this FSM. If any of these events has been emitted with a different value, flag an emission-conflicting error and go to (21).

14. Check all non-preemptive transitions of the current state. If more than one is enabled, flag a non-deterministic error and go to (21). If none is enabled, let the implicit self transition be enabled.

15. Emit the events in the action of the enabled transition. If any of these events has been emitted with a different value, flag an emission-conflicting error and go to (21).

16. Enter the destination state of the enabled transition.

17. If the current state is a hierarchical state, go to (21).

18. Check all instantaneous transitions of the current state. If more than one is enabled, flag a non-deterministic error and go to (21). If none is enabled, go to (21).

19. Emit the events in the action of the enabled instantaneous transition. If any of these events has been emitted with a different value, flag an emission-conflicting error and go to (21).

20. Enter the destination state of the enabled instantaneous transition. Go to (17).

21. One reaction is complete.

3

Integration with Concurrency Models

3.1 Hierarchical Combination

With support of hierarchy and heterogeneity, an FSM can be combined with almost any concurrency model. Our goal is the hierarchical nesting of FSMs with concurrency models, as shown in figure 3.1. We schematically illustrate the states of the FSM with elliptic nodes and the modules of the concurrency model with rectangular blocks. The depth and order of the nesting is arbitrary. We wish for an FSM to be able to describe a

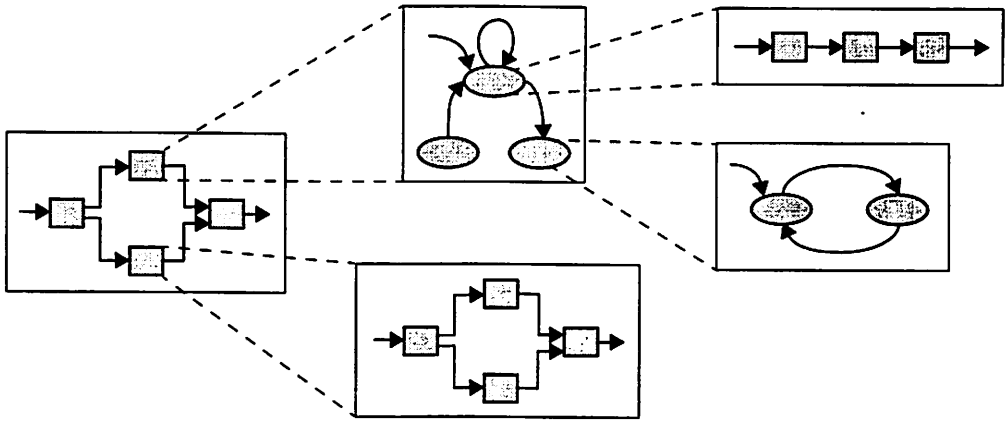


Figure 3.1 Hierarchical nesting of FSMs with concurrency models.

module in a concurrency model and for a state to be able to be refined to a concurrent subsystem.

When two models are brought together to interact with each other in the hierarchical combination, differences between them can lead to ambiguities. For example, how should the FSM deal with the notion of time when it interacts with a timed model? The exact semantics of this interaction must be defined in terms of the semantics of both models. In general, as shown in figure 3.2 [26], the interaction with information flowing from model X to model Y should be handled by one of the following conversions:

- The common semantic properties of both models are translated.
- The semantic properties in model X but not in model Y are ignored.
- The semantic properties in model Y but not in model X are created.

In this chapter, we explore the interaction of the FSM with various concurrency models, namely *discrete-event* (DE), *synchronous/reactive* (SR), *synchronous dataflow* (SDF), *dynamic dataflow* (DDF), and *communicating sequential processes* (CSP). Our objective is to develop semantics that supports arbitrary combinations of these concurrency models with FSMs.

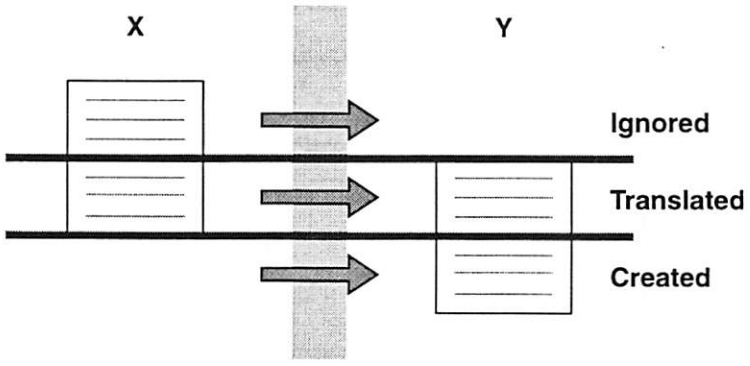


Figure 3.2 Conversions of semantic properties from model X to model Y.

3.2 Discrete Events

Under the discrete-event (DE) [7][24][33] model, a system consists of a network of blocks connected by directed arcs, as depicted in figure 3.3. Blocks communicate among themselves and with the environment through *events*, which may or may not have values associated with them. Each event originates either from an output of some block or from the environment, and is destined either for an input of some block or for the environment. This relationship of source and destination is indicated by the arc. The DE model has a notion of *global time* that is known simultaneously throughout the system. An event occurs at a point in time. In a simulation of such a system, the global time is a value, usually an integer or a real number. Each event needs to carry a *time stamp* that indicates the time at which the event occurs. The time stamp of an event is typically generated by the block that produces the event, and is determined by the time stamp of input events and the latency of the block. The DE simulator needs a global event queue that sorts the events by their time stamps, and chronologically processes each event by sending it to the appropriate input of a block, which reacts to the event (*fires*). In addition, it maintains the global time as the *current time* of the system, which is constantly updated with the time stamp of the event being processed. In figure 3.3, for example, reacting to an input event, block A produces an event with a time stamp t at its upper output. This event is sorted in the event

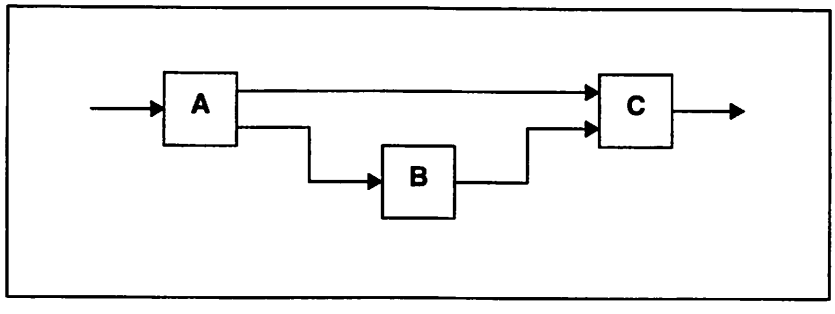


Figure 3.3 A discrete-event (DE) system.

queue, and when its time stamp t becomes the smallest in the event queue, block **C** fires with this event at its upper input.

However, *simultaneous events* (those with the same time stamp) can cause difficulties in a DE system. In figure 3.3, reacting to an input event, block **A** may produce an event at each of its two outputs with the same time stamp t. In this case, there exists an ambiguity about whether block **B** or **C** should fire first at time t since there is more than one event in the event queue with the same smallest time stamp. Suppose that **B** is a *zero-delay* block. This means that if an output event of block **B** is produced in a firing, it is assigned the same time stamp as the input event that invokes that firing. Suppose further that block **B** always produces an output event in response to the input event. The two possible situations are described as follows:

- If block **C** fires first, block **B** subsequently fires and produces an output event with the time stamp t. This event causes block **C** to fire again. Hence, block **C** fires twice.
- If block **B** fires first, it produces an event with the time stamp t at its output. As a consequence, block **C** has an event at each of its two inputs with the same time stamp t. Hence, block **C** may either fire once by observing both input events or fire twice by observing one input event at a time.

Therefore, the resulting behavior of the system depends on how the simultaneous events are handled by the DE simulator.

To eliminate the ambiguity, one solution provided by some DE simulators, such as in VHDL [5], is a *delta delay* that represents an infinitesimal amount of delay. In the previous example, if block **B** has a delta delay δt, its output event will have a time stamp $t+\delta t$. This output event will be ordered after the event with the time stamp t even though both $t+\delta t$ and t represent the same simulation time t. Thus, no matter which block fires first, block **C** always fires twice in response to the input events with time stamps t and $t+\delta t$,

respectively. However, in this solution, there is no way to ensure that block **B** fires before block **C**, in case we wish for block **C** to see both input events at once. This leads to another solution adopted by the discrete-event domain in Ptolemy II [31], which utilizes the topology of the system to determine the order of block firings when simultaneous events occur. In the previous example, since block **B** could produce events with zero delay affecting block **C**, block **B** always fires first when simultaneous events occur. Subsequently, block **C** always fires once by observing the events at both inputs.

In addition to simultaneous events, there can be a problem with directed loops that have zero delay. For example, figure 3.4 shows a DE system with a directed loop. Suppose that blocks **A** and **B** both are zero-delay and each of them always produces an output event reacting to the input event. When an event occurs at the upper input, block **A** starts a process by which an event will circulate through the loop forever without advance of simulation time. According to [60], there is no way to avoid this so-called *Zeno condition* without adding a finite amount of delay in the directed loop.

3.2.1 FSM inside DE

An FSM embedded inside a DE block performs one reaction when the DE block fires, which occurs when there is an event present at one of its inputs. Presence or absence of an

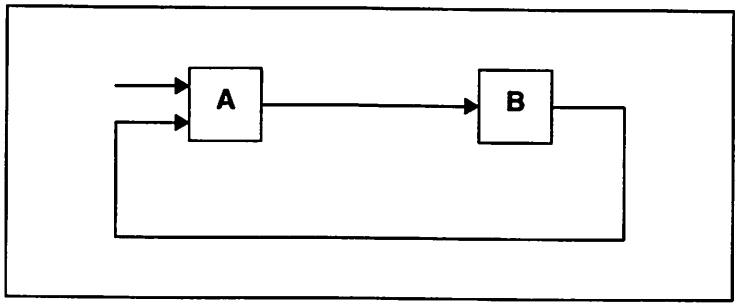


Figure 3.4 A DE system with a directed loop.

event at an input of the DE block translates directly into the status of the corresponding input event in the inner FSM. In addition, if the event is present, its value (if it exists) is also passed from the DE to the FSM. If the event is absent in the DE, the FSM retains the most recent value for the corresponding input event. On the other hand, the FSM embedded inside a DE block may emit output events in the reaction. These emitted output events translate directly into events in the outer DE system. However, in DE, every event needs a time stamp, something not provided by the FSM. We choose the semantics where the FSM appears to the DE system as a zero-delay block. I.e. the event passed to the DE system in a reaction of the FSM is assigned the same time stamp as the input event that invokes that reaction. If we wish to model a time delay associated with a reaction, we may explicitly connect the FSM block with a delay block to simulate the delay occurring inside the FSM subsystem.

Consider an example shown in figure 3.5, where two FSMs are embedded within a DE system. The name (u, v, w, etc.) on the arc of the DE system denotes the name for the nearest input or output of a block, which in turn denotes the name for the input or output event of the FSM inside that block. Suppose that an event for input u with a time stamp t is the

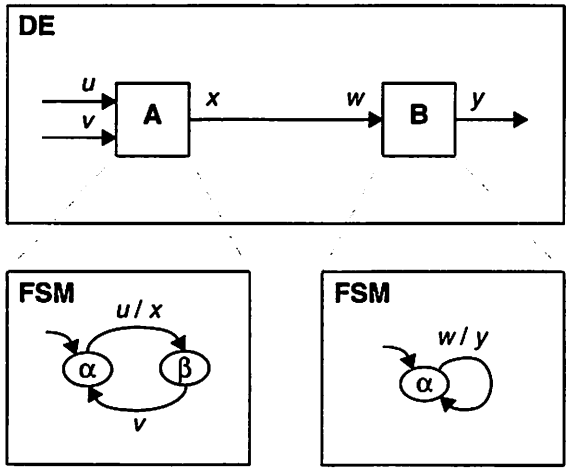


Figure 3.5 Two FSMs are embedded within a DE system.

next to be processed in the event queue, and both FSMs are in state α. The DE system executes as follows:

- Fire **A**: Since there exists an event for input *u*, FSM **A** takes the transition from state α to state β, and emits the output event *x*. In DE, this event will have the time stamp *t*. Therefore, it will be the next to be processed in the event queue, and will be sent to input *w* of block **B**.
- Fire **B**: Since there exists an event for input *w*, FSM **B** takes the transition back to state α, and emits the output event *y*. In DE, the event at output *y* will have the time stamp *t*.

Due to event-driven semantics of the DE model, a block does not fire if there are no events at its inputs. This leads to some subtleties with the guards of the FSM. For example, in figure 3.6, suppose that the FSM is in state α. The guard "¬*u*" on the outgoing transition to state β indicates that input event *u* must be absent for this transition to be taken. Implicitly, however, input event *v* must be present at the same time, otherwise the FSM will not even react since there is no event to invoke the block firing. Hence, it would be

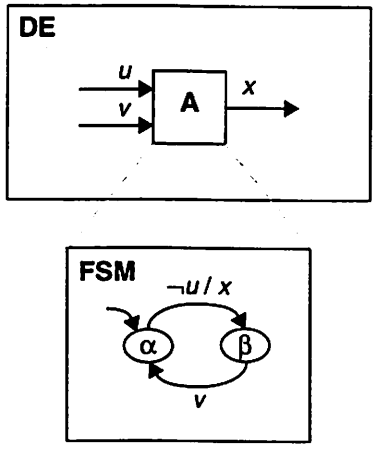


Figure 3.6 The guard "¬*u*" of the upper transition is incomplete due to event-driven semantics of the DE model.

clearer to give the guard as "$\neg u \wedge v$". If the guard were given as "$\neg u \wedge \neg v$" instead, the transition would never be taken.

Since we treat FSMs as zero-delay blocks in a DE system, a directed loop connecting only FSM blocks always has zero delay. As mentioned earlier, a directed loop with zero delay can cause a problem in a DE system. Figure 3.7 shows such a directed loop. Reacting to an event at input u, FSM **A** emits the event x. This event causes FSM **B** to emit the event y, which will let FSM **A** emit the event x again. Therefore, FSMs **A** and **B** keep reacting and emitting the events without advancing time in the DE system.

3.2.2 DE inside FSM

When a DE model refines a state of an FSM, one step of the slave DE subsystem is the simulation of that subsystem until there is no event in the event queue with the same time stamp as the input events. In particular, if previous FSM describes a block of another DE model, the input events passed to invoke the inner DE subsystem by the FSM will have the time stamp that is the current time of the outer DE system. Therefore, the inner DE subsystem will never execute ahead of the outer DE system. In other words, the notion of current time keeps consistent throughout all DE models in the hierarchy.

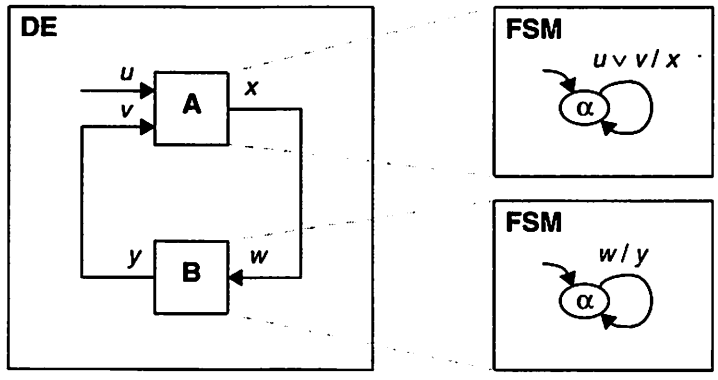


Figure 3.7 A directed loop that has zero delay can cause a problem in a DE system.

Consider an example shown in figure 3.8. Suppose that the FSM is in state α and the DE subsystem has an empty event queue. One possible reaction of the FSM is described as follows: If an input event *u* occurs with a time stamp *t*, it is passed to the DE subsystem and is processed in the event queue. Reacting to the event at input *u*, block **A** produces an event with the time stamp *t* at its output *x*. Since this event still has the same time stamp as the input event *u*, it is processed in the event queue. Subsequently, block **B** fires with the event at its input *v* and produces an event with a time stamp *t*+1 at its output *y*. At this moment, since there is no more event with the time stamp *t* in the event queue, one step of the slave DE subsystem is complete. Then, the master FSM reacts to the input event *u* by taking the implicit self transition back to state α.

However, *future events* (those with time stamps greater than current time) in a slave DE subsystem can cause difficulties. Consider the example of figure 3.8. In the previous mentioned reaction, the event with the time stamp *t*+1 at output *y* is a future event and is left in the event queue of the DE subsystem. Subsequently, if another input event *u* occurs with a time stamp *t*+2 for the FSM, it is passed to the DE subsystem. At this moment, the

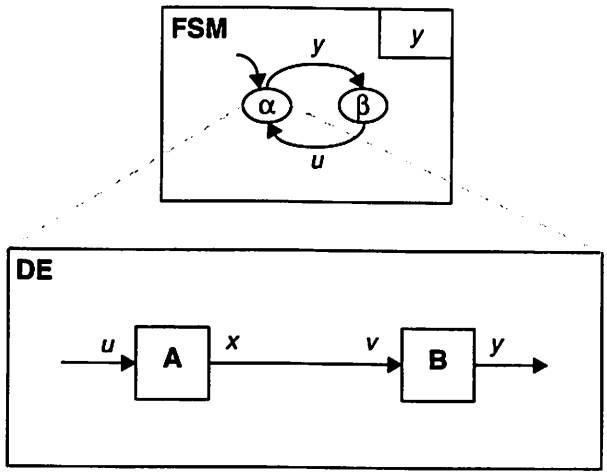


Figure 3.8 A DE subsystem refines a state of an FSM.

event with the time stamp t+1 left in the DE subsystem becomes *expired* in the sense that its time stamp is smaller than the time stamp of current input event.

To avoid the expired events, the slave DE subsystem should request of its master FSM that it needs to be fired again at the time marked by the time stamp of every future event. This request should in turn be propagated by the FSM up in the hierarchy. In the previous example, the FSM needs to have an additional reaction at time t+1 of its environment as requested by the DE subsystem. Thus, the event with the time stamp t+1 can be properly processed in the DE subsystem, and is passed to the FSM as a local event y. As a consequence, the FSM takes the transition from state α to state β.

This request mechanism is not enough by itself because it cannot prevent expired events that arise from state changes of the master FSM. Following the previous example, suppose that a future event with a time stamp t+2 is left in the DE subsystem before the FSM takes the transition from state α to state β. Even though the FSM reacts at time t+2 of its environment as requested by the DE subsystem, the event with the time stamp t+2 remains in the DE subsystem, which is not invoked due to the state change of the FSM. Suppose further that the FSM returns to state α after a period of time. Subsequently, if an input event u occurs with a time stamp t+5, the event with the time stamp t+2 left in the DE subsystem becomes an expired event now.

For those expired events resulting from state changes of the master FSM, one solution is to simply purge them before the slave DE subsystem continues execution. However, this does not really preserve the history of the DE subsystem after its suspension because the expired events are ignored. The solution we prefer is to update the time stamps of the expired events with the time stamp of current input events in the DE subsystem. Hence, in the previous example, the event with the time stamp t+2 is updated with the time stamp t+5, and then the DE subsystem can continue execution without a problem.

3.3 Synchronous/Reactive Models

A system in the synchronous/reactive (SR) [10][42] model consists of communicating blocks, as illustrated in figure 3.9. Execution of the system occurs at a sequence of discrete instants, called *ticks* (as in ticks of a clock). Each tick is initiated by the environment, and nothing happens between ticks. In each tick, each block instantly computes a function that determines its outputs by observing its inputs, and each of the inputs and the outputs either is absent (has no event) or is present (has exactly one event) possibly with a value. Blocks communicate through unbuffered unidirectional arcs. Communication among blocks is instantaneous since an output determined by a block in a tick is observed by other blocks at their inputs in that same tick.

Directed loops in the SR systems intrinsically have zero delay because computation within blocks and communication among blocks are instantaneous. As a result, two problems may arise if there exist directed loops in an SR system.

- Contradiction: Consider the example in figure 3.9. Suppose that blocks **A** and **B** compute the following functions, respectively.

$$\text{output of } \mathbf{A} = \begin{cases} \text{present}; & \text{upper input is present and lower input is absent,} \\ \text{absent}; & \text{otherwise.} \end{cases}$$

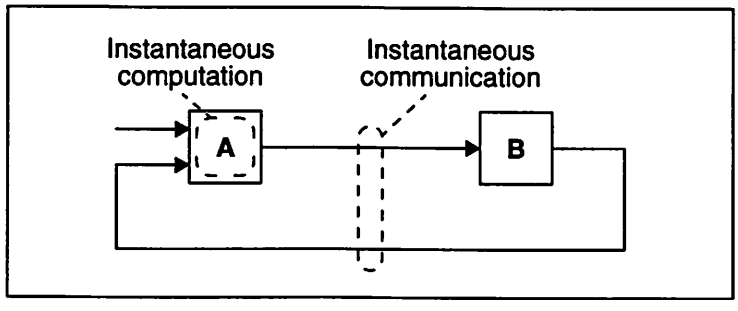


Figure 3.9 A synchronous/reactive (SR) system.

$$\text{output of } \mathbf{B} = \begin{cases} \text{present}; & \text{input is present}, \\ \text{absent}; & \text{otherwise}. \end{cases}$$

When the upper input of block **A** is present in a tick of the system, its output is present if its lower input is assumed absent. However, this causes the output of block **B** to be present, which contradicts the assumption that the lower input of block **A** is absent. Even if the lower input of block **A** is assumed present, the output of block **B** will still contradict the assumption. Hence, under this circumstance, the system appears to lack a behavior.

- Ambiguity: In figure 3.9, suppose that blocks **A** and **B** compute the following functions, respectively.

$$\text{output of } \mathbf{A} = \begin{cases} \text{present}; & \text{both inputs are present}, \\ \text{absent}; & \text{otherwise}. \end{cases}$$

$$\text{output of } \mathbf{B} = \begin{cases} \text{present}; & \text{input is present}, \\ \text{absent}; & \text{otherwise}. \end{cases}$$

When the upper input of block **A** is present in a tick of the system, its output is present if its lower input is assumed present. This causes the output of block **B** to be present, which agrees with the assumption. However, even if the lower input of block **A** is assumed absent, the output of block **B** will still agree with the assumption. Hence, under this circumstance, the system exhibits multiple behaviors.

In the SR model, these directed loops that appear to have no or multiple behaviors are often called *causality loops*.

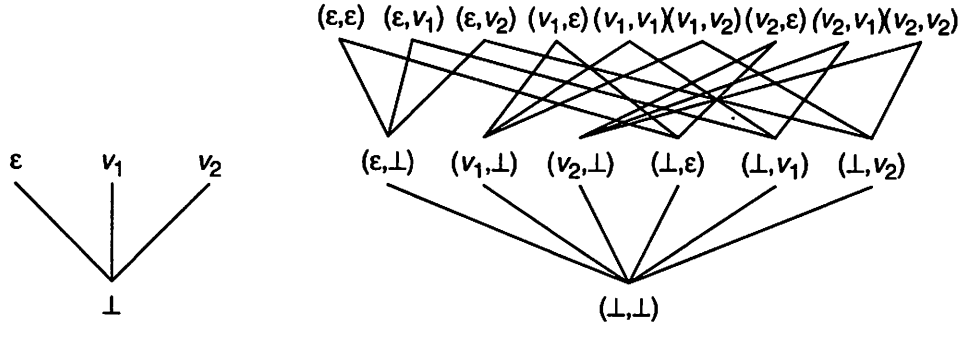
To allow directed loops and maintain determinism (i.e. exactly one behavior given the same inputs) for the systems, two constraints are imposed in the SR model. First, for each input or output of each block, its alphabet is augmented with a special symbol $\perp$, inter-

preted to mean "unknown", and a partial order is defined on the augmented alphabet. For example, suppose that the augmented alphabet is $\{\perp, \varepsilon, v_1, v_2\}$, which means {unknown, absent, present with value v_1, present with value v_2}. As shown in figure 3.10(a), a "flat" partial order is defined on the augmented alphabet. In this partial order, $\perp$ is always the bottom element, which is below ("weaker than", usually denoted as $\sqsubseteq$) everything else in the alphabet. The partial order is easily generalized on multiple augmented alphabets. For example, the partial order in figure 3.10(b) is defined on two of the augmented alphabets. Secondly, all functions computed by the blocks are required to be *monotonic*. A function f is monotonic if

$$(x_1, x_2, \ldots) \sqsubseteq (y_1, y_2, \ldots) \text{ implies that } f(x_1, x_2, \ldots) \sqsubseteq f(y_1, y_2, \ldots)$$

where $\sqsubseteq$ is interpreted with respect to the partial order. With these two constraints, any SR system (a network of monotonic functions) has a least fixed point, where "least" is with respect to the partial order, according to the Knaster-Tarski fixed point theorem [30]. This least fixed point is taken to be the behavior of such an SR system.

Finding the least fixed point is straightforward, in principle, in an SR system. Starting with all inputs and outputs as unknown ($\perp$), the blocks simply take turns to compute their



(a) On an augmented alphabet (b) On two augmented alphabets

Figure 3.10 Partial orders defined in the SR model.

functions in a given order repeatedly until all inputs and outputs converge to a fixed point. Even though choosing a good order for the blocks can greatly impact performance [35], the resulting fixed point is always the same.

In SR systems, the functions computed by the blocks are allowed to change between ticks, as long as they are monotonic in every tick. Thus, in a tick of the SR system, the behavior of a block is divided into two phases, which we call *produce* and *transition*. In the produce phase, the current function of the block is evaluated to determine its outputs given the information about its inputs. In the transition phase, the function may be changed in preparation for the next tick. Moreover, execution of the SR system also has two phases. In the produce phase, each block can be invoked in its produce phase more than once such that more outputs can be determined if newer information about inputs is observed from outputs of other blocks. In the transition phase, each block is invoked in its transition phase only once after the fixed point is found.

Most familiar functions are *strict*, meaning that all inputs must be known before any outputs can be determined. Strict functions are always monotonic. In the SR systems, a directed loop of blocks with strict functions always has the least fixed point with all unknown ($\perp$), which means that the directed loop is a causality loop. Nevertheless, it is not uncommon to have functions where the outputs can be determined even if some of the inputs are not known. The use of non-strict functions allows directed loops with less trivial least fixed point in the SR systems.

3.3.1 FSM inside SR

When embedding an FSM as an SR block, we need to distinguish two phases in the reaction of the FSM corresponding to the two phases of the SR block. Let us first focus on the FSM without states being refined. It is simple if all inputs to the SR block are known in a tick. In the produce phase, the FSM reacts to input events (translated directly from inputs

of the block) by possibly emitting output events (translated directly to outputs of the block). In the transition phase, the FSM takes the enabled transition to change state. For example, as depicted in figure 3.11, two FSMs are embedded within an SR system. In a tick of the SR system, suppose that both inputs *u* and *v* are present and both FSMs are in state α. The SR system executes as follows in an order with block **A** before block **B**.

- Produce phase of **A**: Since input *u* is present, FSM **A** emits an output event *x*. In SR, this makes output *x* known to be present.
- Produce phase of **B**: Since input *w* is present observed from output *x*, FSM **B** emits an output event *y*. Then, in SR, output *y* is determined to be present.
- Transition phase of **A**: FSM **A** takes the transition from state α to state β.
- Transition phase of **B**: FSM **B** takes the transition back to state α.

Frequently, the inputs to the FSM as an SR block are not completely known. As in the previous example, if we choose an order with block **B** before block **A**, the input *w* will be unknown for the first produce phase of block **B**. Thus, we need to define how the FSM behaves when there exist unknown inputs. One direct approach is to treat the FSM as a

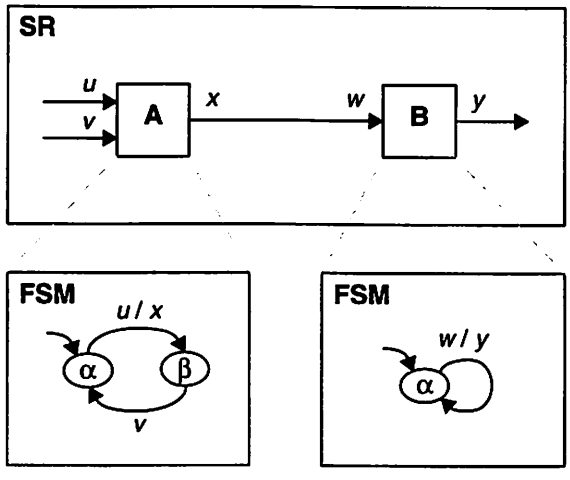


Figure 3.11 Two FSMs are embedded within an SR system.

strict function. If any inputs are unknown, the FSM simply has all outputs as unknown in the produce phase, or stays at current state in the transition phase. However, this approach has an undesirable and unnecessary side effect: A directed loop connecting only FSMs is always considered as a causality loop since the FSMs are treated as strict functions.

The approach we prefer is to enable the FSM inside an SR block to be evaluated as a non-strict function if possible. Consider an example illustrated in figure 3.12, where two FSMs are embedded within an SR system and are enclosed in a directed loop. When the FSM in block **A** is in state α, the function mapping from inputs u and v to output x is

$$f_{\alpha}^{x}(u, v) = (u \wedge v) \vee (\neg u \wedge v) = u.$$

This simplified function is not strict since it does not depend on input v. Therefore, if the FSM is in state α and input u is observed to be present or absent, we can determine whether output x will be present or absent without observing input v. The above analysis can be automated to get a simplified function for each output at each state of an FSM using standard techniques from digital logic design. For example, *binary decision diagrams* (BDDs) [21] can be used for analyzing the pure FSM, and *multi-valued decision diagrams*

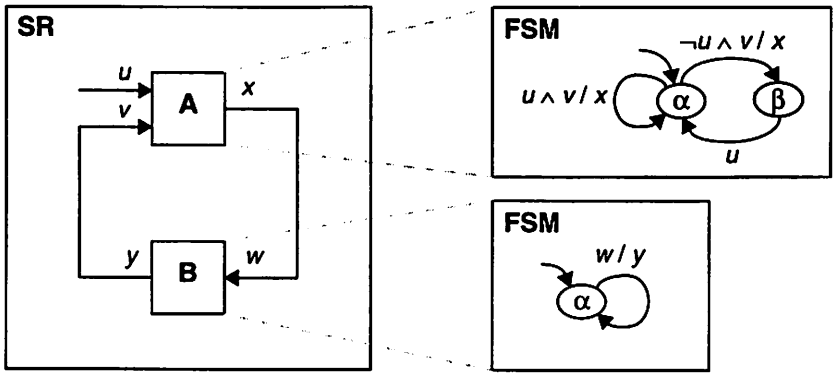


Figure 3.12 Two FSMs are embedded within an SR system and are enclosed in a directed loop.

(MDDs) [55] can be used for the valued FSM. These simplified functions will indicate for each state what inputs need to be known to determine an output.

With the simplified functions, the two phases of reaction for the FSM are modified as follows. In the produce phase, the FSM examines whether the simplified function for any output can be evaluated by observing whether enough inputs are known. If yes, the simplified function is evaluated and the output is determined. If not, the output remains unknown. In the transition phase, the FSM takes the transition enabled by current inputs, but ignores the action of that transition. Consider the example of figure 3.12. Suppose that input u is present and both FSMs are in state α. The SR system executes as follows in an order with block **B** before block **A**.

- Produce phase of **B**: Since input w is unknown, the simplified function $f_{\alpha}^{y}(w) = w$ cannot be evaluated, and thus output y is unknown.
- Produce phase of **A**: Although input v is unknown observed from output y, FSM **A** can still determine output x to be present from the simplified function $f_{\alpha}^{x}(u, v) = u$ since input u is present.
- Produce phase of **B**: Since input w is present observed from output x, FSM **B** can determine output y to be present from the simplified function $f_{\alpha}^{y}(w) = w$.
- Transition phase of **A**: Under the situation that both inputs u and v are present, FSM **A** takes the transition from state α to state β.
- Transition phase of **B**: Under the situation that input w is present, FSM **B** takes the transition back to state α.

Now consider that embedded within an SR system is a hierarchical FSM. Since a slave subsystem of the master FSM may or may not support two phases of execution, we distinguish two cases for the hierarchical FSM. If the slave subsystem of the current state sup-

ports two phases of execution, the produce phase of the hierarchical FSM should consist of the produce phase of the slave subsystem followed by the produce phase of the master FSM. Similarly, the transition phase of the hierarchical FSM should consist of the transition phase of the slave subsystem followed by the transition phase of the master FSM. If the slave subsystem of the current state does not support two phases of execution, then we have to be more cautious. The function computed by the slave subsystem must be monotonic in every tick of the outer SR system. To ensure this, one approach is to consider the slave subsystem as a strict function. Therefore, in the produce phase of the hierarchical FSM, one step of the slave subsystem will not be invoked until all of its inputs are known.

3.3.2 SR inside FSM

Embedding an SR model inside a state of the FSM is straightforward. When a state of the FSM is refined into an SR subsystem, the semantics of SR are simply exported to the outer model in which the FSM is embedded. If the outer system of the FSM supports two phases of execution, the produce and the transition phases of the SR subsystem will be invoked respectively corresponding to those of the outer system. If the outer system of the FSM does not support two phases of execution, then one step of the SR subsystem is taken to be one tick, which consists of the produce and the transition phases of the SR subsystem.

3.4 Synchronous Dataflow

As depicted in figure 3.13, a system specified in synchronous dataflow (SDF) [62][63] consists of a set of blocks communicating through directed arcs. A block represents a computational function that maps input data into output data. The data are divided into *tokens*, which are treated as indivisible units. An arc represents a sequence of tokens conceptually carried by a unidirectional first-in-first-out (FIFO) queue. A *firing* of a block is

an indivisible 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 or produced on each input or output arc can be viewed as part of the *type signature* of the block (along with the data type of the tokens, of course). These numbers can be used to unambiguously define an *iteration*, or minimal set of firings that return the FIFO queues to their original sizes. This is done by writing for each arc a balance equation

$$p_i r_i = c_j r_j$$

where the arc here is assumed to go from block i to block j, and on this arc, block i produces p_i tokens and block j consumes c_j tokens. The variables r_i and r_j are unknowns that represent the numbers of firings of blocks i and j, respectively. The balance equation tells us that within one iteration the total number of tokens produced by the source block equals the total number of tokens consumed by the destination block.

Suppose that there are M arcs and N blocks in an SDF graph. There will be M balance equations in N unknowns. It can be shown [62] that either there exists a unique smallest positive solution for the unknowns, called the *minimal solution*, or the only solution is all zeros. If the minimal solution exists, an iteration of the SDF graph is defined to consist of exactly r_i firings for each block i. For the example of figure 3.13, we have the minimal

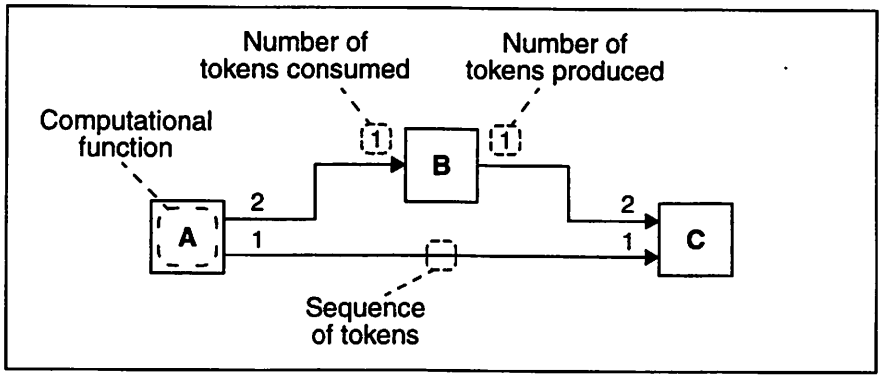


Figure 3.13 A synchronous dataflow (SDF) system.

solution where $r_A = 1$, $r_B = 2$, and $r_C = 1$, and thus the number of firings for each block is determined accordingly. If the solution is all zeros, the SDF graph is said to be *inconsistent*, and it will require unbounded sizes of FIFO queues. This SDF graph is considered invalid. In figure 3.14, for example, each firing of blocks **A** and **B** produces one token on the arc from **B** to **C**, and two tokens on the arc from **A** to **C**. However, each firing of block **C** only consumes one token from each arc. Hence, repeated firings will cause tokens to accumulate on the arc from **A** to **C**. By finding whether the minimal solution exists, it is decidable whether a given SDF graph can be executed in bounded memory.

Once the number of block firings in an iteration is found for an SDF graph, the firing schedule can be determined by simulating the execution of the SDF graph. This is done by selecting any block that has enough tokens on its input arcs, simulating its firing effect on the sizes of FIFO queues, and continuing until either all blocks have been selected as many times as they should fire or no more block has enough input tokens. If each block is selected as many times as it should fire, the sequential list of block selections forms one possible firing schedule. Otherwise, the SDF graph is said to be *deadlocked* since all blocks will halt firings and wait for tokens on some input arcs before an iteration is completed. This SDF graph is also considered invalid. Figure 3.15 illustrates such an SDF graph. Blocks A and B are both waiting for one token from each other before they can fire.

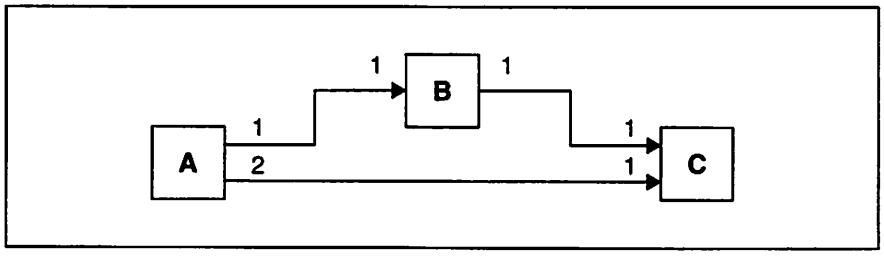


Figure 3.14 An inconsistent SDF graph.

The simplest SDF graphs are *homogeneous*, where every block consumes or produces exactly one token on each input or output arc. For such graphs, an iteration always consists of exactly one firing of each block. The schedule of the firings must obey the data precedences (a token must be in an FIFO queue before it can be consumed). Therefore, to avoid deadlock, all directed loops in a homogeneous SDF must have at least one initial token (often called a *delay*) on at least one arc in the loop. For example, the homogeneous SDF graph in figure 3.15 deadlocks. The deadlock can be avoided by adding an initial token on either arc, allowing one of the blocks to fire first. Arbitrary SDF may require more than one token on some arcs. Nevertheless, it is decidable whether a given set of initial tokens is sufficient to prevent deadlock.

3.4.1 FSM inside SDF

When an FSM describes a block of an SDF graph, we relate one firing of the SDF block to one reaction of the embedded FSM. Moreover, the FSM must externally obey SDF semantics. Hence, the FSM performs one reaction in one firing of the SDF block, which must consume and produce a fixed number of tokens at every input and output. A subtlety for SDF is that absence of a token is not a well-defined, testable condition. Thus, the absence of an event in FSM must appear explicitly as a token in SDF. A possible approach is described as follows. For a pure event, its presence and absence is encoded

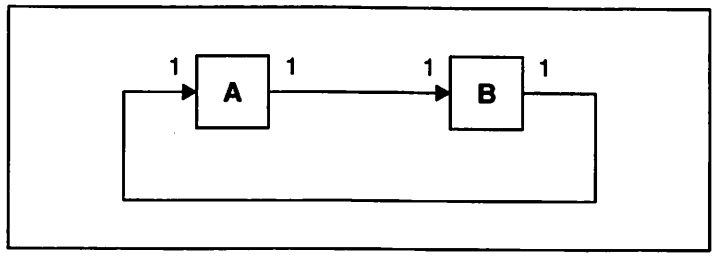


Figure 3.15 A deadlocked SDF graph.

using a boolean-valued token. A false-valued token means the event is absent and a true-valued token means it is present. For a valued event, both its status and value are encoded using a valued token. One rarely-used value (such as "infinity") of the token is reserved to mean the event is absent, and each of other values simply means the event is present with that value.

As shown in figure 3.16, two FSMs are embedded within an SDF system. Since this SDF graph is homogeneous, one iteration consists of a single firing of each block. Suppose that in some iteration the input tokens have values indicating that input event u is present and input event v is absent, and both FSMs are in state α. The SDF system executes as follows.

- Fire **A**: Since input event u is present, FSM **A** takes the transition from state α to state β, and emits the output event x. In SDF, presence of the event is encoded as a token at output x.

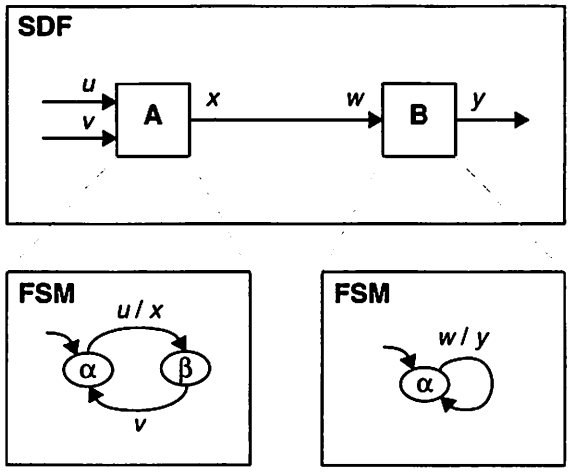


Figure 3.16 Two FSMs are embedded within a homogeneous SDF system.

- Fire **B**: Since input event w is present indicated by the token passed from output x, FSM **B** takes the transition back to state α, and emits the output event y. In SDF, presence of the event is encoded as a token at output y.

In fact, the SDF block in which an FSM is embedded can be non-homogeneous. I.e. the block can consume and produce multiple tokens at some inputs and outputs. Under this circumstance, each token at a given input or output of the SDF block needs to be distinguished as an input or output event of the FSM. This allows the FSM to refer to those events in the guards and the actions. Therefore, we syntactically differentiate each token at an input or output as an event by concatenating its occurrence to its name. Borrowed from the notation used in the Signal language [11], "u" denotes an event corresponding to the last (newest) token consumed or produced at input or output u, "u\$1" denotes an event corresponding to the second last token, "u\$2" denotes an event corresponding to the third last token, etc. Consider an example shown in figure 3.17. In the SDF, the number following a dot after an input or output name indicates the number of tokens consumed or produced by the corresponding block. The guard of the transition from state α to state β in

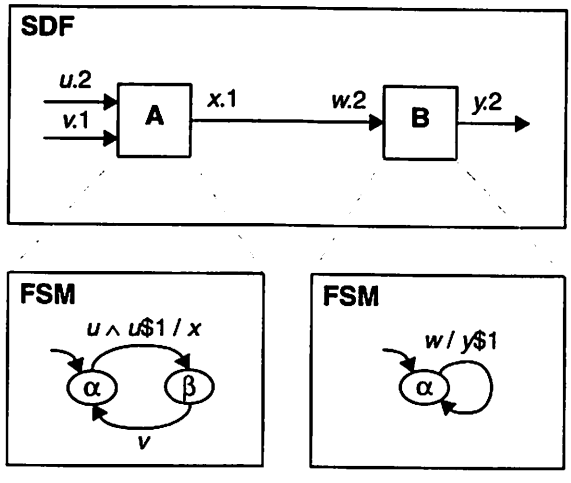


Figure 3.17 Two FSMs are embedded within a non-homogeneous SDF system.

FSM **A** is "$u \wedge u\$1$". Thus, for this transition to be enabled, both tokens consumed from input u must have the value representing a present event. The action of the transition from state α to state α in FSM **B** is "$y\$1$". Thus, when this transition is taken, the first (older) token on output y has a value representing a present event (because $y\$1$ is mentioned), and the second (newer) token has a value representing an absent event (because y is not mentioned).

3.4.2 SDF inside FSM

When an SDF graph refines a state of an FSM, one step of the slave SDF graph is taken to be one iteration. Thus, when the refined state is the current state, the hierarchical FSM reacts with an iteration of the slave SDF graph followed by a reaction of the master FSM. If the SDF graph is homogeneous, at each iteration each input of the SDF subsystem will have an explicit token translated from the corresponding input event of the FSM even if the event is absent.

If the SDF graph is not homogeneous, the semantics becomes more subtle because the SDF subsystem may not have enough input tokens to complete one iteration in a reaction of the FSM. One possible approach is that when input tokens are not sufficient to cycle through one iteration, the SDF subsystem will simply return and produce no output tokens. Only when enough input tokens have accumulated will one iteration of the SDF subsystem be executed and output tokens be produced. However, this is not always the most efficient approach, especially when the FSM is within another SDF system.

In figure 3.18, an SDF subsystem is embedded in an FSM that is within another SDF system. Solving the single balance equation[1] for the inner SDF subsystem indicates that one iteration will consist of two firings of block **C** and one firing of block **D**. Therefore,

1. There is only one arc entirely inside the SDF subsystem. Hence, there is only one balance equation, which is $r_{\mathbf{C}} = 2r_{\mathbf{D}}$.

the type signature for this subsystem indicates that four tokens will be consumed from input *u* and two from input *v*, and two tokens will be produced to output *y*. An alternative approach we choose is that the type signature of the inner SDF subsystem becomes the type signature of the FSM itself. In other words, the outer SDF system must treat the FSM as an SDF block with the given type signature. As a result, the FSM will not react until there are sufficient input tokens for the inner SDF subsystem to cycle through one iteration.

Since there may be more than one state of the FSM refined into an SDF graph, the type signatures may not be the same in different states. In this case, the FSM cannot be treated as an SDF actor because the number of tokens consumed or produced is dependent on the

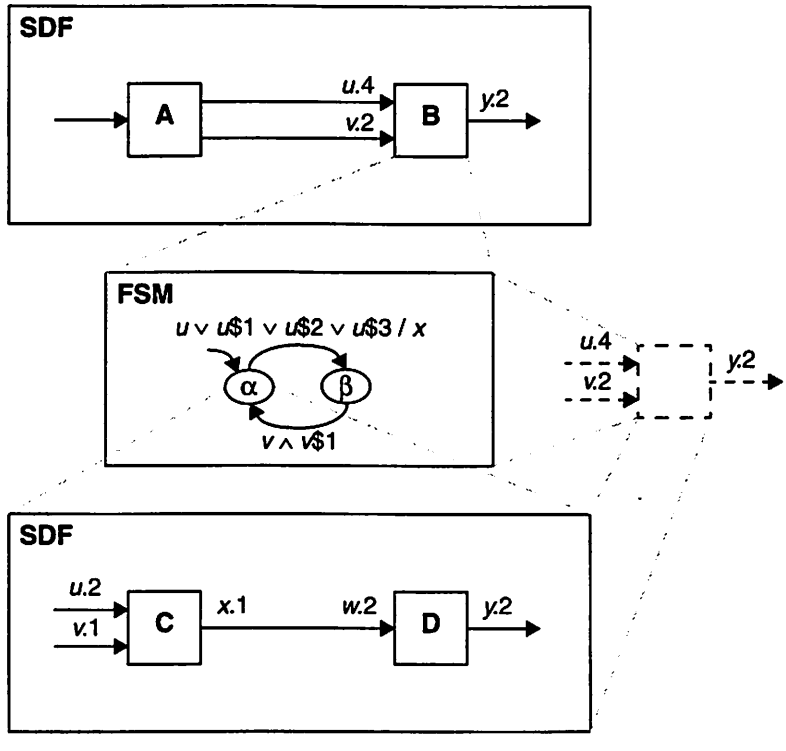


Figure 3.18 The type signature of the inner SDF subsystem becomes the type signature of the FSM itself. Thus, the outer SDF system treats the FSM as a non-homogeneous block with the given type signature.

state of the FSM. Therefore, if the outer system of the FSM is again an SDF graph, all slave SDF graphs in the FSM are required to consume or produce the same number of tokens at each input or output.

3.5 Dynamic Dataflow

Dynamic dataflow (DDF) [61][58][41] is a superset of SDF. In an SDF graph, each block is required to consume and produce a fixed number of tokens at all its firings. In a DDF graph, however, each block is permitted to consume and produce a varying number of tokens at distinct firings. This enhancement by itself is sufficient to let the DDF model have expressive power equivalent to a universal Turing machine [22]. The trade-off is that bounded memory and deadlock properties become undecidable for a given DDF graph.

Furthermore, since the number of tokens consumed and produced by a block can vary from one firing to another, compile-time scheduling as in SDF is no longer possible. Instead, a run-time scheduler dynamically detects a block to fire according to its *firing rules*, each of them specifying how many tokens are required on each input prior to a firing of the block. Hence, the basic operation for the DDF scheduler is to repeatedly scan the list of blocks in a given DDF graph, and to execute the block if it has sufficient input tokens to fire according to its current firing rule.

3.5.1 FSM inside DDF

When an FSM is embedded inside a DDF block, the key is to determine how many tokens need to be consumed and produced at each input and each output of the block for a firing. For each state of the FSM, we find the number of tokens to be consumed at each input of the block by looking at the guards of all outgoing transitions. At least one token is consumed at an input if the corresponding input event is mentioned in any of the guards. In addition, if any input events refer to multiple tokens on the input using the notation

"*u*$*i*", we find the largest integer *i* mentioned in the guards, and this integer *i* plus one is taken to be the number of tokens to be consumed. Figure 3.19 shows an FSM embedded within a DDF system. In state α, two tokens are required from input *u* (because *u*$1 is mentioned) and no token from input *v* (because no *v* is mentioned). In state β, only one token is required from each of the inputs *u* and *v*. These numbers specify the number of tokens that must be present on the inputs for the DDF block to fire when the FSM is in the given state.

Similarly, for each state of the FSM, we find the number of tokens to be produced at each output of the block by looking at the actions of all outgoing transitions. Consider the example of figure 3.19 again. In state α, output *x* will produce two tokens (because *x*$1 is mentioned). In state β, output *x* will produce no token (because no *x* is mentioned).

3.5.2 DDF inside FSM

When a DDF graph refines a state of an FSM, the firing rules of the DDF graph are exported to the environment of the FSM. Hence, if the refined state is the current state, only when the firing rules of the DDF subsystem are met will the FSM react. Moreover, in

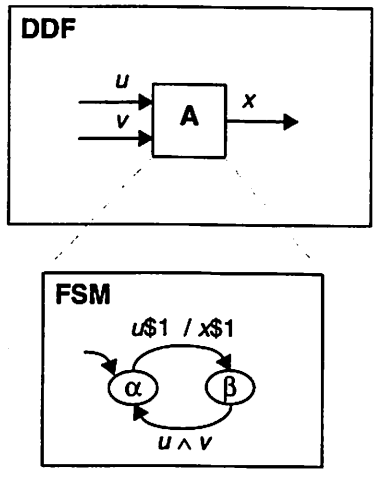


Figure 3.19 An FSM are embedded within a DDF system.

SDF, one iteration is determined by solving the balance equations. However, in DDF, the balance equations do not apply in the same way. Therefore, one iteration of the DDF graph inside an FSM must be clearly defined in order to be one step of the slave subsystem. In Ptolemy [23], for example, one iteration in the DDF domain consists of one firing of each source block (one without inputs) followed by the firings of non-source blocks as many times as possible. In case those non-source blocks have tokens but do not have enough yet at some inputs, source blocks may be invoked more than once to provide the required tokens in one iteration. This ensures that the relative number of firings for each source block is the same in DDF as it would be in SDF.

3.6 Communicating Sequential Processes

Under communicating sequential processes (CSP) [48][49] paradigm, a system is modeled as a network of concurrently operating sequential processes (blocks), such as in figure 3.20. The processes communicate with each other through unidirectional channels (directed arcs). Each channel provides a direct communication between two processes via *rendezvous* (or *synchronous message passing* [4]). This means that both sending and

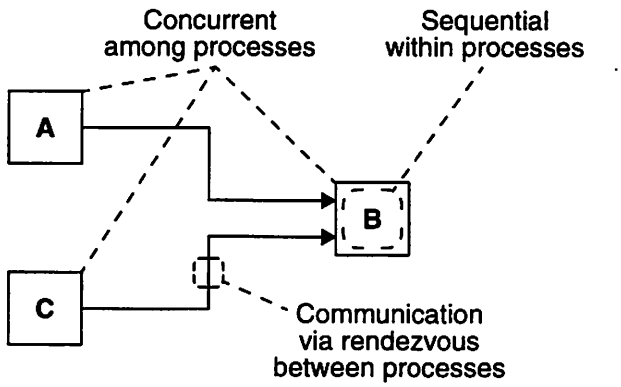


Figure 3.20 A communicating sequential processes (CSP) system.

receiving of a message on the channel must synchronize at every communication point. If one process is first ready to send a message on a channel, it stalls until the other process is also ready to receive the message on that channel. Similarly, if one process is first ready to receive a message on a channel, it stalls until the other process is also ready to send the message on that channel. When both processes are ready, the message transfer is initiated and completed as an indivisible action.

Figure 3.21 illustrates an example of rendezvous communication among those three processes of figure 3.20, where **A** and **B** are connected by a channel *x*, and **B** and **C** are

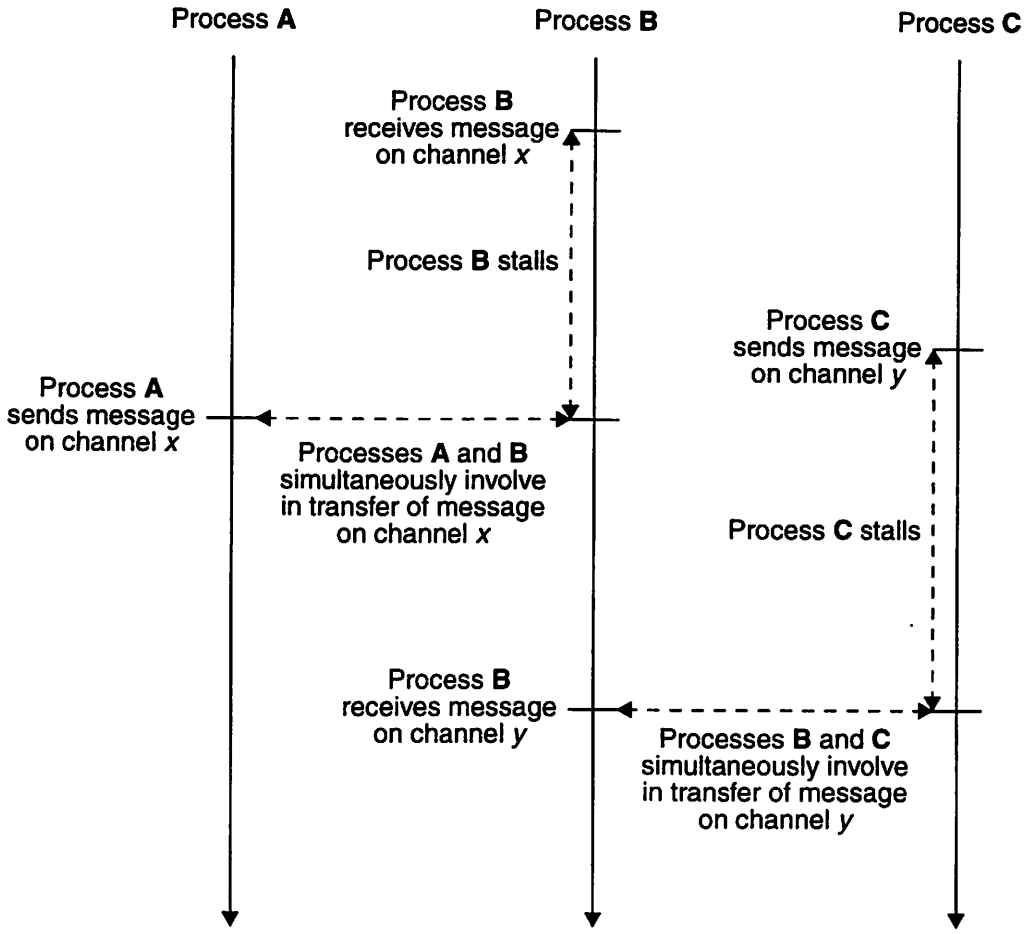


Figure 3.21 An example of rendezvous communication among three processes.

connected by a channel *y*. When process **B** wishes to communicate with process **A** by receiving a message through the channel *x*, it stalls since process **A** does not want to send the message yet. Subsequently, when process **C** tries to send a message on the channel *y*, it stalls too and waits to rendezvous with process **B**. Later when process **A** decides to send the message on channel *x*, processes **A** and **B** simultaneously involve in transfer of the message on channel *x*, and then both processes proceed as normal. When process **B** proceeds to receive the message on the channel *y*, rendezvous between processes **B** and **C** occurs on channel *y*.

The basic rendezvous mechanism only allows a process to wait on one channel at a time. If a process wishes to communicate with other processes through multiple channels, the order in which those channels will rendezvous has to be known and fixed. In the example of figure 3.21, process **B** wishes to receive the message on channel *x* before channel *y*. Hence, even though process **C** sends a message before process **A** does, process **B** waits to rendezvous with process **A** first.

Often, a process wishes to wait on any one of multiple channels. This can be achieved by nondeterministic rendezvous, which is an extension of the basic rendezvous. Nondeterministic rendezvous allows a process to conditionally receive or send messages and thus to monitor on multiple channels until one of the monitored channels completes a rendezvous. Consider the example of figure 3.20. Suppose that process **B** conditionally receives a message on channel *x* or *y*. As shown in figure 3.22, when process **C** first sends the message on channel *y*, process **B** will rendezvous with process **C**. Later when process **A** sends a message on channel *x*, it stalls until process **B** wishes to conditionally receive the message again.

3.6.1 FSM inside CSP

When an FSM is embedded as a CSP process, an input event of the FSM is present only when the corresponding input of the CSP process completes a rendezvous. Thus, the trigger of the guard in each transition of the FSM specifies a set of inputs that have to rendezvous for the transition to be enabled. For each state of the FSM, we get a list of inputs including those mentioned by the triggers of the guards in any outgoing transitions. We call these inputs the *guarded inputs*, which need to rendezvous when the FSM is in the given state. In a reaction, the FSM keeps conditionally receiving the message from each

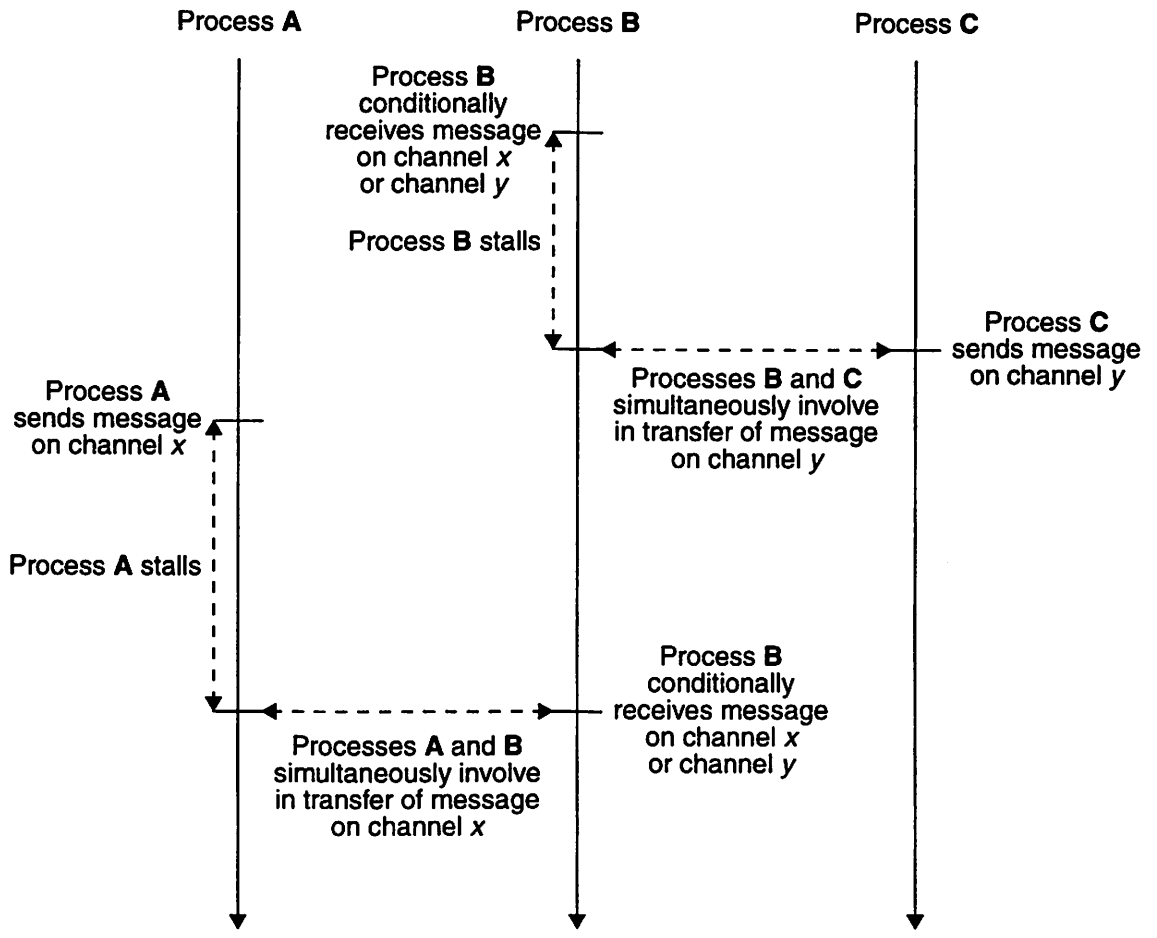


Figure 3.22 An example of nondeterministic rendezvous among three processes.

guarded input until one transition is enabled or every guarded input completes a rendezvous at least once. To maintain fairness among guarded inputs, we restrict each guarded input to rendezvous at most once in a reaction. Similarly, when an enabled transition is found in a reaction, the FSM keeps conditionally sending the message on each output listed in the action of the enabled transition until each of those outputs completes a rendezvous once.

3.6.2 CSP inside FSM

When a CSP model refines a state of an FSM, one step of the slave subsystem needs to be defined. A possible approach is to let the deadlock of the CSP model mark the end of a step for the subsystem [80]. Hence, one step of a slave CSP subsystem is the execution of that subsystem until no processes can make progress (i.e. all processes stall).

Furthermore, the CSP semantics needs to be exported to the environment of the FSM. In particular, if the outer model of the FSM is another CSP system, the outer CSP system must treat the FSM as a CSP process, which will rendezvous first on the inputs required by the inner CSP subsystem and then on the guarded inputs of the FSM. One subtlety is that the CSP model by itself is not compositional [80]. Hence, the processes embedded within the inner CSP subsystem may not have the same behavior as the processes embedded directly within the outer CSP system.

4

Applications

4.1 Embedded Systems

An embedded system [56] is a complex device with at least a general-purpose or application-specific microprocessor built into the device. Unlike a desktop or laptop computer that also utilizes a microprocessor, the functionality of an embedded system is solely dedicated to a specialized purpose, which typically involves the interaction with the real world (such as human users, motor vehicles, and the physical environment) through sensing and actuating interfaces. Therefore, a sizable portion of the specification requirements [37] for designing the embedded system tends to focus on the control logic that governs the interaction of the system with the real world.

Applications of embedded systems are abundant and quite diverse. Examples of the applications include consumer electronics (digital watches, digital cameras, compact disk players, etc.), household appliances (microwave ovens, washing machines, etc.), real-time controllers (railroad controllers, anti-lock braking systems, automotive engine controllers, etc.), and telecommunication systems (modems, cellular phones, answering machines, etc.).

In the following sections, we will demonstrate how to apply *charts to embedded systems in two examples: the digital watch (interacting with the human user) and the railroad controller (interacting with trains).

4.1.1 Example: Digital Watches

4.1.1.1 Problem Description

The digital watch is a commonly used example for specification of control functionality by various specialized languages, such as Statecharts [43], Argos [53], and Esterel [15]. Our version of the digital watch (see figure 4.1) contains a display area, a two-tone beeper, a light, and four control buttons as the user interface. This watch can display the time in either am/pm or 24-hour format. The beeper can be enabled to alarm on a preset time daily, to chime on the hour, or both. The light is included for illumination and has an automatic shut-off feature. The four buttons are marked as *mode*, *update*, *select*, and *adjust*, respectively, for distinction.

The human user interacts with the digital watch through certain sequential combinations of button pressing, which are summarized as follows.

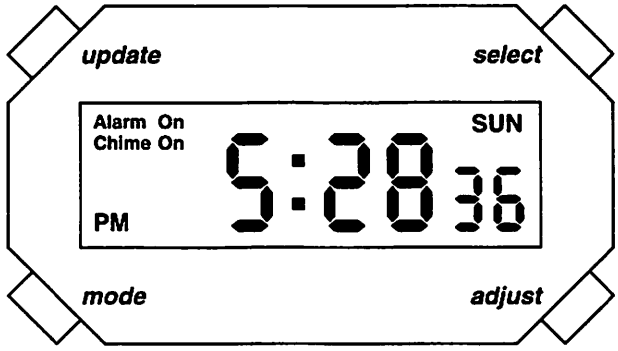


Figure 4.1 Our version of the digital watch.

- Time mode setting:

  1. Press *mode* to switch to time display if the watch is not in time display.
  2. Press *adjust* to switch between am/pm and 24-hour formats.

- Alarm mode setting:

  1. Press *mode* to switch to alarm display if the watch is not in alarm display.
  2. Press *adjust* to enable the daily alarm, the hourly chime, or both.

- Time and day updating:

  1. Press *mode* to switch to time display if the watch is not in time display.
  2. Press *update* to switch to time update in which second digits will flash.
  3. Press *select* to switch the flashing to second digits, minute digits, hour digits, or day of week letters.
  4. Press *adjust* to reset the second as zero or to advance the minute, the hour, or the day of week, when the digits or letters are flashing.

- Alarm updating:

  1. Press *mode* to switch to alarm display if the watch is not in alarm display.
  2. Press *update* to switch to alarm update in which minute digits will flash.
  3. Press *select* to switch the flashing to minute or hour digits.
  4. Press *adjust* to advance the minute or the hour, when the digits are flashing.

- Light illumination:

  1. Press *mode* twice to turn on the light. The light is automatically shut off after a certain period of illumination.

4.1.1.2 *charts Realization

To simulate the real-time behavior of the digital watch, we use DE as the topmost level (see figure 4.2) of our *charts realization to model the environment of the watch (including the human user). This DE model consists of a **display**, a **beeper**, a **light**, a **clock** to

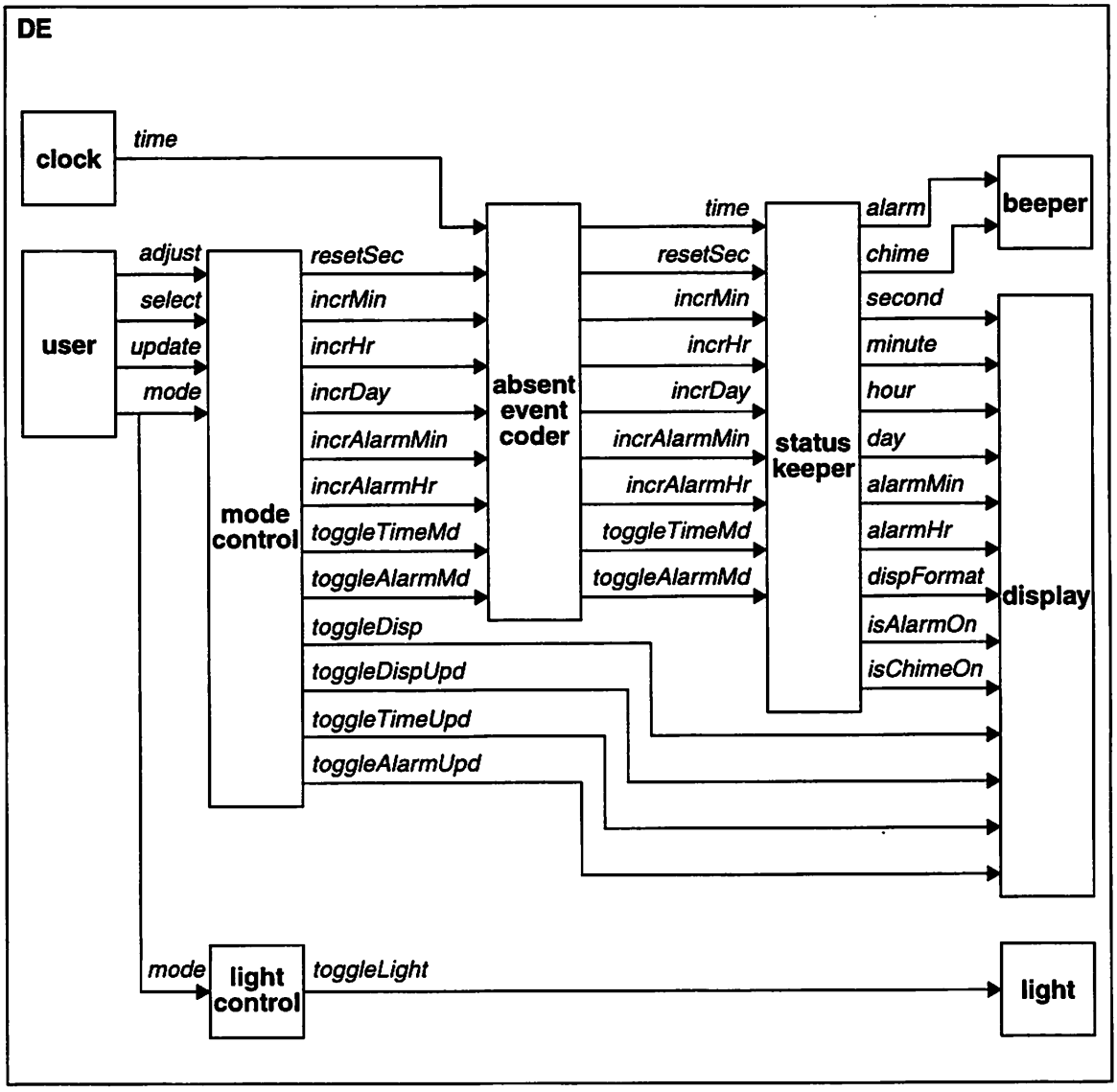


Figure 4.2 The topmost level of the *charts realization for our digital watch.

generate time ticks, a **user** that asserts the button pressing events, a **status keeper** to maintain current time and alarm settings, an **absent event coder** to perform encoding of absent events to zero-valued tokens (for the SDF model within the **status keeper**, which will be detailed later), and two control units: **light control** and **mode control.**

The computation part of the watch mainly resides in the **status keeper**, which is modeled by an SDF graph illustrated in figure 4.3(a). This SDF graph contains a **time keeper** and an **alarm keeper** to keep track of current time and alarm settings, respectively. It also includes the comparison and logic blocks for detecting the time to sound the alarm and the chime, respectively. Moreover, as shown in figure 4.3(b) and (c), these two SDF graphs further refine the **time keeper** and the **alarm keeper**, respectively, which are basically composed of counters with various maximum counts.

Inside the **light control** unit of the watch, we have a hierarchical FSM depicted in Figure 4.4 to control the illumination of the light. In figure 4.4(a), this top-level FSM has three states: **light off**, **wait mode**, and **light on**. It starts in the **light off** state. When a *mode* event arrives, the FSM takes the transition to the **wait mode** state and waits for another *mode* event to arrive. If the *mode* event arrives in time, the FSM emits a *toggleLight* event to turn on the light and takes the transition to the **light on** state. Otherwise, if the *mode* event does not arrive before a local *timeout* event occurs, the FSM gives up waiting and takes the transition to the **light off** state. This *timeout* event is asserted by a slave SDF graph (see figure 4.4(b)) shared by the **wait mode** and the **light on** states. In the **light on** state, the FSM waits until the *timeout* event occurs. Then, the FSM emits the *toggleLight* event to turn off the light and takes the transition to the **light off** state. Even though the **wait mode** and the **light on** states share the same slave SDF graph, the counting for time outs in the SDF graph is reset every time when either state is entered with the initial entry.

Inside the **mode control** unit of the watch, we have another hierarchical FSM (see figure 4.5) to govern mode switching of the watch in response to the button pressing events

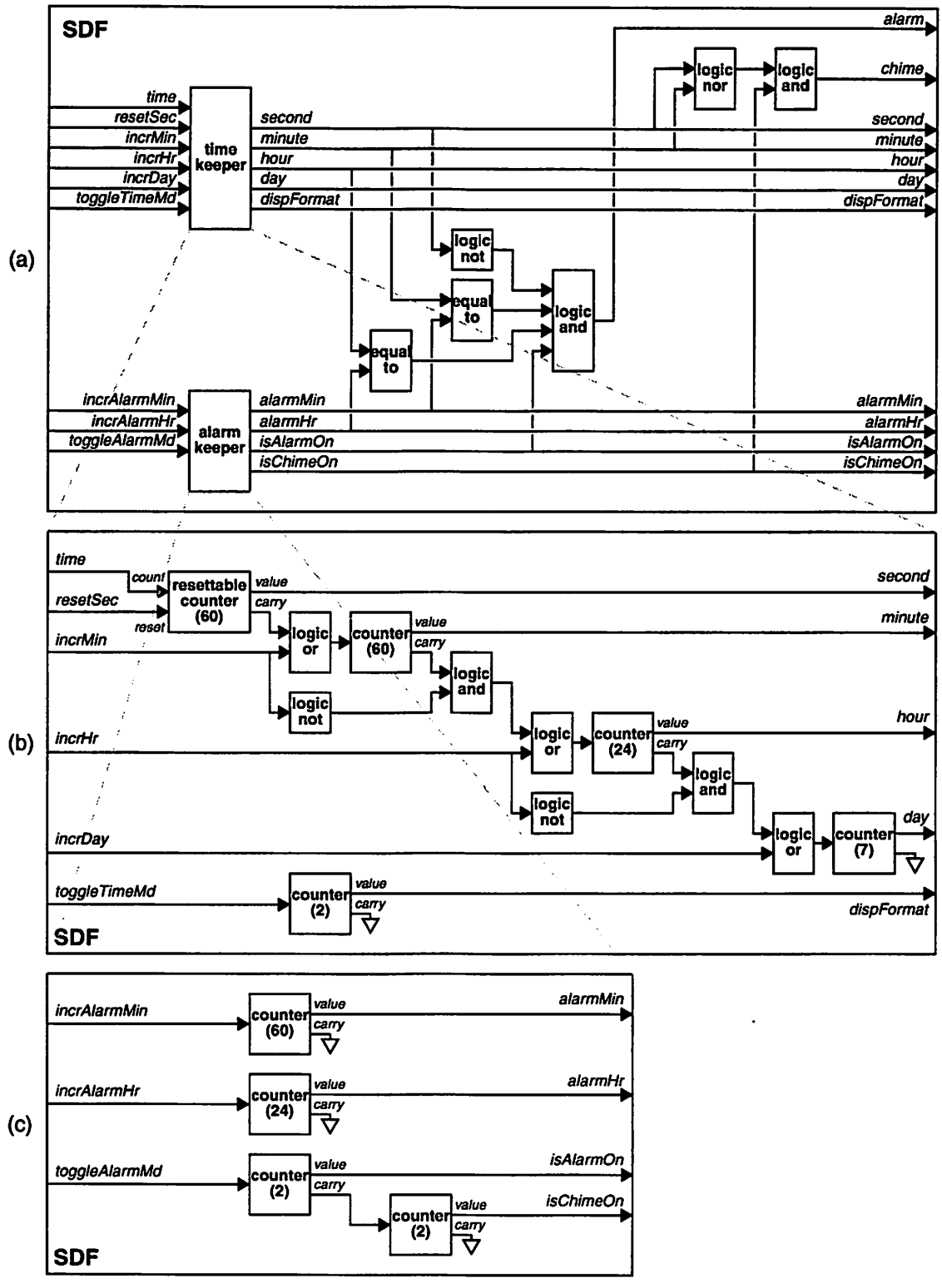


Figure 4.3 The status keeper of figure 4.2 is modeled by this hierarchical SDF graph.

and to generate the corresponding control events. The top-level FSM in figure 4.5(a) has four states: **time display, alarm display, time update,** and **alarm update.** It starts in **time display.** The *mode* event switches the watch between **time display** and **alarm display.** In addition, if in **time display** or **alarm display,** the watch is switched by an *update* event to **time update** or **alarm update** accordingly, and vice versa. Moreover, the four states of this FSM are all further refined. The FSM shown in figure 4.5(b) refines the **time display** state and manages the display format (either am/pm or 24-hour). The **alarm display** state is refined by the FSM in figure 4.5(c), which takes care of enabling or disabling the daily alarm or the hourly chime. Figure 4.5(d) and (e) shows the FSMs that refine the **time update** and the **alarm update** states, respectively. These two FSMs are in charge of the time and the alarm updating, respectively.

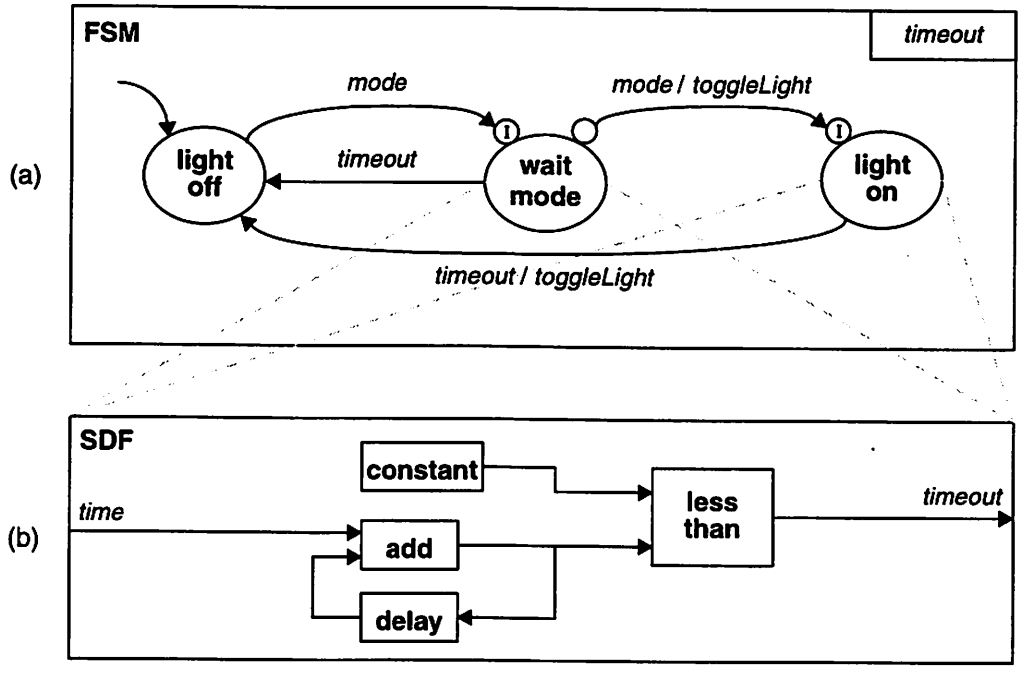


Figure 4.4 Inside the light control of figure 4.2, we have this hierarchical FSM.

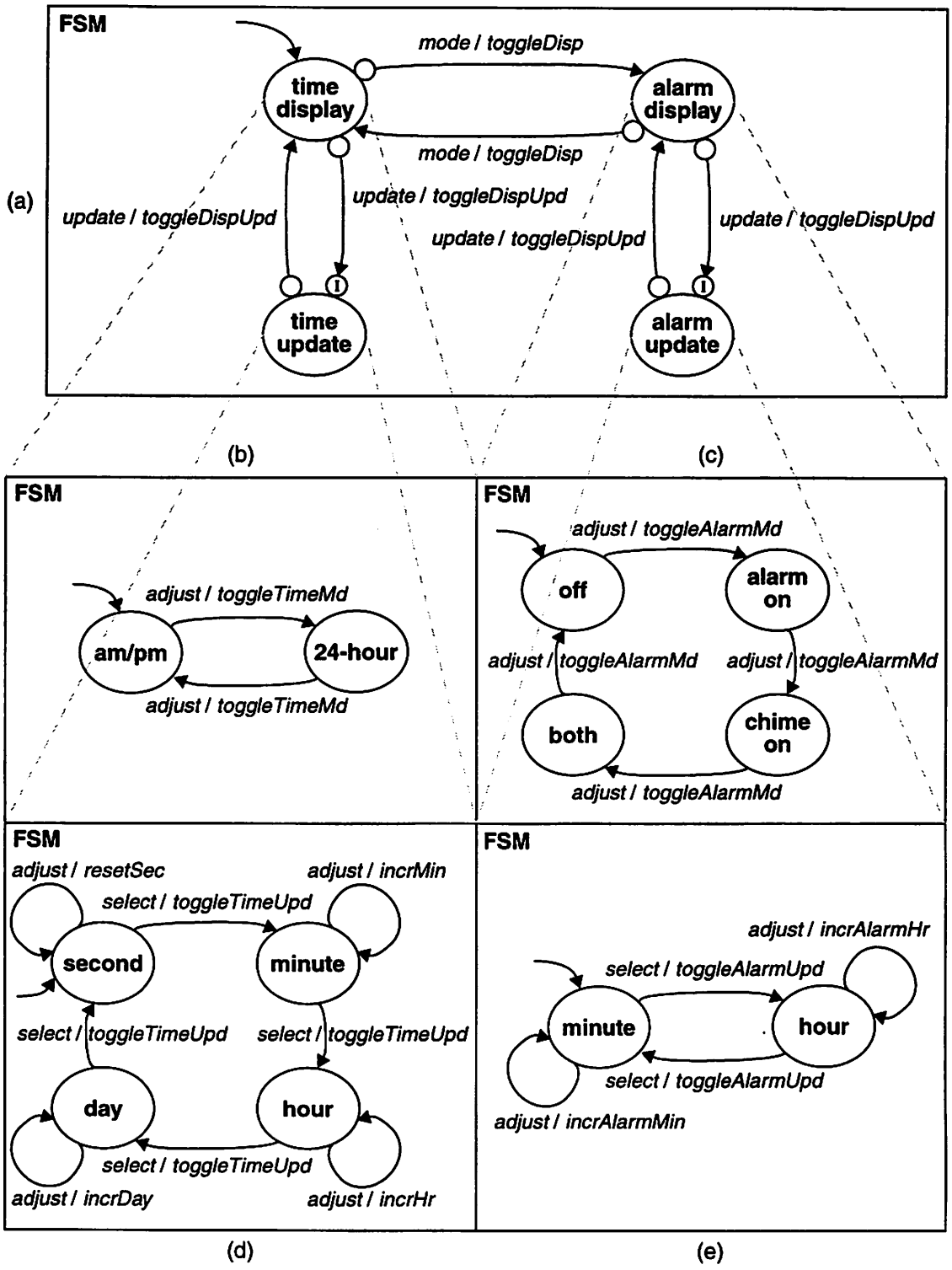


Figure 4.5 Inside the mode control of figure 4.2, we have this hierarchical FSM.

4.1.2 Example: Railroad Controllers

4.1.2.1 Problem Description

As shown in figure 4.6, there are two circular railroad tracks in the railroad control problem [3]. Two trains (A and B) travel counterclockwise on the two tracks, which merge on a bridge that can accommodate only one track. In addition, two signals (A and B) are placed at both entrances of the bridge, respectively, for controlling the access to the bridge. When train A arrives at the entrance of the bridge, if signal A is green, train A may enter the bridge. Otherwise, if signal A is red, train A must wait until the signal turns green, and then it may enter the bridge. Signal B operates in the same fashion as signal A does. However, a head-on train crash may happen if the signals are both green when the trains both arrive at the entrances of the bridge. Therefore, a railroad controller needs to be carefully designed to coordinate the two signals.

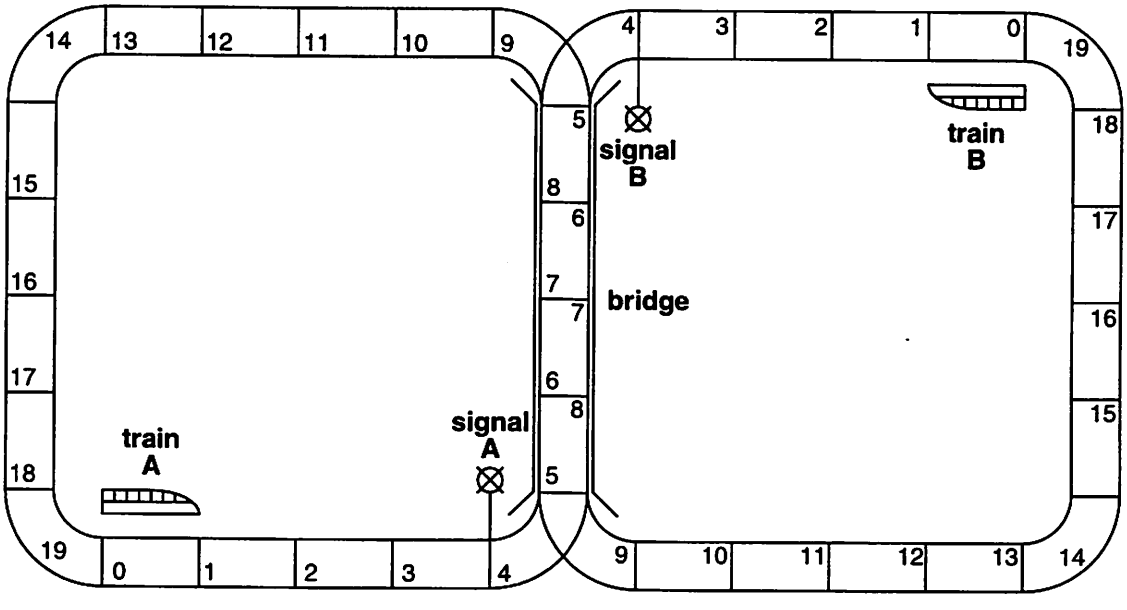


Figure 4.6 The railroad control problem.

One simple and feasible railroad controller operates as follows, assuming that both signals start as green and no train is on the bridge yet.

- Case I (Only one train arrives): Suppose that train A arrives at the entrance of the bridge. The controller keeps signal A as green but changes signal B to red. Then, train A may proceed onto the bridge. When train A leaves the bridge, the controller switches signal B back to green.
- Case II (One train arrives before the other): Suppose that train A arrives at the entrance of the bridge first. The controller keeps signal A as green but turns signal B into red. This allows train A to access to the bridge. At this point, suppose that train B also arrives, this train must stop and wait since signal B is red now. When train A exits from the bridge, signals A and B are changed to red and green, respectively, by the controller. Then, train B may enter the bridge. When train B leaves the bridge, the controller turns signal A back into green.
- Case III (Both trains arrive at the same time): When both trains arrive at the entrances of the bridge at the same time, we choose to grant the bridge-accessing privilege to train A. Therefore, the controller keeps signal A as green but switches signal B to red. When train A leaves the bridge, the controller turns signal A and B into red and green, respectively. Then, train B may proceed onto the bridge. When train B exits from the bridge, signal A is changed back to green by the controller.

4.1.2.2 *charts Realization

Our *charts realization for simulating the railroad control problem is shown in figure 4.7. To resolve the directed loops due to communication between the railroad controller and the trains, we use the SR model as the topmost level (a). This SR system contains a **clock** to generate ticks, the railroad **controller**, the two signals, and the two trains. In addi-

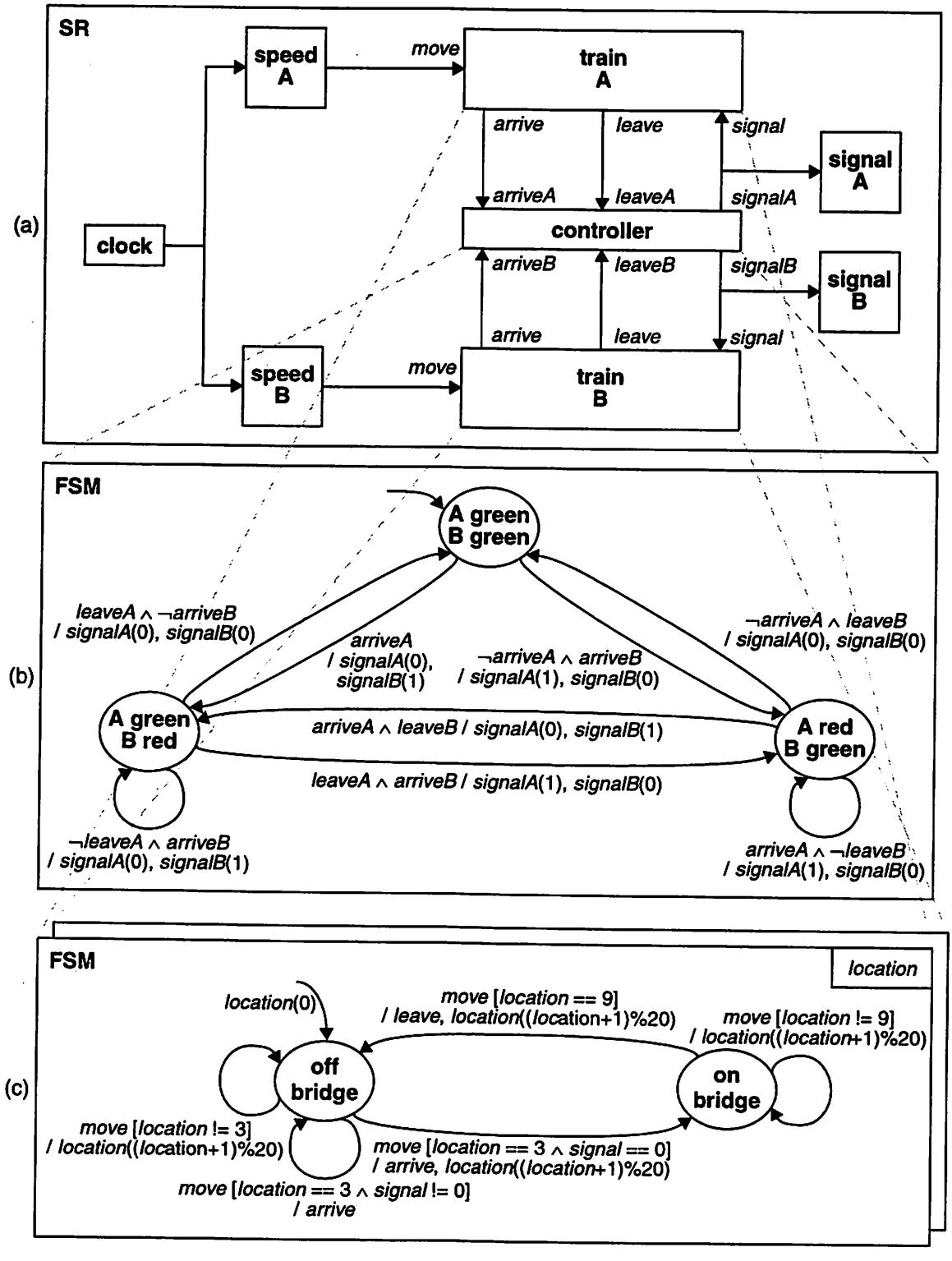


Figure 4.7 Our *charts realization for simulating the railroad control problem.

tion, two blocks, **speed A** and **speed B**, are included to adjust the speeds of the two trains, respectively. As shown in figure 4.6, each railroad track is divided into 20 segments. Every segment is marked with a number (or two numbers for those shared by two tracks) in order to indicate the location of the train on the track. The speed of a train is given by the number of ticks for the train to move one segment. In other words, in every certain number of ticks, each of **speed A** and **speed B** generates a *move* event to allow the corresponding train to move one segment.

The **controller** is modeled by the three state FSM illustrated in figure 4.7(b). The states are named by the statuses of the two signals: **A green B green**, **A green B red**, and **A red B green**. The **controller** starts in the **A green B green** state. On the other hand, each of the trains, **train A** and **train B**, is modeled by an FSM (see figure 4.7(c)) with two states: **off bridge** and **on bridge**. Each train starts in its **off bridge** state with its *location* initialized to segment 0. Before the *location* reaches segment 3 (i.e. the entrance of the bridge), each train simply moves one segment whenever the corresponding *move* event is present. Then, we distinguish the three cases as described in previous section.

- Case I (Only one train arrives): Suppose that the *location* of **train A** reaches the segment 3. When a *move* event is present, even though **train A** cannot decide which transition would be taken for now, the *arrive* event is known to be present since it actually only depends on the *move* event. This helps the **controller** know that the *arriveA* event is present and thus that the *signalA* and the *signalB* events are present with the values 0 (meaning green) and 1 (meaning red), respectively. Now, **train A** can decide which transition is to be taken, and as a result it moves its *location* by one segment and proceeds to the **on bridge** state. In the mean time, the **controller** takes the transition to the **A green B red** state. In the **on bridge** state, whenever a *move* event is present, **train A** simply moves its *location* by one segment until its *location* reaches segment 9 (i.e. the

train is leaving the bridge). Then, **train A** emits a *leave* event, moves its *location* by one segment, and proceeds to the **off bridge** state. This causes the **controller** to emit the *signalA* and the *signalB* events both with the value 0 and to take the transition to the **A green B green** state.

- Case II (One train arrives before the other): Suppose that the *location* of **train A** reaches the segment 3 first. When a *move* event is present for **train A**, as discussed in case I, **train A** and the **controller** proceed to the **on bridge** and the **A green B red** states, respectively. At this point, suppose that the *location* of **train B** also reaches the segment 3. Whenever a corresponding *move* event is present, similarly, the *arrive* event of **train B** is known to be present since it actually only depends on the *move* event, and thus the *arriveB* event is present in the **controller**. Now, the **controller** still needs to know whether the *leaveA* event is present or not, i.e. whether the *location* of **train A** reaches the segment 9 or not. If **train A** does not arrive at the segment 9 yet, the **controller** remains emitting the *signalA* and the *signalB* events with the values 0 and 1, respectively, and thus **train B** remains in the **off bridge** state. Otherwise, if **train A** arrives at the segment 9, the **controller** emits the *signalA* and the *signalB* events with the values 1 and 0, respectively, and takes the transition to the **A red B green** state. This allows **train B** to enter the **on bridge** state. In this state, whenever a corresponding *move* event is present, the *location* of **train B** is increased by one segment until it reaches the segment 9. Then, **train B** emits a *leave* event, moves its *location* by one segment, and proceeds to the **off bridge** state. This causes the **controller** to emit the *signalA* and the *signalB* events both with the value 0 and to take the transition to the **A green B green** state.

- Case III (Both trains arrive at the same time): By observing the two outgoing transitions of the **A green B green** state in the **controller**, we can see that **train A** is given higher priority to access to the bridge. Therefore, even when both **train A** and **train B** arrives at the segment 3 at the same time, the **controller** proceeds to the **A green B red** state, and only **train A** can proceed to the **on bridge** state. The rest of the behavior is basically the same as discussed in case II.

4.2 Image Processing

In addition to performing intensive numeric computation, image processing systems frequently have sophisticated control functionality for sequencing computation tasks, switching among operation modes, coordinating interaction, and managing configuration, etc.

An interesting example is the MPEG [19] standard for compression and decompression. In this standard, there are various situations that are more suitable to be realized by a control-oriented model, such as the FSM, than by a computation-oriented model. Two of those scenarios are described as follows.

- The MPEG *bitstream* is a serial stream of bits, and has a well-defined syntax. It consists of intermixed different layers, each of them with its own headers and data. To construct or parse the MPEG bitstream involves a significant amount of decision making.
- The MPEG standard has three different frame formats: I, P, and B frames. Each of the frame formats corresponds to a specific set of encoding and decoding schemes. The MPEG encoder or decoder needs to switch among three different operation modes depending on which one of the three frame formats is currently processed.

In the following section, we will show how to apply our *charts using the example of a simple coding method: run-length coding.

4.2.1 Example: Run-Length Coding

4.2.1.1 Problem Description

Run-length coding [52] is one of the simplest variable rate coding methods [38]. This coding can lead to significant data compression, especially when the source tends to contain long clusters of repeated symbols. The key idea is to code the source into a sequence of symbols, each of them followed by the number of its repetitions, the *run length*. For example, a binary source, such as the black and white pixels in digital facsimile, may contain long runs of zeros and, occasionally, ones, such as 000001100000000. As a result, its run-length code is 051208.

4.2.1.2 *charts Realization

Figure 4.8 shows our realization of run-length coding using *charts. Since the length of the resulting code using this coding method is changeable, we need to use DDF (instead of SDF) as the topmost level depicted in figure 4.8(a). This DDF system models the environment for simulating the run-length coding. It includes a **source**, an **encoder**, a **decoder**, and a **display**.

The **encoder** is modeled by an FSM, as shown in figure 4.8(b), with two states: **start** and **continue**. It begins in the **start** state. When an *input* is read from the source, the encoder saves the *input* as a local *symbol*, initializes a local *count* to 1 for storing the run length of the symbol, and takes the transition to the **continue** state. In the **continue** state, the encoder reads another *input* from the source, and distinguishes three cases.

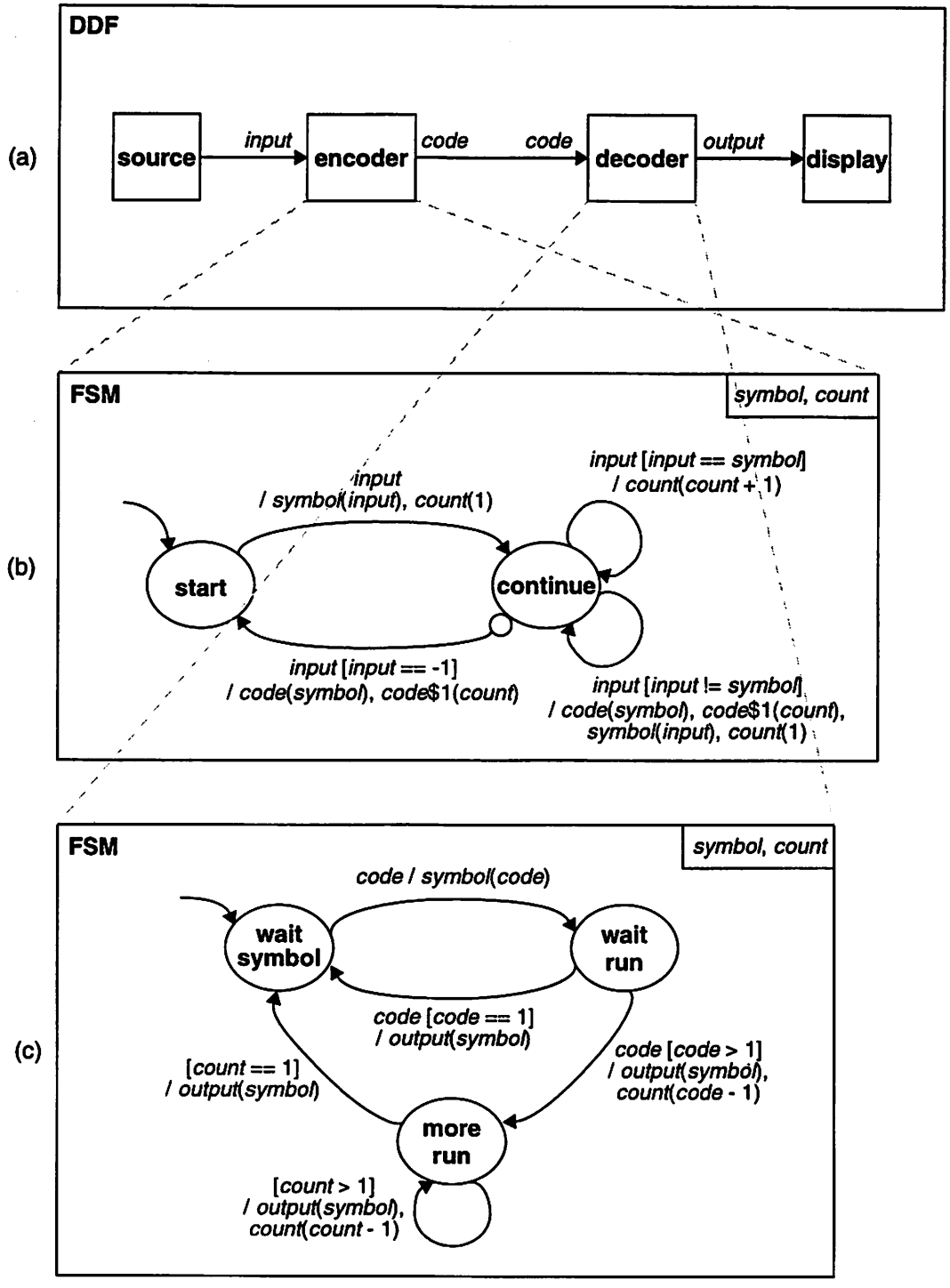


Figure 4.8 Our realization of the run-length coding using *charts.

- If the *input* is a mark for the end of source (assumed to be -1 here), the encoder writes to the *code* with the *symbol* and its *count*, respectively. Then, the encoder takes the transition to the **start** state, and is ready to encode another source.
- If the *input* is not an end-of-source mark but is the same as the current *symbol*, the encoder simply increases the *count* of the run length by 1.
- Otherwise, the encoder writes to the *code* with the *symbol* and its *count*, respectively, updates the *symbol* with the new *input*, and resets the *count* of the run length to 1.

To model the **decoder**, we have an FSM (see figure 4.8(c)) with three states: **wait symbol**, **wait run**, and **more run**. The decoder starts in the **wait symbol** state, and when a *code* is read, saves the *code* as a local *symbol* and takes the transition to the **wait run** state. Then, the decoder reads another *code*, which is the run length of the symbol. If the *code* is equal to 1, the decoder writes to the *output* with the *symbol* only once and takes the transition to the **wait symbol** state. Otherwise, if the *code* is greater than 1, the decoder needs to output the *symbol* more than once. The decoder achieves this by writing the *symbol* to the *output* the first time, storing the remaining run length (i.e. the *code* decreased by 1) to a local *count*, and taking the transition to an instantaneous state, the **more run** state. In the **more run** state, the decoder keeps writing to the *output* with the *symbol* and decreasing the *count* of the run length by 1 until the *count* is equal to 1. Then, the decoder writes the *symbol* to the *output* the last time and takes the transition to the **wait symbol** state.

4.3 Communication Protocols

In communication networks [85], such as computer networks, communication between two or more parties takes place frequently by the exchange of information in a wide range of particular formats. Most of the time, this communication activity needs to strictly follow some sort of order to ensure that the transferred information is complete

and correct. Therefore, a *communication protocol* [50][79], or a set of precisely-defined rules, is required to regulate the exchange of information. If the communication protocol is not followed correctly, the communication activity will not be successful.

Conceptually, a communication protocol can be described as two or more distributed components coordinated by a concurrency model [25]. These components elaborate the logical control of the protocol and the corresponding guarded actions. Thus, they are best characterized as a set of conditional sequences and input/output actions [9]. The concurrency model governs the concurrency and synchrony among those distributed components to accomplish the communication activity.

In terms of *charts, the FSM is the most appropriate model for specification of the protocol components. As for the choice of the concurrency model, although there are various potential candidates, two of them are particularly of interest. The first one is the DE model. This model is known for its strength in modeling the communication infrastructure of distributed systems. In particular, the components within a DE system evolve at the time instants of consecutive events, and their reaction time can be assumed less than the duration between events. This notion serves well to model the interaction of the protocol components, which usually respond to occurring events with a negligible latency as compared to the duration between events [25].

In fact, we often do not care about the exact timing of the communication activity to occur in the simulation of a protocol. What we need for the concurrency model is its ability to maintain the communication activity among the protocol components and to detect the interruption if it occurs. This leads to another concurrency model of interest: the CSP model. The rendezvous mechanism of the CSP model provides sufficient communication infrastructure for the protocol components without the hassle of knowing the exacting timing. If any components try to exchange information with the others but fail to reach rendezvous, they will stall. Therefore, this allows us to detect the situation that the

communication activity fails to continue if all components stall, i.e. if the simulation of the model is deadlocked.

In the following sections, we will apply *charts to two examples of the communication protocols for demonstration. We will first use the DE model and then the CSP model as the concurrency models in these two examples.

4.3.1 Example: Two-Phase Commit Protocol

4.3.1.1 Problem Description

The two-phase commit protocol [13] is a widely used protocol in distributed database systems to maintain the data consistency among them. A coordinator manages a group of participating databases, and can request all participants to commit a particular transaction. The goal of the two-phase commit protocol is to ensure the atomicity of the transaction that accesses multiple participating databases. "Atomicity" means that either all participants commit the transaction or none of them does.

This protocol contains two phases and in normal situations proceeds as follows.

- The voting phase:
  1. The coordinator sends a message to all participants to request them to vote for yes if they are ready to commit a transaction or no if they are not.
  2. The participant receives the vote request and responds by sending to the coordinator a message containing that participant's vote: yes or no.
- The decision phase:
  1. The coordinator collects the vote messages from all participants. If all of the votes are yes, then the coordinator decides to commit and sends a message to inform all participants. Otherwise, the coordinator decides to abort and sends a message to inform all participants.

2. The participant waits for a message containing the coordinator's decision: to commit or to abort. After the message is received, the participant either commits or aborts the transaction depending on the decision.

In the protocol, there are three places that the coordinator or the participant needs to wait for a message before it proceeds.

- In the voting phase (2), a participant waits for a vote request from the coordinator.
- In the decision phase (1), the coordinator waits for a vote from a participant.
- In the decision phase (2), a participant waits for a decision from the coordinator.

However, the expected message may never arrive due to failures. This may cause the protocol to stall indefinitely. To avoid this, a timeout mechanism can be used. When the waiting is more than a certain amount of time, the coordinator or the participant is interrupted by a timeout, and then it takes some special action to allow the protocol to continue. Following are possible timeout actions, one for each of the previous three scenarios.

- In the voting phase (2), since the waiting prevents the participant from voting, it eventually prevents the coordinator from receiving that vote in the decision phase (1). Therefore, we can defer the timeout handling until then to save the effort.
- In the decision phase (1), when the coordinator waits for a vote from a participant, it has not reached any decision, and no participant can commit yet. Hence, if the timeout occurs, the coordinator can simply decide to abort and send a message to inform all participants to follow this decision.
- In the decision phase (2), when a participant times out waiting for the coordinator's decision, if the participant voted no before, it can abort unilaterally since the coordinator's decision is presumably to abort. However, if the participant voted yes before, it cannot unilaterally commit or abort and must find out the coordinator's decision before

it can proceed. One solution is that the participant replies with an additional message to the coordinator after it receives the coordinator's decision. This defers the timeout handling to the coordinator: If the coordinator times out for collecting the replies, it retransmits the decision to the participants.

4.3.1.2 *charts Realization

The topmost level of our *charts realization for the two-phase commit protocol is shown in figure 4.9. This DE system contains a **coordinator**, two participants, a **client** that instructs the coordinator to start a transaction at a randomly chosen time, a **clock** to gener-

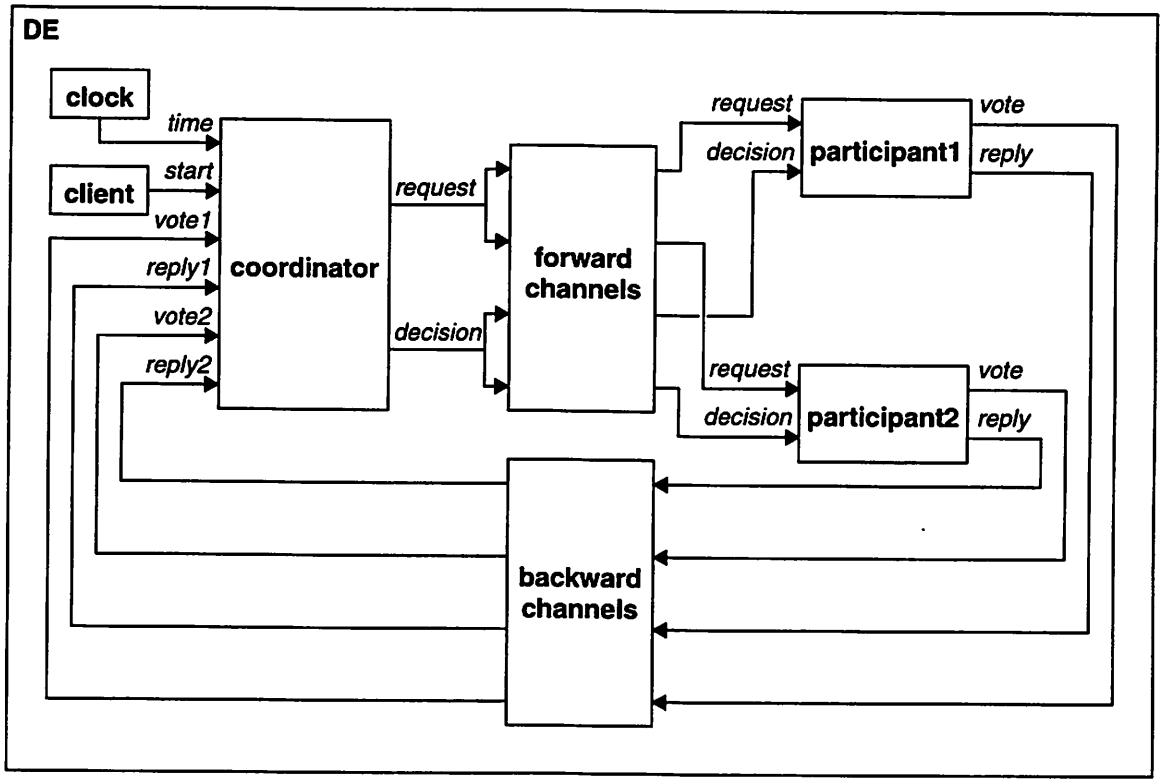


Figure 4.9 The topmost level of our *charts realization for the two-phase commit protocol.

ate time ticks, and the **forward channels** and the **backward channels** to simulate the lossy channels that randomly drop or delay the transferred messages.

Both of the participants, **participant1** and **participant2**, are modeled by the two state FSM depicted in figure 4.10(a). The states are **wait request** and **wait decision**. Each participant starts in its **wait request** state. When a participant receives a *request* event, it sends a *vote* event with the value of a local *ready* event and takes the transition to the **wait decision** state. The *ready* event is generated by an SDF graph (see figure 4.10(b)) that refines the **wait request** state, and its value is randomly either 1 or 0 to represent that the participant is ready to commit or not, respectively. When in the **wait decision** state, if the participant receives a *decision* event, it responds by sending a *reply* event and making transition to the **wait request** state.

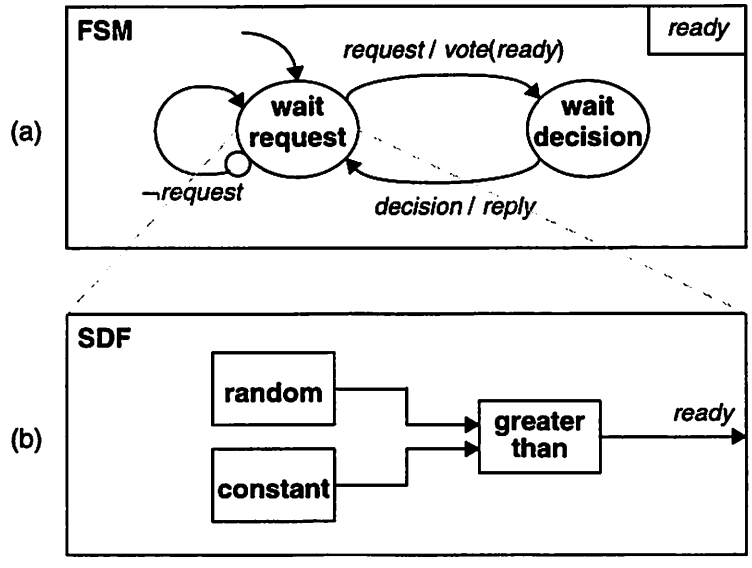


Figure 4.10 The participant1 and the participant2 in figure 4.9 are both modeled by this hierarchical FSM.

Figure 4.11(a) shows an FSM that models the **coordinator** in figure 4.9. This FSM has three states: **idle, wait votes**, and **wait replies**. The coordinator begins in the **idle** state, and when a *start* event is received, sends a *request* event and takes the transition to the **wait votes** state. Then, the coordinator stays in the **wait votes** state until a local *result* event occurs. This *result* event contains a value either 1 or 0 representing that the result of the votes is to either commit or abort, respectively. When the *result* event occurs, the coordinator sends the value of *result* as a *decision* event and takes the transition to the **wait replies** state. When in the **wait replies** state, the coordinator stays until a local *done* event occurs. The value of the *done* event is either 1 or 0 meaning that the collection of replies is complete or not, respectively. If the collection is not complete, the coordinator sends the *decision* again. Otherwise, the coordinator takes the transition to the **idle** state.

Inside the **wait votes** state of figure 4.11(a), we have a three state FSM illustrated in figure 4.11(b). The main functionality of this FSM is to find out the result of the votes and then to inform its master FSM by a *result* event. In general, there are two situations in which the FSM can send the *result* event. If the *vote1* and the *vote2* events both have arrived in time, the result of the votes is the logical "and" of these two votes. Otherwise, if one or both of the votes do not arrive before a local *timeout* event occurs, the result is assigned a value 0 that is considered as to abort. The *timeout* event is asserted by a slave SDF graph (see figure 4.11(d)) shared by the three states of this FSM.

Similarly, inside the **wait replies** state of figure 4.11(a), we have another three state FSM, as shown in figure 4.11(c), to find out whether the collection of the replies is complete or not and then to inform its master FSM by a *done* event. All three states of this FSM also share the same slave SDF graph shown in figure 4.11(d).

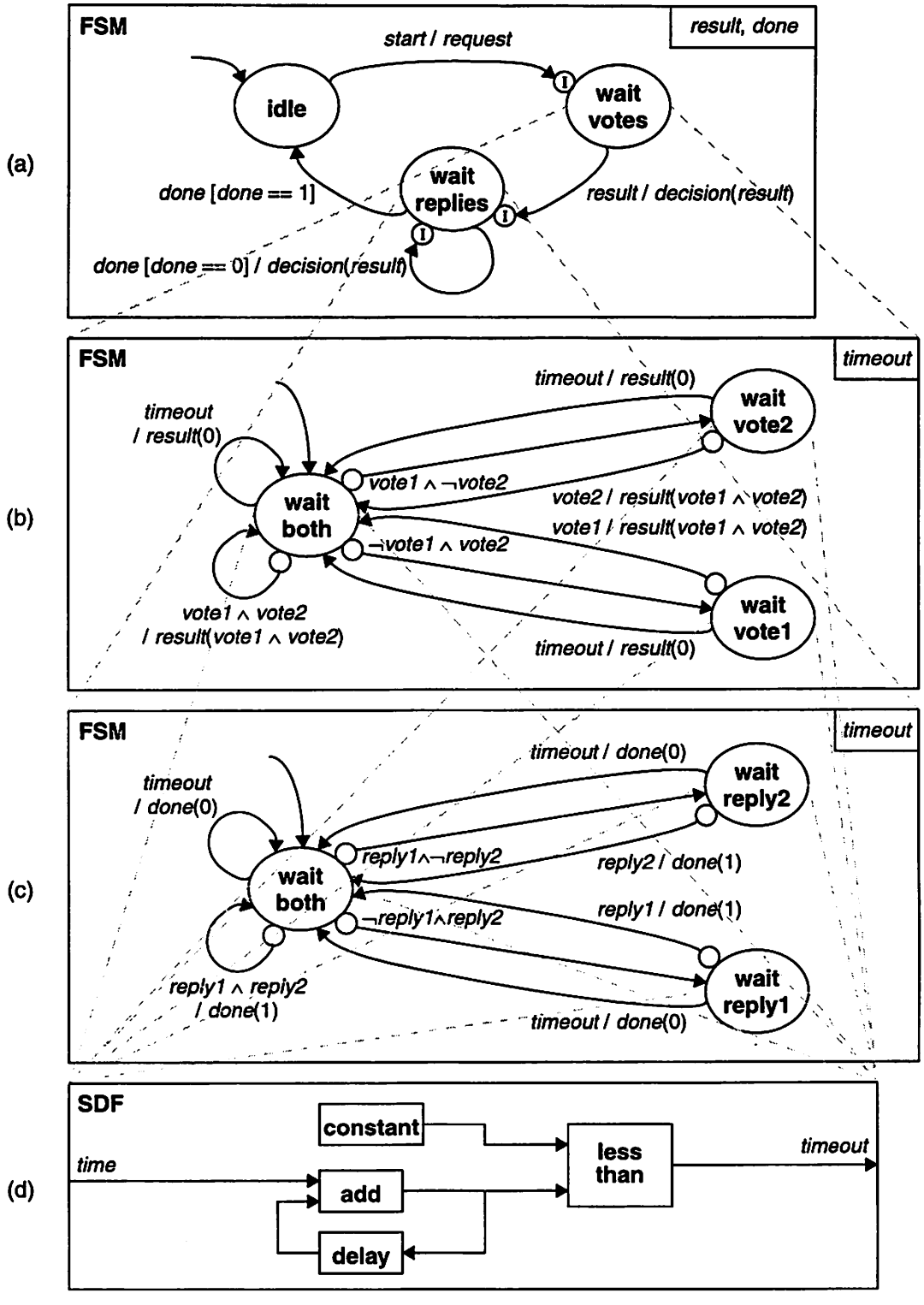


Figure 4.11 This hierarchical FSM models the coordinator in figure 4.9.

4.3.2 Example: Alternating Bit Protocol

4.3.2.1 Problem Description

Because physical communication is often not completely reliable over the networks, some of the messages transmitted between a sender and a receiver may not actually arrive. The alternating bit protocol [66] is a simple retransmission protocol, which is to ensure that when a message is lost, it is retransmitted by the sender, and when duplicate messages are delivered due to retransmission, they are identified and ignored by the receiver.

The alternating bit protocol works as follows.

- The sender sends a data packet that contains a protocol bit.
- When the receiver receives the data packet, it replies with an acknowledgement containing the same protocol bit as that of the received packet. In addition, the receiver needs to identify whether this data packet is a new or a duplicate copy, and then to decide whether to deliver the data for processing or not accordingly. If it is the first time that a data packet is received, the receiver identifies this packet as a new copy and delivers its data for processing. Afterwards, whenever a data packet is received, the receiver compares the protocol bits of this newly received and the last received packets. If the protocol bits are different, then this newly received packet is a new copy, and its data are delivered for processing. Otherwise, the packet is a duplicate copy and is ignored.
- The sender waits for the acknowledgement from the receiver. If the acknowledgement does not reach the sender before time outs, or the acknowledgement received does not have the same protocol bit as that of the data packet transmitted, the sender retransmit the data packet (with the same protocol bit) repeatedly until a desired acknowledge-

ment is received. Then, the sender flips the protocol bit and is ready to transmit the next data packet.

4.3.2.2 *charts Realization

Our *charts realization of the alternating bit protocol is shown in figure 4.12. We use CSP as the topmost level (a), modeling the environment for simulating the protocol. This CSP system consists of a **sender**, a **receiver**, a **client** that requests the sender to transmit its data, a **processor** that processes the data delivered from the receiver, a **timer** to watch for time outs, and a **forward channel** and a **backward channel** to simulate the lossy channels that randomly drop the transferred messages.

In figure 4.12(b), we have a two state FSM to model the **sender**. The states are **ready** and **wait ack**. The sender starts in the **ready** state with a local protocol *bit* set to 0 initially. When the *data* arrives, the sender sends a *packet* with the *data* left shifted[1] one bit to accommodate the protocol *bit*, stores the *data* as a local *dataBuffer* event, and takes the transition to the **wait ack** state. In this state, the sender waits until either an acknowledgement *ack* arrives or time outs. Whenever either an acknowledgement *ack* arrives with different protocol *bit* or time outs, the sender retransmits a *packet* that contains the data (retrieved from the *dataBuffer*) and the current protocol *bit*. Only when an acknowledgement *ack* arrives with the same protocol *bit*, does the sender change the protocol *bit* from 0 to 1 (or vice versa) and take the transition to the **ready** state.

The three state FSM in figure 4.12(c) is to model the **receiver**. The states are **initial**, **wait packet**, and **compare bits**. The receiver starts in the **initial** state. When a *packet* arrives the first time, the receiver replies with an acknowledgement *ack* that contains the protocol bit restored from the *packet*, delivers the *data* restored from the *packet* for pro-

1. Without loss of generality, the value of the *data* is assumed to be a positive integer and have enough bits to prevent overflow.

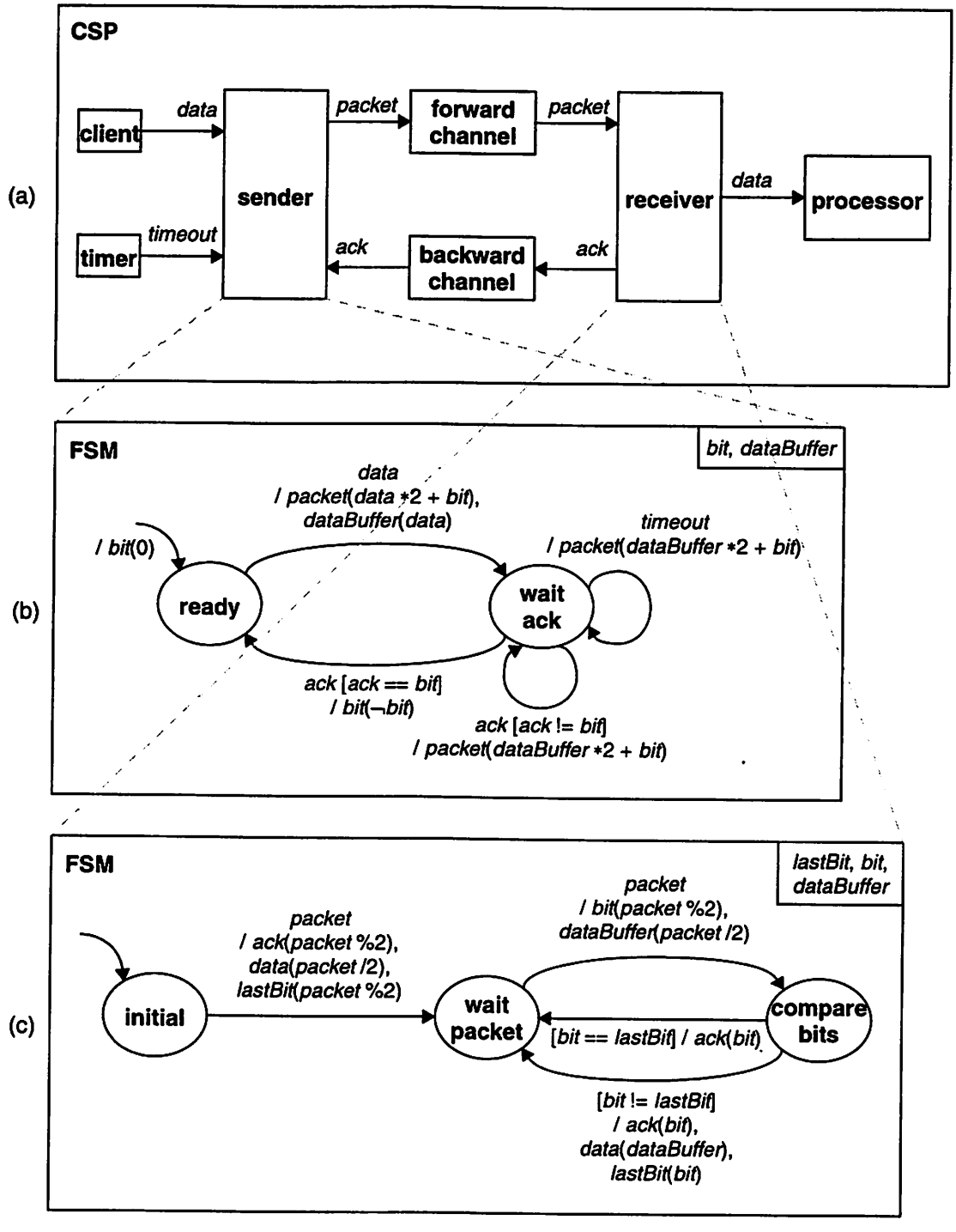


Figure 4.12 Our *charts realization of the alternating bit protocol: (a) the topmost level, (b) the sender, and (c) the receiver.

cessing, remembers the protocol bit as a local *lastBit* event, and takes the transition to the **wait packet** state. Afterwards, whenever a *packet* arrives, the receiver restores the protocol bit and the data from the *packet* to the local *bit* and *dataBuffer* events, respectively, and then takes the transition to the **compare bits** state. At this point, the receiver compares the current protocol *bit* with the last one stored in the *lastBit* event. If both of the protocol bits are the same, the receiver simply replies with an acknowledgement *ack* containing the current protocol *bit*. Otherwise, in addition to replying with an acknowledgement, the receiver delivers the data retrieved from the *dataBuffer* for processing, and updates the *lastBit* event with the current *bit*. In both cases, the FSM takes the transition to the **wait packet** state. Since both explicit outgoing transitions of the **compare bits** state do not have triggers in the guards and are completely specified, this state serves as an instantaneous state.

4.4 Linear Hybrid Systems

A *hybrid system* [1][2] consists of a discrete program interacting with an analog environment. It is generally modeled as a finite automaton equipped with *dynamic variables* that evolve continuously with time according to dynamic laws (often given as differential equations). In particular, when the given differential equations are restricted to first-order, this categorizes a special class of hybrid systems: the *linear hybrid system*. For the linear hybrid system, because the differential equations for the dynamic laws are all first-order, the dynamic variables will follow piecewise-linear trajectories, and those points at the first-order discontinuity in the trajectories correspond to the state changes of the finite automaton.

As a matter of fact, since the trajectories of the dynamic variables are piecewise-linear in the linear hybrid system, the differential equations for the dynamic laws can be substi-

tuted by the discretized equations (see figure 4.13) without changing the trajectories of the dynamic variables. Therefore, a continuous-time (CT) [65] model is no longer necessary to simulate the dynamic laws. Instead, a discrete-event (DE) model can be used and may execute even more efficiently in a discrete-time manner. Therefore, in terms of *charts, the linear hybrid system can be simulated by the combination of the FSM and the DE models.

4.4.1 Example: Water-Level Monitors

4.4.1.1 Problem Description

A commonly used example for the linear hybrid system is the "water-level monitor" problem: The water level in a tank is controlled by a monitor that detects the water level and signals a pump to switch on and off. When the pump is on, the water level rises by 1 unit per second; when the pump is off, the water level falls by 2 unit per second. Suppose that initially the water level is 1 unit and the pump is on. The goal is to keep the water level between 1 and 12 units with the help of the monitor. One key point to complicate this problem is that there is a delay of 2 seconds between the time that the monitor signals the pump to switch status and the time that the switch becomes effective. Thus, the monitor has to signal the pump to switch off when the water level reaches 10 units and to switch on when the water level falls to 5 units.

$$\frac{dx_1}{dt} = v_1 \qquad\qquad x_1(t+dt) - x_1(t) = v_1 dt$$

$$\cdots \qquad\qquad \cdots$$

$$\frac{dx_N}{dt} = v_N \qquad\qquad x_N(t+dt) - x_N(t) = v_N dt$$

(a) Differential equations (b) Discretized equations

Figure 4.13 First-order differential equations and their trajectory-equivalent discretized equations.

4.4.1.2 *charts Realization

Our realization of the water-level monitor using *charts is depicted in figure 4.14. At the bottommost level (c), the DE subsystem is to model the evolution of the dynamic variables, i.e. the water level and the signal delay, according to the dynamic laws in terms of discretized equations. It serves as a slave shared by the four states of the FSM at level (b). There are two main functions performed in this DE subsystem. First, current water level (denoted by *y*) and current signal delay (denoted by *x*) are computed by adding their previous values to the multiplication of their velocities (denoted by *vy* and *vx*, respectively) with current time increment (denoted by *dt*). The current time increment is actually generated at previous step from the next time increment (denoted by *nt*). Moreover, if the reset event for the water level or the signal delay (denoted by *yset* or *xset*, respectively) is present, the corresponding **reset** block will reset the value of the water level or the signal delay, respectively. Second, to prepare for the next step of execution, future events of the next time increment, the water level, and the signal delay are generated by the corresponding **var delay** blocks, which produce future events with values from the *var* inputs and with delays from the *delay* inputs.

At the next level (b) inside the **monitor** block of level (a) is described by an FSM that has four states: in states **on1** and **on2**, the pump is on; in states **off1** and **off2**, the pump is off. The initial transition to state **on1** set the initial conditions of the system: the velocity of the water level to be 1 unit per second, the velocity of the signal delay to be 1 second per second, and the water level to be 1 unit. For display purpose, at every transition except for the initial transition, a *level* event is emitted with the current water level *y*. The monitor starts in state **on1**. When the water level reaches 10 units, the monitor emits a *xset* event to reset the signal delay to be 0 second and a *switch* event to signal the pump to switch off, and then transitions to the **on2** state. In **on2** state, the monitor waits until the signal delay reaches 2 seconds. Then, it changes the velocity of the water level to be -2 units per second

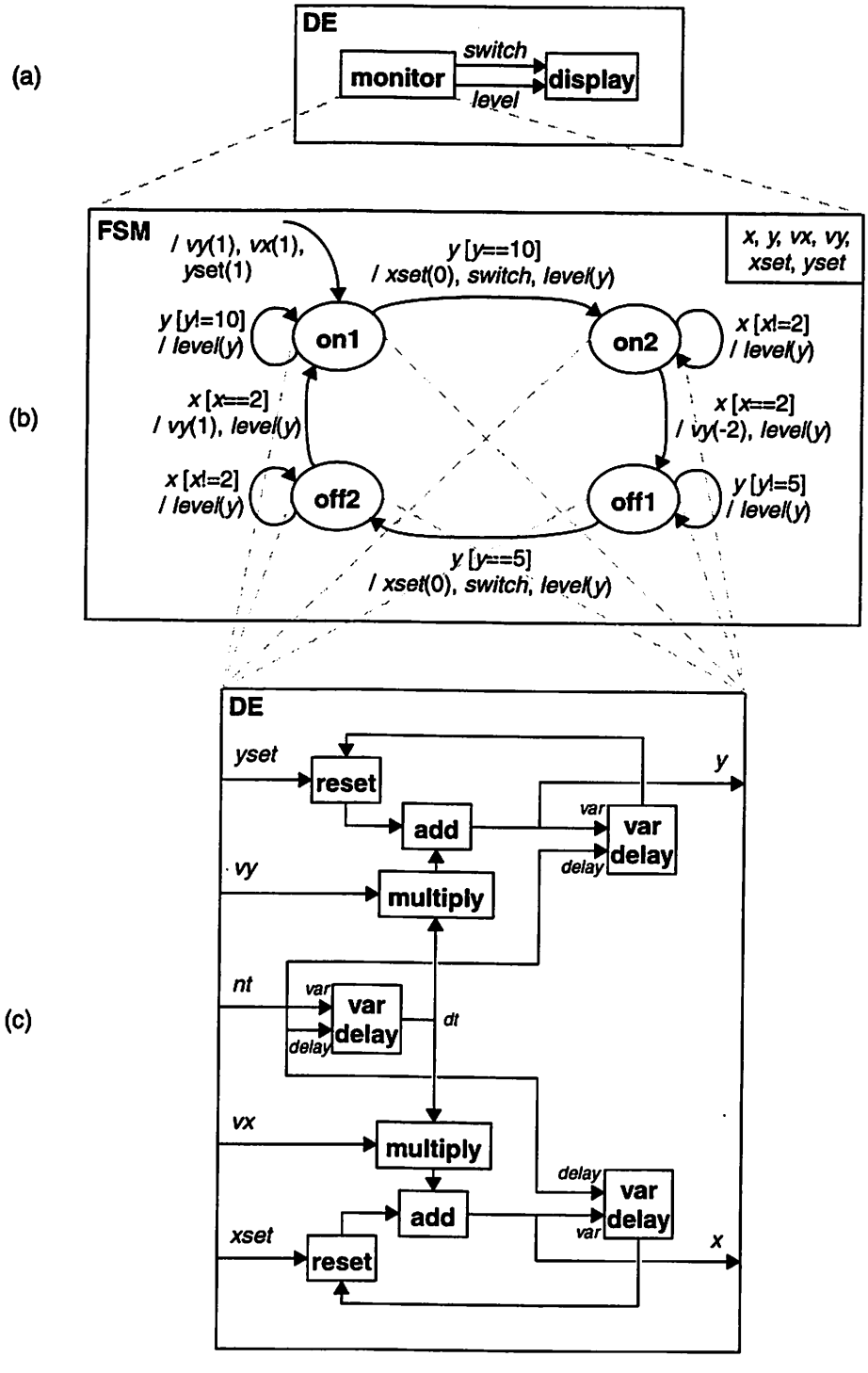


Figure 4.14 The *charts realization of the water-level monitor.

and transitions to the **off1** state. The rest of the behavior at this level should now be evident from the figure.

The DE system at the topmost level (a) is to model the environment of the water-level monitor. It includes a **monitor**, and a **display** to show the statuses of the switch signal and the water level from the monitor.

4.4.1.3 Simulation with Fixed Time Increments

One subtlety for the *charts realization in figure 4.14 is that, at each step of execution, the DE subsystem at level (c) needs to be fed with the next time increment (denoted by *nt*) in order to prepare for the next step of execution. One simple approach is to let the next time increment be a fixed number. For example, when the next time increment is fixed at 0.5 seconds, the simulation result of the *charts realization is shown in figure 4.15. As expected, the water level is kept between 1 and 12 units, and the switch signal is emitted whenever the water level is 10 and 5 units.

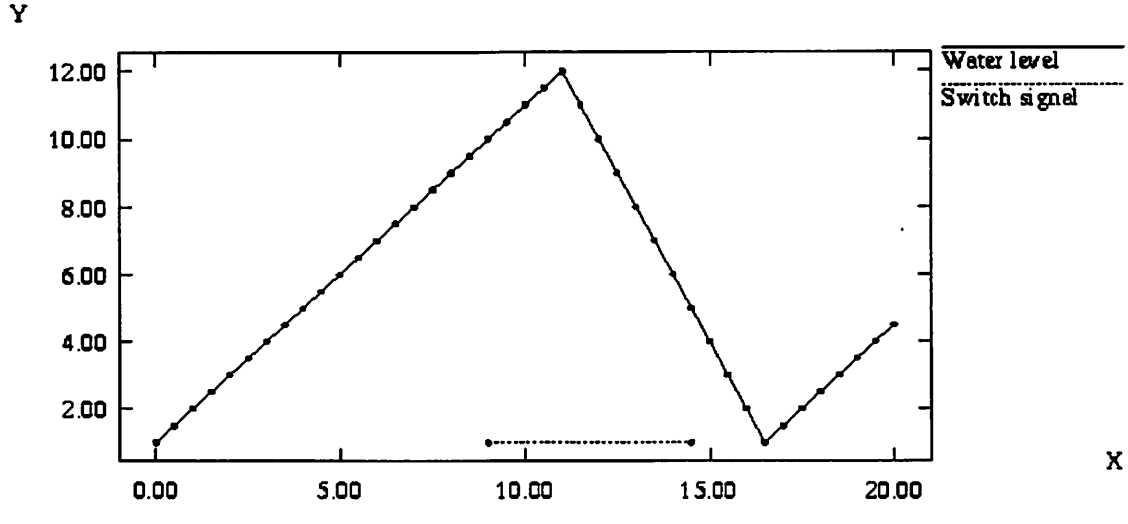


Figure 4.15 Simulation of the water-level monitor with fixed time increments (0.5 seconds).

However, in this approach, the time increments between the consecutive steps of simulation are fixed. Thus, it is possible that the *threshold values*, which are those used for comparison in the guards of the FSM, of the dynamic variables are not detected in simulation even though the resulting trajectories actually pass through them. This may result in incorrect state changes of the FSM and thus incorrect simulation result. For example, when the next time increment is fixed at 0.4 seconds, the threshold value of the water level at 10 units is not detected for comparison in the guard of the FSM. Hence, the FSM stays in the state **on1** forever, and the simulation result shown in figure 4.16 exhibits incorrect behavior.

4.4.1.4 Simulation with Varied Time Increments

In fact, from figure 4.15 we can observe that the points of first-order discontinuity in the trajectories occur only when the dynamic variables are at the threshold values. Furthermore, we can choose only these points for simulation without affecting the correctness of

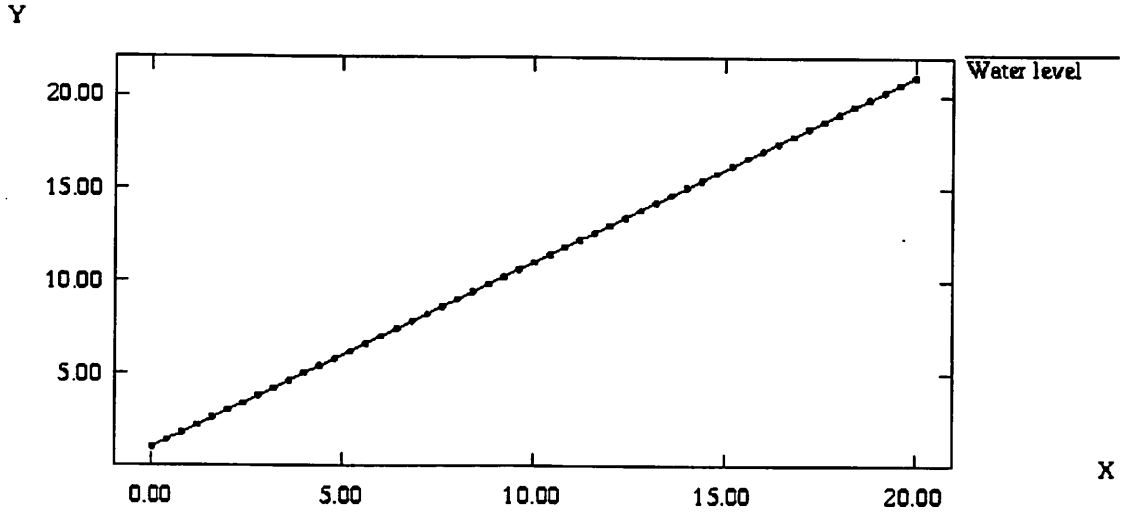


Figure 4.16 Simulation of the water-level monitor with fixed time increments (0.4 seconds).

the result. However, if only selected points are included for simulation, the time increments between the consecutive steps of simulation have to be varied. In other words, in each step of simulation, we need to predict the next time increment based on the threshold values. This approach with varied time increments is more efficient than that with fixed time increments since fewer points are computed to build up the whole trajectories. Moreover, this guarantees that the threshold values of the dynamic variables will always be detected for comparison in the guards of the FSM since the next time increment is computed based on those values.

To support the simulation with varied time increments, we need to distinguish one step of the simulation of the realization in figure 4.14 into following four phases:

1. The DE subsystem at level (c) computes the current values of the dynamic variables, i.e. the water level and the signal delay.
2. The FSM at level (b) takes a transition, which depends on the current values of the dynamic variables.
3. The FSM at level (b) predicts the next time increment (denoted by *nt*), which depends on the threshold values (denoted by T_{nk}, $n = 1, \ldots, N$, $k = 1, \ldots, K_n$) and the velocities (denoted by v_n, $n = 1, \ldots, N$) of the dynamic variables (denoted by x_n, $n = 1, \ldots, N$) at the next state.

$$nt = \min\left\{ \max\left\{ \frac{T_{11} - x_1}{v_1}, 0 \right\}, \ldots, \max\left\{ \frac{T_{1K_1} - x_1}{v_1}, 0 \right\}, \ldots, \max\left\{ \frac{T_{N1} - x_N}{v_N}, 0 \right\}, \ldots, \max\left\{ \frac{T_{NK_N} - x_N}{v_N}, 0 \right\} \right\}$$

4. The DE subsystem at level (c) generates future events at the next time increment to prepare for the next step.

The order of these four phases may not be changed because there exist dependencies between them. In other words, the computation of the dynamic variables and the generation of the future events in the DE subsystem have to be performed in two different phases.

Since one reaction of the FSM invokes the slave DE subsystem at most once, we leave the second phase of the DE subsystem to be performed in the next consecutive reaction of the FSM. I.e. the completion of one step of the simulation actually involves two reactions of the FSM. Figure 4.17 shows the simulation result of the *charts realization using the approach with varied time increments.

4.5 Comparison

In this section, we will demonstrate the effectiveness of the *charts approach with a simple control-oriented example, the reflex game [12]. Realizations of this example will be illustrated and compared, using our *charts and other languages, in particular, Esterel,

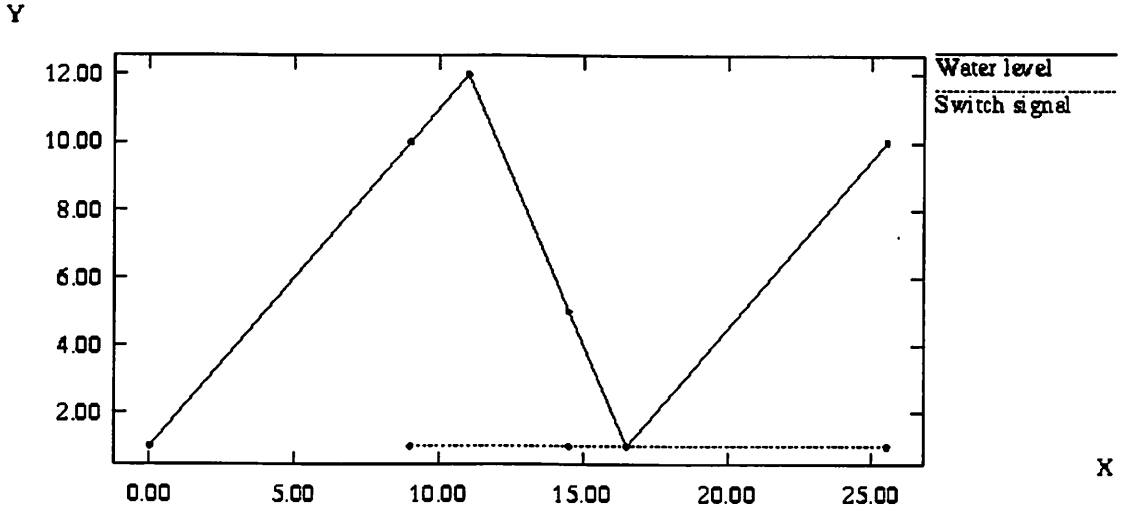


Figure 4.17 Simulation of the water-level monitor with varied time increments.

VHDL and C. The reflex game has a normal behavior and a reasonable set of exception situations. It is simple enough to be summarized on a page, but rich enough to encompass the usage of multiple models.

4.5.1 Problem Description

Our version of the reflex game is a two-player game (to introduce more concurrency). Each player has two buttons to press during the game: *coin* and *go* buttons for player 1; *ready* and *stop* buttons for player 2.

Normal play proceeds as follows:

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

The game measures the reflex time of player 2 by reporting the time between the *go* and *stop* events.

There are some situations where the game ends abnormally and a "tilt" light flashes. These are:

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

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

4.5.2 *charts Realization

Our *charts realization of the reflex game is shown in figure 4.18. To simulate the real-time behavior of the game, a DE system is a good choice for the topmost level (a), modeling the environment of the game (including the players). This DE system contains a **clock** to generate a sequence of time ticks, the two players (**player1** and **player2**) that assert the button pressing events, a **display** to create the lights and to report the reflex time of player 2, and a **reflex** block that models the behavior of the game.

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 rules for the two players. These are interconnected with a directed loop, and thus form an instantaneous dialog between the two players.

At level (d), the two rules are refined into concurrent FSMs. The **rule2** starts in the **wait ready** state, and when a *ready* event is present, emits a *start* event and takes the transition to the **wait go** state. This causes the **rule1** to emit a *yellowLt* event and to take the transition from the **idle** state to the **wait** state. The rest of the behavior at this level should now be evident from the figure.

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

4.5.3 Esterel Realization

Esterel [18] is a programming language dedicated to describe control-oriented reactive systems. Figure 4.19 shows an Esterel realization of the two-player reflex game. The description in Esterel is concise, taking slightly less space than the one in figure 4.18. This application is a good match for the concurrent semantics of Esterel, which is synchronous/

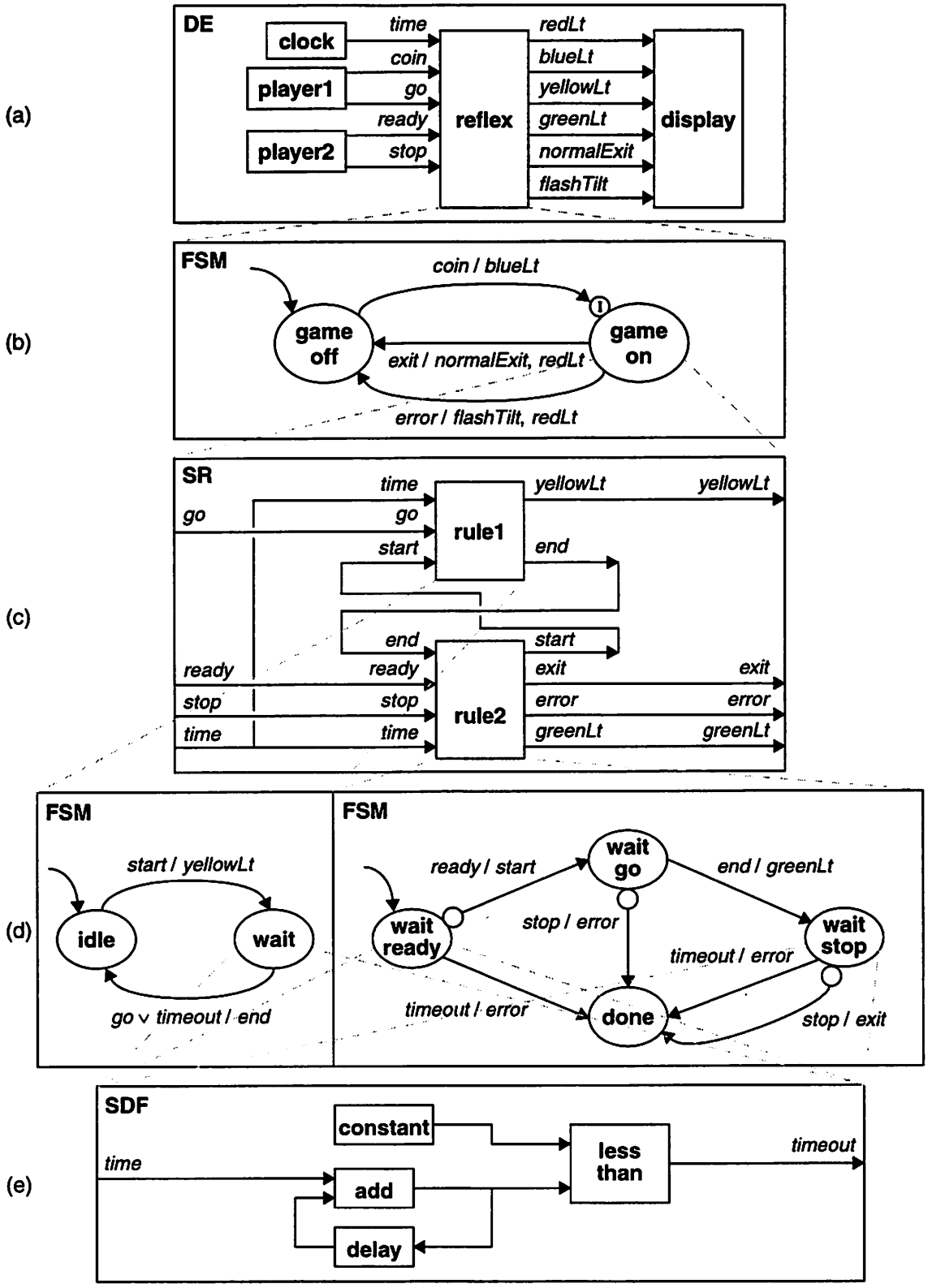


Figure 4.18 Our *charts realization of the two-player reflex game.

```
module REFLEX :

% timeout limit declaration
constant L : integer;
% inputs and outputs
input TIME, COIN, GO, READY, STOP;
relation COIN # GO, READY # STOP;
output REDLT, BLUELT, YELLOWLT, GREENLT, NORMALEXIT, FLASHTILT;
% overall initialization
emit REDLT;

loop
   % game off state
   await COIN;
   emit BLUELT;
   % game on state
   signal START, END in
      trap EXIT, ERROR in
         [ % rule 1 begins
            % idle state
            await START;
            emit YELLOWLT;
            % wait state
            await
               case GO do emit END
               case L TIME do emit END
            end await
         ] % rule 1 ends
         ||
         [ % rule 2 begins
            % wait ready state
            await
               case READY do emit START
               case L TIME do exit ERROR
            end await;
            % wait go state
            await
               case STOP do exit ERROR
               case END do emit GREENLT
            end await;
            % wait stop state
            await
               case STOP do exit EXIT
               case L TIME do exit ERROR
            end await
         ] % rule 2 ends
      handle EXIT do
         emit NORMALEXIT;
         emit REDLT
      handle ERROR do
         emit FLASHTILT;
         emit REDLT
      end trap
   end signal
end loop

end module
```

Figure 4.19 Esterel realization of the two-player reflex game.

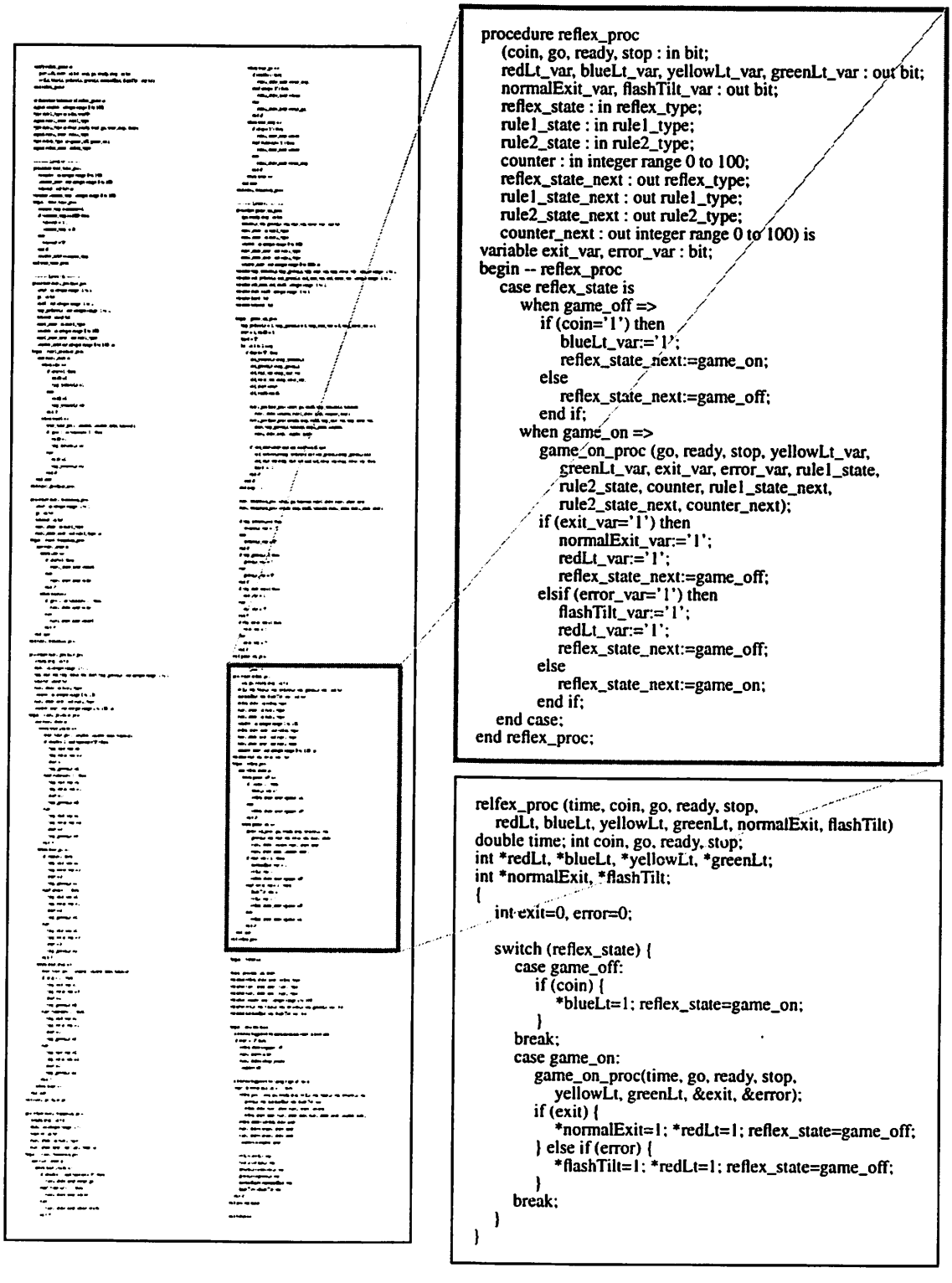
reactive. However, this Esterel module does not include a model of the environment. Esterel programs generally specify modules that are intended to reside within some foreign realization of the environment, such as a C program. Moreover, there is no support for other concurrency modeling, such as discrete events, in Esterel.

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 our *charts, a designer could use dataflow models, which are 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 4.18 will be more easily understood.

4.5.4 VHDL and C Realizations

VHDL and C are two languages that are commonly used as synthesized codes for hardware and software, respectively. The complete VHDL description is shown in figure 4.20 although not in a readable font. VHDL is a relatively verbose language, and this description, which includes almost no comments, occupies more than five pages, and like the Esterel program, does not model the environment. The C description is shorter, occupying less than four pages. In figure 4.20 at the right, we show the VHDL and C descriptions of the level (b) FSM from figure 4.18. This FSM is implemented very directly as switch-case and if-then-else constructs in both cases. Our conclusion is that VHDL and C should be back-end languages, synthesized from higher-level descriptions for the purpose of interfacing to lower-level synthesis tools, and that *charts provides a reasonable higher-level description.



```
procedure reflex_proc
    (coin, go, ready, stop : in bit;
    redLt_var, blueLt_var, yellowLt_var, greenLt_var : out bit;
    normalExit_var, flashTilt_var : out bit;
    reflex_state : in reflex_type;
    rule1_state : in rule1_type;
    rule2_state : in rule2_type;
    counter : in integer range 0 to 100;
    reflex_state_next : out reflex_type;
    rule1_state_next : out rule1_type;
    rule2_state_next : out rule2_type;
    counter_next : out integer range 0 to 100) is
variable exit_var, error_var : bit;
begin -- reflex_proc
    case reflex_state is
        when game_off =>
            if (coin='1') then
                blueLt_var:='1';
                reflex_state_next:=game_on;
            else
                reflex_state_next:=game_off;
            end if;
        when game_on =>
            game_on_proc (go, ready, stop, yellowLt_var,
                greenLt_var, exit_var, error_var, rule1_state,
                rule2_state, counter, rule1_state_next,
                rule2_state_next, counter_next);
            if (exit_var='1') then
                normalExit_var:='1';
                redLt_var:='1';
                reflex_state_next:=game_off;
            elsif (error_var='1') then
                flashTilt_var:='1';
                redLt_var:='1';
                reflex_state_next:=game_off;
            else
                reflex_state_next:=game_on;
            end if;
    end case;
end reflex_proc;
```

```
relfex_proc (time, coin, go, ready, stop,
    redLt, blueLt, yellowLt, greenLt, normalExit, flashTilt)
double time; int coin, go, ready, stop;
int *redLt, *blueLt, *yellowLt, *greenLt;
int *normalExit, *flashTilt;
{
    int exit=0, error=0;

    switch (reflex_state) {
        case game_off:
            if (coin) {
                *blueLt=1; reflex_state=game_on;
            }
        break;
        case game_on:
            game_on_proc(time, go, ready, stop,
                yellowLt, greenLt, &exit, &error);
            if (exit) {
                *normalExit=1; *redLt=1; reflex_state=game_off;
            } else if (error) {
                *flashTilt=1; *redLt=1; reflex_state=game_off;
            }
        break;
    }
}
```

Figure 4.20 VHDL description of the two-player reflex game, with the segment corresponding to level (b) in figure 4.18 shown in a readable font at the upper right. At the lower right is the C version.

5

Conclusion

The goals of this thesis is to have a system design scheme that is capable of describing both control logic and computation tasks, specifying composite behaviors for concurrency and hierarchy, and enabling the selection of different concurrency semantics. To achieve these goals, we present a heterogeneous HCFSM formalism, called *charts, which is aimed at specification and simulation of reactive systems with control logic. Instead of defining a grand-unified concurrency semantics in *charts, we explore the hierarchical combination of FSMs with multiple concurrency models, particularly discrete events (DE), synchronous/reactive (SR), synchronous dataflow (SDF), dynamic dataflow (DDF), and communicating sequential processes (CSP).

As shown in figure 5.1, these models have different strengths and weaknesses. Hence, they are applicable in different situations. The DE model is a natural way to model distributed or parallel systems and their communication infrastructure. The SR model is well-suited to concurrent behaviors of control-intensive systems. Dataflow models are ideal for computation-intensive systems, such as signal processing systems. The CSP model is useful for resource management when modeling at the system level. Finally, the FSM model complements those concurrency models such that sequential control behaviors are intuitively described and easily analyzed.

Our *charts formalism can be utilized as a design framework, in which subsystems with distinct functionalities are separately specified and designed by mixing and matching the best suitable models. The simple and determinate mechanisms we provide are used to combine the subsystems as a whole for validation using simulation. For example, the design of a digital cellular phone often involves specifying distinct functionalities. It includes intensive signal processing and sophisticated control logic in both the speech coder and the radio modem. These would be appropriately constructed using the combination of SDF with FSM. The communication protocols for call processing and multiple-access schemes can be specified by mixing CSP and FSM. The integration of SR and FSM

Model	Strengths	Weaknesses
Finite State Machines	• Good for sequential control • Can be made deterministic • Maps well to hardware and software	• Computation-intensive systems are hard to specify
Discrete Events	• Good for digital hardware • Global time • Can be made deterministic	• Expensive to implement in software • May over-specify systems
Synchronous/ Reactive Model	• Good for control-intensive systems • Tightly synchronized • Deterministic • Maps well to hardware and software	• Computation-intensive system are over-specified
Dataflow	• Good for signal processing • Loosely synchronized • Deterministic • Maps well to hardware and software	• Control-intensive systems are hard to specify
Communicating Sequential Processes	• Models resource sharing well • Partial-order synchronization • Supports naturally nondeterministic interactions	• Some systems are oversynchronized • Difficult to be made deterministic

Figure 5.1 Strengths and weaknesses of different models.

can be utilized to describe user interfaces with concurrent behaviors. The environment of the digital cellular phone can be conveniently modeled under DE. Therefore, *charts provides a good framework for coordinating among such efforts.

In conclusion, the major advantages of *charts are

- Heterogeneous: Diverse models can coexist and interact by hierarchical combination.
- Modular: Distinct portions of a system can be separately modeled, choosing the best appropriate modeling technique.
- Extensible: Additional concurrency models can be included as long as we provide the interaction between different models.

5.1 Open Issues

5.1.1 Software/Hardware Synthesis

Specification of a system focuses on description and interpretation of behaviors without the implementation details. Eventually, system behaviors need to be mapped and implemented by software components (e.g. programmable devices), hardware components (e.g. ASICs), or both. The bridge from specification to synthesis is the code generation that translates system behaviors into formats mappable to software and hardware, such as C code for software synthesis and VHDL code for hardware synthesis.

Synthesis from the FSM model has been a popular technique, demonstrated in [44] for software and [72] for hardware. Moreover, dataflow models have been shown synthesizable in software [8] and hardware [40]. Synthesis from the SR model has also been demonstrated for software [18] and hardware [14]. As for the DE and CSP models, they are used more for modeling, and thus synthesis is not much of an issue.

Our *charts formalism allows the designers to use established synthesis techniques within each model, and provides simple and determinate mechanisms to combine the results. Nevertheless, approaches that do co-synthesis directly from the mixed models are likely to show more value than approaches to synthesize separately from each model.

5.1.2 Formal Semantics

Formal approaches to define semantics have been evolving in two different ways: *operational* and *denotational*. Operational semantics, which dates back to Turing machines [82], describes how a system executes on an abstract machine. It focuses on what the system does (the internal view of the system). Denotational semantics, pioneered by Scott and Strachey [78], describes a system in terms of functions and relations in a semantic domain, such as a complete partial order set. It focuses on what an outside observer sees (the external view of the system).

Various formal semantics for the models we discuss have been given in the literature, such as [47][68] for the FSM model, [60] for the DE model, [35] for the SR model, [57] for the dataflow models, and [49] for the CSP model. Before the interaction among different models can be denoted in formal semantics, a mathematical framework, such as the tagged signal model [64], is required such that common terms of all models are unified and distinct properties of each model are identified. Then, we can leverage the existing formal semantics of the FSM, DE, SR, dataflow, and CSP models within this framework instead of defining new formal semantics.

Bibliography

[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, p. 3-4, February 1995.

[2] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems," *Hybrid Systems*, October 1991.

[3] R. Alur and T. A. Henzinger, *Computer-Aided Verification: An Introduction to Model Building and Model Checking for Concurrent Systems*, Draft, 1998.

[4] G. R. Andrews, *Concurrent Programming: Principles and Practice*, Benjamin/ Cummings Publishing Company, 1991.

[5] J. R. Armstrong and F. G. Gray, *Structured Logic Design with VHDL*, PTR Prentice Hall, 1993.

[6] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press, June 1997.

[7] J. Banks, J. S. Carson II, and B. L. Nelson, *Discrete-Event System Simulation*, Prentice Hall, 1996.

[8] S. S. Battacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[9] F. Belina, D. Hogrefe, and A. Sarma, *SDL with Applications from Protocol Specification*, Prentice Hall International, 1991.

[10] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, vol. 79, no. 9, p. 1270-1282, September 1991.

[11] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the Signal Language," *IEEE Transactions on Automatic Control*, vol. 35, no. 5, p. 535-546, May 1990.

[12] R. Bernhard, G. Berry, F. Boussinot, G. Gonthier, A. Ressouche, J.-P. Rigault, and J.-M. Tanzi, "Programming a Reflex Game in Esterel v3," May 1989.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Company, 1987.

[14] G. Berry, "A Hardware Implementation of Pure ESTEREL," *Sadhana*, vol. 17, no. 1, p. 95-130, March 1992.

[15] G. Berry, "Programming a Digital Wristwatch in Esterel v3.2," Rapport de Recherche no. 8, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, 1991.

[16] G. Berry, "Real Time Programming: Special Purpose or General Purpose Languages," *Information Processing 89*, September 1989.

[17] G. Berry, *The Esterel v5 Language Primer, Version 5.21 Release 2.0*, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, April 1999.

[18] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming*, vol. 19, no. 2, p. 87-152, November 1992.

[19] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, 2nd edition, Kluwer Academic Publishers, 1997.

[20] G. V. Bochmann and J. Gecsei, "A Unified Method for the Specification and Verification of Protocols," *Proceedings of IFIP Congress 77*, p. 229-234, August 1977.

[21] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, p. 677-691, August 1986.

[22] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Memorandum UCB/ERL M93/69, Electronics Research Laboratory, University of California, Berkeley, September 1993.

[23] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, special issue on "Simulation Software Development," vol. 4, p. 155-182, April 1994.

[24] C. G. Cassandras, *Discrete Event Systems: Modeling and Performance Analysis*, Richard D. Irwin, Inc., and Aksen Associates, Inc., 1993.

[25] S.-P. Chang, *System-Level Modeling and Evaluation of Network Protocols*, Memorandum UCB/ERL M98/73, Electronics Research Laboratory, University of California, Berkeley, December 1998.

[26] W.-T. Chang, S. Ha, and E. A. Lee, "Heterogeneous Simulation - Mixing Discrete-Event Models with Dataflow," *Journal of VLSI Signal Processing*, vol. 15, no. 1-2, p. 127-144, January 1997.

[27] K.-T. Cheng and A. S. Krishnakumar, "Automatic Functional Test Generation Using the Extended Finite State Machine Model," *30th Design Automation Conference*, June 1993.

[28] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "A Formal Specification Model for Hardware/Software Codesign," *International Workshop on Hardware-Software Codesign*, October 1993.

[29] R. Cleaveland, S. A. Smolka, R. Alur, J. Baeren, J. A. Bergstra, E. Best, R. de Nicola, R. Gorrieri, M. G. Gouda, J. F. Groote, T. A. Henzinger, C. A. R. Hoare, D. Luginbuhl, A. Meyer, D. Miller, J. Misra, F. Moller, U. Montanari, A. Pnueli, S. Prasad, V. R. Pratt, J. Sifakis, B. Steffen, B. Thomsen, F. Vaandrager, M. Vardi, and P. Wolper, "Strategic Directions in Concurrency Research," *ACM Computing Surveys*, vol. 28, no. 4, p. 607-625, December 1996.

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

[31] J. Davis, R. Galicia, M. Goel, C. Hylands, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong, *Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java*, Memorandum UCB/ERL M99/40, Electronics Research Laboratory, University of California, Berkeley, July 1999.

[32] W. S. Davis, *Tools and Techniques for Structured Systems Analysis and Design*, Addison-Wesley Publishing Company, 1983.

[33] W. Delaney and E. Vaccari, *Dynamic Models and Discrete Event Simulation*, Marcel Dekker, Inc., 1989.

[34] J. B. Dennis, "First Version of a Data Flow Procedure Language," *Programming Symposium*, April 1974.

[35] S. A. Edwards, *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*, Memorandum UCB/ERL M97/31, Electronics Research Laboratory, University of California, Berkeley, May 1997.

[36] D. D. Gajski, F. Vahid, and S. Narayan, "A System-Design Methodology: Executable-Specification Refinement," *European Design and Test Conference*, March 1994.

[37] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, PTR Prentice Hall, 1994.

[38] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*, Kluwer Academic Publishers, 1992.

[39] T. Grotker, R. Schoenen, and H. Meyr, "PCC: A Modeling Technique for Mixed Control/Data Flow Systems," *European Design and Test Conference*, March 1997.

[40] T. Grotker, P. Zepter, and H. Meyr, "ADEN: An Environment for Digital Receiver ASIC Design," *International Conference on Acoustics, Speech, and Signal Processing*, May 1995.

[41] S. Ha, *Compile-Time Scheduling of Dataflow Program Graphs with Dynamic Constructs*, Memorandum UCB/ERL M92/43, Electronics Research Laboratory, University of California, Berkeley, April 1992.

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

[43] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, p. 231-274, June 1987.

[44] D. Harel and E. Gery, "Executable Object Modeling with Statecharts," *Computer*, vol. 30, no. 7, p. 31-42, July 1997.

[45] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "Statemate: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, p. 403-414, April 1990.

[46] D. Harel and A. Pnueli, "On the Development of Reactive Systems," *Logics and Models of Concurrent Systems*, NATO ASI series, vol. F13, pp. 477-498, 1985.

[47] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman, "On the Formal Semantics of Statecharts," *Proceedings of the Symposium on Logic in Computer Science*, p. 54-64, June 1987.

[48] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, p. 666-677, August 1978.

[49] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[50] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall Software Series, 1991.

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

[52] N. S. Jayant and P. Noll, *Digital Coding of Waveforms: Principles and Applications to Speech and Video*, Prentice Hall, 1984.

[53] M. Jourdan and F. Maraninchi, "A Modular State/Transition Approach for Programming Reactive Systems," *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.

[54] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proceedings of IFIP Congress 74*, August 1977.

[55] T. Y.-K. Kam, *Multi-Valued Decision Diagrams*, Memorandum UCB/ERL M90/125, Electronics Research Laboratory, University of California, Berkeley, 1990.

[56] L. Lavagno, A. Sangiovanni-Vincentelli, and E. Sentovich, "Models of Computation for Embedded System Design," September 1998.

[57] E. A. Lee, *A Denotational Semantics for Dataflow with Firing*, Memorandum UCB/ERL M97/3, Electronics Research Laboratory, University of California, Berkeley, January 1997.

[58] E. A. Lee, "Consistency in Dataflow Graphs", *Proceedings of the International Conference on Application Specific Array Processors*, p. 355-369, September 1991.

[59] E. A. Lee, "Embedded Software - An Agenda for Research," Memorandum UCB/ERL M99/63, Electronics Research Laboratory, University of California, Berkeley, December 1999.

[60] E. A. Lee, "Modeling Concurrent Real-Time Processes Using Discrete Events," *Annals of Software Engineering*, vol. 7, p. 25-45, 1999.

[61] E. A. Lee, "Recurrences, Iteration, and Conditionals in Statically Scheduled Block Diagram Languages," *VLSI Signal Processing III*, p. 330-340, November 1988.

[62] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, p. 24-35, January 1987.

[63] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, p. 1235-1245, September 1987.

[64] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, p. 1217-1229, December 1998.

[65] J. Liu, *Continuous Time and Mixed-Signal Simulation in Ptolemy II*, Memorandum UCB/ERL M98/74, Electronics Research Laboratory, University of California, Berkeley, December 1998.

[66] W. C. Lynch, "Reliable Full-Duplex File Transmission over Half-Duplex Telephone Lines," *Communication of the ACM*, vol. 11, no. 6, p. 407-410, June 1968.

[67] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.

[68] F. Maraninchi, "Operational and Compositional Semantics of Synchronous Automaton Compositions," *Third International Conference on Concurrency Theory*, August 1992.

[69] F. Maraninchi, "The Argos Language: Graphical Representation of Automata and Description of Reactive Systems," *IEEE Workshop on Visual Languages*, October 1991.

[70] S. Narayan, F. Vahid, and D. D. Gajski, "System Specification and Synthesis with the SpecCharts Language," *International Conference on Computer-Aided Design*, November 1991.

[71] S. Narayan, F. Vahid, and D. D. Gajski, "System Specification with the SpecCharts language," *IEEE Design & Test of Computers*, vol. 9, no. 4, p. 6-13, December 1992.

[72] S. Narayan, F. Vahid, and D. D. Gajski, "Translating System Specifications to VHDL," *European Conference on Design Automation*, February 1991.

[73] M. Pankert, O. Mauss, S. Ritz, and H. Meyr, "Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis," *IEEE International Conference on Acoustics, Speech and Signal Processing*, April 1994.

[74] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

[75] B. Sarikaya, V. Koukoulidis, and G. V. Bochmann, "Method of Analysing Extended Finite-State Machine Specifications," *Computer Communications*, vol. 13, no. 2, p. 83-92, March 1990.

[76] P. Scholz and D. Nazareth, "Communication Concepts for Statecharts: A Semantic Foundation," *International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software*, May 1997.

[77] P. Scholz, D. Nazareth, and F. Regensburger, "Mini-Statecharts: A Compositional Way to Model Parallel Systems," *International Conference on Parallel and Distributed Computing Systems*, September 1996.

[78] D. S. Scott and C. Strachey, "Toward a Mathematical Semantics for Computer Languages," *Symposium on Computers and Automata*, April 1971.

[79] R. Sharp, *Principles of Protocol Design*, Prentice Hall International, 1994.

[80] N. Smyth, *Communicating Sequential Processes Domain in Ptolemy II*, Memorandum UCB/ERL M98/70, Electronics Research Laboratory, University of California, Berkeley, December 1998.

[81] J. Sodhi, *Computer Systems Techniques: Development, Implementation, and Software Maintenance*, TAB Professional and Reference Books, 1990.

[82] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 42, p. 230-265, December 1936.

[83] K. J. Turner, *Using Formal Description Techniques: An Introduction to Estelle, LOTOS and SDL*, John Wiley & Sons Ltd., 1993.

[84] F. Vahid, S. Narayan, and 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, p. 694-706, June 1995.

[85] J. Walrand and P. Varaiya, *High-Performance Communication Networks*, Morgan Kaufmann Publishers, 1996.

[86] M. von der Beeck, "A Comparison of Statecharts Variants," *Formal Techniques in Real-Time and Fault-Tolerant Systems*, September 1994.