Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems

by

Jie Liu

B.E. (Tsinghua University, Beijing, China) 1993
M.E. (Tsinghua University, Beijing, China) 1996
M.S. (University of California, Berkeley) 1998

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Edward A. Lee, Chair
Professor Shankar S. Sastry
Professor John A. Strain

Fall 2001

The dissertation of Jie Liu is approved:

Chair Date

Date

Date

University of California at Berkeley

Fall 2001

Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems

Copyright Fall 2001

by

Jie Liu

Abstract

Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems

by

Jie Liu

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Edward A. Lee, Chair

This dissertation studies modeling and design frameworks for heterogeneous embedded systems. Heterogeneity, in the sense that components in a system have diverse interaction styles, complicates embedded system design and challenges understandability, composability, and scalability of models. Hierarchical heterogeneous modeling approaches tame the design complexity by hierarchically composing semantically different modeling frameworks. Frameworks are software architectures that define component ontology and interaction styles. Formal frameworks for embedded software make programming models and software architectures reusable.

Embedded systems that engage the real world need to be reactive. This dissertation focuses on studying reactivity and its composition in different frameworks. It introduces the reactor model as an abstract operational semantics to capture interactions among com-

ponents and frameworks. Within a framework, a component execution is a precise reaction if all the prerequisites for the reaction are satisfied before it is being triggered. A framework that only triggers precise reactions is a responsible framework. Precise reactions and responsible frameworks allow us to capture compositionality of reactions, answering questions such as how a composition of a framework and components can be treated as an atomic component at a higher level. This compositionality is key for hierarchically composing heterogeneous models.

Precise reactions and responsible frameworks are discussed for timed models. Having a notion of time helps designers define timely reactions. But it also brings challenges to timed frameworks to precisely determine the triggering time. In terms of modeling mixed-signal and hybrid systems, the challenge is how to precisely control the progression of modeling time. We present techniques for a responsible continuous-time framework to have compositional precise reactivity. These techniques involve optimistic look-ahead execution and possible rollback.

We further study precise reaction and responsible frameworks for priority-based run-time embedded software. A timed multitasking (TM) model of computation is proposed for programming reactive real-time embedded software. This model brings time determinism to the programming model level. We sketch a responsible run-time system that preserves the timing semantics of TM models.

Professor Edward A. Lee
Dissertation Committee Chair

To Feng

and my extended family

Contents

List of Figures

List of Tables

Acknowledgements

I am deeply grateful to my advisor Professor Edward A. Lee, not only for his support and guidance of this research, but also for the pleasant and creative group culture he established in the Ptolemy team. The break-through research vision, the deep thinking spirit, the rigorous academic training, and the ingenuous friendship will benefit me my entire life.

I would like to thank Professor Shankar Sastry and Professor John Strain for serving on my dissertation committee. Their comments are highly appreciated. I also thank Dr. Jörn Janneck, Xiaojun Liu, and Elaine Cheong for commenting and proofreading the draft of this dissertation.

I feel really privileged about the cooperative research atmosphere at UC Berkeley. Besides my group colleagues, especially Dr. Jörn Janneck, Dr. Johan Eker, Dr. John Reekie, Xiaojun Liu, Yuhong Xiong, and Stephen Neuendorffer, I have been significantly influenced and inspired by Professor Shankar Sastry and his group members — Dr. John Koo, Dr. Karl Johansson, Dr. George Pappas, now a professor at Univ. of Pennsylvania, Dr. Claire Tomlin, now a professor at Stanford Univ., and Dr. David Shim, Professor Tom Henzinger and his group members — Dr. Christoph M. Kirsch and Ben Horowitz, and Professor Pravin Varaiya and his student Tunc Simsek.

I appreciate the Chinese student community at Berkeley. Many friends, especially Jianghai Hu, Yunjian Jiang, Xiaojun Liu, Dr. Yi Ma, now a professor at UIUC, Z. Morley Mao, Dr. Yuke Wang, a professor at UT Dallas, Jun Zhang, Lizhong Zheng, Yang Zhao, and Xiaoming Zhu, have made my student life very enjoyable.

Finally, I cannot express my gratefulness and love to my wife Feng Wang, my

father Zhiyan Liu, my sister Tong, and my extended family. This work could not have been accomplished without their understanding, support, and enthusiasm.

Chapter 1

Introduction

The main objective of this dissertation is to describe techniques that will help modeling and design of real-time distributed embedded systems that engage the real world. Embedded systems are computer systems that are embedded in other devices, which makes them not first and foremost computers. A large class of these systems interact directly with the real world and with other embedded systems. Examples are automatic control systems (as in automobiles, airplanes, and industrial plants), test and measurement instruments, optical network switches, office equipment, smart home appliances, intelligent toys, and so on. These systems need to take physical inputs, react in real-time, and produce outputs at the right time. The reactive nature and time criticality make computing in embedded systems significantly different from computing in traditional data-processing and transaction-based computer systems, which typically interact with humans and emphasize total throughput and average performance.

Embedded systems are usually highly customized, since the systems are intended

only to perform a limited set of tasks, rather than target all possible applications. Thus, when designing such systems, designers must consider constraints on the physical environment, I/O devices, power consumption, code size, etc., which are typically not well characterized by mainstream computer sciences. As a consequence, the majority of current embedded systems are designed in a hand-crafted manner, which makes designing embedded systems more like an art than an engineering discipline. Typically, designers pick a hardware platform/architecture by guessing whether it will be sufficient for the system, choose or not choose an operating system based on intuitions of the complexity of software tasks, develop code using assembly or some customized high-level language (like C) in order to have a better estimation of timing, and tweak the scheduling algorithms (like the priorities of the tasks), until the system seems to work. Such a design methodology is very time-consuming, fragile, and unscalable. A slight change of hardware platform, a small addition of functionality, a minor miss-estimation of the working conditions, or even a bug fixing, may break the whole system, and force a complete redesign from the beginning.

As embedded systems become ubiquitous with increasingly complicated functionality and networked communications, no single designer can manage a complete design cycle with the time-to-market pressure. The whole system needs to be decomposed into small pieces, and many designers have to work together. However, how to decompose a system and how to compose the components to achieve desired functional and timing properties are big challenges for system designers. Usually, due to the lack of system-level understanding between domain experts and software engineers, the system integration cost at the end of a project become very high.

Embedded systems usually consist of *heterogeneous* components. Typically, there may be hydraulic and/or mechanical parts, analog circuits for sensors and actuators, communication circuits, application specific digital circuits, micro-processors and/or micro-controllers, memories, and embedded software. These components interact with very different styles, which leads the designers of these components to have very different ways of thinking. In a sense, a key challenge of system-level design of embedded systems is how to integrate all these ways of thinking.

My thesis works on modeling and design issues of embedded systems, particularly, those systems that engage the physical world, have multiple modes of operation, involve networked interactions, and react in real-time. The long term objective of this work is two folds:

- to enrich computer sciences with heterogeneous modeling techniques, their interacting semantics, and programming models, and
- to make state-of-the-art computer science theories and software engineering techniques accessible to embedded system designers by providing computer-aided modeling and design frameworks that allows domain experts to easily prototype ideas, reuse previous designs, and generate hardware and software implementations from high-level specifications.

In this thesis, the approach to managing heterogeneous models is a component-based one. A *system* is an aggregation of interacting components, and each component may be decomposed further into smaller components with possibly different interaction styles. The thesis studies reactivity properties in component-based frameworks. An abstract se-

mantics model, called the *reactor model*, is introduced to cover a wide variety of models of computation. New concepts - *precise reaction* and *responsible frameworks* - are introduced to systematically define reactivity and its composition. The concepts are applied in both timed and untimed models. After studying precise reactions in real-time systems, I propose a new programming model for priority-driven multitasking embedded software. This programming model makes time explicit and resource management transparent to programmers.

1.1 Heterogeneity in Embedded System Modeling

The word "heterogeneity" referred to in this thesis is at the modeling level, rather than at the implementation level. For example, although the interaction among mechanical components and that among analog circuits are very different physically, as long as they both can be modeled as ordinary differential equations, there is no heterogeneity. Thus, the heterogeneity at the modeling level is in the sense of component interaction styles, logically or mathematically.

Take an engine control system as an example, shown in Figure 1.1. A cylinder of an internal combustion engine has four working phases: intake (I), compress (C), explode (E), and exhaust (H). The engine generates torque that drives the power train and the car body. Depending on the car body dynamics, the fuel and air supply, and the spark signal timing, the engine works at different speeds, and thus makes phase transitions at various time instances. The job of the engine controller is to control the fuel and air supplies as well as the spark signal timing, corresponding to the drivers demand and available sensor

information from the engine and the car body.

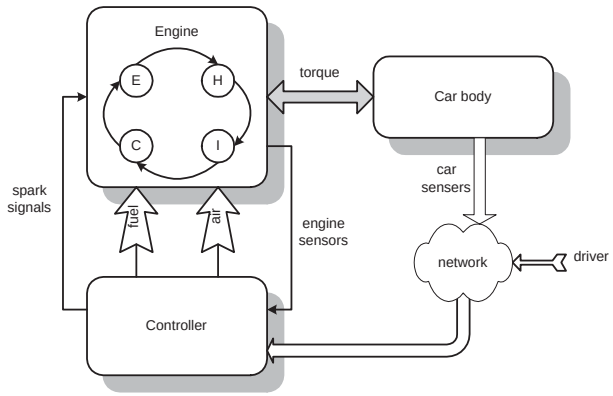

Figure 1.1: An engine control system

When designing the engine controller, one wants to quickly validate the control algorithms before considering the implementation details. So, one may start with modeling and simulating the entire system, including the engine and car dynamics, at a high level of abstraction. The engine and the car body are mechanical systems, which are naturally modeled using differential equations. The four phases of the engine can be modeled as a finite state machine, with a more detailed continuous dynamics for the engine in each of the phases. While all the mechanical parts interact in a continuous-time style, the embedded controller, which may be implemented by some hardware and software, works discretely. In particular, sensor information and driver's demands may arrive through some kind of network. The controller receives this information, computes the control law, controls the air and fuel valves, and produces spark signals, discretely. So, we want to use a model that is suitable for handling discrete events for the network and the controller. Within the discrete controller, the control algorithms may be implemented as software, and there may be multiple software tasks sharing the same CPU and other resources. And, the real-time

scheduling policy may greatly affect the closed-loop performance.

In this not so complicated example, we have seen both continuous-time models and several quite different discrete models - finite state machines, discrete events, and real-time scheduling. All these models have distinct characteristics in terms of what the components are and how those components interact. At an abstract level, we view components as mathematical objects rather than physical devices, and call these characteristics *models of computation* (MoC).

There are many useful models of computation for designing embedded systems. Table 1 is extended from [43], in which Lee has an insightful discussion of several of them. I will define some of them more precisely in later chapters.

Notice that many of the models in the table have various abstraction of time. Some are continuous, like CT, DE, and PDM; some are discrete, like DT and SR; some abstract time away, as in Kahn's process networks and communicating sequential processes. The different notions of time make programming for embedded systems significantly different from programming in desktop, enterprise, and Internet applications.

A natural question to ask, after realizing the diversity of models of computation and heterogeneity of system modeling, is how to use these models coherently in system designs. Our approach is a component-based one. In particular, we use hierarchies to integrate different models and keep models clean at each level.

Table 1.1: Briefs on Models of Computation

MoC	**Brief**	**Possible Applications**
Asynchronous Message Passing (e.g. Kahn's Process Networks)	Processes interact by channels (e.g. FIFO queues) that can buffer messages.	May be used for loosely coupled distributed agents, data-centric algorithms, like signal processing, system identification, and streaming data application, etc.
Continuous-Time (CT)	Functional and storage components communicate with continuous waveforms.	Physical environment, analog circuits, and continuous control laws, etc.
Discrete Events (DE)	Components communicate via signals that carry events placed in time, which is continuous and globally known.	Digital circuits, communication network, queuing systems, and embedded software at the I/O level, etc.
Discrete Time (DT)	Global notion of time, periodical discretized. Every signal has a value at every clock tick.	Periodically sampled data systems and cycle-accurate modeling.
Finite State Machines (FSM)	States and transitions among them. Transitions are triggered by events.	Operational modes and control sequences.
Priority-Driven Multitasking (PDM)	Software tasks sharing resources. Tasks may be preempted.	Embedded software modeled at the operating system level.
Synchronous Message Passing (e.g. Communicating Sequential Processes)	Processes rendezvous, communicating in atomic instantaneous actions.	Concurrent processes accessing critical sections, resource management, etc.
Synchronous/ Reactive (S/R)	Global clock triggers computations that are conceptually simultaneous and instantaneous. Signals may have well-defined empty value.	High-level modeling for reactive real-time hardware and software.

1.2 Component-Based Design in Embedded Software

The principle of component-based design is essential to engineering and has existed long before the invention of computers. It advocates designing components to fit a wide range of applications, and building applications by assembling standard components together with a small number of application-specific components. We have seen this in mechanical engineering over centuries, where a wide variety of standard components has been defined internationally. We have also seen these in electronics, especially in the personal computer (PC) industry, where many components such as processors, memories, disk drives, and extension boards are standardized and highly interchangeable.

Components encapsulate expertise and induce certain formal properties. Component-based design achieves the system qualities by inheriting the expertise, shortens the design cycles by reusing building blocks, and reduces system cost by mass production. Modularization and software reuse are always the main themes of software engineering. However, unlike mechanical systems and electronics hardware, which primarily have only one interaction style among components (force/acceleration for mechanical components, and current/voltages for electronics), software components can interact in much more abstract forms and diverse styles. Thus, it is not immediately obvious how to define a software component or to capture their interactions. Over the years, there have been many attempts to define reusable components in software engineering. Examples include subroutines, objects, software services, and frameworks.

1.2.1 Subroutines

Subroutines are probably the most common type of reusable software component. A subroutine is a finite computation that processes (a predefined type of) input data and produces final results. A big problem with subroutines is the weak management of internal states and the lack of encapsulation. There could easily be unspecified requirements and/or side effects.

1.2.2 Objects

Object orientation improves on subroutines by introducing well-defined boundaries and encapsulation of states and behaviors. In [9], the Object Management Group (OMG) defines an object in object-oriented design as:

An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships; behavior is represented by operations, methods, and state machines.

Object orientation matches well with system decomposition in many problem domains, and raises the abstraction of programming by advocating object encapsulation and class hierarchies. However, the basic object model only offers one mechanism of component interactions – method calls. A method call immediately transfers the flow of control from one object to another. It is up to the programmers to manage concurrency and persistence. This issue becomes more cumbersome for multi-threading programs and distributed object models, like Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM). In these models, the only primitives – synchronous

and asynchronous method calls – make it very hard to reason about flows of control, order of events, and deadlock in distributed object-oriented systems.

1.2.3 Services

Software services are abstractions of one or more objects or procedures that together perform some functions. The main goal of building software services is reuse. Services have a well-defined interface, and are composable with other services to build higher-level systems. This seemingly small step from object orientation introduces a paradigm shift in reusing software components, in the following senses:

- Software services usually impose programming models. For example, the CORBA event service [57], Ninja [23], and JavaSpaces [19] all impose an event-driven programming model. Thus, interaction among components becomes a first-level concern.
- Software services are usually distributed, and concurrency issues become explicit.
- Software services are active processes rather than passive subroutines or methods. Services typically never terminate. They wait for requests, perform their computation, and produce replies.

However, service-based programming leaves the integration of services completely to programmers, and it is weak at managing resources when the service is used by many clients. As a consequence, it is hard to analyze real-time performance of service-based systems.

1.2.4 Framework

In system design communities, the term "framework" generally refers to software architectures that integrate components. Ralph Johnson defines frameworks in the object-oriented programming context [37] as:

a reusable design expressed as a set of abstract classes and the way their instances collaborate.

We use this term in a broader sense, which does not necessarily tie to object orientation and "classes." In this thesis, a framework is a software architecture that imposes a set of constraints on the interactions of components, provides a set of services that components may use, and may induce a set of benefits (e.g. formal properties) for the system. A model of computation can be implemented as a framework, so are many ad hoc software architectures. A "good" framework makes software architectures and programming models reusable (as opposite to simply making code reusable). By solving meta-level problems, like communication styles, scheduling, flow of control, and resource management, good frameworks allow designers to focus on the development of individual components, which are typically small and easy to manage.

Many software frameworks have been developed over years. Agha's actor model [1] is a framework. It defines distributed components (called actors) and their communication styles - unstructured event passing. Pree's framelet [59] model is a framework. It defines components as objects with call-back functions, and the framework provides real-time scheduling services. Stewart's port-based object (PBO) model [69] is anther example of frameworks. It defines components as port-based objects interacting through buffers

of length one, and schedules the execution of the components. The open control platform (OCP) [80] is a framework that provides a component model, and extends real-time scheduling techniques to a distributed system using a real-time CORBA [64] and its event service [58]. There are also many commercial frameworks. Sometimes, designers use frameworks without even realizing that. For example, Simulink [29] is a modeling and simulation environment for continuous-time dynamic systems with discrete events. Like many other timed frameworks, Simulink has a specific way of controlling the execution of components (i.e. *blocks*) and a specific way of modeling time.

1.3 Hierarchical Heterogeneity

In many component-based design frameworks, hierarchy refers to the containment relation, where as in object-orientation, hierarchy refers to the inheritance relation. In a containment relation, an aggregation of components can be treated as a (composite) component at a higher level. In general, hierarchies help manage the complexity of a model by information hiding — to make the aggregation details invisible from the outside and thus a model can be more modularized and understandable.

A framework, together with components contained by it, can be a component of a bigger framework. If these frameworks represent heterogeneous models of computation, the approach is called *hierarchical heterogeneity*. An example of modeling the engine control system in the hierarchical heterogeneous approach is shown in Figure 1.2. The top-level is a discrete event (DE) model, where a discrete controller interacts with a discrete abstraction of the car model. Inside the controller, a priority-driven multitasking model is used to model

multiple software tasks. The discrete car model is internally implemented by a continuous dynamics of the engine model and the car body model. The engine is further modeled as a finite state machine (FSM), and within each state, there is a continuous-time subsystem modeling the engine working in that phase. In this hierarchical heterogeneous approach, the model of computation within each layer is well-defined. The interface among layers can be taken care of by the frameworks designers, instead of by component designers. If we can solve the framework integration problem, component designers can work within their familiar frameworks, and designs become highly manageable and understandable. So, the challenge remains to study and implement frameworks that support hierarchical heterogeneity.

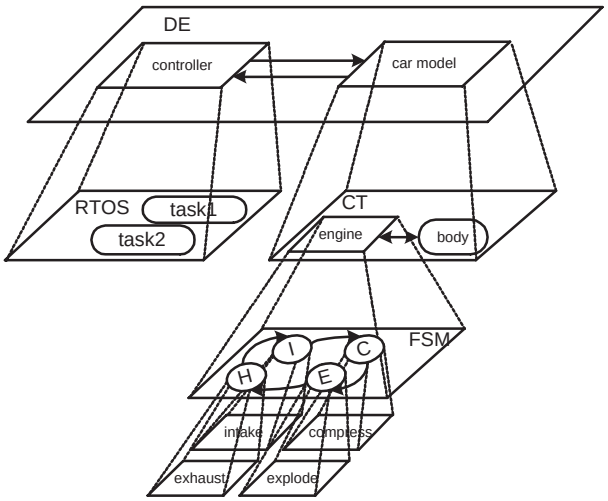

Figure 1.2: A hierarchical model for the engine control system

1.4 Thesis Outline

The remainder of this dissertation starts in Chapter 2 with an introduction of the reactor model, which is an abstract operational semantics model that targets the study of

reactivity and its compositionality. A distinctive feature of this model is the separation of components and frameworks. Components exist in a framework, communicate with one another through the framework, and react to the triggers sent by the framework. Such a component is called a reactor. Within the reactor model, we develop the concepts of reactions, precise reactions, responsible triggers, and responsible frameworks to formally study reactivity across models of computation. A core concept – precise reaction – states that the reaction solely depends on the triggers sent by the framework. A responsible framework can guarantee that all executions within it are precise reactions. We compare some models of computation for responsibleness, and argue that some are responsible, while some are not.

The precise reaction problem may seem trivial for atomic reactors, which are reactors with a single thread of control and finite firings. However, since a reactor can be implemented by a framework containing multiple other actors, and the framework may execute these components concurrently, it is not trivial to make a concurrent reaction precise. Chapter 3 studies the compositionality of reactions. A composite reactor implemented by a responsible framework can easily achieve compositional precise reactions. This allows hierarchical composition of models of computation to have a well-defined semantics. It also becomes possible to precisely integrate concurrent models, like dataflow models, discrete-event models, and continuous-time models, with sequential models like state machines.

The notion of time is very important for embedded systems interacting with the real world. Chapter 4 focuses on a particular class of frameworks which have a continuous notion of time. It shows how having a notion of time helps in defining precise reaction

points. It presents techniques to implement responsible continuous-time frameworks and to make a continuous-time framework precisely reactive. This study provides a semantic insight to model and simulate two particularly useful models that integrate both continuous and discrete dynamics – the mixed-signal model and the hybrid system model.

Chapter 5 studies the precise reaction and responsible frameworks issue in priority-based multitasking real-time programs. It shows that having the notion of precise reaction can avoid the priority inversion problem. With the precise reaction property, the response time of a component is much easier to analyze and control. I also present a real-time programming model, called *timed multitasking* (TM), which integrates the concept of precise reaction with priority-based scheduling and preemptive execution. A real-time responsible framework can help embedded software to achieve precise mode switches and both time- and value-determinism.

1.4.1 Contribution

In summary, this dissertation makes the following primary contributions:

- introducing the reactor model and characterizing precise reaction and responsible frameworks;
- analyzing the advantages of responsible frameworks in the context of compositional reactivity and hierarchical heterogeneous design;
- studying the timed precise reaction problem and the integration of timed models and presenting the implementation of timed responsible frameworks;

- proposing a real-time programming model that allows run-time resource management and prioritized precise reactions.

Chapter 2

Reactors and Frameworks

In this chapter, we present the basic structure of our component model – *actors*, *frameworks*, and the interaction among them. This architecture, called the *reactor model*, clearly distinguishes the activities among actors and frameworks in terms of *computations*, *communication*, and *control*.

The reactor model is an abstract operational semantics model for component-based computation. A fundamental distinction of this model is the concept of frameworks. Actors reside in frameworks and interact with other actors through frameworks. Frameworks control the execution of actors by sending them triggers. An actor defines a set of partially ordered computation and communication. A framework gives the semantics of communications and defines a set of partial order relations on communication and controls. When triggered by a framework, the execution of an actor is constrained by the conjunction of the two sets of partial ordering relations.

An actor is *reactive*, thus called a *reactor*, if the triggered execution is finite.

Therefore, a reaction always finishes in a finite amount of time. However, this does not necessarily mean that a response has been fully completed. For example, a reaction may not be completed because of lack of enough inputs. Intuitively, a reaction is *precise*, if it completes the desired computation and reaches a *quiescent state*. If a framework only triggers an actor when the actor can perform a precise reaction, then the framework is said to be *responsible*. A framework may need actors' cooperation to be responsible.

2.1 Model Structure

In this section, we introduce the basic entities of the reactor model – actors, connections, and frameworks.

2.1.1 Actors, Connections, and Frameworks

An *actor* A, as depicted in Figure 2.1, has a set of *variables*, denoted by X. We write $A.X$ if distinguishing of actors is needed. Variables contain *values*, which encapsulate arbitrary data. We denote the set of values by V. Among the variables, some are called *interface variables*, or *ports*, and partitioned into a set of input ports, P, and a set of output ports, Q. Other variables are *internal variables*, S. That is, $P \cap Q = \emptyset$, $P \cap S = \emptyset$, $Q \cap S = \emptyset$, and $P \cup Q \cup S = X$.

An *evaluation* of a variable is a function that gives the value contained by a variable, i.e. $\sigma_X : X \to V$. By definition, $\perp \in V$, where $\perp$ is the empty value. A variable evaluating to $\perp$ means that there is no meaningful value in that variable. We write $[X \to V]$ for the set of all functions mapping X to V. Thus, $\sigma_X \in [X \to V]$, which is also called the

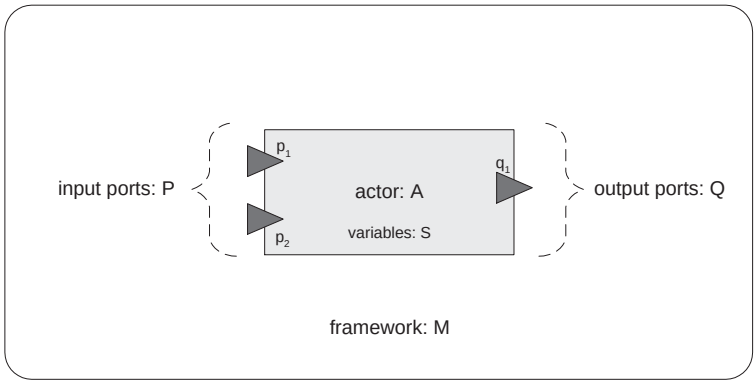

Figure 2.1: A framework that contains one actor.

state of an actor.

Let's look at an example. `AddMultiply` is an actor that computes the sum and product of two numbers. The actor has two input ports and two output ports, as shown in Figure 2.2.

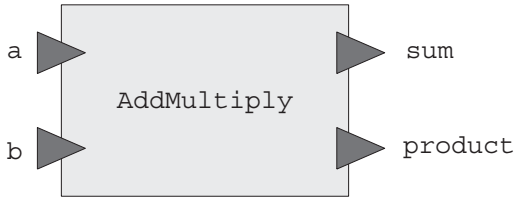

Figure 2.2: An `AddMultiply` actor that reads two inputs and produces their sum and product.

It has variables `a` and `b` as inputs, variables `sum` and `product` as outputs, and no internal variables. I.e.

$$\texttt{AddMultiply}.P = \{\texttt{a}, \texttt{b}\}$$

$$\texttt{AddMultiply}.Q = \{\texttt{sum}, \texttt{product}\}$$

$$\texttt{AddMultiply}.S = \emptyset$$

An actor is controlled by a *framework*. We write $A \Subset M$, if the actor A is controlled by the framework M. A framework can control many actors. We define $M.\mathbf{Actors}$ to be the set of all actors controlled by M, i.e. $M.\mathbf{Actors} = \{A | A \Subset M\}$.

A framework has a set of variables, called *framework variables*, denoted by $M.Z$. We similarly define the evaluation of framework variables $\sigma_Z : Z \to V$ and the set of all possible evaluations $[Z \to V]$.

Actors in the same framework can be composed by connecting their ports. For example, Figure 2.3 shows the output of an actor A connecting to the input of an actor B, within a framework M. A connection is called a *channel*. A channel c is simply a pair of ports. We write $c = (q \rightsquigarrow p) \in A.Q \times B.P$ for a channel that connect output port q of actor A to port p of actor B. A channel defines a set of communication variables, $Z_c \subseteq M.Z$.

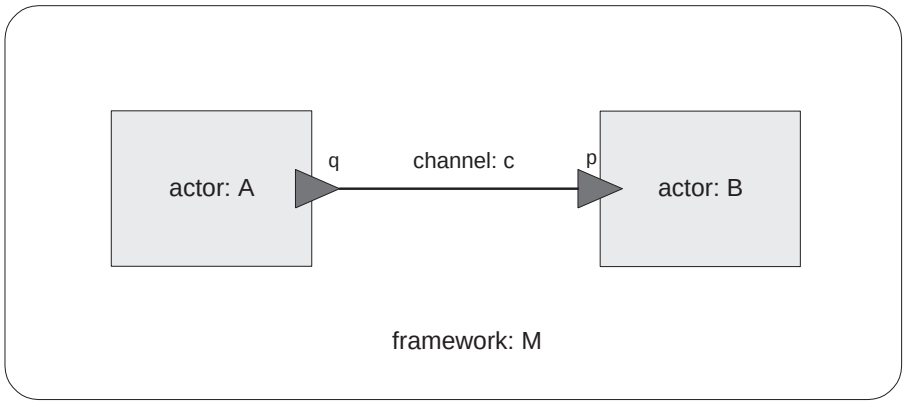

Figure 2.3: Composing two actors.

The aggregation of a framework, M, the actors under its control, $M.\mathbf{Actors}$, and the connections among these actors $M.\mathbf{Connections}$, is called a *composite*, $\Theta = (M, M.\mathbf{Actors}, M.\mathbf{Connections})$.

2.1.2 Operations

Over the framework variables, actor variables, and their evaluations, we define a set of operations, **Oper**. Within the set **Oper**, some operations are performed by the actors and some are performed by the framework. And, operations may be ordered.

Some operations capture the *dataflow* aspect of the computing. These operations deal with *how* to compute new data and send data around. These operations typically change the evaluation of some variables. Other operations do not directly change the value in any variables, but they affect the order among other operations. This is the *control flow* aspect of the computing, which deals with *when* (instead of *how*) computation and communication happen. In the reactor model, control flow only takes place between actors and frameworks, through sets of control-flow operations.

Dataflow Operations

Among the operations performed by an actor A, some serve for computing new data from old data. These are called *computational* operations, $A.\mathbf{Comp}$. Each element of $A.\mathbf{Comp}$ is a partial function $f : [X \to V] \to [X \to V]$ satisfying:

$$f(\sigma)_{[P]} = \sigma_{[P]}, \forall \sigma \in [X \to V], \tag{2.1}$$

where $\sigma_{[P]}$ is the projection of function σ on $P \subset X$. That is, the operations in **Comp** can only change the values in internal variables and output ports, and must leave the values in input variables unchanged. We write $(y_1, y_2) = f(x_1, x_2, x_3)$ for a computation that uses the values of variables x_1, x_2, and x_3, and changes the values of variables y_1 and y_2.

For example, for the `AddMultiply` actor, there may be two computational opera-

tions: $\texttt{sum} = \texttt{add}(\texttt{a}, \texttt{b})$ and $\texttt{product} = \texttt{multiply}(\texttt{a}, \texttt{b})$.

Ports of an actor are the communication interface to other actors through its framework. The value in an input port can only be changed by the framework, while the value in an output port can only be changed by the actor. Thus, we define a set of communication operations for actor A with its framework M, called $A.\textbf{Comm}$, including:

- A set of *read* operations, denoted by $A.\textbf{Read}$. We write $\texttt{read\_p} \in A.\textbf{Read}$ for an operation that reads from input port p. And, $\texttt{read\_p} : [Z \to V] \to [\{p\} \to V] \times [Z \to V]$, which changes the value in an input port $p \in P$ based on the state of the framework, and may change the state of the framework. Strictly speaking, a read operation on p can only access and change values in the framework variables that are connected to p, i.e. let $Z_{\tilde{p}} \subset M.Z$ be set of framework variables for channels that do **not** connect to p, then `read_p` should satisfy,

$$\texttt{read\_p}(\sigma)_{[Z_{\tilde{p}}]} = \sigma_{[Z_{\tilde{p}}]}, \forall \sigma \in [Z \to V]. \tag{2.2}$$

- A set of *write* operations, denoted by $A.\textbf{Write}$. We write $\texttt{write\_q} \in A.\textbf{Write}$ for an operation that writes through output port q. The write operation cannot change the values in q, nor any framework variables for channels that do not connect to q. So, $\texttt{write\_q} : [\{q\} \to V] \times [Z \to V] \to [Z \to V]$ satisfies:

$$\texttt{write\_q}(\sigma)_{[Z_{\tilde{q}}]} = \sigma_{[Z_{\tilde{q}}]}, \forall \sigma \in [Z \to V]. \tag{2.3}$$

  The exact behaviors of read and write operations are determined by the framework.

- The set of communication operations, $A.\textbf{Comm} = A.\textbf{Read} \cup A.\textbf{Write}$.

For the `AddMultiply` actor, there may be four communication operations: `read_a`, `read_b`, `write_sum`, and `write_product`.

Control-flow Operations

A framework M controls the activities of actors by a set of *control* operations, $M.\mathbf{Ctrl} = \bigcup_{A \Subset M} A.\mathbf{Ctrl}$, where $A.\mathbf{Ctrl}$ are the control operations for actor $A \Subset M$. A key for the reactor model is that the activities of actors are always *triggered* by frameworks. For any interesting framework M containing an actor A, there is at least one element $A.\texttt{trigger} \in A.\mathbf{Ctrl}$. Thus, the set $M.\mathbf{Ctrl}$ is never empty. The contents of $M.\mathbf{Ctrl}$ may be enriched to enhance the capability of a framework. When to issue a control operation to an actor is a key issue for a framework. We will discuss more about triggers in section 2.2.3, and about other control operations when we introduce them.

A framework also provides a set of *callback* operations, $M.\mathbf{Clbk} = \bigcup_{A \Subset M} A.\mathbf{Clbk}$, which the actors in it may use to affect the activities of the framework . The set $A.\mathbf{Clbk}$ has at least one element $A.\texttt{finish\_trigger}$, that actor A can use to indicate that it has no more operations to perform for a trigger, `trigger`.

As shown in Figure 2.4, the operations that an actor A can perform are exactly the computation, communication, and callback operations, i.e.

$$A.\mathbf{Oper} = A.\mathbf{Comp} \cup A.\mathbf{Comm} \cup A.\mathbf{Clbk}$$

And the operation that a framework M can perform are $M.\mathbf{Oper} = M.\mathbf{Ctrl}$. Recall that these are all the operations that can be performed inside a composite, so

$$\mathbf{Oper} = M.\mathbf{Ctrl} \cup \left\{ \bigcup_{A \Subset M} A.\mathbf{Oper} \right\}.$$

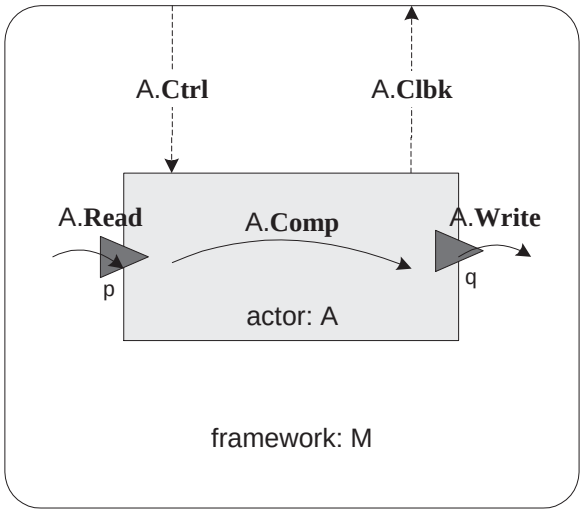

Figure 2.4: Sets of operations between actors and frameworks.

2.1.3 Ordering among Operations

We use an ordering relation $\prec \subseteq$ **Oper** $\times$ **Oper** to model the *causality* among operations. For $f, g \in$ **Oper**, we say f *preceeds* g, denoted by $f \prec g$, if the operation f must be performed before the operation g. With this relation, the set **Oper** is a partially ordered set.

Partially Ordered Sets

Definition 2.1. *For a (ground) set* Γ*, a relation* $\prec \subseteq \Gamma \times \Gamma$ *is called a* ***strict partial order relation*** *if it satisfies (for any* $f, g, h \in \Gamma$*):*

- *Irreflexive:* $f \not\prec f$*;*
- *Anti-symmetric: if* $f \prec g$*, then* $g \not\prec f$*;*
- *Transitive: if* $f \prec g, g \prec h$*, then* $f \prec h$*.*

We also write $g \succ f$ for $f \prec g$. A set with a partial order relation, $(\Gamma, \prec)$, is called a *partially ordered set* (or, *poset* for short). This relation is so called, because in a poset, there may exist elements f and g, such that neither $f \prec g$ nor $g \prec f$. These elements are *incomparable*, denoted by $f \| g$. If all elements in the ground set are comparable, then the set is called a *totally ordered set*, or a *chain*. In a poset Γ, we say g *covers* f, denoted by $f \lhd g$, if $f \prec g$ and there is no such element $h \in \Gamma$, s.t. $f \prec h \prec g$. The partial order relation is the *transitive closure* of the covering relations.

Posets can be visually represented by *Hasse* diagrams. In a Hasse diagram, as depicted in Figure 2.5, nodes represent elements in a set, and arrows represent ordering relations. An arrow is drawn from node f to node g if $f \lhd g$ in the set. For example, in Figure 2.5, $a \prec c$, $b \prec e$, but $c \| d$.

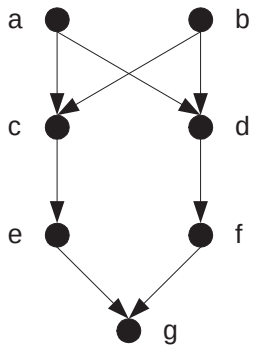

Figure 2.5: A Hasse diagram for a partially ordered set.

Synchronization point

Definition 2.2. *For a poset* $(\Gamma, \prec)$*, an element* $w \in \Gamma$ *is called a* ***synchronization point*** *of* Γ *if for any* $f \in \Gamma, f \neq w$*, either* $f \prec w$ *or* $w \prec f$*.*

For example, in the poset shown in Figure 2.5, the node g is a synchronization point. Obviously, synchronization points may not exist for arbitrary posets. But, if they do exist, they give a total order to a subset of the poset. It is easy to show that the following property holds:

Proposition 2.1. *Let W be a set of synchronization points of $(\Gamma, \prec)$, then $(W, \prec')$ is a chain, where $\prec'$ is the projection of $\prec$ on W.*

Compatibility of partial ordering relations

The union and transitive closure of two sets of partial ordering relations on the same ground set may not define a poset. It is not hard to conceive that, there may be a conflict such that $f \prec g$ in one ordering relation and $g \prec f$ in another ordering relation. In order to define compatibility of posets, we introduce refinements of posets.

Definition 2.3. *Let $\prec$ and $\prec'$ be two partial order relations on the same ground set Γ. Then, $\prec'$ **refines** $\prec$ if $\prec \subset \prec'$.*

Definition 2.4. *Two partial order relations are **compatible** if they have a common refinement.*

For two compatible partial order relations $\prec$ and $\prec'$ on a ground set Γ, $(\Gamma, \textbf{Closure}(\prec \cup \prec'))$ is a poset, where **Closure** is the operator for transitive closure.

Two incompatible partial order relations may be made compatible by removing elements from the ground set, which, apparently, also removes element pairs from the relations. However, in general, there may not exist a minimum set of elements, such that removing them from the ground set can make two incompatible relations compatible. For

example, for ground set $\Gamma = \{a, b, c, d\}$ shown in Figure 2.6, $\texttt{prec}_1 = \{(a, b), (b, c)\}$, and $\texttt{prec}_2 = \{(c, d), (d, a)\}$, removing either a or c from Γ will make $\texttt{prec}_1$ and $\texttt{prec}_2$ compatible. However, neither $\{a\}$ nor $\{c\}$ contains the other.

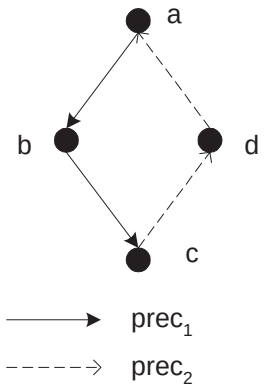

Figure 2.6: The union of two incompatible partial orders.

Indexing Operations

The set of all operations **Oper** is a poset, and we assume some basic ordering relations on it. For example, in order to distinguish different executions of an operation with the same name, we assign index numbers to these operations so that each execution of the operation is a distinct element in **Oper**. For any operation with name `f`, we use `f_i` and `f_j` to denote the i^{th} and the j^{th} execution of `f`. Thus, if $i < j \in \mathbb{N}$, then $\texttt{f\_i} \prec \texttt{f\_j}$. We denote this partial ordering by $\prec_0$. This relation applies to all computation, communication, and control operations. In particular, for actor A with input port `p` and

output port q, trigger trigger, and $i < j \in \mathbb{N}$,

$$A.\texttt{trigger\_i} \prec A.\texttt{trigger\_j}$$

$$A.\texttt{read\_p\_i} \prec A.\texttt{read\_p\_j}$$

$$A.\texttt{write\_q\_i} \prec A.\texttt{write\_q\_j}$$

Now, we can write down the full sets of operations for the AddMultiply actor:

$$\texttt{AddMultiply}.\mathbf{Comm} = \{\texttt{read\_a\_i}, \texttt{read\_b\_i}, \texttt{write\_sum\_i}, \texttt{write\_product\_i}, \texttt{i} \in \mathbb{N}\}$$

$$\texttt{AddMultiply}.\mathbf{Comp} = \{\texttt{sum} = \texttt{add\_i}(\texttt{a}, \texttt{b}), \texttt{product} = \texttt{multiply\_i}(\texttt{a}, \texttt{b}), \texttt{i} \in \mathbb{N}\}$$

$$\texttt{AddMultiply}.\mathbf{Clbk} = \{\texttt{finish\_i}, \texttt{i} \in \mathbb{N}\}$$

The control-operation set, which can only be used by a framework, could be:

$$\texttt{AddMultiply}.\mathbf{Ctrl} = \{\texttt{trigger\_i}, \texttt{i} \in \mathbb{N}\}$$

2.2 Execution

The execution of an actor is a set of operations. For reactive executions, this set of operation is finite. The execution of actors not only depends on the computation defined by the actor designer, but also depends on the semantics and ordering relations that a framework imposes on communication and control.

2.2.1 Firing Sets

A firing set of actor A, denoted as A.fire, is simply a partially ordered subset of computation and communication performed by A, i.e. $\texttt{A.fire} \subseteq A.\mathbf{Comp} \cup A.\mathbf{Comm}$. It

is the set of desired operations and their causality orders that the designer would like the actor to perform. In the reactor model, the execution of a firing set must be triggered by the framework. We write $\texttt{A.fire}|_{\texttt{r}}$ for a firing set triggered by r. The ordering relation $\prec_{A|_r}$ among elements of $\texttt{A.fire}|_{\texttt{r}}$ defines the order that the operations should be performed.

A firing set $\texttt{A.fire}|_{\texttt{r}}$ is (designed to be) *reactive* if it contains finite operations. And,

Definition 2.5. *An actor is **reactive**, or is a **reactor**, if its firing sets are reactive for all triggers.*

So, a firing set defines a desired set of operations that the actor performs when it is triggered. For a reactor, this set of operations is at most finite. The partial order relations within a firing set should at least be compatible with the indexing relations among read, write, and computational operations. An actor designer may add further ordering relations depending on the algorithms that the actor implements.

For example, a firing set of `AddMultiply` for the k^{th} trigger, `r_k` from the framework could be:

$$
\begin{aligned}
&\texttt{AddMultiply.fire}|_{\texttt{r\_k}} = \{ \\
&\qquad\qquad \texttt{read\_a\_k}, \\
&\qquad\qquad \texttt{read\_b\_k}, \\
&\qquad\qquad \texttt{sum} = \texttt{add\_k}(\texttt{a}, \texttt{b}), \\
&\qquad\qquad \texttt{product} = \texttt{multiply\_k}(\texttt{a}, \texttt{b}), \\
&\qquad\qquad \texttt{write\_sum\_k}, \\
&\qquad\qquad \texttt{write\_product\_k}, \\
&\qquad \} \\
\end{aligned}
$$

Obviously, this firing set is finite for all triggers, and thus the actor `AddMultiply` is a reactor. And one possible ordering relation $\prec_{\texttt{AddMultiply}}$ on this set may be the transitive

closure of:

$$\begin{aligned}
\texttt{read_a_k} &\prec \texttt{read_b_k} \\
\texttt{read_b_k} &\prec \texttt{sum} = \texttt{add_k}(\texttt{a}, \texttt{b}) \\
\texttt{sum} = \texttt{add_k}(\texttt{a}, \texttt{b}) &\prec \texttt{product} = \texttt{multiply_k}(\texttt{a}, \texttt{b}) \\
\texttt{product} = \texttt{multiply_k}(\texttt{a}, \texttt{b}) &\prec \texttt{write_sum_k} \\
\texttt{write_sum_k} &\prec \texttt{write_product_k}
\end{aligned}$$

which essentially defines a chain. A Hasse diagram of this partial ordering relation, is shown in Figure 2.7.

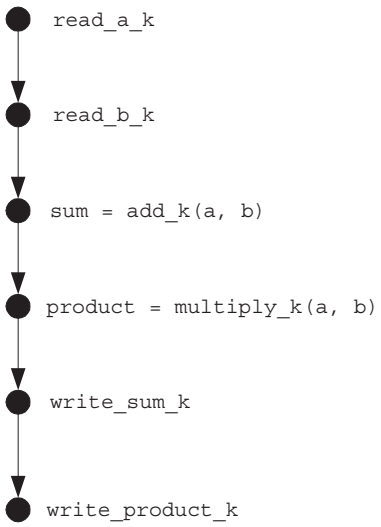

Figure 2.7: An ordering of operations in `AddMultiply.fire`.

If the firing set is totally ordered, we also write the firing set using a conventional imperative language syntax (with ";" denoting sequencing of operations). Noticing that the firing sets are essentially the same for all triggers, we omit the indexing on operations, and have,

```
AddMultiply.fire = {
                     read_a;
                     read_b;
                     sum = add(a, b);
                     product = multiply(a, b);
                     write_sum;
                     write_product;
                   }
```

The total order sometimes over-specifies the relations among operations. Suppose that the `AddMultiply` is implemented using hardware; forcing the operations to be totally ordered may not be the best choice. As depicted in Figure 2.8, the (transitive closure of the) following ordering relations, denoted by $\prec'_{\texttt{AddMultiply}}$, exhibit the minimum causality constraints among operations in `AddMultiply`. Any ordering relations of these operations that refine $\prec'_{\texttt{AddMultiply}}$ is a valid implementation. This specification allows many parallel implementation choices.

$$
\begin{aligned}
\texttt{read\_a\_k} &\prec \texttt{sum} = \texttt{add\_k}(\texttt{a}, \texttt{b}) \\
\texttt{read\_b\_k} &\prec \texttt{sum} = \texttt{add\_k}(\texttt{a}, \texttt{b}) \\
\texttt{read\_a\_k} &\prec \texttt{product} = \texttt{multiply\_k}(\texttt{a}, \texttt{b}) \\
\texttt{read\_b\_k} &\prec \texttt{product} = \texttt{multiply\_k}(\texttt{a}, \texttt{b}) \\
\texttt{sum} = \texttt{add\_k}(\texttt{a}, \texttt{b}) &\prec \texttt{write\_sum\_k} \\
\texttt{product} = \texttt{multiply\_k}(\texttt{a}, \texttt{b}) &\prec \texttt{write\_product\_k}
\end{aligned}
$$

A firing set of an actor by no means has to be finite. For example, an actor, `InfiniteAddMultiply`, with the same variable and operation sets as `AddMultiply`, may

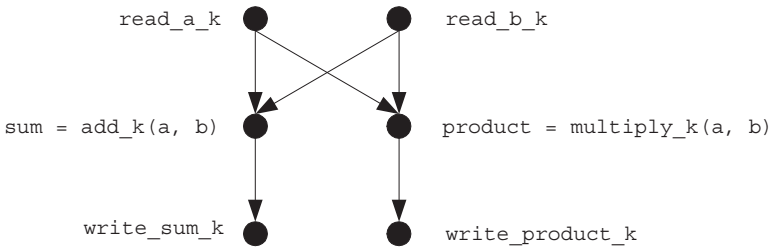

Figure 2.8: Another ordering of operations in `AddMultiply.fire`.

have a firing set like:

$$\begin{aligned}
&\texttt{InfiniteAddMultiply.fire}|_{\texttt{trigger_0}} = \{\\
&\qquad\qquad \texttt{for}(\texttt{k} = 0; \texttt{true}; \texttt{k}++)\ \{\\
&\qquad\qquad\quad \texttt{read_a_k};\\
&\qquad\qquad\quad \texttt{read_b_k};\\
&\qquad\qquad\quad \texttt{sum} = \texttt{add_k}(\texttt{a}, \texttt{b});\\
&\qquad\qquad\quad \texttt{product} = \texttt{multiply_k}(\texttt{a}, \texttt{b});\\
&\qquad\qquad\quad \texttt{write_sum_k};\\
&\qquad\qquad\quad \texttt{write_product_k};\\
&\qquad\qquad \}\\
&\qquad \}
\end{aligned}$$

So, `InfiniteAddMultiply` is not a reactor.

The execution of a firing set starts with a trigger. If an actor is reactive and the execution of the firing set completes, the reactor sends a `finish` callback to the framework. However, depending on the status of the framework, a firing may not always be completed after it is triggered. If it does complete, we call the firing set, together with its trigger and finish operations, a complete reaction.

Definition 2.6. *A* (***complete***) ***reaction*** *of an actor A with respect to a trigger $r \in M.\texttt{Ctrl}_A$ and a firing set $\texttt{A.fire}|_{\texttt{r}}$ is a partially ordered set $\overline{\texttt{A.fire}|_{\texttt{r}}} = \{r\} \cup \texttt{A.fire}|_{\texttt{r}} \cup \{\texttt{finish\_r}\}$ satisfying,*

A. $\texttt{A.fire}|_{\texttt{r}}$ *is reactive;*

B. *for* $f, g \in \texttt{A.fire}|_{\texttt{r}}$, $f \prec g$ *in* $\texttt{A.fire}|_{\texttt{r}}$ *if and only if* $f \prec g$ *in* $\overline{\texttt{A.fire}|_{\texttt{r}}}$;

C. $\forall f \in \texttt{A.fire}|_{\texttt{r}}, r \prec f$;

D. $\forall f \in \texttt{A.fire}|_{\texttt{r}}, f \prec \texttt{finish\_r}$.

Since neither r nor `finish_r` belongs to $\texttt{A.fire}|_{\texttt{r}}$, adding conditions C and D in definition 2.6 does not introduce conflicts with condition B, and thus $\overline{\texttt{A.fire}|_{\texttt{r}}}$ is indeed a poset. The partial order relation defined in $\overline{\texttt{A.fire}|_{\texttt{r}}}$ is denoted by $\overline{\prec_{A|_r}}$. For the short of notation, for any operation $f \in \overline{\texttt{A.fire}|_{\texttt{r}}}$, we also write $\bar{f}$ for the trigger r, and $\underline{f}$ for the corresponding finish operation `finish_r`.

From the ordering defined in Definition 2.6, a complete reaction always has a shape as in Figure 2.9, where all execution starts with the `trigger`, and stop at the `finish`. The operations `trigger` and `finish` are synchronization points of $\overline{\texttt{A.fire}|_r}$.

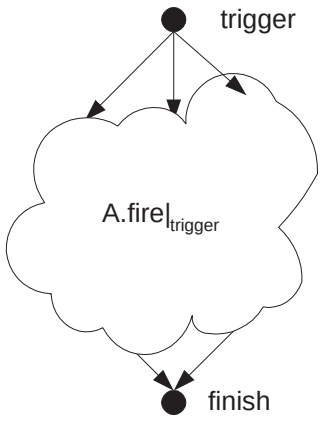

Figure 2.9: The shape of a complete reaction.

Whether a reaction can complete within a framework depends on the semantics of

communication and the triggering rules.

2.2.2 Communications

Communication Semantics

A framework provides communication semantics for read and write operations. The semantics are achieved by the help of framework variables, which essentially record the communication states.

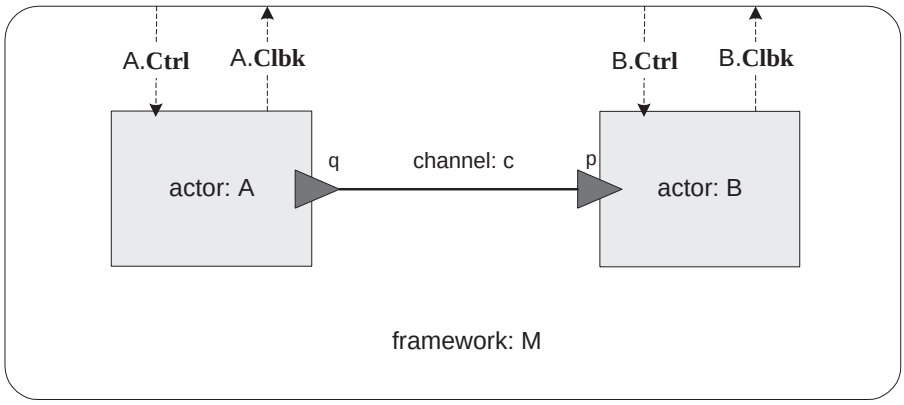

Figure 2.10: Composing two actors.

For example, suppose a channel $c = (q \rightsquigarrow p)$ in Figure 2.10, (which is essentially a redraw of Figure 2.3,) implements a shared memory, such that a write operation performed on port q overrides the old value in the memory, and the reader always reads the latest value. Then, we need one framework variable, $Z_c = \{z\}$ for the channel. Suppose $\sigma(q) = v$ when `write_q` is performed, then the result of the operation is $\sigma(z) = v$. If a read operation `read_p` is performed before any other write operations on q, then $\sigma(p) = v$ after the reading.

For another example, suppose $c = (q \rightsquigarrow p)$ in Figure 2.10 implements a first-in-

first-out (FIFO) queue of size K, then we have $Z_c = \{z[0], z[1], z[2], ..., z[K-1]\}$. The FIFO queue semantics is enforced by the following partial functions:

- `write_q`$(\sigma) = \sigma'$: if $k = 0$ and $\sigma(z[0]) = \perp$, or, $k > 0$, $\sigma(z[k]) = \perp$, and $\sigma(z[k-1]) \neq \perp$, and $\sigma(q) = v$, then $\sigma'(z[k]) = v$, where $\sigma'(z[k])$ is the evaluation of $z[k]$ after the write operation is performed;

- `read_p`$(\sigma) = \sigma'$: if $\sigma(z[0]) = v \neq \perp$, then $\sigma'(p) = v$, and for $k > 0, \sigma'(z[k-1]) = \sigma(z[k])$, and $\sigma'(z[K]) = \perp$. That is, the values inside the queue are shifted to the front.

Ideally, this should implement a queue, such that for $0 < k < K$, if $\sigma(z[k]) \neq \perp$, then $\sigma(z[k-1]) \neq \perp$. Obviously, extra constraints need to be imposed on the order of read and write operations so that the queue does not overflow or underflow.

Communication Orders

The constraints on communication may be imposed by a framework in forms of partial order relations on the read and write operations. Suppose in Figure 2.10, $Z_c = \{z[0], z[1], z[2]\}$ implements a FIFO queue of size three, actor A writes three values in a row when it is triggered, and actor B reads one value a time when it is triggered. Their firing sets are shown in Table 2.1 and Table 2.2.

Then, in order for the communication channel to behave like a FIFO queue without overflow or underflow, the k^{th} writing to port q should be performed earlier than the k^{th} reading from port p, and the $k + 3^{rd}$ writing to port q should be later than the k^{th} reading

Table 2.1: The firing set of an actor that performs three write operations when fired.

```
A.fire|_r_i = {
                q = value;
                write_q_(3(i − 1) + 1);
                write_q_(3(i − 1) + 2);
                write_q_(3(i − 1) + 3);
              }
```

Table 2.2: The firing set of an actor that performs one read operations when fired.

```
B.fire|_r'_j = {
                 read_p_j;
               }
```

from port p. This can be expressed as the following partial order relations: $\forall k \in \mathbb{N}$,

$$A.\texttt{write\_q\_k} \prec B.\texttt{read\_p\_k} \tag{2.4}$$

$$B.\texttt{read\_p\_k} \prec A.\texttt{write\_q\_(k + 3)}. \tag{2.5}$$

Obviously, we need cooperations between the framework and the actors to satisfy these communication orders. The contributions of the framework are the triggers, and the contributions of actors are the contents of their firing sets. The cooperation is reflected in triggering rules.

2.2.3 Triggers

A framework can only issue triggers based on its own activities and the observable activities of actors, which are their communication operations and callback operations. A

trigger may also depend on the states of the framework, (but not the states of actors), which can be expressed as a predicate on framework variables. In general, a trigger r may be *conditionally activated* by an operation and a predicates. A *triggering rule* for r is a pair $(g, \rho(\sigma_Z))$, also written as:

$$g \xrightarrow{\rho(\sigma_Z)} r, \tag{2.6}$$

where,

$$g \in M.\mathbf{Ctrl} \cup \left\{ \bigcup_{A \in M} (A.\mathbf{Comm} \cup A.\mathbf{Clbk}) \right\},$$

is an operation that is observable by the framework M; and $\rho(\sigma_Z)$ is a predicate on the values of variables in Z. The interpretation is that after the operation g is performed, the predicate $\rho(\sigma_Z)$ will be evaluated on the current state of the framework. If the evaluation is `true`, then the trigger r is performed immediately, i.e. $g \triangleleft r$.

Notice that, a trigger may be activated by more than one rule. In this case, we write:

$$r \triangleq \{(g_1, \rho_1(\sigma_Z)), (g_2, \rho_2(\sigma_Z)), ...\}.$$

If any one of these rules is satisfied, then r is activated. Also notice that an operation g may activate multiple triggers. These triggers must be incomparable, such that for any r among them, $g \triangleleft r$. In essence, triggering rules define a set of (conditioned) ordering relations $\prec_R$ on operations.

Sometimes, an singleton initial operation `Init` is needed for a framework to start all the activities in it. This initial operation is a framework operation, and actors may built their triggering rules using it. For example, if an actor A is triggered at the beginning of

the execution without any preconditions, then the actor may have a triggering rule:

$$\texttt{Init} \xrightarrow{\texttt{true}} A.\texttt{trigger\_1}.$$

Static Scheduling

In some models, the triggers are simply unconditioned on `finish` callbacks from another reactor. That is, the triggering rules may look like:

$$\texttt{A.finish\_j} \xrightarrow{\texttt{true}} \texttt{B.trigger\_i}$$

for some actor $A, B \Subset M$, (possibly $A = B$) and some indices i and j. If all triggering rules in the framework have this form, then the framework is called *statically scheduled*.

A static schedule may be sequential, which can be represented as a list of reactors, $\textbf{SequentialSchedule} = \{A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow ...\}$. The meaning of this list is the following triggering rules:

$$\texttt{Init} \xrightarrow{\texttt{true}} \texttt{A}_1\texttt{.trigger}$$

$$\texttt{A}_\texttt{i}\texttt{.finish} \xrightarrow{\texttt{true}} \texttt{A}_{\texttt{i}+1}\texttt{.trigger}$$

If actors are repeated in this list, then corresponding indices can be added to trigger and finish operations.

If there are multiple lists of sequential schedules in parallel, or a finish operation can activate multiple triggers, then a static schedule can have more complex structures. In these cases, a list is not sufficient for representing the schedule. And we will keep the triggering rule representations.

2.3 Precise Reactions and Responsible Frameworks

2.3.1 Composite Execution

Summarizing the discussions in the last section, for a composite $\Theta = (M, \mathbf{A}, \mathbf{C})$, there are the following sets of partial ordering relations that constrain the operations within it:

- $\prec_0$: operation index orders;
- $\overline{\prec_{A|_r}}$: reaction orders, imposed by firing sets and reactions of actors;
- $\prec_M$: communication orders, imposed by frameworks
- $\prec_R$: triggering rules, imposed by frameworks in cooperation with actors.

The execution of a composite Θ is *well-defined* if all of the above partial ordering relations are compatible, i.e. there exist a maximum ground set and a common refinement that refines all the partial order relations. In this case, we say that the actors are *compatible* with the framework, and define the firing set of Θ to be the maximum subset of the union of reactor and framework operations.

More precisely, we define an *execution* of Θ be,

$$\texttt{0.exec} \subseteq \bigcup_{r \in R, A \in M} \left\{ \overline{\texttt{A.fire}|_{\texttt{r}}} \right\}$$

satisfying:

[1.] Ordering relations: $\prec =$ **Closure** $\left(\prec_0 \cup \prec_M \cup \prec_R \bigcup_{r \in R, A \in M} \{\overline{\prec_{A|_r}}\} \right)$

[2.] Trimming rules: if $f \prec g$ and $f \notin \texttt{0.exec}$, then $g \notin \texttt{0.exec}$

where R is the set of all triggers activated by the triggering rules during the execution. We denote by Θ.EXEC the set for all executions of Θ. Since all elements in Θ.EXEC satisfy the above two rules, the union of any two elements also belongs to Θ.EXEC. We define the *firing set* of Θ, Θ.fire, to be the union of all elements in Θ.EXEX. That is, Θ.fire $\in$ Θ.EXEC, and if $exec \in \Theta$.EXEC then $exec \subseteq \Theta$.fire.

If the four sets of ordering relations introduces conflicts, then the set of actors is not compatible with the framework, and the firing of Θ is not well-defined. In fact, for an incompatible composition of actors and a framework, there may not exist a unique way to remove a subset from the union of operations to make the firing set a poset. That is, it may not be precisely defined by the set of ordering relations which element is in the firing set and which element is not.

Even when the composite execution is well-defined, it is not necessarily true that a reaction of an individual reactor is completed in the composite execution. An incomplete reaction forces the designer to understand where exactly the reaction stops within a reaction, and how to resume the reaction later. This violation of information hiding not only brings difficulty in the understandability of a model, but also destroys the compositionality of actors and models of computation.

For example, suppose we have the following triggering rules

$$\textbf{Schedule} = \{A \rightarrow B \rightarrow B \rightarrow A\} \tag{2.7}$$

for the system shown in Figure 2.10, where the actor A performs three writes in its firing set and actor B performs one read in its firing set, as shown in Table 2.1 and Table 2.2. Also, suppose that we have a FIFO queue communication with buffer size 3, and use the

communication constraints in (2.4).

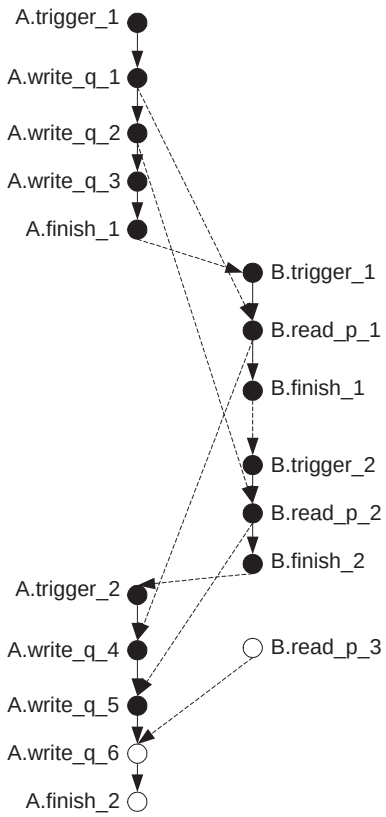

Figure 2.11: An execution trace of the example.

Then, the poset of oeprations in this composite can be represented by Figure 2.11. In the figure, the solid arrows shows the ordering relations imposed by the firing sets of actor A and B, and the dashed arrows show the ordering relations imposed by the framework, including triggering rules and communication constraints. The circles represent some operations that are not performed under this particular schedule. It is obvious that actor A cannot finish its second reaction, since `B.read_p_3` is not performed. Unless we know exactly the content of actor A, it is imposible to know where exactly the execution stops. For this reason, we define precise reactions as a property for composable reactivities.

2.3.2 Precise Reactions

A reaction is activated by a trigger. If a trigger is "smart" enough to summarize all the prerequisites for an actor to complete its reaction, then the reaction can always be completed in the composite execution.

Definition 2.7. *For a composite* $\Theta = (M, \mathbf{A}, \mathbf{C})$*, actor* $A \in \mathbf{A}$*, and a trigger* $r \in \texttt{0.fire}$*, the reaction* $\overline{\texttt{A.fire}|_{\texttt{r}}}$ *is* ***precise*** *if* $\forall f \in \overline{\texttt{A.fire}|_{\texttt{r}}}$*,* $\forall g \notin \overline{\texttt{A.fire}|_{\texttt{r}}}$*,* $g \prec f \Rightarrow g \prec r$*. A trigger for a precise reaction is called a* ***responsible*** *trigger.*

Definition 2.8. *The state of an actor* A*, when the* `finish` *operation is performed, is called a* ***quiescent state*** *of* A*. By definition, the state of* A *is also quiescent if it has never been triggered.*

Figure 2.12 illustrates the shape of a precise reaction activated by a responsible trigger.

A responsible trigger guarantees that all the preconditions for an actor to finish the firing have been satisfied. In terms of the partial ordering relations among operations, this means that for any $f \in \overline{\texttt{A.fire}|_{\texttt{r}}}$, $g \notin \overline{\texttt{A.fire}|_{\texttt{r}}}$, and $g \prec f$, adding the relation $g \prec r$ does not create any conflicts in the execution of the composite. The concept of precise reaction guarantees that a reaction, once triggered, can be finished within the composite execution. And when it finishes, it is at a quiescent state. So, we have,

Proposition 2.2. *For a composite* $\Theta = (M, \mathbf{A}, \mathbf{C})$ *and actor* $A \in \mathbf{A}$*, if* $r \in \texttt{0.fire}$ *is a responsible trigger, then* $\overline{\texttt{A.fire}|_{\texttt{r}}} \subseteq \texttt{0.fire}$.

The proof is straightforward by the definitions of precise reaction and responsible

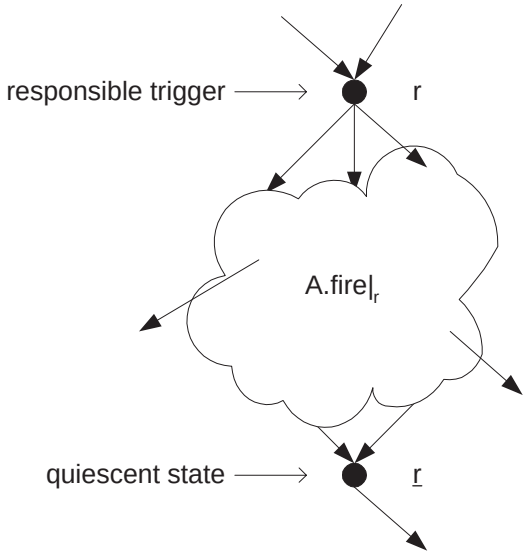

Figure 2.12: The shape of a precise reaction. Note that there can be causality relations pointing out of the firing set, but all the prerequisites of the firing are summarized by the trigger.

trigger. By these definitions, in our previous example, $\overline{\texttt{B.fire}|_{\texttt{trigger\_1}}}$ and $\overline{\texttt{B.fire}|_{\texttt{trigger\_2}}}$ are precise reactions, but $\overline{\texttt{A.fire}|_{\texttt{trigger\_2}}}$ is not. The actor A, at the end of the composite execution is not at a quiescent state.

2.3.3 Responsible Frameworks

Since triggers are operations of a framework, the responsibleness of triggers reflects certain properties of frameworks. We define,

Definition 2.9. *A framework is **responsible** if it requires all triggering rules to be responsible and it respects these triggering rules.*

That is, the execution within a responsible framework consists solely of precise reactions, as shown in Figure 2.13.

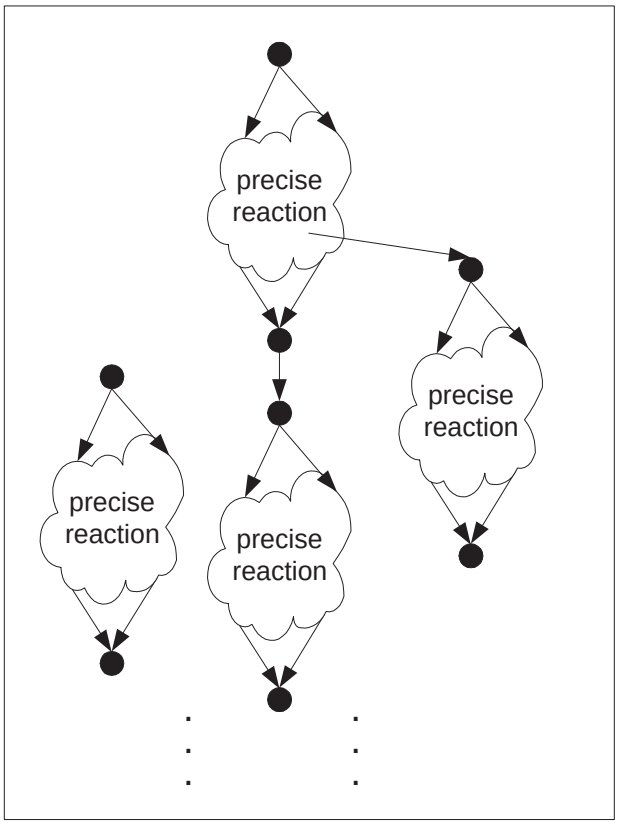

Figure 2.13: The execution within a responsible framework consists only of precise reactions.

Responsible frameworks gives many useful properties. For example, let Θ be a composite with a responsible framework and precise reactors. We have the following properties,

Properties:

[1.] Whenever the framework stops sending triggers, all reactors will settle at their quiescent states within finite operations.

[2.] Θ is deadlocked only if the framework cannot send more triggers. So, deadlocks can be detected by monitoring triggering rules.

[3.] Even when Θ deadlocks, all reactors are at their quiescent states.

The proofs are all straightforward. So, a responsible framework can easily control the progress of execution by sending and holding triggers. And it can detect deadlocks by monitoring triggers.

2.3.4 Atomicity

For a composition of reactors, there is a stronger notion than precise reactions, called atomic reactions.

Definition 2.10. *A reaction* $\overline{\texttt{A.fire}|_{\texttt{r}}}$ *with callback* $\underline{r}$[1] *is* ***atomic****, if it is precise, and* $\forall f \in \overline{\texttt{A.fire}|_{\texttt{r}}}$, $\forall g \notin \overline{\texttt{A.fire}|_{\texttt{r}}}$, $f \prec g \Rightarrow \underline{r} \prec g$.

So, the atomicity of a reaction states that not only all the prerequisites have been satisfied before the reaction, but also, once the reaction is started, all other activities within the composite can wait until the reaction has finished. We will see that, for responsible frameworks, precise reactions are compatible with atomic reactions.

Theorem 2.1. *Under a responsible framework, precise reactions are compatible with atomic reactions.*

Proof. Let $\Theta = (M, \mathbf{A}, \mathbf{C})$ be a composite with a responsible framework M. Let $A \in \mathbf{A}$, r be a (responsible) trigger for $\overline{\texttt{A.fire}|_{\texttt{r}}}$, and $\exists f \in \overline{\texttt{A.fire}|_{\texttt{r}}}$, $g \in \mathbf{Oper}$, and $g \notin \overline{\texttt{A.fire}|_{\texttt{r}}}$, s.t. $f \prec g$. We want to show that adding $\underline{r} \prec g$ does not create any conflicts in the execution of Θ.

Since all the activities in a responsible framework are precise reactions, and $f \prec g$, there must exist $A' \in \mathbf{A}$ and r', s.t. $g \in \overline{\texttt{A'.fire}|_{\texttt{r'}}}$. By the definition of firing sets, it is

[1] Recall that $\underline{r}$ is the `finish` operation for reactor $\overline{\texttt{A.fire}|_{\texttt{r}}}$ corresponding to trigger r.

sufficient to show that adding $\underline{r} \prec r'$ does not create any conflicts. Suppose to the contrary this is not true, then $\exists h \in \mathbf{Oper}$ s.t. $r' \prec h \prec \underline{r}$, as shown in Figure 2.14. Since the framework M is responsible, $h \prec \underline{r} \Rightarrow h \prec r \Rightarrow h \prec f$. But r' is responsible, and $f \prec g$, so $f \prec r' \prec h$. Thus, the existence of h creates a conflict in **Oper**. By contradiction , $\underline{r} \prec g$ should not create any conflict. □

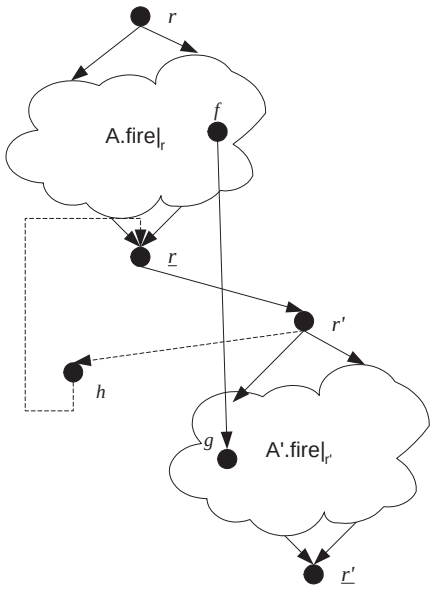

Figure 2.14: Proof of Theorem 2.1. Adding operation h creates a contradiction.

During the proof of Theorem 2.1, we actually showed a slightly stronger result, which is that we can always start triggering a reactor after the finish of another reaction. So, we have,

Corollary 2.1. *The execution of reactors in a responsible framework can be sequentialized.*

The sequentializable execution makes responsible frameworks very easy to understand, in the sense that a complete reaction can be abstracted into one operation without

affecting the overall execution of a composite. This property also provides a semantics foundation for information hiding and component refinements in component-based designs. An abstract atomic operation at a high level can be refined into a composite execution as long as the composite execution is a precise reaction that has the same prerequisites (i.e. triggering rules) as the atomic operation.

2.4 Examples of Untimed Frameworks

In this section, we give some examples of frameworks. These frameworks are called *untimed* to distinguish themselves from *timed frameworks*, which have a notion of time. We will discuss timed frameworks in Chapter 4.

2.4.1 Communicating Sequential Processes (CSP) Frameworks

In the CSP model of computation [32], each component is a *process*. A process is a conceptually unbounded sequence of operations. The communications between processes are *atomic* exchange of data, called *rendezvous*. We consider a simple CSP model, where each rendezvous only involves two processes.

In theory, a CSP framework, `CSP`, can have actors with an unbounded firing set, like the `InfiniteAddMultiply` actor in section 2.2. A reactor A can be made into an infinite sequential process, if the firing set $\texttt{A.fire}|_{\texttt{r}}$ has totally ordered operations w.r.t. all triggers, and the framework adopts the following triggering rules for all $A \Subset M$:

$$\text{initial trigger:} \qquad \texttt{Init} \xrightarrow{\texttt{true}} \texttt{A.trigger\_1} \tag{2.8}$$

$$\text{self-trigger:} \qquad \texttt{A.finish\_k} \xrightarrow{\texttt{true}} \texttt{A.trigger\_(k + 1)}, \text{for } k > 1. \tag{2.9}$$

Thus, a CSP framework triggers a reactor whenever its last reaction is finished, regardless whether the new reaction can be completed. In this sense, CSP frameworks are not responsible.

The communication on each channel has a shared variable semantics. That is, for each communication channel $c = (q \rightsquigarrow p), q \in A.Q, p \in B.P$, there is one framework variable z_c, such that `write_q` will assign $\sigma(q)$ to z_c, and `read_p` will assign $\sigma(z_c)$ to p. The *rendezvous* style of communication also requires ordering relations on corresponding read and write operations. Suppose actors A and B in Figure 2.3 communicate via atomic *rendezvous*. Then the framework imposes the following constraints to implement atomic *rendezvous*: for $k \in \mathbb{N}$,

- $A.\texttt{write\_q\_k} \triangleleft B.\texttt{read\_p\_k}$;
- $\forall A.f \in A.\mathbf{Oper}, (A.f \prec A.\texttt{write\_q\_k}) \Rightarrow (A.f \prec B.\texttt{read\_p\_k}), \quad (A.\texttt{write\_q\_k} \prec A.f) \Rightarrow (B.\texttt{read\_p\_k} \prec A.f)$.
- $\forall B.g \in B.\mathbf{Oper}, (B.g \prec B.\texttt{read\_p\_k}) \Rightarrow (B.g \prec A.\texttt{write\_q\_k}), \quad (B.\texttt{read\_p\_k} \prec B.g) \Rightarrow (A.\texttt{write\_q\_k} \prec B.g)$.

Visually, Figure 2.15 shows a communication section in CSP models.

In summary, for a framework **CSP** with actors **A** and connections **C**, we have the following ordering relations:

- Initialization: $\forall A \in \mathbf{A}$, $\texttt{Init} \xrightarrow{\texttt{true}} A.\texttt{trigger\_1}$;
- Sequential processes: $\forall A \in \mathbf{A}, \forall f, g \in A.\mathbf{Oper}$, either $f \prec g$, or $g \prec f$;
- Self-triggering rules: $\texttt{A.finish\_k} \xrightarrow{\texttt{true}} \texttt{A.trigger\_(k + 1)}$, for $k > 1$;

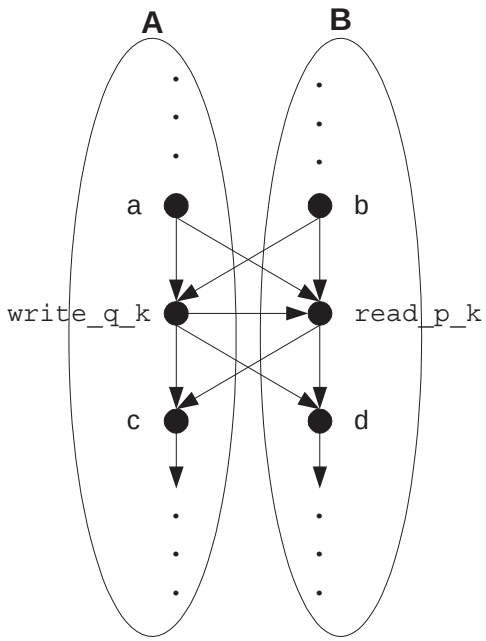

Figure 2.15: Ordering relation of rendezvous communication.

- Rendezvous communication: as shown above.

A CSP composite may deadlock, depending on the connections of ports and ordering relations inside each actor. And when it deadlocks, the actors may not at their quiescent states. This deadlock is identified by observing conflicts when composing partial orders. I.e. $\exists f, g \in \mathbf{Oper}$, s.t. $f \prec g$ and $g \prec f$.

2.4.2 Process Network (PN) Frameworks

In a process network model of computation [39], as in CSP models, components are processes. But, unlike CSP models, the communication style on each channel has a FIFO queue semantics. That is, for each channel $c = (q \rightsquigarrow p)$, there is a (potentially infinite) set of framework variables $Z_c = \{z_i, i \in \mathbb{N}\}$. The k^{th} writing to port q will assign the value to

the k^{th} variable in Z_c; the k^{th} reading from port q will return the value of the k^{th} variable in Z_c. The communication requires that writing to a variable preceeds, but not necessarily immediately preceeds, the corresponding reading from it, as shown in Figure 2.16.

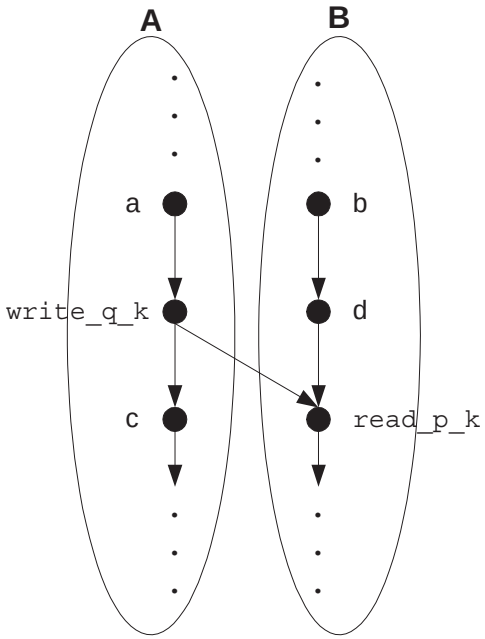

Figure 2.16: Ordering relation of FIFO queue communications.

Together with the irresponsible triggering strategy shown in (2.8) and (2.9), we have the following ordering relations for a PN framework `PN` with actors $\mathbf{A}$ and connections $\mathbf{C}$:

- Initialization: $\forall A \in \mathbf{A}$, $\texttt{Init} \xrightarrow{\texttt{true}} A.\texttt{trigger\_1}$;
- Sequential processes: $\forall A \in \mathbf{A}, \forall f, g \in A.\mathbf{Oper}$, either $f \prec g$, or $g \prec f$;
- Self-triggering rules: $\texttt{A.finish\_k} \xrightarrow{\texttt{true}} \texttt{A.trigger\_(k + 1)}$, for $k > 1$;
- FIFO communication: for $A, B \in \mathbf{A}$, $q \in A.Q, p \in B.P$, and $c = (q \rightsquigarrow p) \in \mathbf{C}$,

$A.\texttt{write\_q\_k} \prec B.\texttt{read\_p\_k}, \forall k \in \mathbb{N}.$

2.4.3 Dataflow (DF) Frameworks

In dataflow with firing [46], components are reactors with finite firing sets, and the communication channels are FIFO queues. A reactor A may have one or more firing rules, each having the form of a predicate on framework variables associated with the channels connected to the input ports of the actor, i.e. $\rho(Z_{(q \rightsquigarrow p)})$, for $p \in A.P$ and any port q. These firing rules precisely specify the requirements for the actor to finish the execution of one firing. That is, if the one the rules is satisfied, and the actor is triggered accordingly, then the actor can always execute to the `finish` of the firing set. Thus, DF frameworks are responsible.

For example, Figure 2.17 shows a `Select` actor, with ports `input0`, `input1`, `control`, and `output`. For the simplicity of representation, we call the channel connected to `input0` the channel *input*0, and so on. Depending on the value, `true` or `false`, of the first variable in the *control* channel, the actor transfers the first value of the *input*1 or *input*0 to the *output* channel.

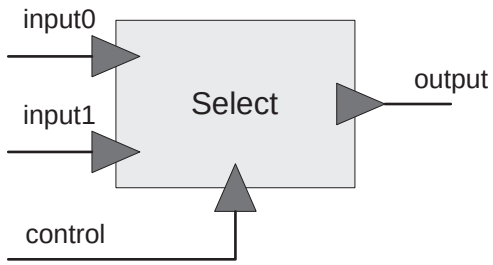

Figure 2.17: A `Select` actor in dataflow models.

We also assume that the framework shifts the contents in the variables to the front when the first variable is read by the consumer of the channel. The firing rules in this case is

$$\rho_0 : (\sigma(z_{control}) == \texttt{false}) \wedge (\sigma(z_{input0}) \neq \perp) \tag{2.10}$$

$$\rho_1 : (\sigma(z_{control}) == \texttt{true}) \wedge (\sigma(z_{input1}) \neq \perp) \tag{2.11}$$

where, $z_{control}$ is the first variable in channel *control*, z_{input0} is the first variable in channel *input*0, and z_{input1} is the first variable in channel *input*1. These firing rules map cleanly to the triggering rules in the reactor model. For example,

$$g \xrightarrow{\rho_0} r_F$$

$$g \xrightarrow{\rho_1} r_T$$

where ρ_0 and ρ_1 are the predicates in (2.10) and (2.11), and g is any `write` or `finish` operation observed by the framework. The corresponding firing sets (omitting the indices) are:

```
Select.fire|rF = {
                    read_control;
                    read_input0;
                    output = σ(input0);
                    write_output;
                }
```

and,

```
Select.fire|rT = {
                    read_control;
                    read_input1;
                    output = σ(input1);
                    write_output;
                }
```

2.4.4 Synchronous Dataflow (SDF) Framework

The synchronous dataflow model [45] is a special case of dataflow models. The actors in this model are so regular that the number of reads and writes in each firing is fixed and known for all triggers. For example, the `AddMultiply` actor is a SDF actor, but the `Select` actor is not.

Because of this regularity, each SDF actor only have one firing rule, and the firing rule depends totally on the non-emptiness of framework variables. As a consequence, a SDF composite can be statically scheduled, such that the triggering of one actor only depends on the finishing of the actor proceeds it in the schedule [45]. We have met a SDF model in section 2.2.2 with two actor, one performing three writes in its firing and one performing one read in its firing. In general, for a SDF composite $\Theta_{SDF} = (\texttt{SDF}, \mathbf{A}, \mathbf{C})$, a sequential schedule of a SDF model[2] is $\mathbf{SDFSchedule} = \{A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow ... \rightarrow A_n\}$, where A_i refer to an actor in $\mathbf{A}$. Note that depending on the number of reads and writes for each actor, an actor may appear multiple times in $\mathbf{S}$. The sequence $\mathbf{S}$ is called one *iteration* of an SDF model. For each iteration, the triggering rule of the SDF model may look like:

$$A_i.\texttt{finish} \xrightarrow{\texttt{true}} A_{i+1}.\texttt{trigger}$$

And this rule simply repeats for more iterations.

[2]For simplicity, we assume the firing of actors is sequentialized. In general, the schedules are partially ordered sets rather than sequences. For multiple CPU scenarios, the partial ordering can be exploited to introduce parallelism in the execution.

2.5 Implementation

In this section, we briefly discuss the implementation of the reactor model in the Ptolemy II software environment. In Ptolemy II, each model of computation is implemented as a domain, and a director controls the component interaction in that domain. Ptolemy II models can be hierarchical, where different models of computations are nested through composite actors. However, the reactor model we discussed in this chapter does not support hierarchy yet. We will study hierarchical frameworks in Chapter 3. Thus, the implementation discussed in this section only covers "flat" models in Ptolemy II.

In Ptolemy II, the basic building blocks are atomic actors, which are reactors that have totally ordered firing sets. Actors have ports, which can connect to other ports through relations. Relations do not have semantic properties other than keeping track of connections. The communication mechanisms among ports are provided by directors and implemented as receivers. In the Ptolemy II model, receivers always reside in input ports. Receivers may implement rendezvous points, FIFO queues, buffers, or proxies to a global queue.

Actors are executable. As shown in Figure 2.18, there are seven methods in the Executable interface:

- `preinitialize`(): performs structural and pre-type-resolution initialization. This method is called exactly once at the beginning of an execution.
- `initialize`(): performs scheduling- and type-dependent initialization. This method is called once at the beginning of an execution. It may be called again to reset an actor to its initial state.

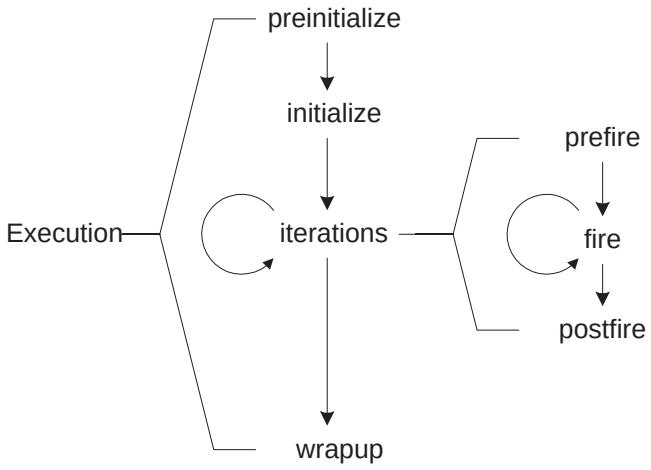

Figure 2.18: Ptolemy II execution.

- `prefire`(): returns true if the fire() method can successfully finish.
- `fire`(): performs the fire function without updating persistent states.
- `postfire`(): updates states, and return true if the actor can be further fired. This method is designed for models that use fixed-point iterations.
- `wrapup`(): releases resource and wrap up. This method is called exactly once at the end of an execution.
- `stopFire`(): interrupts the firing and requests that the actor to return the flow of control to the director.

The reactor model captures the structural and execution properties of Ptolemy II models in the following senses:

- *Ptolemy actors are reactors.* In particular, they are reactors that have at least two firing sets: `fire`() and `postfire`(). In Ptolemy II, ports are objects instead of variables,

so that they can contain receivers and keep track of connections. [3]

- *Directors, Receivers, and Relations implement a framework.* Receivers are framework variables, and their evaluations and assignments are performed through `get`() and `put`() methods. Relations capture the connections among ports. Directors issue triggers to actors, by calling their execution methods.

- *The* `prefire`() *method helps directors to provide precise triggers.* In Ptolemy II, the `fire`() and `postfire`() methods are called only if the `prefire`() method returns `true`, i.e. the actor can successfully proceed for one iteration. Responsible frameworks can use these methods to achieve responsible triggers. There are also ways to hide these preconditions for irresponsible frameworks. Typically, the hiding is performed by receivers that may always inform an actor that there are enough input data.

2.6 Related Work

A distinctive semantic characteristic of embedded software is reactivity. A pioneer work in this area is the study of reactive systems in the formalism of statecharts [27], [28] and its variants [78]. The statecharts model uses hierarchical finite state machines to formulate reactive systems. The transitions among states are triggered by input events, and transitions are always atomic. For every triggering event, the system has exactly one transition to take, from a well-defined state to another. Although the statecharts model is a big step of improvement from traditional flat state machines, in terms of hierarchy and

[3] Ptolemy II ports can be multiports, meaning that one port can contain many channels. From the reactor model point of view, they are simply many ports.

some concurrency, it lacks the notion of abstraction to handle large-scale designs and may not be intuitive to model computational intensive systems.

Synchronous languages, like Esterel [7], Lustre [26], Signal [25], and Argos [54], make the synchrony assumption in the modeling of reactive systems. The synchrony assumption states that all reactions take no time to execute. This extreme abstraction allows designers to separate functional properties from timing concerns. However, it also brings semantic subtitles like zero-delayed feedback loops, which requires the introduction of "non-strict" components[4] and fixed-point semantics to make a model well-defined. The reactive modules formalism [2] also takes the synchronous assumption of reactivity, and focuses on nondeterminism and verifiability. One advantage of synchronous languages is that they can be compiled into sequential programs, such that the concurrency at the modeling level are compiled away. On the other side, it introduces difficulties when used in distributed systems. Although many work has been done on distributing synchronous models [6], using these models at large-scale multitasking systems remains challenging.

The reactor model presented in this chapter is greatly influenced by event structure [82] and the dataflow process model [46]. Winskel's event structure inspires the idea of using partially ordered sets and precedence relations to capture concurrent actions in a system. The dependency-based action refinements [62] implies the feasibility of modeling compositional action semantics using event structures. In a sense, a precise reaction is refined to a composition of individual actions in a firing set. The dataflow model with firing inspires me to study the significance of firing rules. In dataflow models, like SDF [45], DDF [12], and the Moses' actor model [36], the firing rules are restricted to be patterns in

[4] A non-strict component does not require all the inputs to be present to produce outputs.

terms of the presence of data. As will be seen in later chapters, we relaxed this restriction in reactor models to include time and physical events which are closer to the interaction with the real world.

Chapter 3

Compositional Precise Reaction

As discussed in the last chapter, reactors and frameworks can implement models of computation. Hierarchical heterogeneity requires that a composite itself be an actor and be controlled by a higher-level framework. This chapter claims that an open composite is an actor but compositional reactions may not always be precise. Responsible frameworks help reactivize composite actors.

3.1 Composite Actor

Recall that a *composite*, $\mathbf{\Theta} = (M, \mathbf{A}, \mathbf{C})$, is an aggregation of a framework M, the actors $\mathbf{A}$ controlled by the framework, and the connections $\mathbf{C}$ among the actors. Abstracting the activities of actors and frameworks into partially ordered operations allow us to treat a composite as an actor.

3.1.1 Open Composite

An *open composite* $\mathbf{\Omega}$ extends a composite $\mathbf{\Theta} = (M, \mathbf{A}, \mathbf{C})$ with ports. These ports are called *composite ports*, which are variables of $\mathbf{\Omega}$. These ports also introduce additional connections, which require more framework variables for the additional communication channels. A framework within an open composite is called an *open framework*.

Figure 3.1 shows an open composite Ω with two components A and B. It has an open framework O, three composite ports, p_1', p_2', and q_2', four channels, $c_1 = (p_1' \rightsquigarrow p_1)$, $c_2 = (p_2' \rightsquigarrow p_2)$, $c_3 = (q_1 \rightsquigarrow p_3)$, and $c_4 = (q_2 \rightsquigarrow q_2')$, and four sets of framework variables Z_{c_1}, Z_{c_2}, Z_{c_3}, and Z_{c_4}.

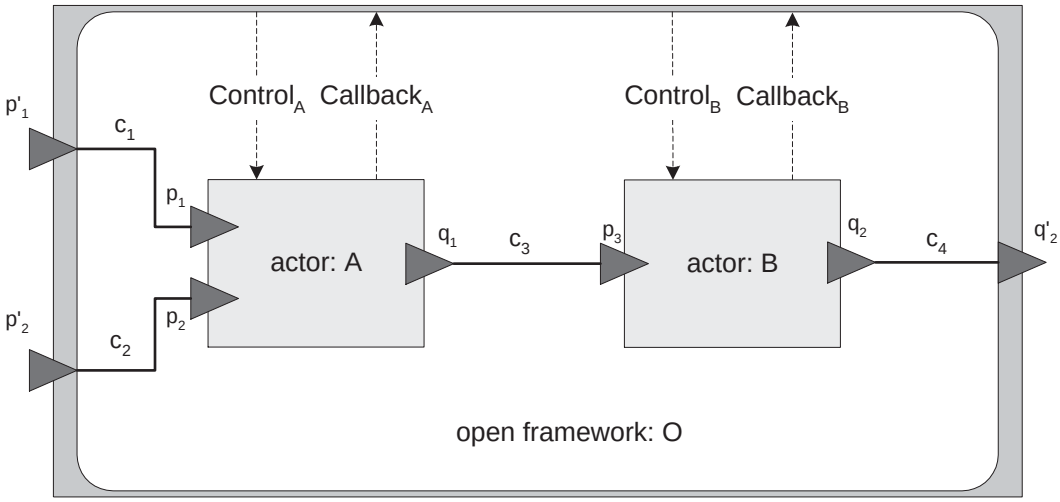

Figure 3.1: An open composite with two actors.

All these framework variables and the variables of actor A and B are the internal

variables of Ω. So,

$$\begin{aligned} \Omega.S &= A.X \cup B.X \cup Z_{c_1} \cup Z_{c_2} \cup Z_{c_3} \cup Z_{c_4} \\ \Omega.P &= \{p_1', p_2'\} \\ \Omega.Q &= \{q_2'\} \end{aligned}$$

All the operations of O, A, and B are operations on these internal variables of Ω, and thus they belong to Ω.**Comp**. Notice that the control-flow operations between the framework O and the actors are also parts of the computational operations of Ω, even though they do not change the evaluation of any variables. So, in general. we have the following proposition:

Proposition 3.1. *An open composite is an actor, and thus is called a* ***composite actor****.*

Being an actor, an open composite can be put into a framework. Thus, a composite actor involves two frameworks, as shown in Figure 3.2. The one outside the composite actor is called the composite actor's *executive framework*. The executive framework controls the composite actor. The one inside the composite actor is called the composite actor's *local framework*. The local framework controls the actors contained by the composite actor. This formalism hides the operations inside a composite actor from other actors in the executive framework. A composite actor will be triggered by its executive framework, and in response to that, it consumes inputs, invokes its local framework for reactions, and produces outputs. Obviously, when the two frameworks are not the same, we obtain hierarchical heterogeneity.

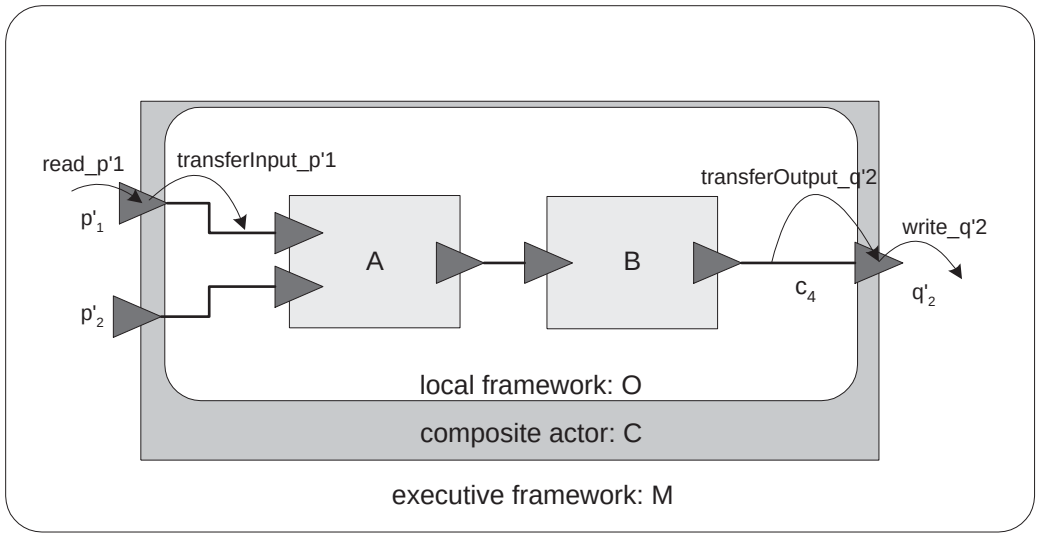

Figure 3.2: A composite actor can have a local framework and an executive framework.

3.1.2 Boundary Operations

In order for a composite actor to communicate with its executive framework, read and write operations must be defined on composite ports. Since these operations happen at the boundaries of composite actors, we call them *boundary operations*. Boundary operations need to be integrated with other operations of the local framework.

When a composite actor Ω performs a `read` on its port p, it communicates with the executive framework and may change the evaluation of variable p. The value in p must be transferred to the channel that connects the inside of this port to an input port of an inside actor. For example, in Figure 3.2, after `read_p`, the value in p must be transferred to one of the variables in Z_{c_1}, so that actor A can use it later. We call this operation `transferInput_p` : $[\{p\} \rightarrow V] \rightarrow [\{p\} \rightarrow V] \times [O.Z \rightarrow V]$. This operation is an internal operation for Ω, and is performed by the local framework.

Similarly, after some inside actors, say B in Figure 3.1, performing a write operation, the value in the framework variable on the channel from the output of B and port q needs to be transferred to the composite output port q. We call this operation `transferOutput_q` : $[O.Z \to V] \to [\{q\} \to V] \times [O.Z \to V]$, and it is also performed by the local framework. A `write_q` operation can then be performed to emit the value to the executive framework.

How to order these boundary operations with local and executive framework triggers is critical for integrating models, and sometimes could be framework/application dependent. One way of doing it, which certainly is not the only way, is to transfer the inputs at the beginning of a reaction of the composite actor, and transfer output after the internal execution has "settled". In this case, we have the ordering relation in the firing set of a composite actor that is shown in Figure 3.3.

However, defining the settlement of a composite execution may not be easy. Depending on the framework activities, the `transferOutput` part of the firing set, and thus the `finish` operation of the composite actor, may not always be reachable. In some cases, even these operations *are* reachable in finite steps, or the framework *inserts* `transferOutput` operations at some point of the execution, the internal actors' reaction may not be precise within the composite reaction.

3.2 Compositional Precise Reaction

As we know from chapter 2, depending on the triggering rules, the firing set of a composite actor with respect to a trigger may not be finite, even though all internal actors

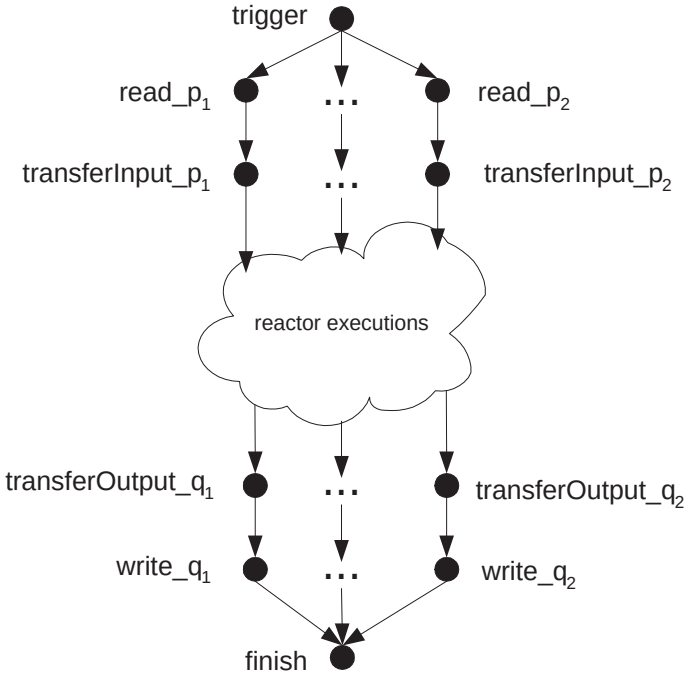

Figure 3.3: A general structure of a composite firing.

are reactive. In addition, even the composite firing is finite, it is not necessarily true that, when the composite finishes firing, all the actors in it are at their quiescent states. Thus, we characterize the following properties for compositional reaction,

Definition 3.1. *An open composite* $\Omega = (O, \mathbf{A}, \mathbf{C})$ *is* ***reactive*** *with respect to a trigger* r*, if* $\Omega.\texttt{fire}|_r$ *is finite. The reaction is* (***compositionally***) ***precise****, if at the end of the reaction, all the actors* $A \in \mathbf{A}$ *are in quiescent states.*

So, a reactive composite actor can be treated as a reactor by its executive framework, and a quiescent state of such a composite actor is the aggregation of all the quiescent states of the actors contained in it.

The reactiveness of a composite actor may not be obvious, even when all the

actors are reactive and the local framework is responsible. For example, Figure 3.4 shows a composite actor C connecting to an actor A within a dataflow framework $DF1$. The composite actor also contains a dataflow framework $DF2$, and a reactor B.

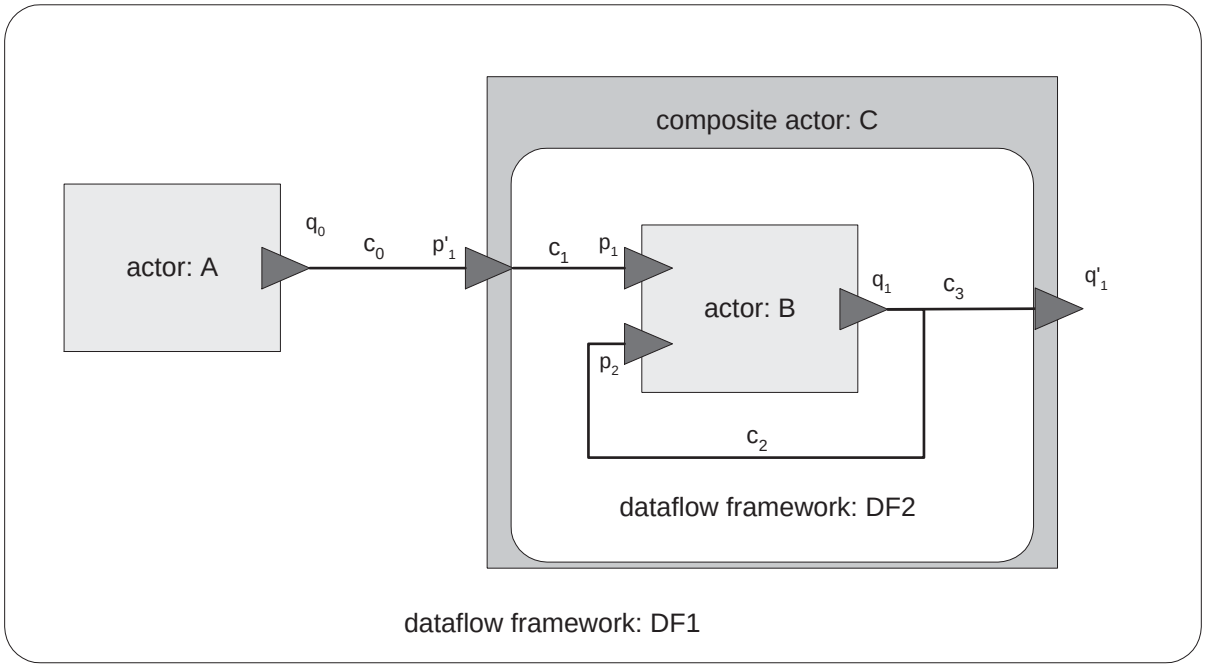

Figure 3.4: A composite actor containing a reactor and a dataflow framework.

Let the firing of reactor A perform one write operation and then finish, and the reactor B first reacts to an input at port p_1 and later reacts to inputs at p_2. That is, B has the following firing sets:

$$\begin{aligned} \texttt{B.fire}|_{\texttt{trigger\_1}} = \{ & \\ & \texttt{read\_p}_1\texttt{\_1}; \\ & \texttt{q}_1 = \sigma(\texttt{p}_1); \\ & \texttt{write\_q}_1\texttt{\_1}; \\ \} & \end{aligned}$$

and for $i > 1$,

$$\texttt{B.fire}|_{\texttt{trigger\_i}} = \{ \texttt{read\_p}_2\texttt{\_i}; \; \texttt{q}_1 = \sigma(\texttt{p}_2); \; \texttt{write\_q}_1\texttt{\_i}; \}$$

Let Actor A have an initial trigger $\texttt{Init} \xrightarrow{\texttt{true}} \texttt{A.trigger}_1$, the composite actor C have a triggering rule

$$\texttt{A.finish\_1} \xrightarrow{\texttt{z}[\texttt{c}_0]\neq\perp} \texttt{C.trigger\_1}$$

and the triggering rules for reactor B are:

$$g \xrightarrow{z[c_1]\neq\perp} \texttt{B.trigger\_1}$$

$$\texttt{B.finish\_(i} - 1) \xrightarrow{\texttt{z}[\texttt{c}_2]\neq\perp} \texttt{B.trigger\_i}, \text{ for } i > 1$$

where g is any operation observable by the framework $DF2$.

During the execution of this model, reactor A is first triggered and writes one value to $z[c_0]$, then the composite actor C is triggered. Suppose the boundary operations shown in Figure 3.3 are taken, so the value of $z[c_0]$ is transferred to $z[c_1]$ by performing a read operation and a `transferInput` operation. Reactor B is then triggered and it reads the value from $z[c_1]$ and writes it to both $z[c_2]$ and $z[c_3]$. Since $z[c_2]$ is not empty now, the reactor B can be triggered again. And the composite actor can keep this loop without reaching an end point. So, C is not reactive. A firing set for the composite actor C is shown in Figure 3.5.

A non-reactive composite actor may be made reactive if its local framework inserts a termination point into the infinite firing set to stop the execution of the internal model

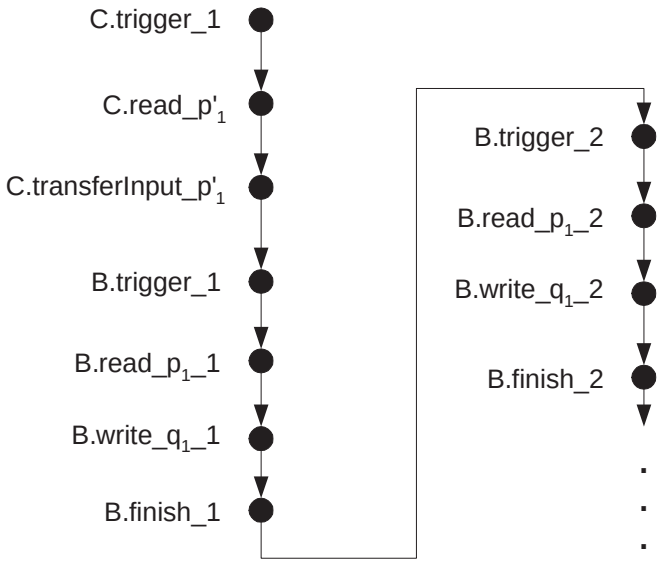

Figure 3.5: An ordering relation for the firing set of composite actor C.

and forces it to transfer outputs. This processes is called *reactivization* of a composite actor. In general, this "termination" of execution may lead the composite actor to a non-quiescent state. However, if the framework is responsible and the actors are precisely reactive, then it is relatively easy for the local framework to restrict the number of triggers for each actor so that the total firing set of the composite actor is finite. And it guarantees that the composite will reach a quiescent state when all the reactions to these triggers are finished. Thus, a responsible framework can *precisely reactivize* a composite actor.

For example, the $DF2$ framework in the previous example can restrict the number of triggers for actor B to be one for each firing of composite actor C. By doing this, C becomes reactive, and at the end of the reaction, reactor B is at its quiescent state. The firing set with respect to `C.trigger_1` is shown in Figure 3.6

Notice that reactivization may not always be unique. For example, in the above

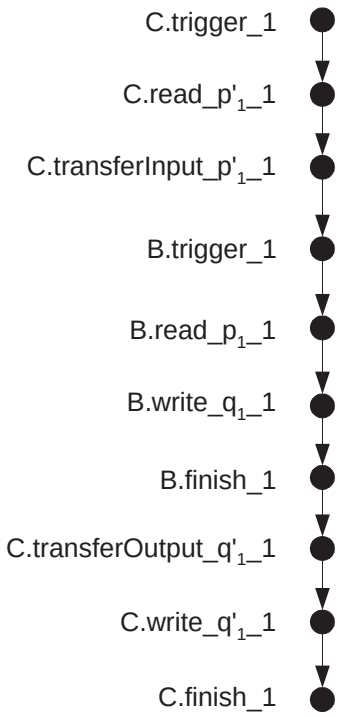

Figure 3.6: A precisely reactive firing set of composite actor C.

model, triggering reactor B twice per firing of C can equally make composite actor C reactive. A more systematic way may rely on the knowledge of the local framework and actors under its control. For example, the framework can keep track on the output data items produced, and, say, restricts the number of output data to be at most (or at least) one at each output port.

There are certain computational models whose precise reactivity can be well-defined. For example, a synchronous dataflow (SDF) actor consumes a fixed amount of data once triggered, and produces a fixed amount of data when it finishes. This property makes a composition of SDF reactors under a SDF framework a SDF composite reactor. Such a composite reactor can specify triggering rules that guarantee that there are at least

enough input data for one internal SDF iteration. In other cases, the notion of time can be used to define reactivity. We will discuss more on that in Chapter 4.

3.3 Modal Models

Some modeling techniques have the notion of *operation modes* and *mode switchings*. That is, the system operates in a certain configuration until some mode switching event occurs, then the system enters another configuration. A different operation mode, in terms of the reactor model, could be a different set of actors, connections, framework variables, communication semantics, and triggering rules.

A systematic way of constructing modal models is to hierarchically combine concurrent models with state-machine-based sequential models [21]. The states in the state machine represent operation modes, and the mode switching events trigger the state transitions. These combinations are called *modal models*. Figure 3.7 shows an example of a modal model, where at the top-level, the composite actor C_0 contains a state machine framework FSM with two states, s_1 and s_2. Each of the states further contains a sub-composite actor C_1 and C_2, which are called the *refinements* of the states. In general, the frameworks inside each refinements may not be of the same kind. The meaning of the model is that, when FSM is at one of its states, the composite actor C_0 is functionally replaced by the refinement of that state. The transition from one state to another may depend on the values of the input of C_0 as well as the output from the current refinement.

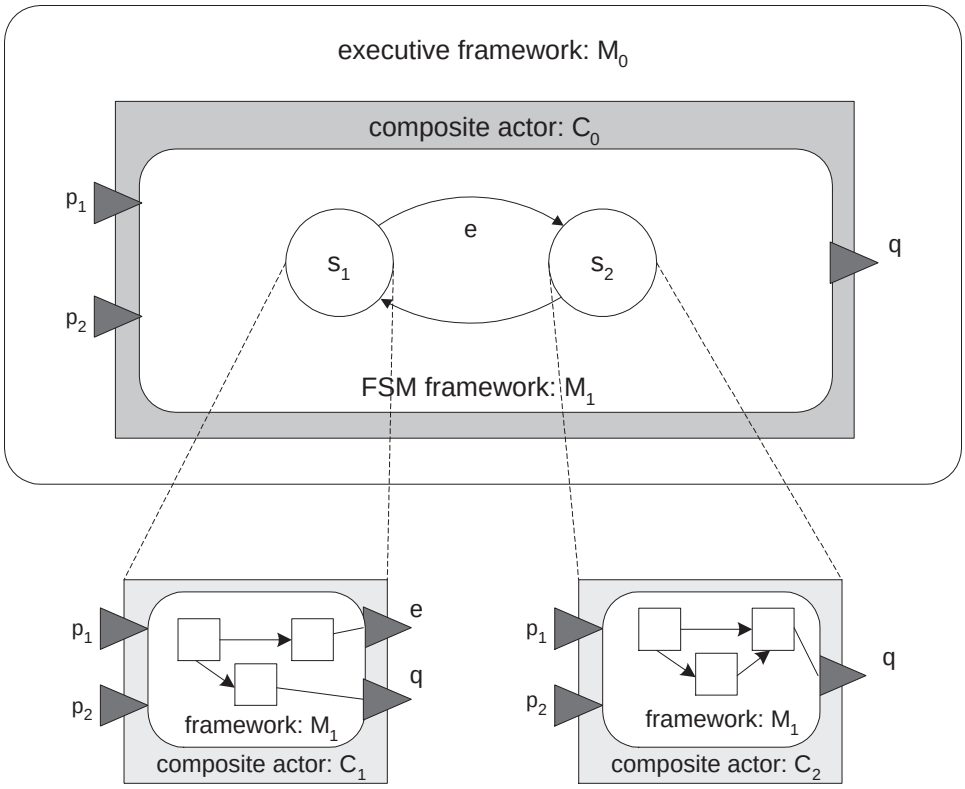

Figure 3.7: A modal model with three levels of hierarchy.

3.3.1 Precise Mode Switching

A key requirement of building modal models is to precisely define the mode switching points. Mode switching may not be safe to perform at arbitrary execution points. Some reactors may not exist after the mode switching, communication channels may change, semantics of communication operations may vary, and the triggering rules may be different.

Suppose, in Figure 3.7, C_0 is triggered to execute and its current state is s_1, so the refinement C_1 is triggered to execute respectively. If C_1 is not compositionally precisely reactive, and an output at port e triggers a state transition to s_2, then when the transition is taken, it is not obvious what states the internal actors of C_1 are in. Moreover, if later the

finite state machine switches back from s_2 to state s_1, it is not clear what states C_1 should start with.

Ideally, a mode switching operation should be a synchronization point (as defined in section 2.1.3) of the entire execution of the composite. Mode switchings should only be performed when all the reactors are at their quiescent state in the current operation mode. In this circumstance, there is a well-defined and consistent semantics of the framework during the time that an reactor executes its firing set. For this reason, we introduce the notion of precise mode switching,

Definition 3.2. *A mode switching operation w is **precise** if all the reactions before w are precise, i.e. if $f \prec w$, then $\underline{f} \prec w$.*

Given the properties described in section 2.3.3, precise mode switching are relatively straightforward to achieve in responsible frameworks. The framework can hold its triggers when receiving a mode switching event. Then all the reactors will eventually finish execution and reach a quiescent state. At this point, the framework can perform the mode switching. Notice that, since no more triggers are sent after the mode switching request, the execution before the mode switching point is well-defined. Furthermore, if the refinement of a state is compositionally precisely reactive, then the FSM composite actor, say C_0 in the example, is also precisely reactive.

3.4 Implementation

In Ptolemy II, the compositional execution is achieved by the `CompositeActor` and the `Director` classes. A composite actor is an actor that contains other actors. A

composite actor can have a local director, which makes it *opaque*. An opaque composite actor is executable. Thus, an opaque composite actor involves two director, a local director and an executive director, just like in our framework model.

As discussed in section 2.5, the Ptolemy II `Executable` interface has seven methods. An opaque composite actor implements these methods and delegates them to its local director. The local director, depending on the model of computation it implements, may implement these execution method differently. In general, in the `fire()` method of the composite actor, it first transfers inputs for all input ports, if there is any data in them, then calls the `fire()` method of its local director, and after the director finishes its execution, it transfers the data in its output ports to the outside model.

A key concept to achieve compositional execution is the notion of "iteration" for a model of computation. An iteration is one compositional precise reaction that start with trigger of the composite actor and finishes at a compositional quiescent state. Obviously, as we shown in previous discussions, for an irresponsible framework, an iteration may not be well-defined.

3.5 Related Work

Compositionality is a big challenge for modeling paradigm and languages. Some models lose properties once composed. For example, a finite sequential processes can always execute to its end, but two finite sequential processes, composed under the CSP model, may never reach their end points, and thus deadlock. Composable models are useful in two ways: *top-down* or *bottom up*. The top-down view of compositionality advocates modeling and

design of a system through *refinement*. A system is first modeled at a high abstraction level, typically with nondeterminism. Then, a module of a high level model is refined into a more concrete implementation. Semantics studies, like structured programming [83], action refinement [76], assume-guarantee model checking [31], and interface theories [17], are based on this point of view. A bottom-up approach advocates composing existing modules into a larger-scale module, through the notion of *containment*. This has been practiced in circuit design for many years. Diposet [33] is a semantics model to study concurrency and containment relations in models of computation.

Many design languages support *homogeneous* compositionality. For example, a composition of automata yields another automaton [67]; a composition of statechart models yields another statechart [28]; a composition of reactive modules is another reactive module [2], and so on.

Complex engineering systems are heterogeneous, and some design methodologies advocate *heterogeneous* compositionality. One example is the globally asynchronous and locally synchronous (GALS) approach, like implemented in POLIS [5] and distributed synchronous languages [6]. These approaches typically integrate two kinds of models, and they have a fixed containment relation. A more aggressive approach is taken by systems like Ptolemy Classic [11], Ptolemy II [44], and El Greco [13], which advocate the integration of multiple sequential and concurrent models. The work presented in this chapter follows the hierarchical heterogeneous modeling approach and focuses on the compositionality of reactivity.

Chapter 4

Timed Responsible Frameworks

In this chapter, we focus on frameworks that have a *continuous* notion of time. In particular, we study continuous-time models, discrete-event models, and the interaction among them, and with some untimed models.

4.1 Time

Time in the physical world is continuous and evolves at a constant rate. Embedded systems with timing constraints usually need to be modeled in a timed framework to reflect their timing behavior. A timed framework is a framework with a notion of time, represented by a framework variable $t \in Z$. In this chapter, we consider the evaluation $\sigma(t) \in \mathbb{R}$, a real number. With this variable, all operations are tagged with a value of t. We say that an operation is performed at time τ, if when the operation is performed, $\sigma(t) = \tau$. We define $T : \mathbf{Oper} \to \mathbb{R}$ to be the function that gives the time stamp of an operation.

Although time in the physical world only increases, it is not necessary to be so in

modeling frameworks. The value of t can only be set by the framework though an operation `setTime` : $\mathbb{R} \rightarrow [\{t\} \rightarrow \mathbb{R}]$, which set a value $v \in \mathbb{R}$ to the variable t. The value of t can be observed by both the framework and the actors, and actors can build its triggering rules and change their firing set based on the evaluation.

The notion of time loosely sequentializes the operations in a timed model. If an operation f is performed at $t = t_1$, and operation g is performed at $t = t_2 > t_1$, then $f \prec g$. Only when $t_1 = t_2$, there may be parallelism between the two operations. Two operations $f \| g$ only if $T(f) = T(g)$ and neither $f \prec g$ nor $g \prec f$.

4.2 Continuous-Time Frameworks

A continuous-time (CT) framework models continuous dynamic systems that can be represented by ordinary differential equations (ODEs).

4.2.1 Conceptual View

ODE-based continuous-time models can be expressed as

$$\dot{x} = F(x, u, t) \tag{4.1}$$

$$y = G(x, u, t) \tag{4.2}$$

$$x(t_0) = x_0 \tag{4.3}$$

where,

- $t \in \mathbb{R}$, $t \geq t_0$ is continuous time;
- $x : \mathbb{R} \rightarrow \mathbb{R}^n$ is the n-dimensional state trajectory;

- $u : \mathbb{R} \to \mathbb{R}^m$ is the m-dimensional input signal;
- $y : \mathbb{R} \to \mathbb{R}^l$ is the l-dimensional output signal;
- $\dot{x} = dx/dt$ is the derivative of x w.r.t. t;
- $x_0 \in \mathbb{R}^n$ is the initial state.

We call F the right-hand side (RHS) of the ODE, and G the output map. We assume that F is *globally Lipschitz continuous* on x for any bounded and piecewise-continuous input u, such that, by the existence and uniqueness theorems of ODEs (see e.g. [70]), for any $t_f \in \mathbb{R}, t_f \geq t_0$, there is a unique trajectory x on $[t_0, t_f]$ satisfying the system dynamics (4.1) and initial condition (4.3) for that input u, except on finitely many discontinuous points. Such an ODE system is called *well-formed*. We only consider well-formed ODE systems in this dissertation.

A set of ODEs can be built using components as shown in Figure 4.1. Conceptually, components in this framework communicate via (piecewise-)continuous waveforms, which are functions on a closed interval of $\mathbb{R}$. The components are piecewise-continuous maps from input waveforms to output waveforms. A special component, the *integrator*, makes a feedback loop an ODE. The output of the integrator is the state trajectory x, and the input of the integrator is the derivative $\dot{x}$. The functions F and G can consist of a feed-forward composition of components that implement the piecewise-continuous maps. High-order ODEs may involve multiple integrators connected in serial or parallel.

Strictly speaking, the conceptual view of differential systems does not fit in our actor and framework concepts. In this model, variables change their values continuously,

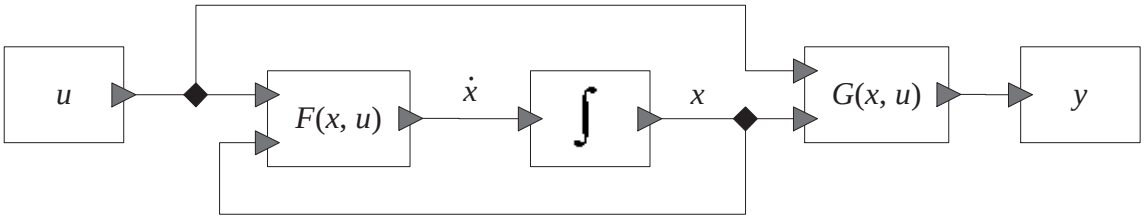

Figure 4.1: A component-based construction of ODEs.

and concepts of read and write operations do not apply. To build continuous-time frameworks in computers, time must be discretized into discrete instants, and ODEs must be solved numerically at these time instants. Nevertheless, the implementation of a CT framework should stay as close as possible to the conceptual semantics of the CT models. For example, the continuous notions of time and waveforms make it proper to ask for the value of any signal at any time instant. The continuous-time framework should be able to find it computationally.

4.2.2 Operational View

The execution of a continuous-time model involves solving the ODEs numerically. A widely used class of algorithms, called *time-marching algorithms*, discretize the continuous time line into an increasing sequence of time instants, and numerically computes the state variable values at these time instants in that increasing order. The discretization of time usually reflects the speed and accuracy trade-offs of a numerical algorithm, and is determined based on the error tolerance of the solutions and the "order" of the algorithm[1].

[1] Different classes of numerical algorithms may define "order" differently. But roughly, an ODE solution algorithm has order n if the numerical error $e = O(h^n)$, where h is the step size.

To compute the values of state variables at each time instant, the right-hand side of the ODE needs to be evaluated with different x and u values. For example, the simplest ODE solver is possibly the *forward Euler* (FE) algorithm,

$$x(t+h) = x(t) + h \cdot F(x(t), u(t), t), \tag{4.4}$$

where t is the last time instant where the solution is computed, so $x(t)$ and $F(x, u, t)$ are known; h is the integration step size; and $x(t+h)$ is the to-be-computed value of x at $t+h$.

This algorithm can be achieved by making the integrator implement (4.4), and scheduling the components in Figure 4.1 by sending triggers properly. More precisely, an integrator, `FEIntegrator`, which implements the FE algorithm, has the following variables:

$$\begin{aligned} FEIntegrator.P &= \{input\} \\ FEIntegrator.Q &= \{x\} \\ FEIntegrator.S &= \{lastTimeInstant\}, \end{aligned}$$

the firing set for time t_0:

```
FEIntegrator.fire|t0 = {
                x = x0;
                lastTimeInstant = σ(t);
                write_x;
            }
```

and for any $t > t_0$,

```
FEIntegrator.fire|t = {
                read_input;
                h = σ(t) − lastTimeInstant;
                lastTimeInstant = σ(t);
                x = x + h ∗ input;
                write_x;
            }
```

The scheduling resembles a data-driven style, where the triggers are sent to actors according to their data-dependency order, using integrators to break the feedback loops. For example,

$$\textbf{ODESchedule} = \{Integrator \rightarrow u \rightarrow F(x,u)\} \tag{4.5}$$

is a proper schedule for the system in Figure 4.1 for the FE algorithm.

Under this ODE solver, the communication channels between the ports represent the values of the continuous waveforms at a particular time. A framework variable for a channel is a buffer of size one.

Multi-iteration algorithms

More advanced ODE solvers may require firing the integrators and the actors that build the RHS of (4.1) multiple times at several intermediate time instants for a single integration step. For example, a 2^{nd} order Runge-Kutta method (RK2) [66] has the form:

$$k_0 = F(x,u,t) \tag{4.6}$$

$$k_1 = F\left(x+\frac{h}{2}k_0, u(t+\frac{h}{2}), t+\frac{h}{2}\right) \tag{4.7}$$

$$k_2 = F\left(x+\frac{3h}{4}k_1, u(t+\frac{3h}{4}), t+\frac{3h}{4}\right) \tag{4.8}$$

$$x(t+h) = x(t)+h\cdot\left(\frac{2}{9}k_0+\frac{1}{3}k_1+\frac{4}{9}k_2\right) \tag{4.9}$$

This algorithm requires that the framework iterate the schedule like (4.5) four times before the integrator can complete the computation of $x(t+h)$. During the four iterations, the value of t needs to be increased accordingly, and the integrator needs to

implement different firing functions, (4.6) - (4.9) accordingly. As a consequence, there are several intermediate values appear in the framework variables.

Algorithms like FE and RK2 are called *explicit* algorithms, which have a fixed number of iterations for each time instant. There are also *implicit* algorithms, which reduce the ODEs to a set of algebraic equations, and rely on fixed-point or Newton iterations to find the solution. For certain ODE problems, implicit algorithms may provide better numerical stability, and thus allow larger step sizes [20]. Choosing what ODE algorithm to use is largely application dependent.

An example of implicit algorithms is the *backward Euler* (BE) algorithm,

$$x(t+h) = x(t) + h \cdot F(x(t+h), u(t+h), t+h) \tag{4.10}$$

Obviously, this is an algebraic equation, which involves computing $F(x(t+h), u(t+h), t+h)$ without knowing $x(t+h)$. A fixed-point iteration to solve this algebraic equation starts with a guess of $x(t+h)$, say $x_0(t+h) = x(t)$, then iterates

$$x_{k+1}(t+h) = x(t) + h \cdot F(x_k(t+h), u(t+h), t+h), \text{ for } k \geq 0 \tag{4.11}$$

until for some m, $||x_{m+1}(t+h) - x_m(t+h)|| \leq \epsilon$, where ϵ is a small positive number that measures the convergence of the sequence $\{x_k(t+h)\}_{k\geq 0}$, and $||\bullet||$ is some norm. If the sequence converges, then $x(t+h) = x_{m+1}(t+h)$ is the numerical solution of x at time $t+h$.

Fixed-point iterations (and Newton iterations) may not converge for arbitrary step size h. However, the contraction mapping theorems [10] implies that if the trajectory is continuous on $[t, t+h]$, and h is small enough, then the fixed-point iteration always converges. Computationally, this implies that we start with a guessed step size, and an

upper bound for the number of iterations, M. If the fixed-point iteration does not converge in up to M iterations, then we reduce the step size, say by half, and try again, until the process converges. The convergence can be observed by the framework by looking at the values in framework variables that represent the states of the ODE system.

Fixed-point semantics

For a continuous-time framework, both explicit and implicit ODE solvers should be thought of in terms of fixed-point semantics. The framework starts with a known state of the ODE system, it performs a finite (either known or unknown) number of scheduled firing of actors, and reaches a fixed-point, which gives the state of the ODE system at the new time instant. During the scheduled firing of the actors, the values of the framework variables and the actor variables may be inconsistent, the value of the time variable may increase or decrease, and their values may not be used by the rest of the system (e.g. the output map). For this reason, we introduce an operation for a CT framework to indicate the successful resolution of the state at a time instant – `stateResolved`. This is a framework operation, and actors (e.g. those that construct the output map) can build triggering rules using it.

After the state values at time $t + h$ have resolved, the output value y can be computed by triggering the reactors that construct the output map G in their data-dependency order. After that, the computation at time $t + h$ has completed. We introduce an operation `commit` to indicate the finishing on the computation at one time instant. The framework can then advance time to the next step, and repeat the ODE solution process again.

So, in general, the operations in a CT framework may look like Figure 4.2, where

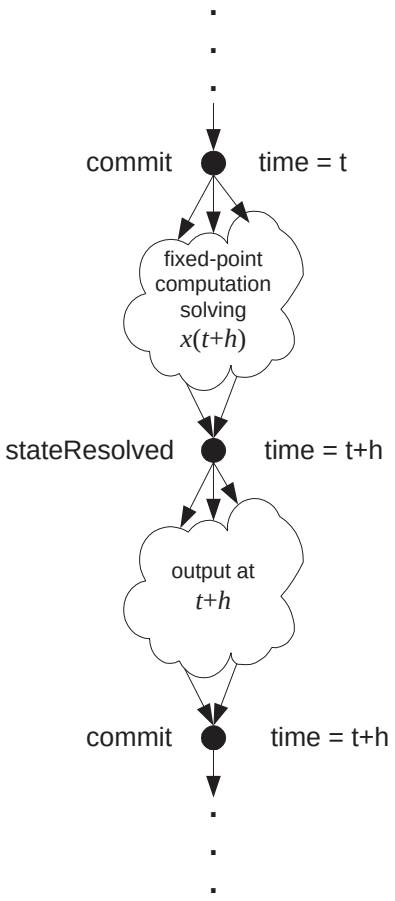

Figure 4.2: Illustrating the fixed-point semantics of CT frameworks.

`stateResolved` and `commit` at each time instant are synchronization points for this model. There is a discrete set of time points $\{t_0, ..., t, t+h, ...t_f\}$, and the values of x at these time points are computed by the ODE solvers. In the process of solving the ODE and producing the output, there may be some partially ordered operations, but the `stateResolved` and the `commit` operations are always in a total order.

4.2.3 Hybrid Components in CT Frameworks

A *timed event* (or, *event* for short) is a tuple $e = (\tau, v) \in T \times V$, where $T \subset \mathbb{R}$ is the set of time stamps, and V is the set of values. The time stamps make timed events totally comparable. For $e = (\tau, v)$ and $e' = (\tau', v')$, we say $e < e'$ if $\tau < \tau'$.

A set of events is *discrete* if they are ordered isomorphic to a (subset) of integers[2]. This means that there is an 1-1 and onto function that maps the events to the subset of the integers, and the mapping preserves the ordering relation among the events.

Some continuous-time frameworks embrace discrete events on a subset of communication channels[3]. A communication channel that represents discrete events will have an empty value at all but those event times. That is, let E be a set of discrete events on a channel $c = (p \rightsquigarrow q)$, and T_E be the set of time stamps for those events, then $z_c \neq \perp$ only when $\sigma(t) \in T_E$. These events may trigger components that only reacts to events, rather than involves in the computation of continuous-time waveforms.

A *discrete actor* is an actor that only has discrete events at its input and output channels. The firing of a discrete actor is triggered by the presence of an input event. A *precisely reactive discrete actor* is a discrete actor whose firing set is finite and the presence of any input event is a responsible trigger. That is, the reaction does not have to wait for more than one input event.

Discrete actors are special cases of *hybrid actors*, which are actors that may have both continuous and discrete inputs and outputs. Hybrid actors fundamentally increase the expressiveness of ODEs. For example, they can be used to model discontinuous inputs,

[2]This definition is due to W.T. Chang [42].

[3]These frameworks are better called *mixed-signal frameworks*. But for consistency reasons, we keep calling them continuous-time frameworks.

discontinuities on the RHS of ODE, A/D and D/A converters, and event detectors.

Since a discrete event is defined by both a time stamp τ and a value $v = \gamma(x(\tau), u(\tau), \tau)$, which may depend on the state and the input at τ, it is important to determine both of them in a continuous-time framework. An *event generator*, E, is a reactor that has continuous waveform input and discrete event output. An event generator implements the function $\gamma(x(\tau), u(\tau), \tau)$, and is triggered only when time is equal to the event time, by the `stateResolved` operation.

In a continuous-time framework, there are two ways to define the time stamp of an event:

- One way is to give the time stamps τ directly. Events of this type are called *timed events*. A triggering rule of an event generator EG that produces this type of event can be written as,

$$\texttt{stateResolved} \xrightarrow{t==\tau} \texttt{EG.trigger}$$

- Another way is to give a predicate $\beta(x, u, t) == 0$ on the value of the state variables, the input variables, and time. We assume $\beta(\cdot)$ to be continuous on its arguments. For example, an event e can be defined as occuring whenever the state trajectory crosses zero, i.e. $x == 0$. Events of this type are called *state events*. A triggering rule of an event generator EG' that produces this type of event can be written as,

$$\texttt{stateResolved} \xrightarrow{\beta(x,u,t)==0} \texttt{EG}'\texttt{.trigger}$$

It is the responsibility of the framework to trigger such reactors at the correct time, no matter how they specify the rules.

Reactors that respond to discrete events are relatively easy to handle. The write operation of an event is an observable operation to the framework. And a discrete actor D can be triggered by that write operation. I.e. the triggering rule may look like,

$$\texttt{write\_q} \xrightarrow{\texttt{true}} \texttt{D.trigger}$$

for some output port q.

Sometimes, a hybrid actor may decide to produce a discrete event in the future time. This can be achieved by allowing discrete actors to register their triggering rules dynamically during the execution. For example, a discrete actor D that is expected to be triggered at time $\tau' > t$, can register a triggering rule,

$$\texttt{stateResolved} \xrightarrow{t==\tau'} \texttt{D.trigger}.$$

4.2.4 Responsible Continuous-Time Frameworks

A CT framework controls the progression of time. A *responsible continuous-time framework* not only needs to keep the computational results close to the real solution of a CT model in terms of acceptable numerical errors, but also needs to trigger the hybrid actors according to the rules they specify. In general, to achieve correct computation of continuous-time models, a responsible CT framework must control the modeling time according to the following three issues:

(A.) **Numerical performance**: The numerical errors of ODE solvers significantly depends on the choice of step sizes. Although choosing smaller step sizes always improves accuracy, it may elongate the computational time. So, how to choose the step

sizes to achieve maximum computation speed subject to tolerable numerical errors must be considered.

(B.) **Unsmoothness**: Numerical ODE solvers assume a certain order of smoothness on the RHS function F. If this assumption is broken, then the numerical algorithm should not cross the non-smooth point in one step. For example, if

$$F(x,u,t) = \begin{cases} 1 & : \quad t \geq 1 \\ -1 & : \quad t < 1 \end{cases}$$

then the numerical algorithm should not cross the time instant 1 in one step.

(C.) **Event generation**: Event generators need to be triggered at specified time instants. Timed event generators are easy to handle, since the framework can examine the rules to adjust the increase of time instant so that no events are missed. State events are much harder, since the framework cannot predict exactly when a predicate that involves the state of the ODE is true.

To implement responsible CT frameworks, we define the concept of *breakpoints*.

Definition 4.1. *A **breakpoint** in a continuous-time model is a time instant when the right-hand side F of the ODE or the output map G are not sufficiently smooth.*

The "sufficiency" of smoothness may depend on the ODE solvers used. By the nature of ODE problems, breakpoints should not be crossed in one integration step. In fact, the values of the state variables may not be well-defined at these points. The numerical algorithms should instead compute the left and right limits of the state values.

Depending on whether a breakpoint is known before the modeling time reaches that instant, we classify *predictable* and *unpredictable* breakpoints. All unsmooth points

that explicitly depend on time are predictable breakpoints, while the unsmooth points that depend on the values of the state variables are unpredictable.

Predictable breakpoints are easy to handle. They can be stored in a table by the framework. Before the framework chooses the next integration step size, it can look at the table and possibly reduce the step size to make sure that no predictable breakpoints is crossed in this step. Unpredictable breakpoint can only be detected after an integration step has finished. Actors, whose behaviors depend on unpredictable breakpoint, may register predicates, like $\beta(x, u, t) == 0$, to the framework. After a tentative numerical integration step, the framework can examine these predicates, and see whether any predicate is `true`. If so, an unpredictable breakpoint is found. It may also check if any predicate has changed sign in this step. Typically, by the continuity of function $\beta(\bullet)$ on x, this change of sign means there is an unpredictable breakpoint that has just been crossed by the integration step. Thus, the result at $t + h$ is not valid. The framework should roll back to time t, reduce the step size to some h' (probably by examining the property of $\beta(\bullet)$), and start the integration from t to $t + h'$ again.

Strictly speaking, there is always a risk of missing unpredictable breakpoints if $\beta(x, u, t)$ crosses zero twice in one integration step. Restricting the rate of change of $\beta(\cdot)$ may reduce the risk. Specifying high numerical accuracy requirements, which essentially reduce general step sizes, may also help.

4.3 Discrete-Event Frameworks

A discrete-event framework has a continuous notion of time but only discrete actors. Since there are no continuous waveforms, no ODE solvers and fixed-point semantics are needed. Conceptually, a discrete actor responds to a set of discrete-event inputs and produces a set of discrete events as outputs. These events are time stamped, so the framework knows exactly *when* to trigger reactors to process the (next) events. The reactors are required to be *causal*, which roughly means that time stamps of output events should be no earlier in time than the corresponding input events. This requirement, although intuitive, has profound semantics implication on the existence and uniqueness of "behaviors" of discrete event systems. A formal discussion of this causality is given in [42]. We will only use the intuitive definition in this dissertation, and restrict all actors to be causal in our discussion.

4.3.1 Operational View

Recall that a discrete reactor is conceptually triggered by time-stamped input events. It has a precise reaction for any input. That is, it does not wait for another input once an input is available. As shown in Figure 4.3, suppose an output port q of a DE reactor D is connected to an input port $p'$of a reactor D', then the triggering rules of D' may simply be,

$$\texttt{write\_q} \xrightarrow{\texttt{true}} \texttt{D}'.\texttt{trigger\_p}'. \tag{4.12}$$

Suppose that the reactor D' implements a time delay, which delays its input events by time duration δ and produces the same value. That is, if the reactor D' is triggered by

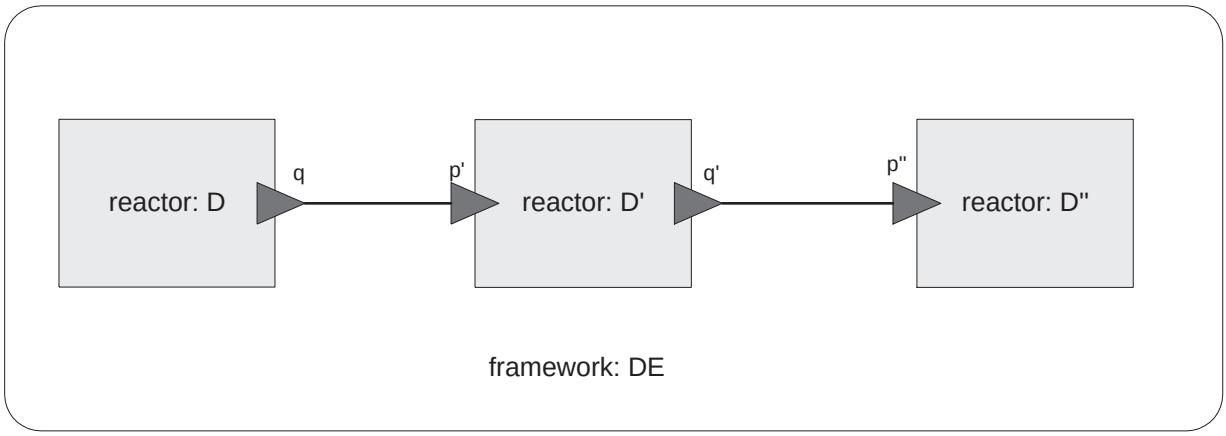

Figure 4.3: Three reactors in a DE framework.

the rule (4.12) at time τ, and it reads a value v from the channel $(q \rightsquigarrow p')$, then it should assign v to port q', and do a `write_q'` at time $\tau + \delta$. This can be achieved by allowing reactor D' to register for a trigger at the time it wishes to produce an output. We use the syntax:

$$\texttt{fireAt}(\texttt{D}', \tau + \delta)$$

for this registration, which registers the following triggering rule with the framework,

$$\texttt{setTime}(\tau + \delta) \xrightarrow{\texttt{true}} \texttt{D}'.\texttt{trigger\_q}',$$

so that it can be triggered again when time reaches $\tau + \delta$. In response to that trigger, it can perform `write_q'`, and reactor D'' can be triggered accordingly. So, suppose that the minimum interval between two successive events produced by D is greater than the time delay δ, then D' may have the following firing sets[4]: for trigger $\texttt{D}'.\texttt{trigger\_p}'$,

[4] A general time delay reactor may need to implement a queue to locally store the timed events to be produced.

$$\begin{aligned}\texttt{D}'.\texttt{fire}|_{\texttt{trigger\_p}'} = \{ \\ &\texttt{read\_p}'; \\ &\texttt{q}' = \sigma(\texttt{p}'); \\ &\tau = \sigma(\texttt{t}) \\ &\texttt{fireAt}(\texttt{D}', \tau + \delta); \\ \}\end{aligned}$$

and for trigger $\texttt{D}'.\texttt{trigger\_q}'$,

$$\begin{aligned}\texttt{D}'.\texttt{fire}|_{trigger\_q'} = \{ \\ &\texttt{write\_q}'; \\ \}\end{aligned}$$

This mechanism of registering a triggering rule in the future to trigger the reactor itself is called *self-triggering*. It is also a useful mechanism for source reactors, which are reactors that have no input port, like D in Figure 4.3. Source reactors have no input events to trigger them, but by using self-triggers, they can be triggered according to the progression of time. For example, suppose reactor D implements a *Poisson* processes. At the beginning of the execution, it can request a trigger $\texttt{trigger\_t}_0$,

$$\texttt{Init} \xrightarrow{true} \texttt{trigger\_t}_0.$$

In response to that trigger, it computes a Poisson-distributed random time value t_1 at which it produces the next output. It then registers a self-triggering rule:

$$\texttt{setTime}(\texttt{t}_1) \xrightarrow{\texttt{true}} \texttt{D.trigger\_q}$$

to emit the output at time t_1 from port q, and at the same time register a self-trigger at the next random time.

So, the job of a DE framework is to iteratively look at all the triggering rules and find the smallest value τ it should assign to the time variable, and trigger all reactors

that are activated at time τ. We also introduce a `commit` operation in the DE framework to indicate the completion of activities at a specific time. Because of the causality of all reactors, after the commit operation at τ, no timed event earlier than τ can occur. The framework can then increase the time variable value to the next smallest value, and repeat the triggering process.

4.3.2 Precise-Reactive CT Composite

Discrete event reactors may be implemented compositionally. In particular, it can be implemented by an open composite with a continuous-time framework. The situation is not trivial, since time in the CT framework conceptually progresses continuously. As a composite actor, the CT composite actor is only triggered by the DE framework at discrete time instants. Furthermore, the CT composite actor should be reactive and causal to obey the discrete event semantics, which implies that time in the CT framework should always be ahead of the DE framework time, and the CT time should not progress beyond the time stamp of the "next" input event.

More precisely, let's consider the configuration shown in Figure 4.4, where a CT composite actor is controlled by a DE framework. So, there are two time variables involved, t_d for the DE framework and t_c for the CT framework. We have the following theorem.

Theorem 4.1. *Let C be a CT composite actor with time variable t_c. In order for C to be a causal discrete-event reactor, whenever C is triggered by the DE framework, the following relation must hold:*

$$\sigma(t_d) \leq \sigma(t_c). \tag{4.13}$$

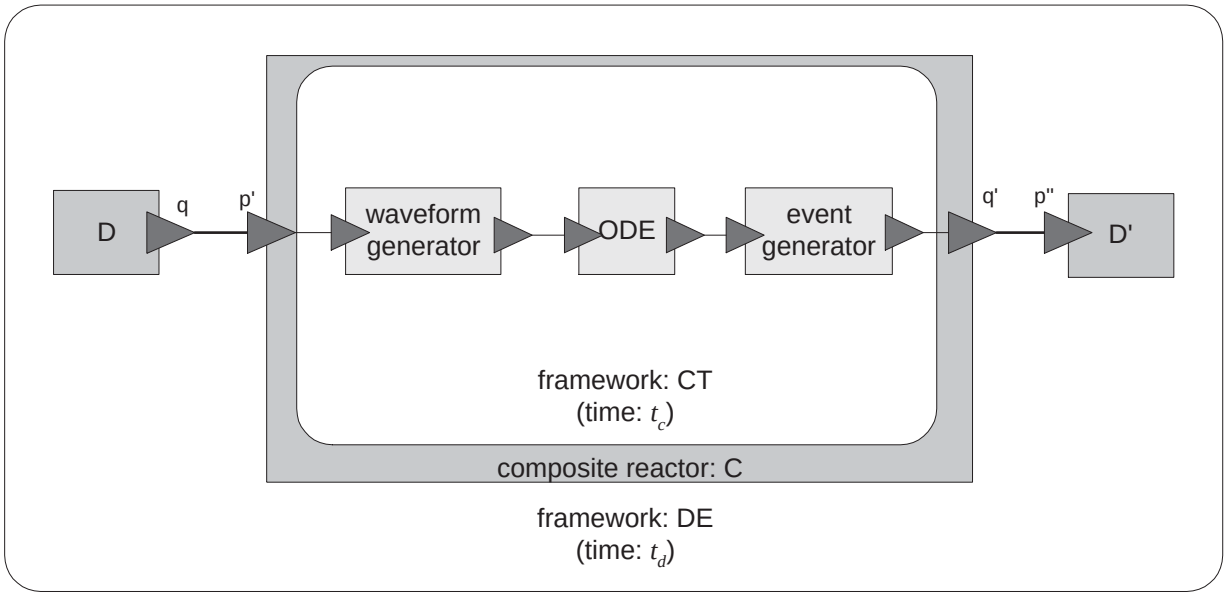

Figure 4.4: A CT composite actor inside a DE framework.

Proof. By contradiction. Suppose on the contrary, when C is triggered and $\sigma(t_d) > \sigma(t_c)$. Let $\sigma(t_d) - \sigma(t_c) = \delta$. By the continuous-time semantics of the CT framework, it will start execute from tc continuously. It is possible that, for some $\epsilon < \delta$, an event generator produces an output, which is also an output of C, at $\sigma(t_c) + \epsilon < \sigma(t_d)$. Then, the composite actor C is not a causal DE reactor. □

Theorem 4.1 indicates that when a CT composite actor is controlled by a DE framework, the CT composite must run ahead of the DE time. This optimistic execution has further implications. There are two cases when the composite actor C is triggered by an input event, $\sigma(t_c) = \sigma(t_d)$ or $\sigma(t_c) > \sigma(t_d)$. The situation $\sigma(t_c) = \sigma(t_d)$ is desirable, which means that the input event makes effect at the time it happens. Now, the question is how far in advance should the CT composite actor execute, or, in other words, *is a CT*

composite actor reactive?

Suppose that when C is triggered, we have $\sigma(t_c) = \sigma(t_d) = \tau$. The reactiveness of discrete reactors seems to imply that the CT composite actor should execute until it generates an output event. However, this may not always be possible. What if there is no more discrete events for output? Futher more, even if it generates an output $o = (\tau', v)$ at time $\sigma(t_c) = \tau' = \tau + \epsilon$ for some positive ϵ, and requests a self-trigger at τ' to the DE framework, what if the next input event for C is earlier than $\tau + \epsilon$. That is, the next time C is triggered, it finds that the time value of the DE framework $\sigma(t_d) = \tau'' < \tau'$, which means that the optimistic execution last time was partly wrong. Thus, the CT framework should not blindly execute until generating an output event. It should restrict the optimistic execution to a certain length, and check with the outside DE framework for the next input. And if the input event time, say at τ'', is earlier than the stop time τ' of the optimistic execution, the CT framework should roll back part of its last execution, to τ'', and recompute the trajectory after τ''. The amount of look-ahead execution is an application-dependent design parameter. A section of interaction between CT and DE frameworks is shown in Figure 4.5.

In summary, a precise-reactive CT framework needs to perform optimistic execution and support rollback.

4.4 Timed Precise Mode Switching

The notion of time naturally provides a set of synchronization points in timed models. These points are the `commit` operations at a particular time instants. These

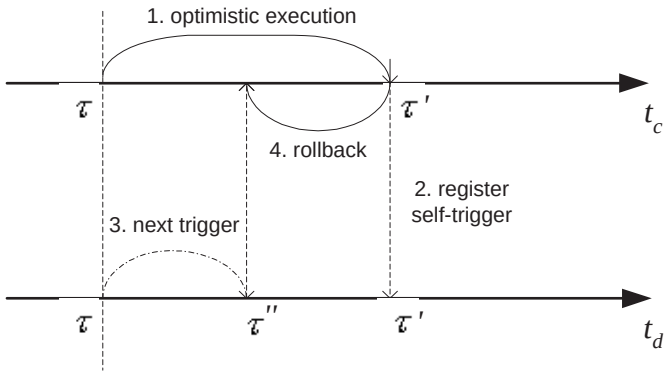

Figure 4.5: A situation that requires the CT framework to roll back its optimistic execution within a DE framework.

synchronization points make mode switching in timed model relatively easy to be defined precisely.

Mode switching in timed models can be defined in terms of the value of the timed variable. Of course, in continuous-time frameworks, it can also be defined in terms of the values of continuous waveforms, like state events. The framework can activate all the triggers before the mode switching time, and make sure that it performs the `commit` operation at the mode switching time. Then, the mode switching can take place.

One situation that needs additional attention is the "zero delay" semantics of the mode switching operation. Conceptually, mode switching takes no time. At the mode switching time τ, all operations before τ are finished. However, the mode switching may activate triggers at exactly time τ again. So, time cannot be advanced immediately after the mode switching. One way of looking at this is the $\tau^- - \tau - \tau^+$ interpretation. That is, the `commit` operation before the mode switching only completes the operations up to τ^-; the mode switching happens at time τ; and the new starting time is τ^+. Although τ^-, τ,

and τ^+ take the same value, they indicate an ordering relation among operations.

4.5 Implementation

We implement the CT and DE frameworks as corresponding domains in Ptolemy II. A detailed documentation of these implementation can be found in [34] and [52]. We only highlight some of the features in the implementation that reflects the discussions in this chapter.

- **CT domain scheduling**. The CT domain in Ptolemy II implements a responsible CT framework. It allows the existence of discrete signals and hybrid actors. The scheduling for a CT model is based on clustering. A model is clustered into a *continuous part* and a *discrete part*. The boundary between the two parts are hybrid actors, like event generators and waveform generators. A signal type system, discussed in the next bullet, performs this clustering. In the continuous part, the actors are further partitioned into the state transition actors, which implement the F function in the ODE 4.1, and the output actors, which implement the output map G in 4.2. Initial conditions of the ODE are parameters of the integrators. Within each partition, we use a demand-driven topological sort algorithm to schedule the actors. The state transition actors are all the actors whose outputs are needed by the integrators. The topological sort starts with the input ports of the integrators, and backtracks to source actors or the outputs of integrators. The output actors are all actors that are needed by the sink actors, like plotters. The topological sort starts with the sink actors, and backtracks to source actors or integrators. Actors in the discrete cluster are scheduled

in a data-driven manner. These discrete actors cannot introduce delays from their inputs to their outputs, and there cannot be feedback. If these features are needed, a DE composite actor can be used.

- **Signal type system**. When a continuous-time system contain discrete components, the signals at the boundaries have to be converted accordingly. For example, the output of event generators should connect to the input of discrete actors, and the output of waveform generators should connect to continuous actors. In addition, many components can be used in both continuous and discrete parts of a CT system. For example, a `Scale` actor can be used to scale a waveform by a factor, or it can scale all the event values in a set of discrete events. In order to properly cluster actors into continuous and discrete parts and schedule them accordingly, we develop a signal type system.

  The signal type system resolves the signal type for each port in a CT system. The possible types are `UNRESOLVED`, `CONTINUOUS`, `DISCRETE`, and `NOT-A-TYPE`, forming a lattice in Figure 4.6. A type that is lower in the lattice is more specific than the type that is higher in the lattice. This means that the type `UNRESOLVED` can be resolved to either `CONTINUOUS` or `DISCRETE`, and the types `CONTINUOUS` or `DISCRETE` can be resolved to `NOT-A-TYPE`.

  Some components have fixed signal types at their ports. For example, an integrator has a `CONTINUOUS` input and a `CONTINUOUS` output; a periodic sampler has `CONTINUOUS` inputs and `DISCRETE` outputs; a zero-order-hold actor has `DISCRETE` inputs and `CONTINUOUS` outputs, and many actors only works for `DISCRETE` inputs and

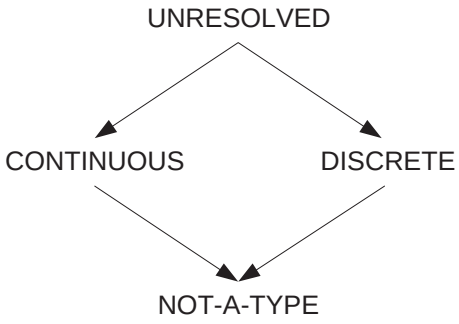

Figure 4.6: The signal type lattice for mixed-signal continuous-time models.

outputs. But for actors that can be used in both continuous and discrete clusters, their signal types are `UNRESOLVED`. The role of the signal type system is to resolve all the `UNRESOLVED` types by converting them to either `CONTINUOUS` or `DISCRETE`. And the rules are simple:

- If a port p is connected to another port q with a more specific type, then the type of p is resolved to the type of the port q.
- If a port p of type `CONTINUOUS` is connected to a port q of type `DISCRETE`, then both of them are resolved to `NOT-A-TYPE`.
- Unless otherwise specified, the types of the input ports and output ports of an actor are the same.

At the end of the signal-type resolution, if any port is of type `UNRESOLVED` or `NOT-A-TYPE`, then the topology of the system is illegal, and the execution is denied.

- **Step size control mechanisms**. The CT domain in Ptolemy II controls the progression of time by the three mechanisms discussed in section 4.2.4. The step size

control mechanisms are achieved by the `fireAt()` method in the `Director` class and a `CTStepSizeControlActor` interface. Actors can use `fireAt()` method to register predictable breakpoint, the CTDirector will make sure that the entire system is executed on that time instant. Actors that can only affect step sizes after the new CT states has been resolved should implement the `CTStepSizeControlActor` interface. These actors includes integrators, which controls the numerical accuracy and convergence, and state event generators, which produces unpredictable breakpoints. After resolving the states, the `CTDirector` will query these actors for the successfulness of the last step. If any of these actors disagree the resolved states, either because of intolerable numerical error or missing of events, the director will recompute the last step with a smaller step size. The smaller step size is also obtained by asking these step size control actors.

4.6 Mixed-Signal and Hybrid System Modeling

The compositional precise reactivity of CT and DE frameworks allow us to build models that hierarchically compose continuous and discrete dynamics. This section gives the modeling structure for such systems, as well as some examples built in Ptolemy II.

4.6.1 Mixed-Signal Models

A mixed-signal system can be built by hierarchically composing CT and DE models. For example, Figure 4.7 shows a scenario where a DE model is embedded in a CT system. This is a natural model for systems like computer-based control application, where

discrete controllers are embedded within continuous plants. An event generator produces discrete events that trigger the execution of the DE subsystem. The response, another set of events, is fed through the waveform generator and converted back to waveforms.

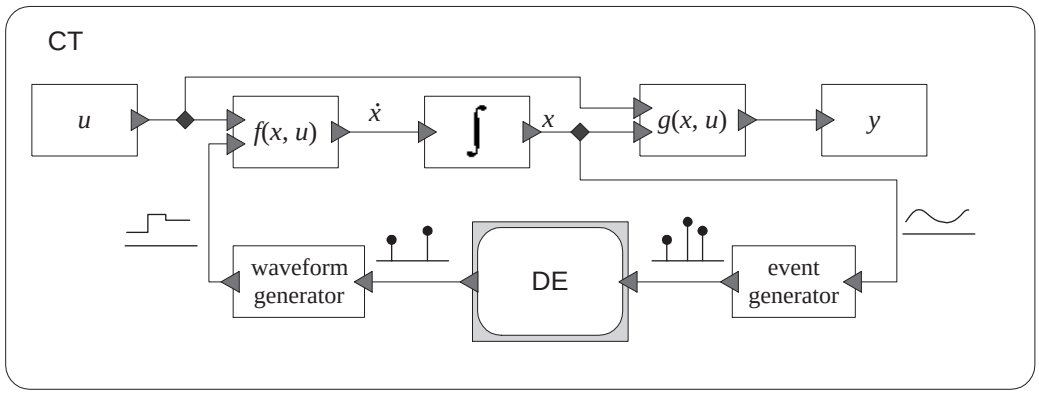

Figure 4.7: A DE composite actor inside a CT model.

Figure 4.8 shows a scenario where a CT model is embedded in a DE system. This is a natural model for systems like mixed-signal circuits and micro-electromechanical (MEM) devices. These systems have large portion of discrete parts and typically provide a discrete interface to larger applications. Event generators and waveform generators are used again at the boundaries of these models.

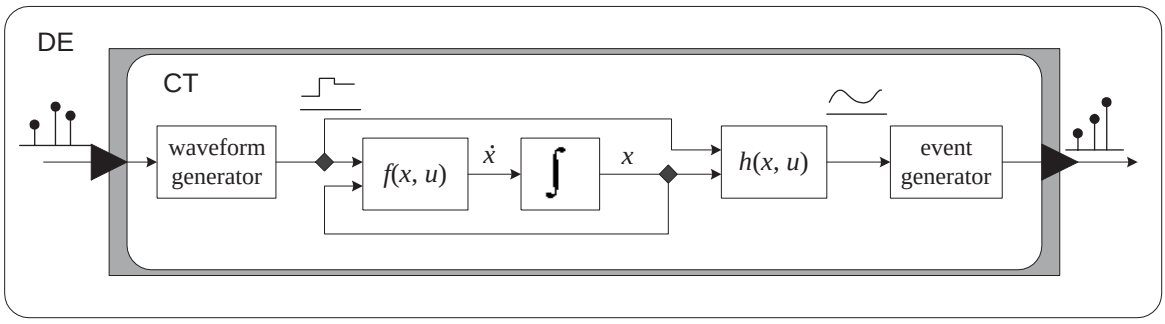

Figure 4.8: A CT composite actor inside a DE model.

4.6.2 Mixed-Signal Examples

Controller with time delay

This example models a discrete controller that controls a continuous plant, as shown in Figure 4.9. The control algorithm is simply a proportional feedback controller. Presumably the controller is implemented in software and it introduces a computational delay from the receiving of input samples to the production of the control values. Depending on whether there are other software tasks running, this delay may vary from sample to sample.

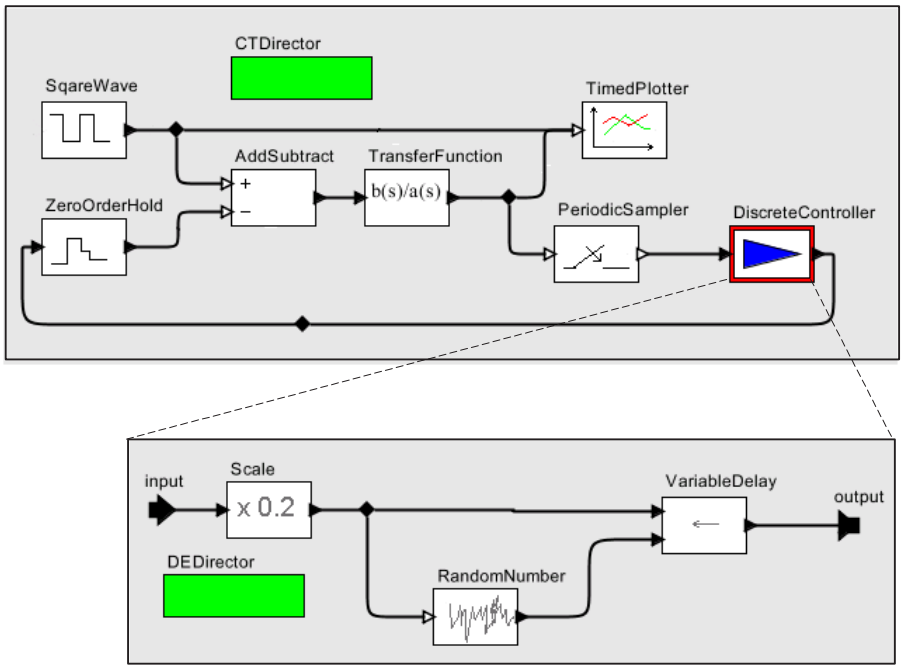

Figure 4.9: A Ptolemy II model for a control system with time delay.

The model has two levels of hierarchy, a CT top-level containing a DE composite actor. A `TransferFunction` actor[5] is used to model the differential equations. The output

[5]This is a syntactic sugar for high order differential equations. Internally, it is built using integrators,

of that actor is periodically sampled and fed into the discrete controller. Inside the discrete controller, the control law is applied. The event also triggers a random number generator, and a variable delay actor, which delays its input events by the amount of time indicated by the value of the second input. We model the delay as a random number that takes two values 0.05 and 0.1 with equal probability, 50%. One execution trace is shown in Figure 4.10.

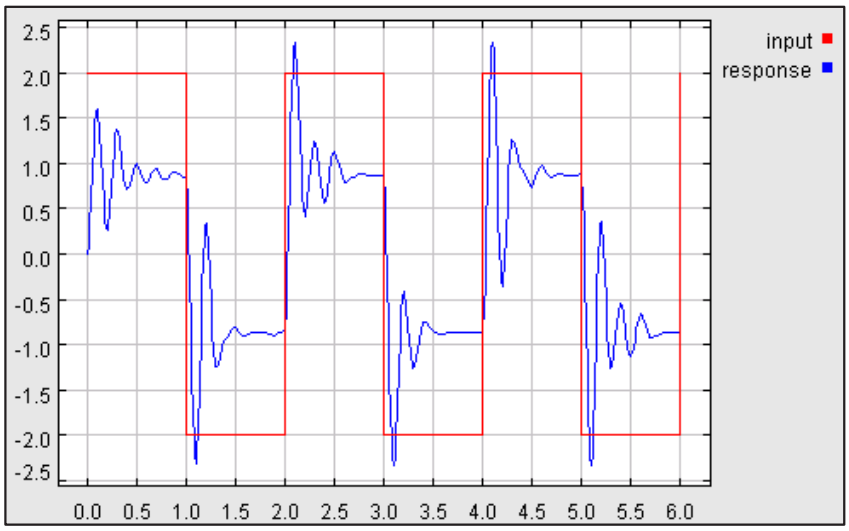

Figure 4.10: The execution result for the model in Figure 4.9.

Micro-accelerometer

Sigma-delta (Σ/Δ) *modulation* [14], also called *pulse density modulation*, is a Bang-Bang controlled over sampling A/D conversion technology. An analog input is over sampled N times faster than the requested digital output frequency, and quantized to one bit, ± 1. The quantized value is fed back to the analog part, as well as accumulated by a digital accumulator. For every N samples, the convertor produces the digital output and reset

adders, and scale actors.

the accumulator. Due to its robustness, Σ/Δ modulated A/D convertors have been extensively developed. Recently, this technique has been applied to micro-electromechanical accelerometers to reduce noise, enhance stability, and improve sensing range [49].

Figure 4.11 illustrates the physical structure of a Σ/Δ modulated micro-accelerometer. The three beams and the gaps between them create a structure that convert acceleration at the vertical direction to changes of capacitance. By sampling and accumulating the capacitance, a digital representation of the acceleration can be computed.

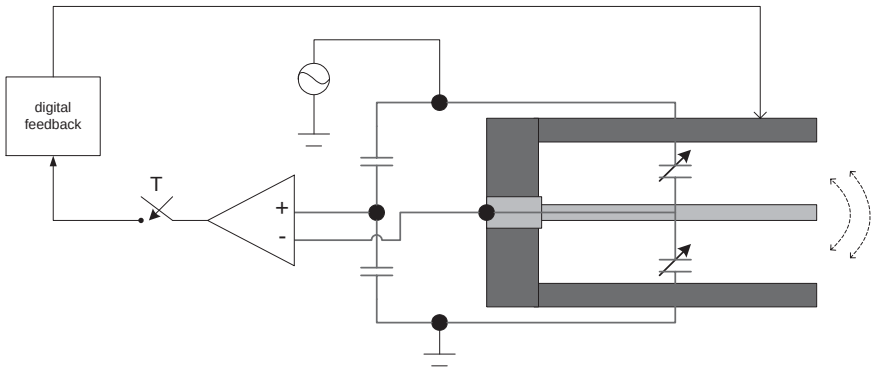

Figure 4.11: A physical structure illustrating a micro-accelerometer.

Figure 4.12 shows a model for the modulated micro-accelerometer. A CT composite actor, `CTSubsystem`, is used to model the capacitance dynamics of the accelerometer, which is simplified to 2^{nd} order ODE. The sensing signal is sampled by the periodic sampler, filtered by a lead compensator (FIR filter), and fed to an one-bit quantizer. A delay actor is used to model the time delay introduced by filtering and quantization. The outputs of the quantizer are fed back to the analog part. The quantized signal is filtered again by the moving average (MA) filter, and accumulated. A digital clock, which produces a trigger every N sampling period, triggers the accumulator to produce the digital output, as well

as resets the accumulator.

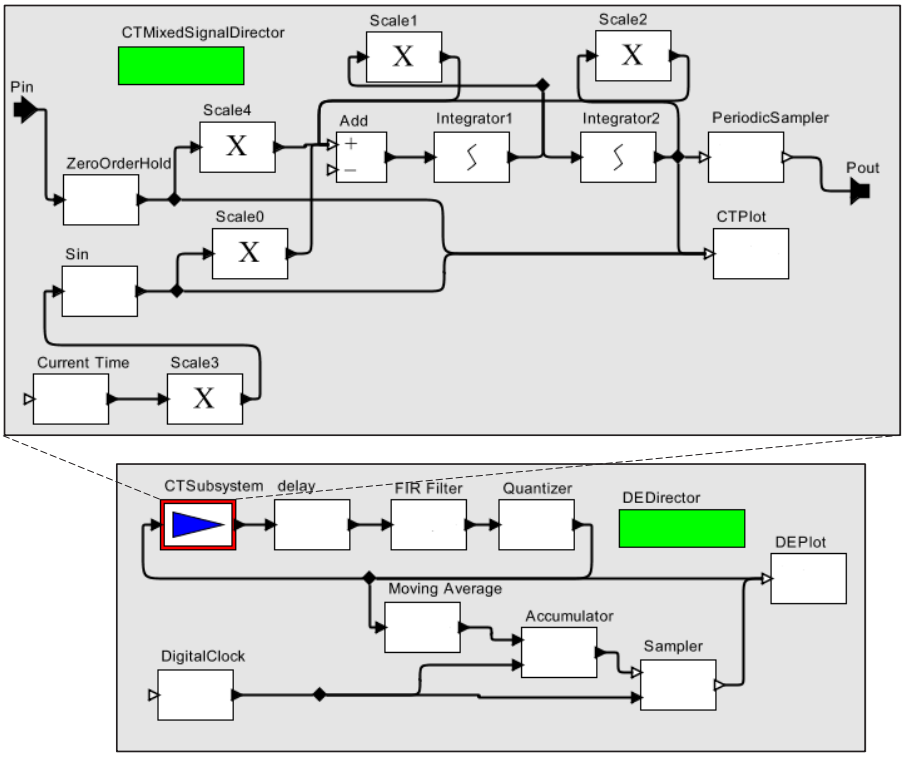

Figure 4.12: A Ptolemy II model for Σ/Δ modulated accelerometer.

Figure 4.13 shows an execution result of the model for a sine wave acceleration input. The upper plot in the figure shows the discrete signals. The dense events, with values ± 1, are the quantization result. The sparse events are the final output of the accumulator, i.e. the digital outputs, and as expected, they have a sinsoidal shape. The lower plot shows the continuous signals, where the low frequency sine wave is the acceleration input, the high frequency waveform is the analog sensing signal, and the square wave is the zero-order hold of the feedback from the digital part.

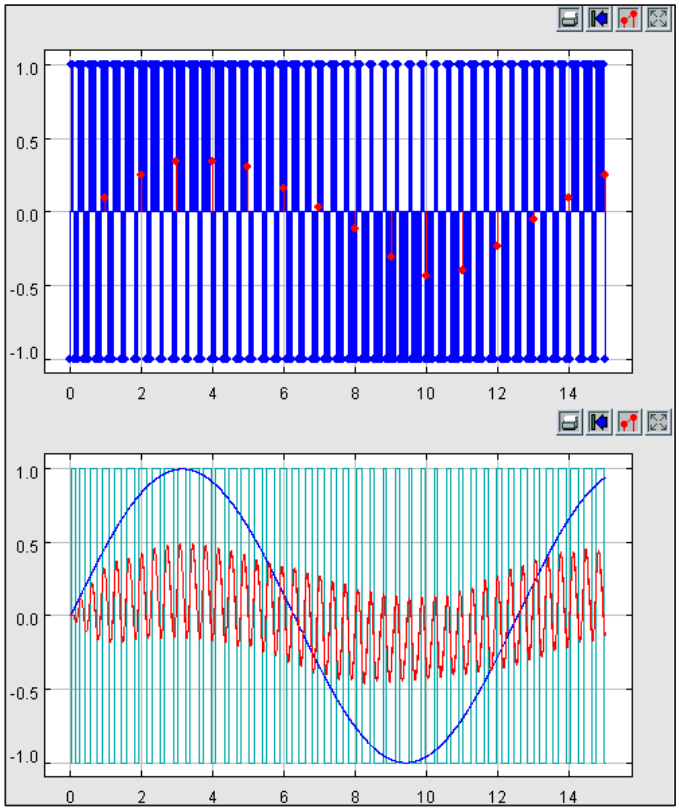

Figure 4.13: An execution result for the model shown in Figure 4.12.

4.6.3 Hybrid System Modeling

A hybrid system is a modal model that consists of finite state machines and continuous-time models, as shown in Figure 4.14, where each state is refined into another CT composite actor. Notice that by adopting the event generation facilities in CT models, a CT subsystem that refines an FSM state can produce discrete events as their outputs, like the port `e` in the figure. State machine transition triggers can be built using these events, as well as continuous signals from the inside and the outside domains.

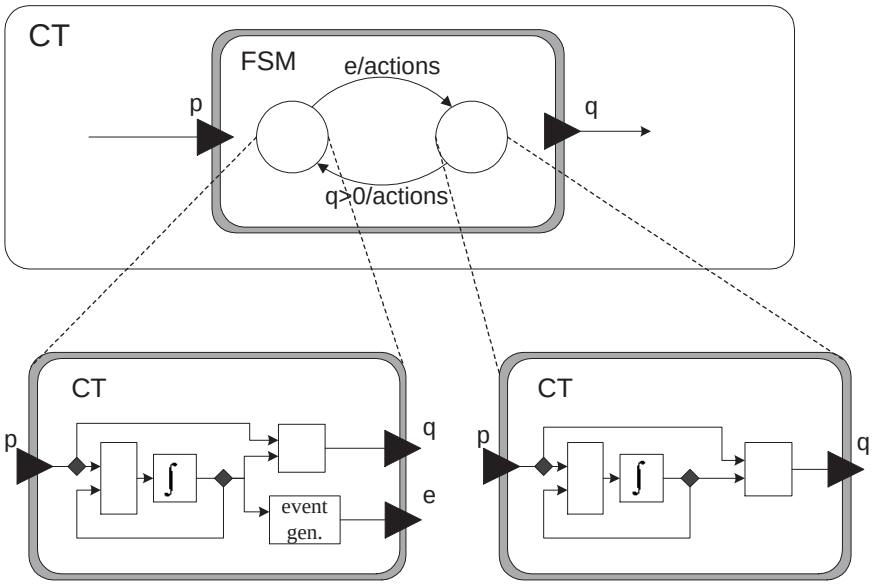

Figure 4.14: A hybrid system is a modal model with hierarchies of FSM and CT.

Hybrid System Example: Sticky Point Masses

This example shows a hybrid system with two states. As shown in Figure4.15, there are two point masses on a frictionless surface with two springs attaching them to fixed walls.

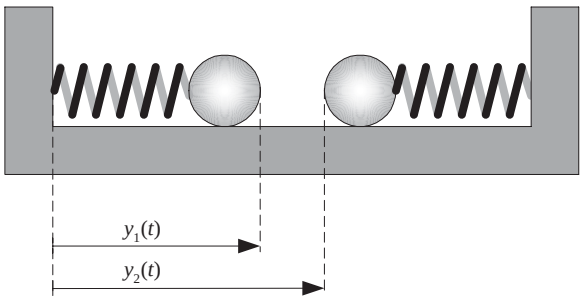

Figure 4.15: A sticky point mass system.

Let $y_1(t)$ denote the right edge of the left mass at time t, and $y_2(t)$ denote the left edge of the right mass at time t. Let p_1 and p_2 denote the neutral positions (i.e. the equilibrium points) of the two masses, so the force is zero. For an ideal spring and frictionless surface, the force at time t on the mass is proportional to $p_1 - y_1(t)$ for the left mass and $p_2 - y_2(t)$ for the right mass, assuming the force is positive to the right. Let the spring constants be k_1 and k_2, and the masses be m_1 and m_2. Then, by Newton's law, we have

$$\ddot{y}_1(t) = k_1(p_1 - y_1(t))/m_1 \tag{4.14}$$

$$\ddot{y}_2(t) = k_2(p_2 - y_2(t))/m_2 \tag{4.15}$$

Now, giving initial positions other than the equilibrium points, the point masses oscillate. The distance between the two walls is small enough that the two point masses may collide. The point masses are sticky. And, when they collide, the situation changes. With the masses stuck together, they behave as a single object with mass $m_1 + m_2$. This single object is pulled in opposite directions by two springs. While the masses are stuck together, $y_1(t) = y_2(t)$.

Let $y(t) = y_1(t) = y_2(t)$, the dynamics are now given by:

$$\ddot{y}(t) = \frac{k_1 p_1 + k_2 p_2 - (k_1 + k_2)y(t)}{(m_1 + m_2)} \tag{4.16}$$

We also assume the stickiness decays exponentially after the collision, such that eventually the pulling force between the two springs is big enough to pull the point masses apart. This gives the two point masses a new set of initial positions and speeds, and they oscillate freely until they collide again.

The system model, as shown in Figure 4.16, has three levels of hierarchy - CT, FSM, and CT. The top level is a continuous-time model with two actors, a composite actor, `SPM dynamics`, which outputs the position of the two point masses, and a plotter that simply plots the trajectories. The composite actor is a finite state machine with two modes, `Apart` and `Together`.

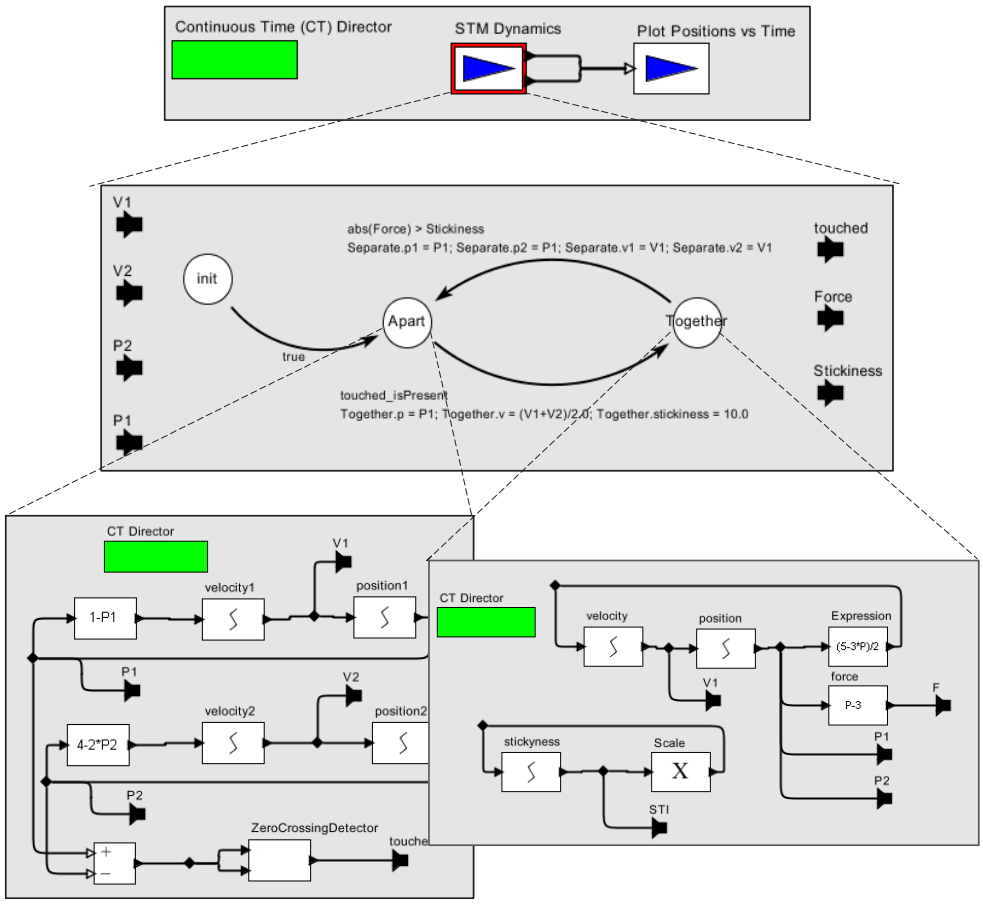

Figure 4.16: A Ptolemy II model for the sticky point mass system.

In the `Apart` state, there are two ODEs modeling two independently oscillating point masses, as in (4.14) and (4.15). An event detector, implemented by subtracting one position from the other and detecting the zero crossing, is used to generate the *collision*

event, `c`. This event will trigger a FSM transition from the `Apart` state to the `Together` state. Besides transferring the point mass position, the actions on the transition set the velocity of the stuck point mass based on law of conservation of momentum:

$$v = \frac{v_1 m_1 + v_2 m_2}{m_1 + m_2}, \tag{4.17}$$

where v_1 and v_2 are the velocities of the point masses before the collision, and v is the velocity of the stuck point mass, after the collision.

In the `Together` state, there is one differential equation implementing (4.16), and another first order differential equation modeling the exponentially decaying stickiness. An expression computes the pulling force between the two springs. The trigger on the transition from the `Together` state to the `Apart` state compares the pulling force to the stickiness. If the pulling force is bigger than the stickiness, then the transition is taken. The positions and velocities of the two separated point masses are equal to those before the separation. An execution result is shown in Figure 4.17.

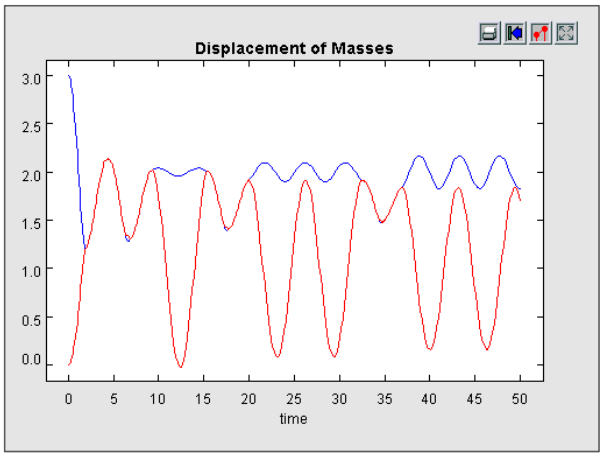

Figure 4.17: An execution result of the sticky point mass model.

4.7 Related Work

Modeling and simulation of timed systems was one of the first applications of computer systems. The work presented in this chapter is greatly influenced by circuit (analog, digital, and mixed-signal) simulation technologies, discrete event simulation technologies, control system simulation technologies, and hybrid system theories.

Early computer simulation tools typically deal with a single model of computation and application domain. For example, SPICE [77] is particularly tuned for analog circuit simulation, VHDL [3] and Verilog [73] simulators are tuned for digital circuits, and Simulink[6] v1.0 is designed for simulating continuous control systems. Heterogeneous systems, like mixed-signal circuits, micro-electromechanic systems (MEMS), and computer control systems, have boosted the theory and practice of integrating continuous and discrete modeling and simulations [65], [56], [74], in particular, mechanisms for event detection [55] and step-size controls [38]. Early tools, like SPLICE [63], only perform a coarse event prediction and cannot handle tight feedback among continuous and discrete parts of a circuit. Hybrid system modeling tools like SHIFT [18] and Teja also simply perform coarse-grained event prediction. Saber [15] and its successive VHDL-AMS [24] and Verilog-AMS [35] simulator VeriasHDL [16] have very sophisticated step-size control mechanisms and handle much complicated circuits. Later versions of Simulink [29] also implemented event detection mechanisms to support the existence of discrete blocks, including finite state machines.

[6]Simulink is a software package of The Mathworks, Inc..

Chapter 5

Real-Time Responsible Frameworks

The ultimate goal of designing embedded systems is to deploy them into the physical world and let them function. This chapter discusses frameworks that interact directly with the physical world. Such frameworks are called *run-time frameworks* in contrast to *design-time frameworks* that at most only have simulated physical world models. A run-time framework needs to operate with respect to the physical time, respond and produce physical events, and manage the computation and communication resources within the embedded system. While design-time frameworks may freely adjust the notion of time and events to achieve computational efficiency, a run-time framework needs to strictly respect the law of physics in the real world. In cases where reactivity cannot be completely fulfilled, it may sacrifice precise reactions of some components for obtaining timely reaction of some other components. A common way to specify criticality of reactions is to assign *priorities*

to software components. However, blindly applying priority-driven execution may introduce many problems, including priority inversion. In this chapter, we formulate prioritized precise reaction and responsible run-time framework to solve the priority inversion problem. After introducing the concepts of time determinism, value determinism, and real-time responsible frameworks, we propose a real-time model of computation — timed multitasking (TM), which makes time explicit at the programming level and leverages a real-time responsible framework to achieve deterministic timing behavior of embedded software tasks.

5.1 Run-Time Composite Actor

The real world is a framework with its laws of physics. In the physical framework, time, called the *real time*, is continuous and flows at a constant rate independent of anything else[1]. The physical framework is like a conceptual continuous-time framework discussed in section 4.2.1, where components (here called *physical processes*) do not need triggers. They evolve continuously and concurrently with respect to the time continuum.

An entire embedded system in the physical world can be viewed as an actor, namely a *run-time actor*, which is "managed" by the physical framework. All operations within the run-time actor are stamped by a value of real time. The triggers and inputs of a run-time actor are typically obtained from sensors; and its outputs are made available to the physical world by actuators. A run-time actor may be a composite actor, with a *run-time framework* as its local framework.

For a run-time actor, it is important to distinguish data I/O from triggering of

[1]We do not consider the relativity effects.

reactions. Not all data acquisition triggers immediate reactions. A trigger creates a thread of execution in the run-time actor, while data inputs do not.

5.1.1 Physical Data I/O

The physical world changes continuously. In order for the embedded system to respond and control the physical processes, some states of the physical world must be acquired by the embedded system, discretely. Similarly, discrete outputs from the embedded system need to be converted to physical activities. These jobs are performed by sensors and actuators, as shown in Figure 5.1.

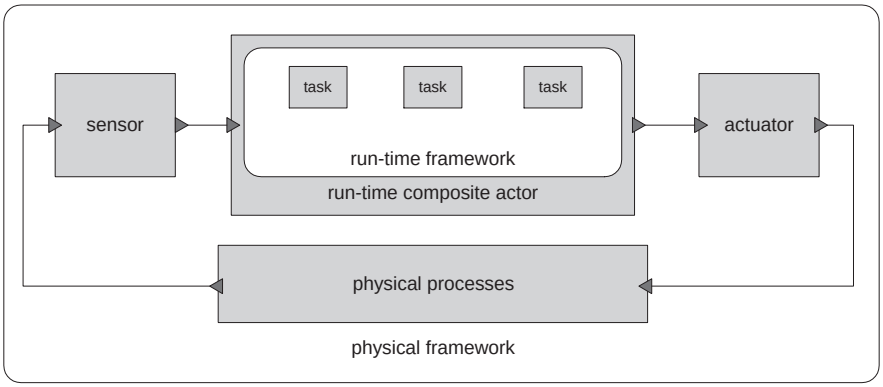

Figure 5.1: The physical world as a framework.

Following [41], we define two kinds of semantics for the I/O data between the embedded system and the physical world — event semantics and state semantics.

- *Event semantics* requires that the receiver of the data process every event exactly once. The loss of a single event may lead to a misunderstanding between senders and receivers. If there is a mismatch between the production and consumption rates of events, a blocking mechanism or a queuing mechanism may be introduced to force

synchronization.

- *State semantics* reflects the current state of the physical world. It is reasonable to only keep the most recent samples of physical states. The rate mismatch between senders and receivers can be solved by preserving and/or overwriting data samples. State data are usually seen in control-oriented real-time systems, where controllers only deal with the latest state of plants.

5.1.2 Run-Time Triggers

Conceptually, computing in a run-time embedded system never terminates. This infinite computing can be segmented into an (infinite) aggregation of finite computation. We view these pieces of finite computation as reactions, and the starting point of each reaction is a trigger. These triggers may or may not directly associate with physical events. We classify three kinds of triggers for a run-time composite actor.

- *Self-triggered.* Self-triggered embedded software has a single thread of control, typically implemented as an infinite outer loop. Once it starts execution, it repeats some computation and communication activities over and over again. Sensor information is pulled from sensors to internal actors. Timing properties of reactions totally depend on the operations performed within the loop, and may differ significantly from time to time.

- *Time-triggered.* Timed-triggered embedded software starts its reaction in response to some (predefined) clock signals. All other sensor information may be pulled from sensors in order not to block the reaction. Time-triggered models have the advantage

that all triggers are predictable in terms of time, and reactions potentially have precise starting times, so that timing analysis may be easy.

- *Event-triggered.* Event-triggered embedded software responds to changes in some physical variables. There are master events that trigger responses, and other sensor information may be pulled. Physical events may sometimes be highly unpredictable. It may be that a second trigger comes before the reaction of the first trigger is finished. But in many cases, the physical dynamics actually guarantees that certain kinds of events do not repeat beyond a particular frequency. Understanding the physical dynamics and choosing what physical event to use as triggers is a important design decision to make for event-triggered systems.

Self-triggered embedded software does not need interrupts of any kind, but it also suffers slow reaction to some events and non-predictable response time. Both time-triggered and event-triggered execution requires interrupts to the system. Time-triggered execution sacrifices promptness of response in favor of predictable timing behavior, while event-triggered execution takes another choice.

5.1.3 Run-Time Frameworks

The internals of a run-time composite actor may be an aggregation of software components and a framework that schedules their execution. Following the real-time programming communities, we call these software components *tasks*. The framework that manages these tasks is a run-time framework. In most embedded systems, the run-time framework is implemented as a real-time operating system (RTOS).

Although time-triggered and event-triggered run-time composite actors can be viewed as reactors from the outside, their internal reactions are seldom sequential. Some internal tasks may have a long execution time and a late deadline to finish, while some other tasks may have to be finished promptly.

It is worthwhile to emphasize that digital embedded systems only interact with the physical world at some discrete time instants. From the physical effect point of view, only those discrete inputs and outputs affect the computation and the physical dynamics. The operations inside the framework, and their ordering are not visible by other physical processes. This information hiding gives run-time frameworks the flexibility to arrange internal actor's operations based on reactivity constraints, as long as it preserves timing properties of its I/O operations.

For example, an embedded controller shown in Figure 5.2 has two tasks: the `controller` task computes control outputs, and the `supervisor` task monitors the control algorithm and updates the controller's parameters.

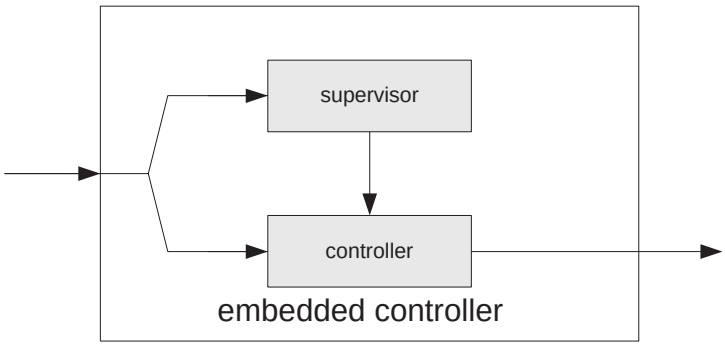

Figure 5.2: Two tasks in a controller.

Suppose that the controller is triggered by periodic samples, say every $2ms$, and

for each sampling input, the controller produces an output with a (fixed) delay, say $1ms$. The supervisor task is only triggered once in a while, e.g. every second, and it takes much more time to compute a new set of parameters to update the control algorithm. Since both tasks are implemented on the same embedded system, they share the computing resources. It is unacceptable that once the supervisor task is started, the controller stop producing any output for a long time. A better strategy is shown in Figure 5.3, where the controller task *preempts* the long run supervisor task. In the figure, each box represents the execution of a task. The shaded parts in the supervisor task indicate that its execution is preempted by the controller task.

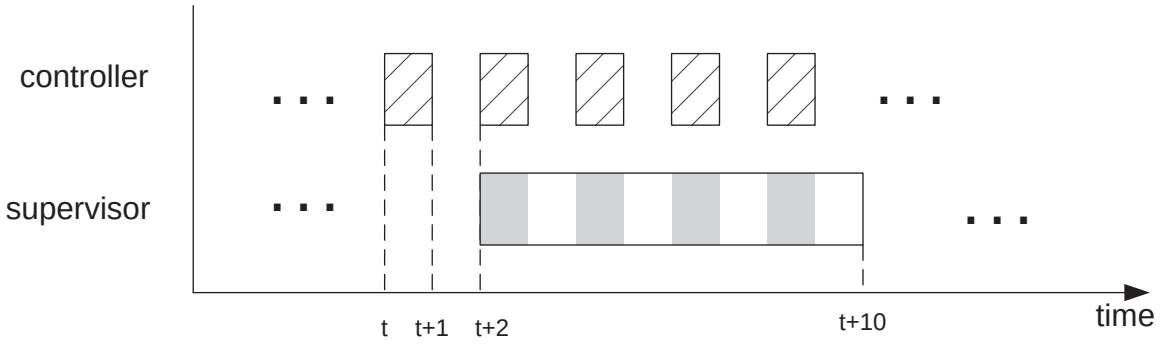

Figure 5.3: Timing diagram of a controller with two tasks.

Preempting long-running tasks to grant resources to a more "important" task is a powerful concept. It may provide better reactivity than nonpreemptive execution. In practice, many real-time operating systems support preemptive execution of tasks. However, preemptivity also brings another level of complexity to real-time programming. It may make real-time programs hard to understand, analyze, and maintain.

Resource Management

Typical embedded systems usually only have limited resources for its multiple tasks. These resources include computing resources, like CPU and memory, and communication resources, like buses, networks, and physical I/O. Sometimes, resources may be simply logical. For example, a critical data section can be viewed as a resource. At any time, there is at most one task that is allowed to write to it.

In order to understand the preemptive execution of tasks, we need to further characterize the preemptability of resources. Some resources are *arbitrarily preemptable*, to a certain granularity. After granting the resource to a task, it can be taken back at any time without waiting for the task to complete. A typical example of preemptable resources is the CPU resource. Some resources may not be arbitrarily preemptable. Once the resource is occupied, its use cannot be interrupted until the task actively release it. During the time that a task is using the resource, the resource is *nonpreemptable*. For example, a shared communication medium is nonpreemptable during the time that a task is sending a packet.

Managing multiple resources is not trivial. For example, a task A may first need resource a, and before releasing a, need resource b. Another task B may first need resource b, and before releasing it, need resource a. Without carefully managing their execution, it may occur that each reactor occupies one resource and waits for the other resource — a typical deadlock situation.

Some resources may be preemptable for some tasks, but none preemptable for another set of tasks. We call them *partially preemptable resource*. For example, writing to a critical data section is nonpreemptable for tasks that share the same data section but

preemptable for other tasks. This partial preemptability may introduce *priority inversion* problems, which will be discussed further in the next section.

5.2 Real-Time Computing: Common Practice

A common practice of embedded systems programming is to adjust *priorities* among the tasks to fulfill timing constraints. Intuitively, priorities represent the relative importance of tasks at run time. Resources should first be granted to high priority tasks in order for it to produce faster response. Priorities can be assigned to tasks statically at design time, or they may be dynamically assigned at run time.

Real-Time Scheduling

How to assign priorities to multiple tasks is the *real-time scheduling* problem and has been an active research area for more than 20 years, starting with the seminal work by Liu and Layland [50]. The goal of real-time scheduling is to come up with a set of priority assignment rules to fulfill timing requirements.

Real-time scheduling algorithms typically make some assumptions on tasks and resources. For example, Liu and Layland's original work makes the following assumptions:

- a single and arbitrarily preemptabe resource (CPU);
- independent tasks;
- fixed and known task execution time;
- periodic task triggers;

- run-ability constraints — i.e. each task must be completed before receiving the next trigger.

Under these assumptions, they derived rate-monotonic scheduling (RMS) and earliest-deadline-first (EDF) algorithms. These algorithms are shown to be optimal (in terms of CPU utilization) for static and dynamic priority assignments. Further work in this area developed more sophisticated timing analysis theories and have relaxed many assumptions in their algorithms [48], [4], [75] [68], [79]. However, most of them still rely on tasks' worst case execution time (WCET) and arbitrary preemptability. Optimal scheduling for multiple resources has also been shown to be NP-complete [8].

In reality, many of these assumptions in scheduling theories do not hold. Tasks may require multiple resources to execute, and they can be strongly coupled. Priority-based real-time programming can be very subtle. One problem example is the priority inversion phenomena.

Priority Inversion

The intuition of assigning priorities to tasks is to prioritize resource utilization and obtain fast response for more critical tasks. However, because of the partial preemptablilty of some resources, blindly following the priority assignment and triggering high priority tasks may cause problems. The *priority inversion* problem breaks the independent task assumption such that a high priority task may be preempted by low priority tasks indefinitely.

Consider the following situation where there are four tasks, A, B, C, and D with

decreasing priorities, i.e A has the highest priority, while D has the lowest one. Tasks A and D share a critical data section s. Suppose at some time instant, D is the only eligible task, and starts to execute. During its execution, it grabs a lock on s and writes to it. Suppose now that the task A is triggered. Since s is locked by D, the execution of A is blocked. Mean while, task B, which does not require resource s, may be triggered. From the view point of task B, D's writing to s is merely some CPU, bus, and memory operations, thus it is preemptable. So, B may preempt D and start executing, which in turn preempts task A. In fact, B and C can be alternatively triggered, and their executions can preempt D from releasing s for an arbitrarily long time. The result is that A is blocked for an arbitrarily long time, even though A has a higher priority than B and C, and A does not share a critical section with B and C.

Priority inversion problems are usually solved by the *priority inheritance* and *priority ceiling* protocols [60]. The basic ideas of these protocols are to look into the content of each tasks, analyze shared critical data sections at compile time, and for each data section, find the highest priority task that may access it. Call this highest priority value π. Then, if a lower priority task enters this section, the priority of the task inherits π, so that no task of priority lower than π can preempt it. When the task leaves the critical section, its priority drops back to its normal value. Priority inheritance and priority ceiling protocols successfully solve the priority inversion problem. And by adding a constraint that all tasks need to grab resources in the same order, it also solves the deadlock problem caused by cross waiting for resources. Thus, these protocols are widely implemented in real-time operating systems, like VxWorks [81], QNX [40], and resource kernel [61]. On the downside,

the priority inheritance protocol may be hard to implement, and it adds run-time overhead to monitor priorities and critical data sections. So, some light-weight real-time kernels omit them and ask software designers to take care of avoiding the priority inversion and deadlock problems themselves.

More Pitfalls

Using priorities as the only tuning factor and brute-force applying priority-driven preemptive run-time rules without considering the status of other tasks introduce many problems. Besides priority inversion problems, other pitfalls exist when the assumptions of real-time scheduling theories do not hold.

- Preemptive executions, especially with static priority assignment, are fundamentally fragile. Timing behavior of tasks may be very sensitive to the task triggering time and the accuracy of WCET estimation. An early arrived higher priority task can have a domino effect and make all subsequent low priority tasks miss their deadlines.

- Chasing fast response may not be optimal. Some hard-real-time algorithms may have strict requirements on the output time. An optimal result may only be achieved by emitting the output at a particular time. Early outputs may result in a sub-optimal (maybe even disastrous) result, as well as late outputs. This is particular the case for multimedia applications and some control systems.

- The results from schedulability analysis may not be very useful. The typical answer from schedulability analysis is the worst case response time between the triggering and the finish of the task. This value is required to be less than the deadline to

pass the schedulability test. However, schedulability analysis does not tell how often the worst case response time is met, what distribution it has, and what happens if it is bigger than the deadline. In many applications, missing a deadline occasionally may not cause catastrophic results. Schedulability theories are no longer applicable in these cases.

- The worst case execution time may not be the best representation of the execution time of a task. It could be much larger than the average execution time. And by using WCET for scheduling analysis, the real-time schedule could be very conservative. As a consequence, the resources are not best utilized.

An example illustrating these pitfalls is the situation called Rechard's anomalies described in [22]. It shows that for a particular set of (precedence-)dependent tasks and an optimal schedule, adding more processors, decreasing task execution times, or reducing the number of precedence constraints will all *increase* the overall response time.

5.3 Real-Time Responsible Frameworks

Two things make real-time programs different from non-real-time programs; one is prioritized execution and the other is the sensitivity of timing behavior. Of course, the first factor is merely an operational decision to achieve the latter. This section studies precise reactions in prioritized execution, and defines real-time responsible frameworks in terms of time determinism.

5.3.1 Prioritized Reactors

Whether or not an operation should be performed at run time may depend on two factors, which essentially impose partial ordering relations among operations:

- Data dependency: A reactor may require inputs from the physical world or results from other operations. These dependencies impose ordering relations such that some communication and computation must be performed before some others, as discussed in Chapter 2. In a run-time framework, some physical inputs are *unpredictable*. They are controllable by neither the framework nor any reactors. Waiting on these inputs may take arbitrarily long time.

- Resource dependency: Reactors need computational and communication resources to execute. For example, in a single CPU system, all the tasks must share the same CPU, which essentially sequentialize all the executions. In a multi-CPU or distributed system, the communication resource, either the bus or the network, must also be shared by some tasks. Choosing what operation to perform is a decision made by a framework. As stated before, resources can be preemptable, partially preemptable, or nonpreemptable. Arbitrarily preemptable resources impose no constraints on the operations that need them, while nonpreemptable resources impose ordering relations like the first occupant must release the resource before another occupant can get it. We will see that partially preemptable resources may introduce conflicts in ordering relations.

Granted with data and resources, a reactor can execute its firing set, which takes time to finish. To bias resource allocation to fulfill timing constraints, *priorities* can be

assigned to the firing set of a reactor. We consider a priority to be a natural number, such that the smaller the number is, the higher priority it represents. We denote by Π : **Oper** $\rightarrow$ $\mathbb{N}$ a function that gives the priorities of to operations. Assigning priority to a firing set essentially assigns this number to all operations in that firing set. So, for $f_1, f_2 \in \texttt{A.fire}|_{\texttt{r}}$, $\Pi(f_1) = \Pi(f_2)$. A reactor, whose firing sets are assigned with priorities, is called a *prioritized reactor.*

5.3.2 Prioritized Precise Reactions

Priorities define ordering relations for operations that share the same resource. For example, let A and B be two reactors, competing for a preemptable resource. Let trigger r, activated at time t, be a responsible trigger for reactor A in the data dependency sense (as discussed in Chapter 2), and trigger r', activated at $t < t' < T(\underline{r})$, be a responsible trigger for reactor B in a data dependency sense[2]. Also, assume that firing set $\texttt{A.fire}|_{\texttt{r}}$ has priority π_A, and firing set $\texttt{B.fire}|_{\texttt{r}'}$ has priority $\pi_B < \pi_A$. Thus, reaction $\texttt{B.fire}|_{\texttt{r}'}$ has a higher priority. Since all operations in a real-time execution are associated with real time stamps, the time instant t' essentially partitions the operations in $\texttt{A.fire}|_{\texttt{r}}$ into two subsets: A_1 and A_2, such that $\forall f \in A_1, T(f) < t'$ and $\forall f \in A_2, T(f) \geq t'$. Then, the preemptive execution, depicted in Figure 5.4, has the following ordering relation:

$$\forall f \in A_1 \text{ and } \forall f' \in \overline{\texttt{B.fire}|_{\texttt{r}'}}, f \prec f'$$

$$\forall g \in A_2, \underline{r'} \prec g$$

Suppose that the resource shared by $\texttt{A.fire}|_{\texttt{r}}$ and $\texttt{B.fire}|_{\texttt{r}'}$ is not preemptable,

[2]Recall that $\underline{r}$ is the `finish` operation in $\overline{\texttt{A.fire}|_{\texttt{r}}}$.

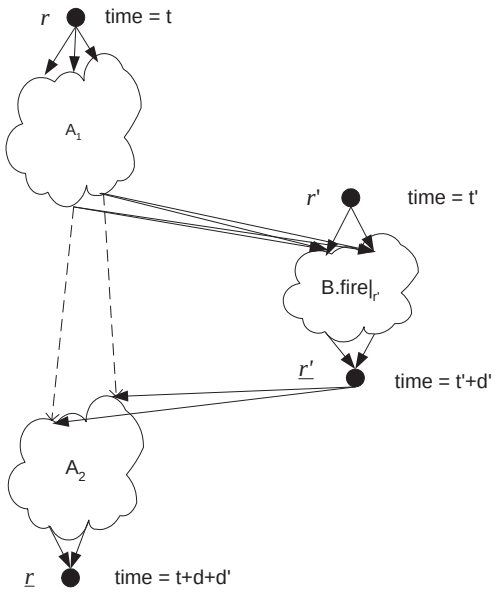

Figure 5.4: An ordering diagram for preemptive execution. The dashed arrows indicate the ordering relation without the preemption. Numbers d and d' are the execution time of reactor A and B, without preemption.

then the execution of $\texttt{B.fire}|_{\texttt{r}'}$ cannot start before the finish of $\texttt{A.fire}|_{\texttt{r}}$. This ordering relation, shown in Figure 5.5, can be written as:

$$\forall f' \in \texttt{B.fire}|_{\texttt{r}'}, \underline{r} \prec f'$$

Now we show that partially preemptable resources may introduce conflicts in the execution order. Suppose reactor A and C needs resource a and b, and reactor B needs resource a only. In addition, suppose b is nonpreemptable for A and C, but is preemptable for B. Let priorities $\pi_A > \pi_B > \pi_C$, and triggering time $T(r) < T(r') < T(r'')$. As shown in Figure 5.6, we get a set of ordering relations like:

$\underline{r'} \prec \underline{r}$, since $\texttt{B.fire}|_{\texttt{r}'}$ preempts $\texttt{A.fire}|_{\texttt{r}}$

$\underline{r''} \prec \underline{r}$, since $\texttt{C.fire}|_{\texttt{r}''}$ preempts $\texttt{B.fire}|_{\texttt{r}'}$

$\underline{r} \prec \underline{r''}$, since resource b is not preemptable for A and C.

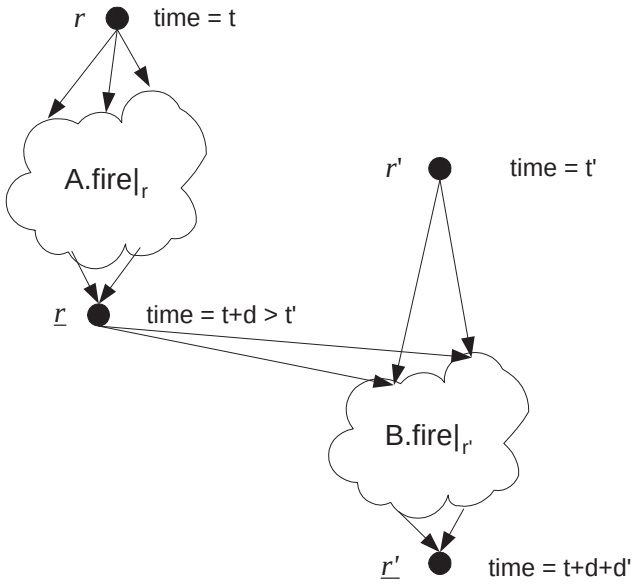

Figure 5.5: An ordering diagram for nonpreemptive execution.

Obviously, these indicate a conflict. So, we have,

Proposition 5.1. *Partially preemptive resources may introduce conflict in prioritized reactions.*

Thus, in order to guarantee that a set of reactors are prioritized precise reactive, the resources they use must either be disjoint, arbitrarily preemptable, or nonpreemptable, but cannot be partially preemptable. This also implies the following corollary.

Corollary 5.1. *In a single CPU system, resources should either be arbitrarily preemptive or nonpreemptive to guarantee prioritized precise reactions.*

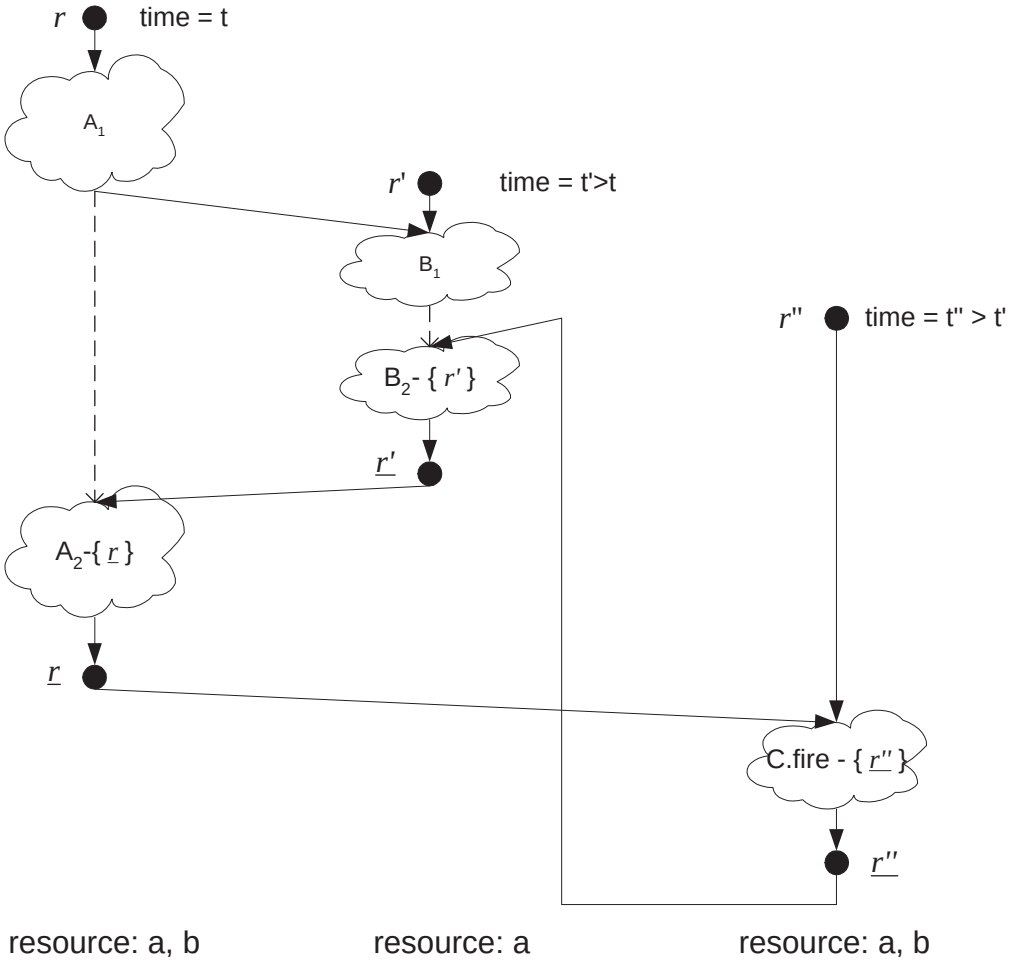

Figure 5.6: An ordering diagram for partially preemptive execution that introduces a conflict.

5.3.3 Time Determinism and Value Determinism

Although priorities order the operations within a run-time composite actor, they do not directly determine the timing properties of reactions. A fundamental problem of common real-time programming models is the isolation of functionality and timing concerns. It assumes that designers can first code the functionality of all tasks, and rely on later priority tuning to achieve timing properties.

Timing should be considered as an intrinsic property for real-time algorithms. When designing algorithms for embedded systems, designers usually expect that result of the computation to make effect at certain time instants. The time instants are defined as either absolute points or relative to some other events and reactions. To formally characterize the behavior of real-time actors, we introduce the concepts of time determinism and value determinism.

A run-time composite actor can be viewed as a discrete-event system, which responds to a set of (real-)timed events and produces a set of (real-)timed events. Since it is not always possible to view the computation of a run-time actor as reactors, we characterize their real-time behaviors in terms of inputs and outputs.

Intuitively, a composite actor should "produce right values at right time." These "right value" and "right time" are denotational requirements imposed by the physical dynamics and the algorithms, instead of the computing resource utilization and scheduling strategies. In order to capture these intuitions, we introduce the following definitions.

Definition 5.1. *Let A be a real-time actor, and, in response to a input signal $I = \{(\tilde{t_i}, \tilde{v_i}), i \in \mathbb{N}\}$, A should conceptually produce outputs $E = \{(t_i, v_i), i \in \mathbb{N}\}$. In reality, an execution of A may produce outputs $E' = \{(t'_i, v'_i), i \in \mathbb{N}\}$. Then, the execution is called* ***time deterministic*** *if $t_i = t'_i, \forall i$. The execution is called* ***value deterministic*** *if $v_i = v'_i, \forall i$.*

According to this definition, E is a set of "desired" outputs that is defined denotationally by the physical constraints, and E' is the set of operational outputs, subject to computing resources and task scheduling. Ideally, an execution should be both time and value deterministic, then the operational semantics is consistent with the denotational

semantics. However, a real execution may only approximate the denotational semantics, in the sense that it may sacrifice one or two aspects of the desired results. For example, in typical best-effort execution models in RTOS, time determinism is probably always sacrificed. But there are also other models that take different trade-offs.

5.3.4 Real-Time Responsible Frameworks

The activity of a run-time composite actor is a composition of the run-time framework and the reactors under its control. This inevitably requires that the value and timing constraints of the composite actor be transferred into the value and time constraints in individual reactors and requirements on the framework.

In order to make timing properties part of the semantics of reactors, we give each reactor time stamps for their triggers and corresponding finish operations. So, at the programming model level, a *real-time reaction* is a reaction whose trigger and finish operations are associated with time stamps, called the *baseline* and the *deadline*, respectively. These time stamps are declarative properties of the reaction, and their values are typically resolved from the physical constraints and the algorithms implemented. A run-time framework should allocate resources and triggers so that reactions are executed between corresponding baselines and deadlines. Figure 5.7 shows that the reaction of an actor is bounded by its baseline and deadline.

At run-time, depending on the resources and task priorities, these declared timing constraints may not always be fulfilled. We call a reaction *real-time precise*, if the real time stamps of its trigger and finish coincide with the declared baseline and deadline. If the outputs are produced just before the finish of the task, real-time precise reactions make

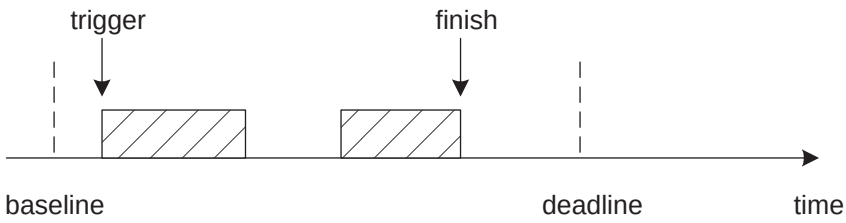

Figure 5.7: A real-time reaction is bounded by its baseline and deadline.

time and value determinism of a program relatively easy to analyze. If all the executions in a run-time framework are precise, in terms of data and resource dependencies, then the timing constraint of the composite actor can be translated into the constraints on the baseline and deadline of individual reactors.

The job of a *real-time responsible framework* is to manage resources and triggers such that each reactor fulfills its baseline and deadline. A real-time responsible framework needs to interrupt at the output time and perform `write` operation for the composite actor. These timed interrupts should have the highest priority and preempt any on going tasks. Ideally, the reaction that produces the output should have finished by this time. A responsible framework may enforce this by properly allocating resources and forcing all component to complete by their deadlines. By the compositionality of deadlines, the response time of the composite actor is guaranteed.

However, the real execution time may vary from reaction to reaction, and the physical event may not happen exactly at the expected time. Thus, the baselines and deadlines of some reactions may inevitably be violated. There are different approaches to reconcile these problems. We describe two models in this section and introduce a new programming model in the next section.

Giotto

Giotto [30] is a time-triggered programming model, where each task has a well-defined (usually periodic) starting time (baseline) and stopping time (deadline). So all triggers are defined with respect to the real time. Between the starting time and the stopping time, the execution of the task shares the resources with other tasks. But the output are only produced when the deadline is reached. A Giotto scheduler, using an estimated WCET of each task, makes sure that all tasks can finish before the deadline, otherwise it will reject the set of tasks. So, if the WCET time is accurate or loose enough, a Giotto program will always be time-and-value deterministic at run time.

Port-based object

The port-based object (PBO) [69] model is also time triggered. But unlike the Giotto model, it only has time-triggered starting point, but does not have time-triggered outputs. It uses shared variables and state semantic communications to solved the execution time miss-matching. For example, there are three tasks shown in Figure 5.8 — the sensor task, the computation task, and the actuator task. Suppose that all tasks are triggered at 10Hz, but there is a phase difference. That is, the sensor task is triggered at $0, 100ms, 200ms, ...$; the computation task is triggered at $10ms, 110ms, 210ms, ...$; and the actuator task is triggered at $70ms, 170ms, 270ms, ...$

The communication represented by the arcs are shared variables. Ideally, the sensor has $10ms$ to update its output, so that it can be used by the computation. The computation task has $60ms$ to compute the actuation, so the output task can read a new

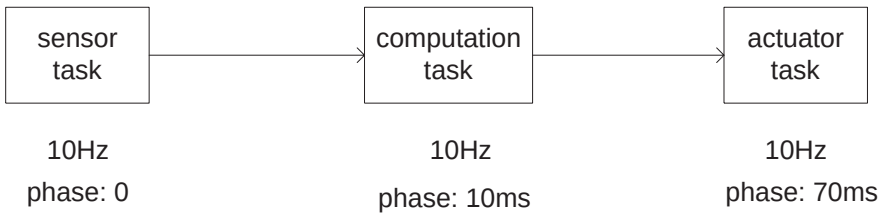

Figure 5.8: Three tasks in the PBO model.

value for every $100ms$. However, if the sensor task misses it $10ms$ deadline, the computation task will use the sensor reading from the last cycle. Similarly, if the computation task misses it deadline, the actuator will simply repeat its last output. Thus, a PBO model is time deterministic—since it always produces something at the output time. But it may not be value deterministic.

5.4 Timed Multitasking Model of Computation

In this section, we introduce a real-time model of computation that embraces the concepts of real-time precise reaction and real-time responsible frameworks. By using this programming model, designers think in terms of both functionalities and I/O time requirements. These I/O timing requirements are preserved by a run-time system — a real-time responsible framework. Thus the execution is time-and-value deterministic, as long as there are sufficient system resources. If there are not enough resources at run time, it preserves time determinism.

5.4.1 Programming Concepts

The basic concepts in this programming model are reactors, resources, and overrun handlers. The data dependencies in a program is handled by tasks and responsible triggers. The resource dependencies in a program are handled by resource and schedulability analysis. The possible misses of deadlines are handled by overrun handlers.

Reactors

The software components in this model are prioritized precise reactors, i.e. their triggering rules allow them to perform a finite computation without further requirements. The communication among the reactors have the event semantics and are typically implemented by FIFO queues. The output side of a communication is never blocked on writing. And a trigger is only activated when there are enough data to complete a reaction. If state semantics is needed by the application, the reactor developers need to code it within the reactor code, like consuming all the inputs and using the last one.

The triggers are the baselines for the reactions. Triggers can be expressed in terms of (possibly combinations of) real time, physical events, and internal operations. Tasks have deadlines, expressed in terms of absolute real time. A trigger of a reactor must be responsible, so that once the execution of the task starts, it does not wait on further unpredictable inputs.

By annotating the timing information, designers know exactly what time delays their programs will introduce at run-time. And this delay will be preserved at run time.

Resources

A reactor may need one or more resources — preemptable or nonpreemptable — to execute. In a TM program, these resources are annotated to the tasks to help schedulability analysis. Following the discussion of prioritized precise reactions in section 5.3.2, we treat the execution of a reactor to be preemptable only if all the resources it needs are preemptable by all other reactors with overlapping resource requirements. So, reactions are either arbitrarily preemptable or nonpreemptable. This essentially prevents partially preemptable resources and avoids priority inversion problems.

A reaction has an execution time, which is the declared nonpreempted execution time when all the resources required by the reaction are available. This execution time may not necessarily be the worst case execution time, if the task can provide meaningful overrun handlers.

Over-run Handling

Overrun handlers are nonpreemptable piece of codes that are triggered by the run-time framework when the corresponding tasks are about to miss deadlines. Unlike the Giotto model, the execution of a TM model is event triggered. In general, it is impossible to guarantee that all deadlines of all tasks can be met. By having these overrun handlers, the TM model can preserve time determinism when resources are not sufficient at run time.

Providing overrun handlers also gives the TM models the possibility of better resource utilization than traditional WCET-based models. In many applications, the WCET may be much longer than typical execution time. Real-time scheduling based on WCET

may lead to a very low resource utilization. The TM scheduling based on typical execution time can improve resource utilization on the average cases, and the overrun handlers deal with exceptional long executions.

In summary, the semantics of the model is that if there are enough resources at run-time, then each reactor will be granted with the declared resources for at least the declared execution time, before the deadline is reached. The output of the execution is only made available to other tasks and the outside physical world when the deadline is reached. This is also called *faster-than-real-time computation* in some literatures [71]. If a task has not finished by the deadline, the task will be stopped and the overrun handler will be triggered. Notice that this semantics does not directly specify the priorities of any tasks. Typically, there may be multiple priority assignment policies that can fulfill the timing requirements. And for any feasible scheduling policy, the execution result of a model is exactly the same, in terms of time and value determinism. In this sense, the TM model is immune to scheduling policies.

5.4.2 Execution Model

The execution model of TM programs is a stylized use of priority-based multitasking execution, as seen in most real-time kernels.

Through compilation, the reactors in the TM model are classified (and possibly merged) into a set of preemptable and nonpreemptable tasks, depending on their resource requirements and triggering rules, similar to the techniques described in [72]. A task may be in one of three states: `idle`, `ready`, or `active`. A task is `idle` if it is not triggered. After being triggered, a task is `ready` to execute, but may wait for resources. When there are

resources available, a ready task may start executing and become an `active` task. An active task may be preempted by another ready task if the ready task has a higher priority and the active task is preemptable. A run-time system typically manages three pools, corresponding to the three states of the tasks, as shown in Figure 5.9.

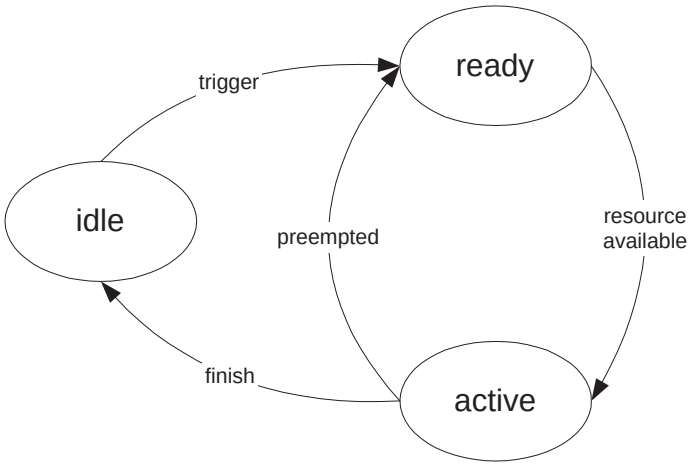

Figure 5.9: Three tasks pools in typical RTOS kernels.

The triggering rules of the reactors are the guards that bring tasks from the idle pool to the ready pool. The predicates on the triggering rule tells the run-time framework when these rules should be examined. At run time, some of these predicates, like those regarding physical events, are implemented as interrupts for the run-time framework.

The priorities of tasks in the `ready` pool and the preemptiveness of the tasks inside the `active` pool determine which task will be moved to the `active` pool. Typically, as long as the tasks are in the `ready` pool, their priorities never change. The priorities may change for each time a task is moved from the `idle` pool to the `ready` pool. Priorities can be statically or dynamically assigned. Dynamic priority assignment usually introduces more run-time overhead than static priority assignment.

The run-time system for TM programs strictly obeys the deadlines for each task. It keeps track of the deadlines for all tasks. And when the deadline is reached, it asks the task to produce its outputs. If at the deadline time, the task is still in the `ready` or `active` pools, the run-time framework will terminate the task and call its overrun handler. The overrun handler's execution are nonpreemptable. A terminated task is put back to the `idle` pool.

5.4.3 Implementation

The TM domain in Ptolemy II implements a very preliminary version of the timed multitasking model of computation. In this domain, actors (conceptually) execute as concurrent threads in reaction to inputs. Actors need to be designed in the way that each input event is a responsible trigger in the data dependency sense. Resources are assumed to be arbitrarily preemptable and actors are statically assigned with `priorities`. An actor specifies a `executionTime`, which is the amount of time for the reaction to complete. Specification of `deadline` and scheduling analysis have not been implemented, and we assume that actors can always finish their execution within the specified execution time.

The `TMDirector` provides an event dispatcher, which maintains a prioritized event queue. The execution of an actor is triggered by the event dispatcher by invoking first its `prefire()` method. The actor may begin execution of a concurrent thread at this time. Some time later, the director will invoke the `fire()` and `postfire()` methods of the actor (unless `prefire()` returns false).

The current implementation only supports one shared resource, the CPU. At one particular time, only one of the actors can get the resource and execute. Execution of one

actor may be preempted by another eligible actor with a higher priority input event. If an actor is not preempted, then the amount of time that elapses between `prefire()` and `fire()` equals the declared `executionTime`. If it is preempted, then it equals the sum of the `executionTime` and the execution times of the actors that preempt it. The current implementation of the TM domain in Ptolemy II only simulates the execution of a TM model of computation, under ideal assumptions. An implementation of a programming environment, together with a run-time framework, that fully support real-time precise reactions is part of future work.

5.5 Examples

We give two examples in this section to illustrate the use of the TM domain. The first example is a simulated control system, where the controller is implemented in the TM model. The second example shows a multitasking execution under the Java^{TM} virtual machine (JVM)[3]. It uses thread schedulers within the JVM to approximate a TM run-time environment.

5.5.1 Shared Resource Controllers

This example, as shown in Figure 5.10, shows two independent controllers sharing the same computation resource.

The top level is a CT model, implementing two continuous dynamic systems,

[3]Java^{TM} is a trademark of Sun Microsystems, Inc.

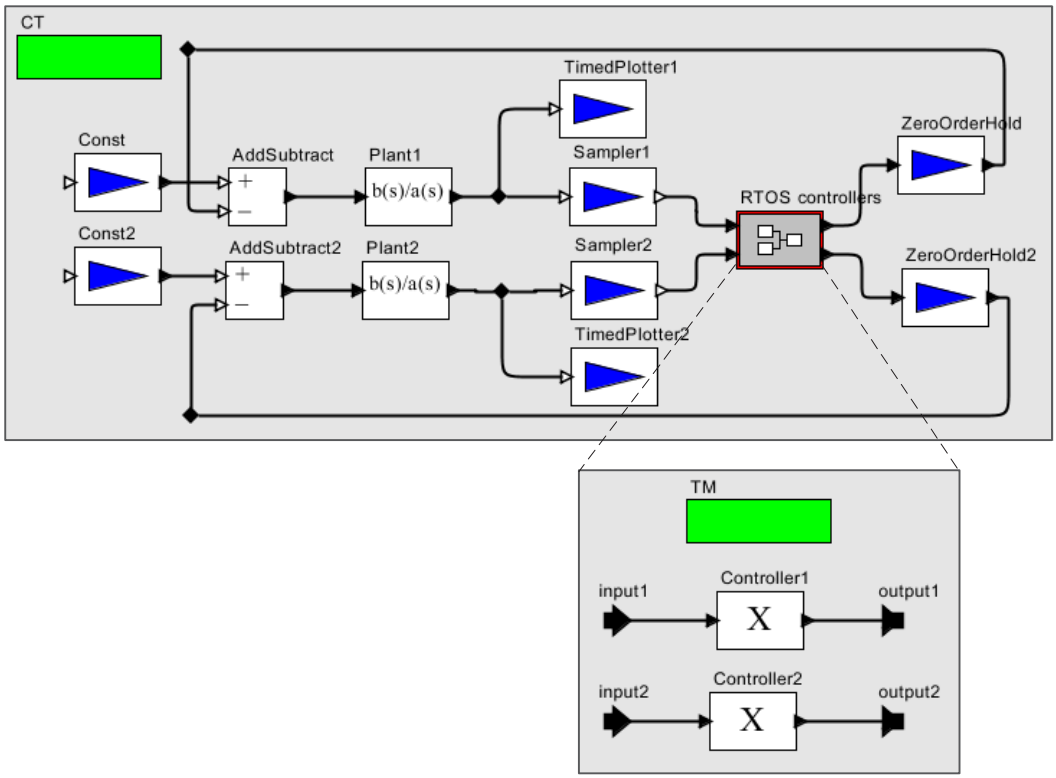

Figure 5.10: Two controllers sharing a computation resource.

`Plant1` and `Plant2`. In this case, the Laplace transfer functions of both plants are,

$$\frac{1}{s^2 + s}. \tag{5.1}$$

The sampling rates are different, so that the triggers of the two controllers are not perfectly aligned.

Two discrete controllers, implemented in the TM domain, share a resource — the CPU. Due to the priorities of the tasks, and the execution policies, each controller may introduce a delay in its reaction. So, the actual delay of a task may not be the execution time it has specified, unless it has the highest priority and the execution is preemptive. The continuous plants have well adjusted parameters such that if the delay is too long, the

Table 5.1: Experimental parameters for the shared resource controllers

plots	**priority1**	**priority2**	**preemptiveness**
(a)	high	low	preemptive
(b)	low	high	preemptive
(c)	high	low	nonpreemptive
(d)	low	high	nonpreemptive

system becomes unstable.

The executions, whose parameters are listed in Table 5.5.1, are shown in Figure 5.11. The results indicate the following observations:

- Real-time execution policies may have big impact on the closed-loop performance of an embedded system. Suppose that the execution times specified are the real execution times of a controller, then depending on the implementation, like priority assignment, we actually get dramatically different results.

- Preemptive execution policies are not necessarily better than nonpreemtive execution. Real-time systems have to be considered in terms of overall performance, rather than sacrificing one control loop for the other.

- For a TM model, given that there are enough resources for tasks to finish before the specified execution time, then a TM run-time can guarantee that the run-time behavior of the system is the same as the ones simulated. This is not true in general real-time programs, where the output delay depends on the finish time of the task, rather than the specified execution time.

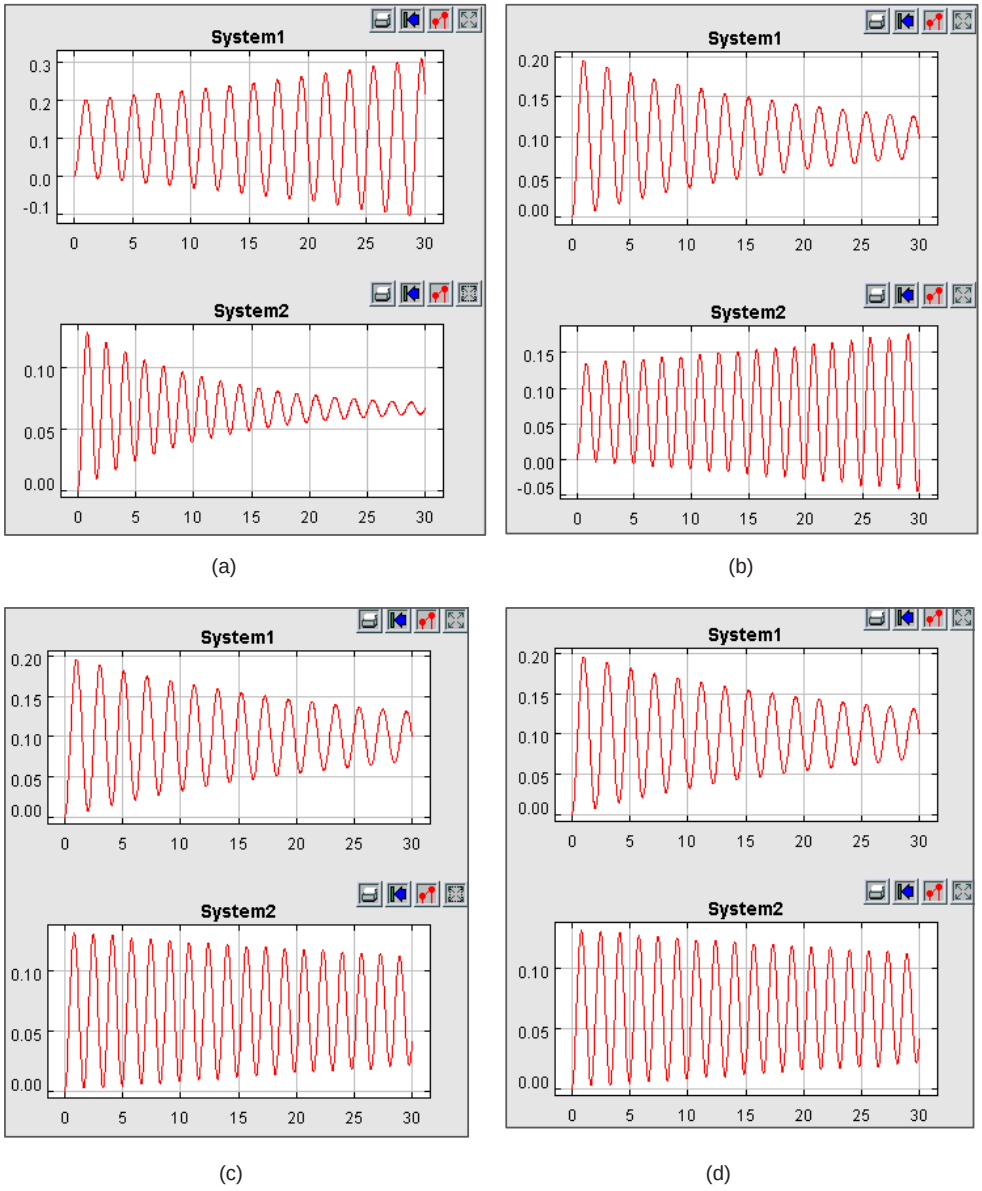

Figure 5.11: For preemptable executions, the control loop with low priority is unstable.

5.5.2 Background Processes

This example, Figure 5.12 shows the use of preemptable and nonpreemptable tasks in the TM domain. The model simply generates a noisy sine wave and performs spectrum analysis using an FFT algorithm.

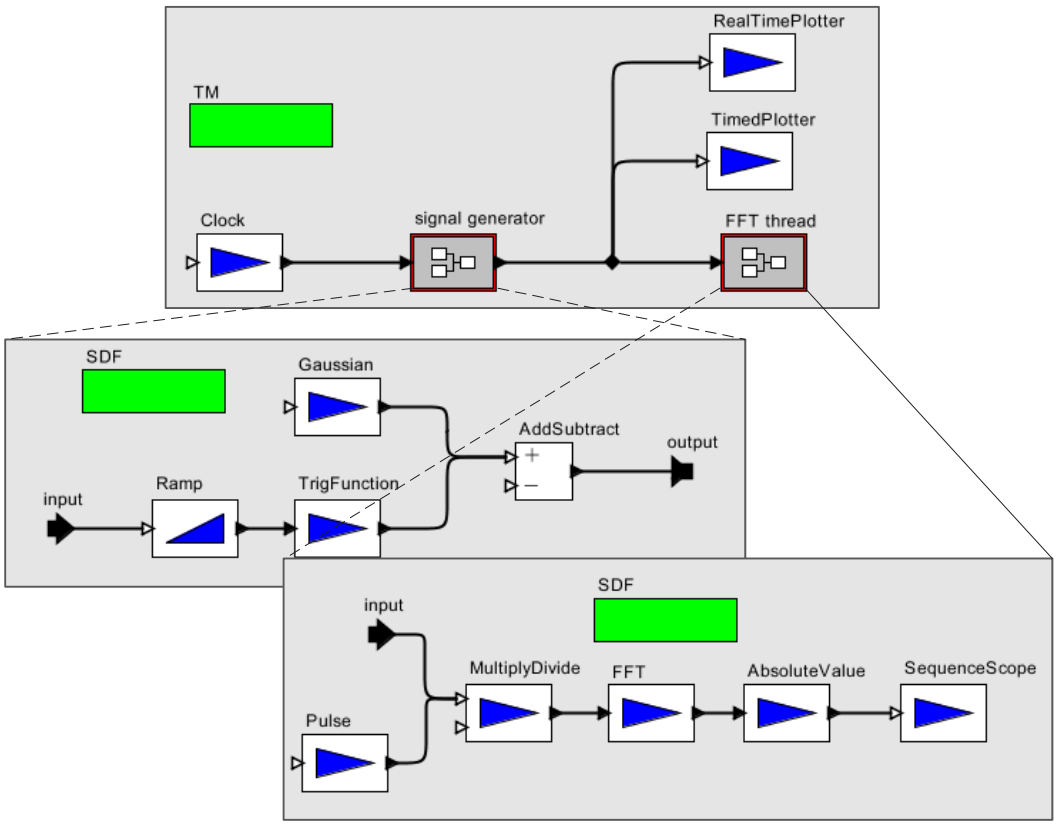

Figure 5.12: Preemptable and nonpreemptable tasks in a TM model.

There are two composite actors in the model. The one labeled `signal generator` is a nonpreemptable task, and the other labeled `FFT thread` is a preemptable task. The nonpreemptable task is executed in the event dispatcher thread, while the preemptable task is executed in a separate thread. Both of these composite actors are internally implemented

using the SDF model. In the example, the `signal generator` actor has priority 5 and execution time 0.0001, which can basically be ignored. The `FFT thread` composite actor has the execution time set to 0.25 seconds, and we explore the effect of its priority and the JVM thread scheduling. The execution is preemptive

The execution results with simulated time are shown in Figure 5.13, where `FFT thread` has a high priority, and in Figure 5.14, where `FFT thread` has a low priority. Remember that the logging may be affected by the FFT process, which contains a large chunk of computation. It is obvious that when `FFT thread` has a low priority, it does not block the logging process.

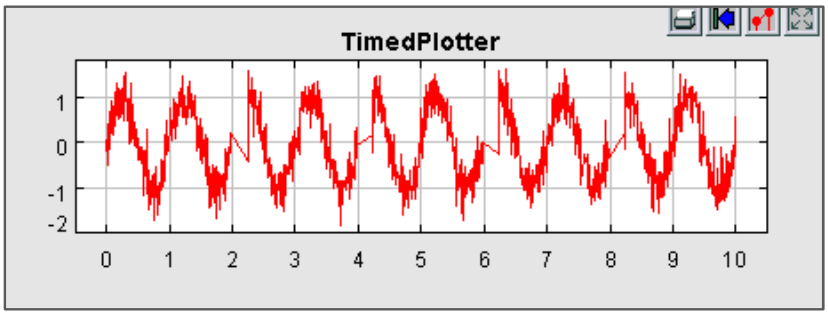

Figure 5.13: Execution result of the background process example, when `FFT thread` has a high priority.

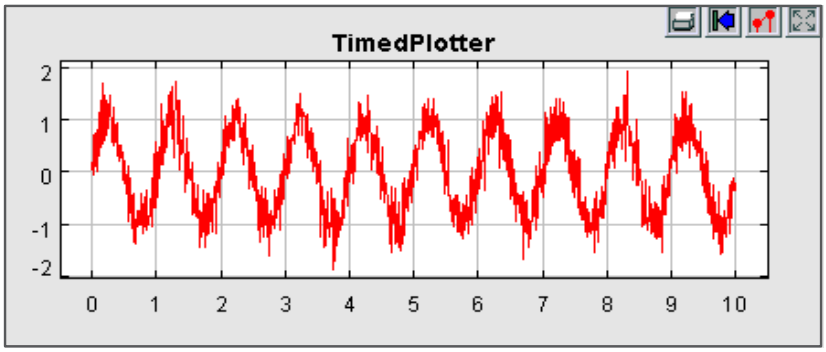

Figure 5.14: Execution result of the background process example, when `FFT thread` has a low priority.

How to map the modeling time to real time in a run-time framework is the next question. A correct run-time framework for this model should arrange system facilities such as timers, locks and semaphores, and scheduling policies to produce exactly the same behaviors at run time. There are possibly many choices to implement such a run-time system. An obvious one is to build the run-time system on top of a hard-real-time operating system, which already provides high-accuracy timers, preemptive multitasking, and resource locks. For resource-rich systems, where the computation power is cheap but the operating system is not so "real-time", a faster-than-real-time strategy may be more applicable. Accurate timing behavior can be achieved by the help of smart sensors and actuators.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This dissertation studies modeling and design of heterogeneous embedded systems. The approach taken to tame heterogeneity in embedded systems is a component-based hierarchical one. An embedded system is considered as an aggregation of components under a framework. Frameworks can be hierarchically composed to combine different semantic models. I focus on abstract semantic properties related to reactivity and its composition. The key issue I address in the dissertation is

what frameworks are "good"?

Given the reactive nature of embedded systems, a good framework should preserve the reactivity of each component, and make this reactivity composable. These ideas are captured in the concepts of *precise reactions* and *responsible frameworks*.

In Chapter 2, I present the reactor model and define precise reactions, atomic reactions, responsible triggers, and responsible frameworks. The reactor model is an abstract operational semantic model that uses partial order relations among operations to capture computation, communication, and flow of control in component-based frameworks. Operations may be grouped into reactions, which are triggered by the framework according to some triggering rules. A reaction is precise if it can be finished completely in a compositional execution. A responsible trigger, which summarizes all the preconditions for a reaction, can always guarantee a precise reaction. A framework is responsible if it requires all triggering rules to be responsible and triggers reactions accordingly. Responsible frameworks have many good properties, like preservation of quiescent states, detectable deadlocks, and sequentializable execution. I also show that among commonly used frameworks, like CSP, PN, and dataflow, some are responsible, while some are not.

In Chapter 3, I consider the compositionality of precise reactions. The goal is to allow a composition of actors in a framework to behave like an atomic actor in a higher level model. I show that responsible frameworks help achieve compositional precise reactions and precise mode switching.

Embedded systems typically have a notion of time, and need to be modeled in timed frameworks. Chapter 4 is devoted to responsible timed frameworks. In fact, the notion of a continuous time helps define precise reactions. In the study of continuous-time frameworks, I recognize the importance of precisely controlling the integration step sizes to obtain responsible triggers to all continuous, discrete, and hybrid components. The study yields simulation techniques for mixed-signal systems, which are compositions of CT and

DE frameworks, and hybrid systems, which are compositions of CT and FSM frameworks. The results in these three chapters also provide insights to formally integrate modeling and simulation tools. In [53], we have claimed that not all tools can be integrated in *ad hoc* ways. Tools need to expose enough semantic information to be used by other tools. The study in these chapters indicates that this semantic information is exactly how an invocation of one tool can finish a precise reaction.

In Chapter 5, I further apply responsible frameworks to real-time systems, where the reactions not only have a physical notion of time, but also have a notion of *priority*. I show that irresponsible triggers in a priority based execution environment may introduce the priority inversion problem. After analyzing time determinism and value determinism in run-time frameworks, I proposed a *timed multitasking* (TM) model of computation for real-time embedded software. This model has the notion of time and resources at the programming level and relies on a responsible real-time framework to preserve it at run time.

6.2 Future Work

6.2.1 Formal Semantics for Component-Based Design

Component-based approaches, with promising properties like composability, scalability, and reusability, have great potential in modeling and design technologies for embedded systems. To understand these promising properties, formal semantic models are required. Among various recent achievements, interface theories [17] formalize component interfaces and their refinements, and behavior type systems [47] formalize dynamic behavior

of components and frameworks into a type system. This work is a starting point towards *framework theories*, which will formalize the dynamics and compositionality of frameworks.

6.2.2 Run-Time Frameworks

The studies of time determinism, value determinism, and real-time responsible frameworks suggest that we can have better approaches to ensure timing properties in embedded software than the current RTOS-based methodologies. Essentially, from high-level models, we can *generate* real-time frameworks that meet the specific timing and resource constraints in the application. Furthermore, the idea of run-time frameworks should not be limited by computer boundaries. A run-time framework may cross many distributed computation platforms and communication channels to achieve coordinations among large-scale systems. The compositionality studied in this dissertation will help hierarchically manage these frameworks to achieve further scalability and determinism [51]. Hierarchical run-time frameworks may be particularly useful in applications like distributed control systems, sensor networks, and real-time information processing.

6.2.3 Software Synthesis for TM Models

A next step for the work on timed multitasking models is to synthesis run-time software from TM specifications. Some ideas are highlighted in the following.

We only consider single processor platforms at this time. The software synthesis has three steps — trigger analysis, resource analysis, and timing analysis.

Trigger analysis

The trigger analysis step looks at the triggering rules of each reactors. It separates physical events, including timers, from internal triggers. It builds an interrupt table for physical events and maps the interrupts to trigger predicates. It also builds a table of what operations should be monitored, together with the corresponding trigger predicates.

Resource analysis

The resource analysis step looks at the resource annotation on each reactor, and analyzes the preemptability of its execution. The results of resource analysis are a set of arbitrarily preemptable tasks and a set of nonpreemptable tasks. In addition, the overrun handlers for the reactors are treated as nonpreemptable tasks.

Timing analysis

The timing analysis step looks for a real-time scheduling policy, static or dynamic, to fulfill the timing requirement of the run-time composite actor, based on the execution time and deadlines of individual reactors. In general, finding an optimal schedule may be an NP-hard problem. However, since the semantics of the model relies on the denotational timing properties instead of task execution times, a sub-optimal scheduling is acceptable as long as it fulfills the timing requirements.

Optimization

Further optimization may be performed to improve timing properties and reduce run-time overhead. For example, it is desirable to merge strongly connected tasks. Strongly

connected tasks are a set of preemptive tasks that can be assigned the same priority, and where only one of these tasks are triggered by external events. It is sometimes desirable to split a task into smaller tasks to improve reactivity, since in the TM model, the outputs are only made available at the deadline time.

6.3 Final Words

三十辐，共一毂，当其无，有车之用。
埏埴以为器，当其无，有器之用。
凿户牖以为室，当其无，有室之用。
故有之以为利，无之以为用。

—— 老子《道德经》

Thirty spokes meet at a nave;

Because of the hole we may use the wheel.

Clay is molded into a vessel;

Because of the hollow we may use the cup.

Walls are built to make a house;

Because of the emptiness we may use the room.

Therefore, what is present is used for profit;

But it is in absence that there is usefulness.

— Lao Zi, *Dao De Jing*,

dates uncertain. Speculations

ranges from 600 BC to 200 BC.

Bibliography

[1] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, MA, 1986.

[2] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[3] P. J. Ashenden. *The Designers Guide to VHDL.* Morgan Kaufmann Publishers, 1996.

[4] N. C. Audsley, A. Burns, M. Richardson, , and A. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings of IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, May 1991.

[5] 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, 1997.

[6] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *10th International Conference on Concurrency Theory (CONCUR'99), LNCS 1664*, pages 162–177. Springer-Verlag, 1999.

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

[8] J. Blazewicz, W. Cellary, R. Slowinski, and J. Weglarz. Schedulingunder resource constraints – deterministic models. *Annals of Operations Research*, 7, 1986. Baltzer Science Publishers.

[9] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language: User's Guide*. Addison-Wesley, 1999.

[10] V. Bryant. *Metric Spaces*. Cambridge University Press, 1985.

[11] 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,"*, 4:155–182, April 1994.

[12] Joseph T. Buck and Edward A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 429–432, Minneapolis, MN, 1993.

[13] Joseph T. Buck and Radha Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES'00)*, pages 142–146, San Diego, CA, May 2000.

[14] James C. Candy. A use of limit cycle oscillations to obtain robust analog-to-digital converters. *IEEE Transaction on Communications*, COM-22(3):298–305, March 1974.

[15] Avant! Corporation. Saber. http://www.avanticorp.com/product/1,1500,65,00.html.

[16] Avant! Corporation. VeriasHDL: Language-independent, mixed-signal simulator. http://www.avanticorp.com/product/1,1500,73,00.html.

[17] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211*. Springer-Verlag, 2001.

[18] Akash Deshpande, Aleks Gollu, and Pravin Varaiya. SHIFT: A formalism and a programming language for dynamic networks of hybrid automata. In *Hybrid Systems V, LNCS 1567*, pages 113–133. Springer, 1997.

[19] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces: Principles, Patterns, and Practive*. Addison-Wesley, 1999.

[20] C.W. Gear. *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice-Hall, Englewood Cliffs, NJ, 1971.

[21] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, 18(6):742–760, June 1999.

[22] R.L. Graham. Bounds on the performance of scheduling algorithms. In *Computer and Job Scheduling Theory*, pages 165–225. John Wiley and Sons, N.Y., 1976.

[23] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The Ninja architecture for robust internet-scale systems and services. *Computer Networks on Pervasive Computing*, 35(4):473–497, 2000.

[24] IEEE 1076.1 Working Group. VHDL 1076.1-1999: Analog and mixed-signal extensions to VHDL, 1999.

[25] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[26] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[27] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO ASI Series*, pages 477–498. Springer-Verlag, 1985.

[28] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proceedings of the Symposium on Logic in Computer Science*, pages 54–64, June 1987.

[29] Thomas L. Harman and James B. Dabney. *Mastering Simulink 4*. Prentice Hall, 2001.

[30] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Embedded control systems development with Giotto. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'01)*, June 2001.

[31] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of the IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 245–252, 2000.

[32] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, April 1985.

[33] John Davis II. *Order and Containment in Concurrent System Design.* PhD thesis, Department of EECS, University of California, Berkeley, September 2000.

[34] John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong. Ptolemy II: Heterogeneous concurrent modeling and design in Java. Technical Memorandum UCB/ERL M01/12, EECS, University of California, Berkeley, March 2001.

[35] Open Verilog International. Verilog-AMS language reference manual 2.0, Jan 2000.

[36] J. Janneck. *Syntax and semantics of graphs - An approach to the specifcation of visual notations for discrete-event systems.* PhD thesis, Institut fr Technische Informatik und Kommunikationsnetze, ETH Zrich, Swaziland, June 2000.

[37] R. Johnson. Framework = (components + patterns). *Communications of the ACM*, 40(10):39–42, Oct. 1997.

[38] J.Vlach and A. Opal. Modern CAD methods for analysis of switched networks. *IEEE Transaction on Circuits and Systems-I: Fundamental Theory and Applications*, 44(8):759–762, 1997.

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

[40] Frank Kolnick. *The QNX 4 Real-time Operating System*. Basis Computer Systems, 2000.

[41] Hermann Kopetz. *Real-Time Systems : Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[42] Edward A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.

[43] Edward A. Lee. What's ahead for embedded software. *IEEE Computer*, 33(9):18–26, Sept. 2000.

[44] Edward A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, March 2001.

[45] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, Sept. 1987.

[46] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[47] Edward A. Lee and Yuhong Xiong. System-level types for component-based design. In *First Workshop on Embedded Software, EMSOFT2001, LNCS 2211*. Springer-Verlag, Oct. 2001.

[48] J.P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 166–171, Dec. 1989.

[49] M. A. Lemkin. *Micro Accelerometer Design with Digital Feedback Control*. PhD thesis, EECS, University of California, Berkeley, Dec. 1997.

[50] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 10(1):46–61, Jan 1973.

[51] Jie Liu, Stan Jefferson, and Edward A. Lee. Motivating hierarchical run-time models in measurement and control systems. In *2001 American Control Conference (ACC'01)*, pages 3457–3462, Arlington, VA, June 2001.

[52] Jie Liu and Edward A. Lee. A component-based approach to modeling and simulating mixed-signal and hybrid systems. submitted to ACM Trans. on Modeling and Computer Simulation, Special Issue on Computer Automated Multi-Paradigm Modeling.

[53] Jie Liu, Bicheng Wu, Xiaojun Liu, and Edward A. Lee. Interoperation of heterogeneous CAD tools in Ptolemy II. In *Symposium on Design, Test, and Microfabrication of MEMS/MOEMs*, March 1999.

[54] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, oct 1991.

[55] C. Moler. Are we there yet? Zero crossing and event handling for differential equations. In *Matlab News and Notes: Simulink2 Special Edition*, pages 16–17. The Mathworks Inc., 1997.

[56] Pieter J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *Hybrid Systems: Computation and Control (HSCC99), LNCS1569*, pages 165–177. Springer, 1999.

[57] Object Management Group (OMG). CORBA event service specification, version 1.1. Document formal/01-03-01, http://www.omg.org/cgi-bin/doc?formal/2001-03-01, March 2001.

[58] Carlos O'Ryan, Douglas C. Schmidt, and J. Russell Noseworthy. Patterns and performance of a CORBA event service for large-scale. *International Journal of Computer Systems Science and Engineering, CRL Publishing*, 2001.

[59] A. Pasetti and W. Pree. A component framework for satellite on-board software. In *IEEE/AIAA 18th Digital Avionics Systems Conference (DASC'99), Saint Louis, MI*, Oct. 1999.

[60] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[61] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shui Oikawa. Resource Kernels: A resource-centric approach to real-time systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, Jan. 1998.

[62] Arend Rensink and Heike Wehrheim. Dependency-based action refinement. In P. Ruzicka, editor, *MFCS'97 Mathematical Foundations of Computer Science, number 1295 in Lecture Notes in Computer Science*. Springer, 1997.

[63] R. A. Saleh and A. R. Newton. *Mixed-Mode Simulation*. Kluwer Academic Publishers, 1990.

[64] Douglas C. Schmidt, David L. Levine, and Chris Cleeland. Architectures and patterns for high-performance, real-time CORBA object request brokers. In Marvin Zelkowitz, editor, *Advances in Computers*. Academic Press, 1999.

[65] S. Senturia. Cad challenges for microsensors, microactuators, and microsystems. *Proceedings of the IEEE*, 86(8):1611–1626, Augest 1998.

[66] Lawrence F. Shampine and Mark W. Reichelt. The MATLAB ODE suite. *SIAM Journal on Scientific Computing*, 18(1):1–22, 1997.

[67] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[68] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems Journal*, 10(2):179–210, 1996.

[69] D. B. Stewart. Software components for real time. *Embedded Systems Programming*, 13(12):100–138, Dec. 2000.

[70] Morris Tenenbaum and Harry Pollard. *Ordinary Differential Equations*. Dover Pubns, 1982.

[71] David Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43–50, May 2000.

[72] F. Thoen, M. Cornero, G. Goossens, and H. De Man. Software synthesis for real-time information processing systems. In *ACM SIGPLAN Workshop on LCT for Real-Time Systems*, La Jolla, CA, June 1995.

[73] D. E. Thomas. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1998.

[74] Michael Tiller. *Introduction to Physical Modeling With Modelica*. Kluwer Academic Publishers, 2001.

[75] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems Journal*, 6(3):133–151, 1994.

[76] Rob J. van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4/5):229–327, 2001.

[77] Andrei Vladimirescu. *The SPICE Book*. John Wiley & Sons, 1993.

[78] M. von der Beeck. A comparison of Statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, September 1994.

[79] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *International Conference on Real-Time Computing Systems and Applications*, Dec. 1999.

[80] Linda Wills, Suresh Kannan, Sam Sander, Murat Guler, Bonnie Heck, J.V.R. Prasad, Daniel Schrage, and George Vachtsevanos. An open platform for reconfigurable control. *IEEE Control Systems Magazine*, pages 49–64, June 2001.

[81] Wind River Systems, Inc. *VxWorks Programmer's Guide*, 1997.

[82] Glynn Winskel. Event structures. In *Advances in Petri Nets, LNCS 255*, pages 325–392. Springer, 1986.

[83] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.