On the Design of Concurrent, Distributed Real-Time Systems

Yang Zhao

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-117
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-117.html

August 13, 2009

On the Design of Concurrent, Distributed Real-Time Systems

by

Yang Zhao

B.Eng. (Tsinghua University) 1997
M.Sci. (UC Berkeley) 2003

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, BERKELEY

Committee in charge:

Professor Edward A. Lee, Chair
Professor Ruzena Bajcsy
Professor Lee W. Schruben

Fall 2009

The dissertation of Yang Zhao is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2009

On the Design of Concurrent, Distributed Real-Time Systems

by

Yang Zhao

Abstract

On the Design of Concurrent, Distributed Real-Time Systems

by

Yang Zhao

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Edward A. Lee, Chair

Achieving determinism in distributed real-time systems is challenging, due to uncertainties in execution time, communication jitter, and resource scheduling. This dissertation presents a concurrent model of computation (MoC) for distributed real-time systems called PTIDES (pronounced tides, for Programming Temporally Integrated Distributed Embedded Systems). PTIDES uses a discrete-event (DE) model as the underlying formal semantics to achieve analyzable deterministic behavior.

PTIDES programs are discrete-event models constructed as networks of concurrent components, called *actors*, communicating via time-stamped events. These time stamps serve as the basis to define the unique order among events. Rather than using DE models for performance modeling and simulation, where time stamps are a modeling property bearing

no relationship to real time during execution of the model, PTIDES uses DE model as a specification language for real-time applications. It extends DE models with the capability of relating events that interact with the physical world with physical time.

Preserving DE semantics at runtime can be challenging, since the global, consistent notion of time may lead to a total ordering of execution in a distributed system, an unnecessary waste of resources. A dependency analysis framework is presented to allow out of order processing of events without compromising determinism and without requiring backtracking. The key idea is that if two events have independent affects, formally defined through causality analysis, then they can be processed in any order. As a result, if the earlier event is delayed due to communication, processing of the later event does not need to be blocked.

General event triggered real-time systems with multiple shared resources are not amenable to compile-time feasibility analysis [34]. However, when the discrete activities can come in predictable patterns, real-time scheduling theories are applicable to many PTIDES models. This dissertation studies a sufficient condition for a PTIDES model to be feasible when the inputs to a PTIDES model are sporadic, i.e. when there is a minimum interval between any two consecutive events of the same input.

Professor Edward A. Lee
Dissertation Committee Chair

To Ye, Karen, and my parents Xiumei Zhao and Shengli Bian.

Contents

List of Figures

Acknowledgements

I am deeply grateful to my adviser, Professor Edward Lee. His constant support and guidance over the years make the writing of this dissertation possible. I am also very grateful to Professor Shankar Sastry, Professor Ruzena Bajcsy, Professor Lee W. Schruben and Professor Raja Sengupta for serving on my qualifying exam and dissertation committees, and for providing valuable advice to my research.

It is my privilege to work with the past and present members of the Ptolemy group. Ye Zhou's work on interface theory provides a formal foundation that this thesis builds on. Xiaojun Liu's research on a semantic foundation for the tagged signal model and Jie Liu's work on timed multitasking model are great sources of ideas. Slobodan Matic, Jia Zhou and Thomas Feng's work on implementing the ideas presented in this thesis on different platforms are important to make it practical. My work has also benefited a lot from discussions and collaborations with Jörn Janneck, Gang Zhou, Haiyang Zheng, Adam Cataldo, Elaine Cheong and Eleftherios Matsikoudis. I would also like to thank Christopher Brooks, Mary Stewart and Ruth Gjerde for their excellent support.

Finally, I want to thank for the constant support and encouragement from my husband Ye Tang, parents Xiumei Zhao and Shengli Bian, sister Haijie Zhao, Qiao Zhao and my extended family.

Chapter 1

Introduction

Real-time systems are computer systems where the correctness of the system behavior depends not only on the logical results of the computation, but also on when the results are produced [25]. A real-time system is typically connected to sensors and actuators where the software reacts to sensor data and issues commands to actuators. The physical system is an integral part of the design and the software must be conceptualized to operate in concert with that physical system. Physical systems are intrinsically concurrent and temporal. Real-time software needs to fulfill not only a set of functional requirements but also timing constraints. Distributed real-time systems consist of multiple real-time systems interconnected by a communication network. Applications of distributed real-time systems include industrial automation, distributed immersive environments, advanced instrumentation systems, networked control systems, and many modern embedded software systems that integrate networking.

Despite the fact that both data values and time affect the behavior of real-time systems,

no widely used programming language integrates a way to specify timing requirements or constraints. Instead, the abstractions they offer are about scalability (inheritance, dynamic binding, polymorphism, memory management). Application designers are required to step outside the programming language semantics to specify timing properties of the applications. In typical real-time software design, time is often treated as an afterthought. The functionality is determined at design time with assumptions such as zero or a fixed non-zero run-time delay. The actual timing properties are determined at run time by a real-time operating system (RTOS) [34]. Typically, an RTOS provides mechanisms for prioritizing tasks. At run-time, multiple tasks often need to share resources including computing resources, like CPU and memory, and communication resources, like buses, networks and I/O. Whether a task can be granted resources to run largely depends on the hardware platform, what other tasks are running, and the relative priorities. In many real-time systems, this time uncertainty is undesirable.

Recent innovations in real-time software provide unconventional ways of programming concurrent and timed systems by introducing the notion of time and concurrency to the programming level. Giotto [19] and port-based objects [48] are examples that take a time-triggered approach, where software components are triggered by some clock signals. Time-triggered software has the advantage that all triggers are predictable in terms of time and timing analysis may be easy, but it does not fit as well for systems that need to respond to irregular activities. Timed Multitasking (TM) [34] takes an event-trigged approach that controls timing properties through deadlines and events rather than time triggers. By

specifying the deadline of each software component, the designer specifies when the computational results are produced to the physical world or to other software components.

This dissertation studies methods for programming distributed real-time systems where time and concurrency are first-class properties of the program. The approach in this thesis leverages the concept of actor-oriented design [29], borrowing ideas from Simulink and Giotto [19]. However, it addresses a number of limitations in Simulink and Giotto by building similar multitasking implementations from specifications that combine dataflow modeling and distributed discrete-event modeling. In discrete-event models, components interact with one another via events that are placed on a time line. Discrete-event models have primarily been used in performance modeling and simulation, where time stamps are a modeling property bearing no relationship to real time during execution of the model. My approach is not to model physical phenomena, but rather to use discrete event models to specify real-time systems. Borrowing ideas from TinyOS and nesC [17], this approach introduces real-time components as thin wrappers around hardware to discrete event models. I propose a programming model, called PTIDES (Programming Temporally Integrated Distributed Embedded Systems), which has discrete-event semantics, but with carefully chosen relations between model time and real time. This approach is closest to timed multitasking (TM)[34], but goes a step further in embracing discrete-event semantics and relating model time to real time only at sensor and actuator interactions based on the observation that in many real-time systems time assurance matters only when they react to or act on the physical world. This offers much flexibility in scheduling the internal software components.

Besides the programming model, this dissertation also studies execution mechanisms

that preserve the timing properties in distributed systems. Execution of the software will first obey discrete-event semantics, just as done in DE simulators, but it will do so with specified real-time constraints on certain actions.

A great deal of work has been done on efficient and distributed execution of such models, much of this work originating in either the so-called "conservative" technique of Chandy and Misra [8] or the speculative execution methods of Jefferson [23]. More interesting is the work in the Croquet Project [46], which focuses on optimistic techniques in the face of unreliable components. Croquet has principally been applied to three-D shared immersion environments on the internet, similar to the ones that might be used in interactive networked gaming.

While distribution of discrete-event models has long been used to exploit parallel computing to accelerate execution [50], we are not concerned here with accelerating execution. The focus is instead on efficient distributed real-time execution. To accomplish this, I develop an execution strategy that obeys DE semantics without the penalty of totally ordered executions based on time stamps. Based on causality analysis of DE models, I define *relevant dependency* and *relevant orders* to enable out-of-order execution without compromising determinism and without requiring backtracking. Some level of agreement about time across distributed components is necessary for this model to have a coherent semantics. This technique relies on having a distributed common notion of time, known to some precision.

In the rest of this chapter, I first review related work on concurrent and distributed real-time software design, and then give an overview of this dissertation.

1.1 Related Work

This section first reviews common practice in real-time programming and some pitfalls. It then discusses several time-triggered computation models, including port-based objects [48], Simulink/RTW Gitto [19], and event-triggered computation models, including TinyOS [17] and Timed Multitasking (TM) [34]. These models bring the notion of time and concurrency to the programming level and provide different levels of timing determinism.

1.1.1 Common Practice in Real-time Programming

A distributed real-time system can be decomposed into a set of communicating computer nodes. Within each node, there is a set of concurrent tasks reacting to inputs from the physical world, performing computation or producing output to the physical world. A *task* is a finite amount of computation that requires some resources and takes some time to perform [34]. Each task can be executed potentially an infinite number of times during a run of a real-time system, and each execution of a task is called a *job*. At run-time, multiple tasks often need to run concurrently and compete for resources, including computing resources, like CPU and memory, and communication resources, like buses, networks and I/O. Real-time systems often incorporate a real-time operating system (RTOS) that provides resource management and task activation. A common practice in real-time system design is to adjust *priorities* among the tasks to fulfill timing constraints, and resources should be

granted preferentially to high priority tasks. Priorities might be statically assigned to tasks, using for example the principle of rate monotonic (RM) scheduling [32], or they might be dynamically computed at run time.

The process of deciding which task to execute is called real-time scheduling. Since rate monotonic scheduling was introduced by Liu and Layland [32], real-time scheduling has been an active research area for more than 30 years, and many scheduling algorithms and analysis techniques have been developed. The complexity of the general scheduling problem, i.e. determining whether an arbitrary task system is schedulable or not, is NP-hard [6]. Hence in the literature, real-time scheduling algorithms have been addressing limited problems based on some assumption on tasks and resources. Real-time scheduling algorithms typically assume that the tasks in a real-time system are independent and can be arbitrarily preempted, and that their worst case execution times (WCET) are fixed and known. In general, real-time scheduling for multiprocessor tasks may also need information about the best case execution times of the tasks to protect against Richard's anomalies [1].

Let $\tau = \{T_1, ..., T_n\}$ be a real-time task system that consists a finite set of tasks that are independent, arbitrarily preemptable and having known WCETs. In the rest of this section, we review some classic results on assigning priority to the tasks and statically checking whether τ is schedulable or not.

Scheduling Periodic Independent Tasks

Rate Monotonic Algorithms

Rate Monotonic scheduling described by Liu and Layland [32] is a priority assignment algorithm used with fixed priority preemptive scheduling. It assigns priorities to tasks according to their period, i.e. tasks with shorter period get higher priority. In the simple case, it assumes that all tasks are periodic with deadlines equal to their periods and that tasks are independent of each other. Based on this scheme, schedulability of a real-time system can be statically checked. A sufficient condition to ensure that a real-time task system $T_1, ..., T_n$ is schedulable is given in [32]:

$$\frac{w_1}{p_1} + ... + \frac{w_n}{p_n} \leq n(\sqrt[n]{2} - 1) \tag{1.1}$$

where w_i is the worst case execution time and p_i is the period of task T_i. The left side, $\frac{w_1}{p_1} + ... + \frac{w_n}{p_n}$, is often called the *utilization factor* of the task system. The right side converges to $ln2 \approx \%69$ as n gets large.

Later, Lehoczky, Sha and Ding [22] give a necessary and sufficient condition for such a task system to be schedulable. Assuming the periods $p_1 \leq ... \leq p_n$, a task system $T_1, ..., T_n$ is schedulable if and only if:

$$\begin{aligned} &\text{for all } 1 \leq i \leq n, min_{t \in S_i} \left(\sum_{j=1}^{i} \frac{w_j}{t} \left\lceil \frac{t}{p_j} \right\rceil \right) \leq 1 \text{ ,where} \\ &S_i = \{k \cdot p_j | 1 \leq j \leq i, k = 1, ..., \left\lfloor \frac{p_i}{p_j} \right\rfloor \} \end{aligned} \tag{1.2}$$

Lehoczky, Sha and Ding [22] discuss *schedulability threshold* for several kinds of task systems, where the schedulability threshold means that the system is schedulable if the utilization factor is below the threshold value. When the periods of tasks are randomly generated from a uniform distribution, the schedulable threshold is less than 88%. When the periods of tasks are close to harmonic, i.e. each task period is an exact multiple of

another one, the schedulability threshold approaches 100%, and the worst case schedulability threshold, $ln2$ arises when the periods are relative primes [6].

Another well-known fixed priority preemptive scheduling algorithm is the deadline monotonic algorithm, which assigns priorities to tasks according to their relative deadlines, i.e. tasks with shorter relative deadline get higher priority [2]. It assumes that all tasks are periodic and independent of each other, but does not require that tasks have deadlines equal to periods. With equal periods and deadlines, the deadline monotonic algorithm is identical to the rate monotonic algorithm. The rate monotonic algorithm is optimal in the sense that if any static priority scheduling algorithm can meet all the deadlines, then the rate monotonic algorithm can also do so. When periods are not equal to deadlines, the deadline monotonic algorithm is optimal.

Earliest Deadline First (EDF) Algorithms

EDF scheduling is a dynamic scheduling algorithm [32]. It assign priorities to tasks according to their absolute deadlines and gives higher priorities to tasks with earlier deadlines. When a task with an earlier deadline is released (i.e. when the task is ready to execute), the currently executing task, if there is any, will be preempted, and the newly released task get executed. In the simple case, it assumes that all tasks are periodic and independent of each other. A real-time task system $T_1, ..., T_n$ is schedulable with EDF if and only if

$$\sum_{n}^{i=1} \frac{w_i}{p_i} \leq 1 \tag{1.3}$$

EDF is optimal in the sense that if a set of periodic tasks can be scheduled by any algorithm to meet all the deadlines, the EDF algorithm can also do so. With periodic tasks

that have deadlines equal to their periods, EDF has a utilization bound of 100%. That is, all deadlines are met provided that the total CPU utilization is no more than 100%.

Scheduling Periodic and Sporadic Independent Tasks

Many real-time embedded systems need to react to both cyclic activities and unpredictable discrete activities. If the discrete activities can come in any pattern, then we can not check the schedulability of the system. Thus, researchers have been focused on the scheduling of a mixture of periodic tasks and sporadic tasks which have a minimum interval between any two consecutive release times.

Baruah et al. [3] give the sufficient and necessary condition for a sporadic task system to be schedulable, and provide a testing algorithm that runs in pseudo-polynomial time. For a sporadic task set $\tau = \{T_1, \cdots T_n\}$ with relative deadlines d_i, minimal intervals p_i, worst execution times w_i, $d_i < p_i, \forall i \in [1, n]$, and $U = \sum_{i=1}^{n} \frac{w_i}{p_i} \leq 1$, let (i, t) denote the task invocation of T_i released at t, R denote a set of task invocations and $Pow(R)$ denote the set of all sets of task invocations. Baruah et al. [3] prove that using EDF, a sporadic task set is schedulable iff $\forall R \in Pow(R)$ (i.e. for all possible task release patterns):

$$\forall t \in [0, t^u), h_{R(t)} \leq t \tag{1.4}$$

where $h_{R(t)}$ is the total processing time request for tasks with deadlines before t, i.e. $h_{R(t)} = \sum_{(i,t_0) \in R \wedge t_0 + d_i \leq t} w_i$, and t^u is an upper bound on the value of t determined by some algebraic manipulations.

Ripoll et al. derived a tighter upper bound on t in [43]:

$$t^u = \frac{\sum_{i=1}^{n}(1 - d_i/p_i)w_i}{1 - U} \tag{1.5}$$

Given a sporadic task system, there can be an infinite number of task invocation sequences. In [5], Baruah et al. shows that it is relatively easy to identify a unique worst-case task invocation sequence, such that all other task invocation sequences can be scheduled to meet all deadlines if and only if this worst-case task invocation sequence can. This worst-case task invocation sequence is exactly the sequence of jobs generated by the synchronous periodic task task system with the exact same parameters as the sporadic task system [5].

Pitfalls with Real-time Scheduling

Many of the assumptions made by real-time scheduling algorithms can be too restrictive in reality. For example, tasks in a real-time system may need to access shared data repositories, typically using a mutual exclusion mechanism. In such a case, higher priority tasks can be blocked by lower priority tasks if the lower priority tasks are holding the mutex and updating the shared data. Even worse, the higher priority tasks may be blocked indefinitely given there are some intermediate priority tasks. As an example, consider a real-time system $\tau = \{T_1, T_2, T_3\}$ with three tasks, and assume the priorities among them satisfy $\rho_1 > \rho_2 > \rho_3$, where ρ_i is the priority of task T_i for $i = 1, 2, 3$. Suppose tasks T_1 and T_3 share some data, while T_2 is independent of T_1 and T_3. As shown in figure 1.1, suppose that at some point, while task T_1 is blocked by T_3 to finish updating the shared data, T_2 is ready to execute. Since T_2 has a higher priority than T_3 and is independent of T_3, T_3 will be preempted and T_2 gets the resource to execute. This causes the execution of the high

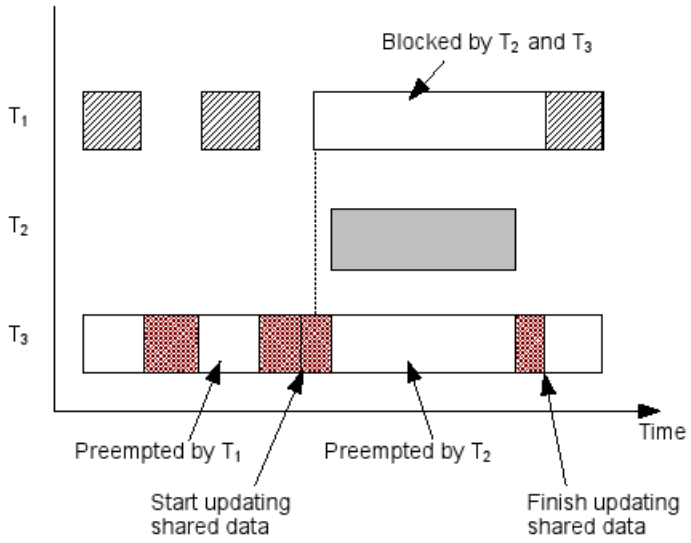


Figure 1.1. An example of priority inversion in real-time scheduling.

priority task T_1 to be further delayed. Assume that there are multiple intermediate-priority tasks like T_2 in the system, then the high priority task T_1 can be blocked indefinitely. This problem is well-known as *priority-inversion* in real-time scheduling.

Priority inversion can be prevented by priority inheritance and priority ceiling protocols [45]. With priority ceilings, each resource is assigned a priority, which equals the highest priority of the tasks that require this resource. When a task locks a resource, the scheduler temporarilly raises the priority of the task to the priority level of the resource. Taking the example above, the mutex for accessing the shared data will have the same priority as T_1, and when T_3 acquires the mutex, its priority is raised to ρ_1, thus keeping intermediate priority tasks like T_2 from preempting T_3 and thereby solving the priority inversion problem. With priority inheritance, a task inherits the highest priority level of all the tasks that are blocked by this task on accessing some shared resource and regains its original priority after

releases the shared resources [45]. The Priority inheritance protocol can also effectively prevent intermediate priority tasks from blocking higher priority task and hence prevent priority inversion. These protocols are widely used in real-time operating systems like VxWorks and QNX. However, the footprint and the run-time overhead of these protocols are not trivial, and many lightweight real-time kernels do not support them [34].

Although there are ways to prevent priority inversion, the problem reveals some more fundamental issues in real-time programming. One key problem is the lack of interaction mechanisms between tasks. Designers are required to implement communication strategies of their own, typically using mutual exclusion to access shared data repositories. Tasks interact both through the real-time scheduler and the mutual exclusion mechanism. Reasoning about the interactions between these two mechanisms can be very difficult, and the result is often fragile designs [28]. Further, there are no coherent compositional semantics between these two mechanisms. As the complexity of real-time applications grows, they will be increasingly difficult to design.

1.1.2 Time-Triggered Computation

Time-Triggered Architecture

The Time-Triggered Architecture (TTA) provides computing infrastructure for safety-critical distributed real-time systems. It addresses the fundamental issues of real-time programming by treating time as a first class quantity. The interaction between software components is fully specified both in the value domain and in the temporal domain, so that

compilers and run-time systems can schedule and optimize the software to achieve timing determinism [34].

TTA includes a fault-tolerant clock synchronization algorithm that establishes a *global time* base. A real-time application is usually decomposed into several sub-systems running on different nodes and a global time is generated at every node. The global time is used as the model of time in TTA to specify the interaction between nodes, to schedule communication and to perform prompt error detection [26].

"A global time is an abstract notion that is approximated by properly selected microticks from the synchronized local physical clocks of an ensemble" explains Kopetz [25], where a *microtick* is a periodically generated event by a oscillation mechanism that increases the counter of a physical clock. Given a set of clocks C synchronized with a precision ϵ, we can select a subset of microticks of each local clock as the local implementation of a global notion of time. For example, every tenth microtick of a local clock k may be selected as the global tick, and the $i \times 10$th microtick of clock k is used as t_i^k, the ith global tick approximated by clock k. Global time is a weaker notion of a universal time reference. The global time t is called *reasonable* if all local implementations of the global time satisfy the condition

$$g > \epsilon \tag{1.6}$$

where g is the granularity, the duration between two consecutive ticks, of the global time. The time-stamp of an event is assigned according to the local global time after the event occurs. Notice that if the global time is reasonable, then the time stamp for a single event e that is observed by any two different clocks of the ensemble can differ by at most one tick.

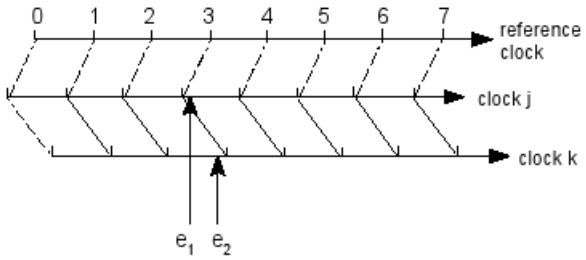


Figure 1.2. Uncertainty in ordering of two events with one tick difference.

The limitation of the global time is that it can not be used to order events whose global time stamps are only 1 tick apart. For example, if we are given two events e_1 and e_2 where e_1 has global time stamp one tick smaller than e_2, can we conclude e_1 happens earlier than e_2? We cannot because it is possible that an event that happens later gets a time stamp smaller than an event happens earlier, as shown in figure 1.2, where e_1 is time stamped by clock j with global time 3, e_2 is time stamped by clock k with global time 2. However, if the time stamp of e_1 is smaller than the time stamp of e_2 by at least two ticks, we can safely conclude that e_1 happens earlier than e_2.

In order to consistently order events based on their global times, TTA introduces a *sparse time* base. In the sparse-time model there is a silence period between any two activity durations [26] as shown in figure 1.3. Events can only happen in the activity interval. Events with the same global time stamp are called simultaneous. Kopetz [25] shows that when the silence interval s satisfies $s \geq 3g$, events that happens during different durations of the activity interval, which hence have different global time stamps, can be consistently ordered based on their global time stamps.

Computation in the Time- Triggered Architecture is based on the time-triggered (TT)

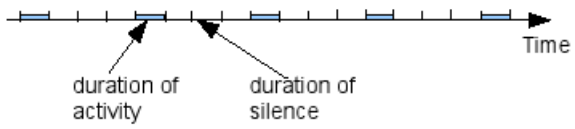


Figure 1.3. Sparse Time Base in TTA.

model, where software components react to events that are repeated with a fixed period. Thanks to the regularity of the computation, the time when a message is sent or fetched from a node is known a priori. TTA includes a communication service to schedule and deliver the messages from the sending nodes to the receiving nodes based on the a priori known time instances of transforming and fetching [26]. The network protocol that provides this communication service in the TTA is the fault-tolerant TTP protocol. The communication in TTP is organized into rounds, where every node must send a message in every round. TTP offers fault-tolerant message transport between distributed nodes with bounded delay by employing a TDMA medium access strategy on replicated communication channels [26].

Time-Triggered Computation Models

Time triggered computation has been widely explored for safely critical systems. It has the advantage that all triggers are predictable in terms of time, and timing analysis becomes easy. I review three examples of time-triggered models in this section.

Port-Based Object

The port-based object (PBO) [48] is a software framework developed in the Advanced Manipulators Laboratory at Carnegie Mellon University (CMU). It was first introduced to

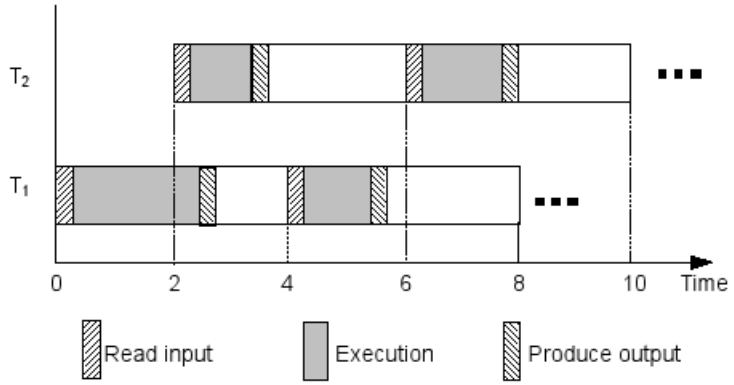


Figure 1.4. An example of task execution sequence in a PBO model.

program reconfigurable robots. In the PBO model, an task is modeled as a *port-based object* with input ports and output ports. The term *object* used in PBO has different meaning from the "object" in object-oriented design. A PBO is an independent concurrent process. The execution of each PBO is activated by time periodically. Communication between PBOs is mediated by input ports and output ports. The communication between output and input ports is via state variables that are stored in a global table. Each PBO stores a subset of the data that is needed from the global table in its own local table. Before executing a PBO, the state variables corresponding to its input ports stored in its local table are updated from the global table. After the execution completes, the state variables corresponding to its output ports are copied from its local table to the global table.

In the PBO model, synchronization is needed to ensure the accesses to the same state variable in the global table are mutually exclusive. The PBO framework provides a mechanism using *spin-locks* [40] for synchronizing access to the global state variable table. When a PBO needs to access the global table, it first locks the processor on which it is executing,

and then waits to obtain the global lock for the global table. The global lock has the highest priority of all the PBOs and the object holding the lock inherits its priority. This way, the object that holds the global lock is assured being on a different processor and will not be preempted. The PBO model assumes that the amount of data exchanged through the state variables are very small, so that each object does not need to hold the global lock long to finish its critical section accessing the global table.

In the PBO model, the communication between PBOs is not deterministic. As the execution finish time of an PBO varies from time to time, the time when the state variables corresponding to its output ports get updated in the global table may not be regular. If another object needs to read these output ports, it may or may not get the results of the current cycle. As an example, consider a simple PBO model with two tasks T_1 and T_2, where T_2 reads the output of T_1. Suppose T_1 and T_2 are running on different CPUs and T_1 get activated at every time instance $t = 2k$ and T_2 get activated at every time instance $t = 2k + 2$, where $k = 0, 1, 2, \cdots$. A possible execution trace of T_1 and T_2 is shown in figure 1.4. In the first period of T_2, the output of T_1 has not been produced, so it reads the stale value of the state variable corresponding to the output of T_1 (in this case, it is the initialized value of the state variable). However, in the second period of T_2, it reads the fresh output of T_1. This communication uncertainty may not be desired in many real-time applications. I review two other time-triggered models below that provide deterministic communication between components.

Giotto

The Giotto programming language [19] provides a time-triggered programming model for periodic, hard real-time systems. Similar to the PBO model, the execution of each task in Giotto is activated by periodic clocks. But the communication between tasks in Giotto is more restricted. Unlike the PBO model where the communication between PBOs may vary from time to time, communication between tasks in Giotto is well defined. The execution of a task in Giotto has a start time, which is the starting time instant the execution period starts, and a stop time, which is the end time instance the execution period ends. A task reads all its inputs at the start time and makes its outputs available to other tasks at its stop time.

In exchange for the communication determinism, Giotto (implicitly) introduces a unit delay at the rate of the slow task on every connection. Consider a Giotto model with two tasks T_1 and T_2 both having a period of 4, and task T_2 reads the output of task T_1. Figure 1.5 (b) shows the timing diagram of the execution of this model. Task B always reads data output by A in the previous cycle. Hence there is a precise 4 seconds delay introduced by task A. This one unit delay may not be desirable for some applications, but it gives Giotto strong formal structure, deterministic behavior in both functional and timing and the ability to perform static schedulability analysis.

Giotto also introduces a composite construct called *mode*. A mode contains a set of tasks and mode switch conditions. Different modes can contain different sets of tasks or have tasks run at different frequencies. At any particular time instant, a Giotto program runs in one mode. Mode switches are also time-triggered and are checked periodically to decide whether to stay in the current mode or switch to another one. Mode switches provide

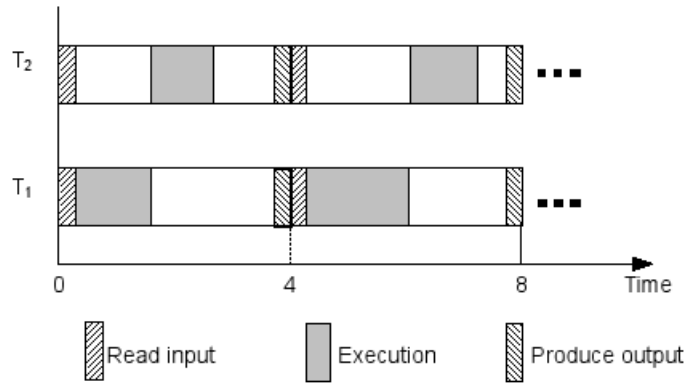


Figure 1.5. In Giotto, the outputs are always produced at the end of the execution cycle.

Giotto the capability to run tasks at different frequencies, to remove tasks or to add tasks dynamically.

Simulink with Real-Time Workshop

Simulink was originally developed as a modeling environment, primarily for control systems. Although initially Simulink focused on simulating continuous dynamics and providing excellent numerical integration, it also has acquired a discrete capability. Semantically, discrete signals are piecewise-constant continuous-time signals which change value only at discrete points on the time line. In addition to discrete signals, Simulink has discrete blocks. These have a sampleTime parameter, which specifies the period of a periodic execution. Any output of a discrete block is a piecewise constant signal. Inputs are sampled at multiples of the sampleTime.

Certain arrangements of discrete blocks turn out to be particularly easy to execute. An interconnection of discrete blocks that all have the same sampleTime value, for example, can

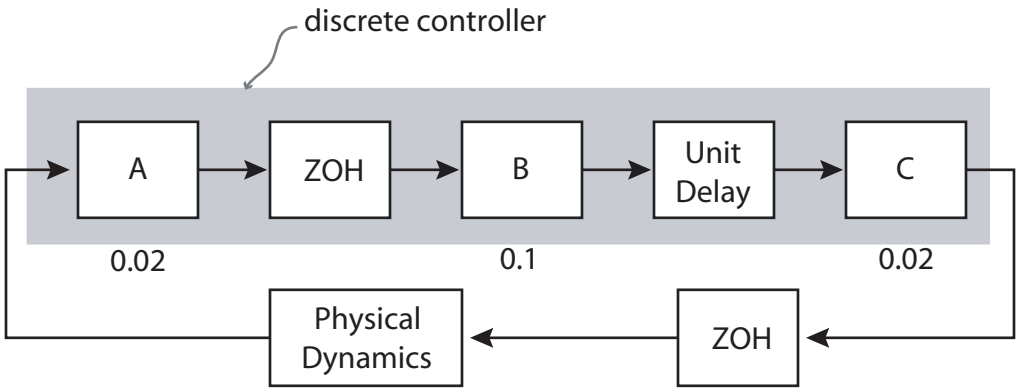


Figure 1.6. A representation of a Simulink program.

be efficiently compiled into real-time software. But even blocks with different sampleTime parameters can yield efficient models, when the sampleTime values are related by simple integer multiples. Fortunately, in the design of control systems (and many other signal processing systems), there is a common design pattern where discrete blocks with harmonically related sampleTime values are commonly used to specify the software of embedded control systems.

Figure 1.6 shows schematically a typical Simulink model of a control system. There is a portion of the model that is a model of the physical dynamics of the system to be controlled. There is no need, usually, to compile that specification into real-time software. There is another portion of the model that represents a discrete controller. In this example, we have shown a controller that involves multiple values of the sampleTime parameter, shown as numbers below the discrete blocks. This controller is a specification for a program that we wish to execute in an real-time system.

Real-Time Workshop is a product from The MathWorks associated with Simulink. It

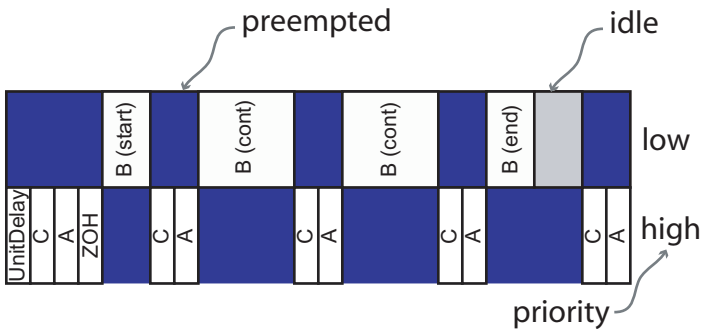


Figure 1.7. A simplified representation of a Simulink schedule.

takes models like that in figure 1.6 and generates code. Although it will generate code for any model, it is intended principally to be used only on the discrete controller, and indeed, this is where its strengths come through.

The discrete controller shown in figure 1.6 has fast running components (with sampleTime values of 0.02, or 20 ms) and slow running components (with sampleTime values of 0.1, or 1/10 of a second). In such situations, it is not unusual for the slow running components to involve much heavier computational loads than the fast running components. It would not do to schedule these computations to execute atomically. This would permit the slow running component to interfere with the responsivity (and time correctness) of the fast running components.

Simulink with Real-Time Workshop uses a clever technique to circumvent this problem. The technique exploits an underlying multitasking operating system with preemptive priority-driven multitasking. The slow running blocks are executed in a separate thread from the fast running blocks, as shown in figure 1.7. The thread for the fast running blocks is given higher priority than that for the slow running blocks, ensuring that the slow run-

ning code cannot block the fast running code. So far, this just follows the principles of rate-monotonic scheduling [33]. But the situation is a bit more subtle than this, because data flows across the rate boundaries. Recall that Simulink signals have continuous-time semantics, and that discrete signals are piecewise constant. The slow running blocks should "see" at their input a piecewise constant signal that changes values at the slow rate. To guarantee that, the model builder is required to put a zero-order hold (ZOH) block at the point of the rate conversion. Failure to do so will trigger an error message. Cleverly, the code for the ZOH runs at the rate of the slow block but at the priority of the fast block. This makes it completely unnecessary to do semaphore synchronization when exchanging data across these threads.

When rate conversions go the other way, from slow blocks to fast blocks, the designer is required to put a UnitDelay block, as shown in figure 1.6. This is because the execution of the slow block will typically stretch over several executions of the fast block, as shown in figure 1.7.

To ensure determinacy, the updated output of the block must be delayed by the worst case, which will occur if the execution stretches over all executions of the fast block in one period of the slow block. The unit delay gives the software the slack it needs in order to be able to permit the execution of the slow block to stretch over several executions of the fast one. The UnitDelay executes at the rate of the slow block but at the priority of the fast block.

Giotto and Simulink/RTW are intended primarily for periodic real-time tasks. These

purely time-triggered approaches require tasks to be periodic and do not fit well with systems reacting to unpredictable irregularly spaced events from the external physical processes. For example, consider a sensor fusion problem where two sensors produce events randomly and we need to calculates a running average of the sensor data using a discounting strategy. While it would be straightforward to construct a discrete multitasking model in Simulink/RTW that polls the sensors at regular (harmonic) rates, reacting to stimulus from the sensors at random times does not fit the semantics very well. I review event-triggered computation models in the next section.

1.1.3 Event-Triggered Computation Models

An event-triggered approach fits well with systems reacting to unpredictable discrete events from external physical processes. The arrival time of events may sometimes be highly unpredictable, which makes timing and schedulability analysis of event-triggered software harder. But in many cases, physical dynamics actually guarantees a lower bound on the interval between certain kind of events, and recent work shows under such constraints event-driven software can be subject to scheduability analysis as well [44]. I review three event-triggered computation models in this section.

TinyOS

TinyOS is a specialized, small-footprint operating system for use on extremely resource-constrained computers, such as 8 bit microcontrollers with small amounts of memory [17]. It is typically used with nesC, a programming language that describes "configurations," which are assemblies of TinyOS components.

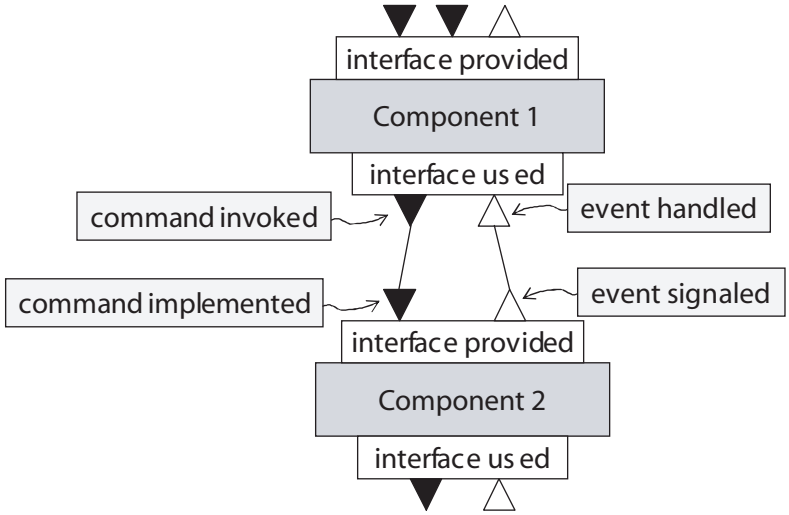


Figure 1.8. A representation of a nesC/TinyOS configuration.

A visual rendition of a two-component configuration is shown in figure 1.8, where the visual notation is that used in [17]. The components are gray boxes with names. Each component has some number of interfaces, some of which it uses and some of which it provides. The interfaces it provides are put on top of the box and the interfaces it uses are put on the bottom. Each interface consists of a number of methods, shown as triangles. The filled triangles represent methods that are called commands and the unfilled triangles represent event handlers. Commands propagate downwards, whereas events propagate upwards.

After initialization, computation typically begins with events. In figure 1.8, Component 2 might be a thin wrapper for hardware, and the interrupt service routine associated with that hardware would call a procedure in Component 1 that would "signal an event." What it means to signal an event is that a procedure call is made upwards in the diagram via the connections between the unfilled triangles. Component 1 provides an event handler

procedure. The event handler can signal an event to another component, passing the event up in the diagram. It can also call a command, downwards in the diagram. A component that provides an interface provides a procedure to implement a command.

Execution of an event handler triggered by an interrupt (and execution of any commands or other event handlers that it calls) may be preempted by another interrupt. This is the principal source of concurrency in the model. It is potentially problematic because event handler procedures may be in the middle of being executed when an interrupt occurs that causes them to begin execution again to handle a new event. Problems are averted through judicious use of the atomic keyword in nesC. Code that is enclosed in an atomic block cannot be interrupted (this is implemented very efficiently by disabling interrupts in the hardware).

Clearly, however, in a real-time system, interrupts should not be disabled for extensive periods of time. In fact, nesC prohibits calling commands or signaling events from within an atomic block. Moreover, no mechanism is provided for an atomic test-and-set, so there is no mechanism besides the atomic keyword for implementing mutual exclusion. The system is a bit like a multithreaded system but with only one mutual exclusion lock. This makes it impossible for the mutual exclusion mechanism to cause deadlock.

Of course, this limited expressiveness means that event handlers cannot perform non-trivial concurrent computation. To regain expressiveness, TinyOS has tasks. An event handler may "post a task." Posted tasks are executed when the machine is idle (no interrupt service routines are being executed). A task may call commands through the interfaces it

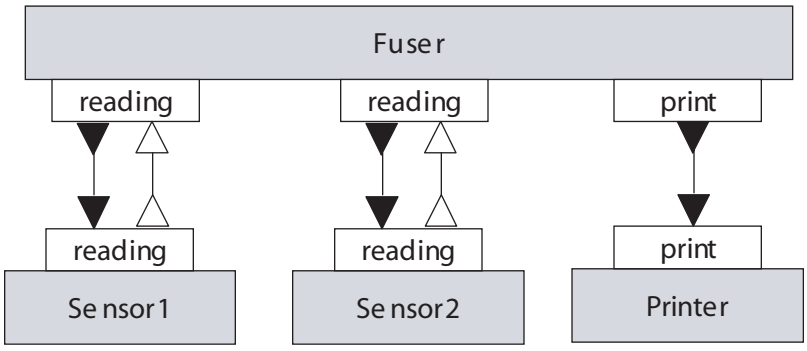


Figure 1.9. A sketch of the sensor fusion problem as a nesC/TinyOS configuration.

uses. It is not expected to signal events, however. Once task execution starts, it completes before any other task execution is started. That is, task execution is atomic with respect to other tasks. This greatly simplifies the concurrency model, because now variables or resources that are shared across tasks do not require mutual exclusion protocols to protect their accesses. Tasks may be preempted by event handlers, however, so some care must be exercised when shared data is accessed here to avoid race conditions. Interestingly, it is relatively easy to statically analyze a program for potential race conditions [17].

Consider the sensor fusion example mentioned in section 1.1.2 where two sensors produce events randomly and we need to calculates a running average of the sensor data using a discounting strategy. A configuration for this is sketched in figure 1.9. The two sensors have interfaces called "reading" that accept a command a signal an event. The command is used to configure the sensors. The event is signaled when an interrupt from the sensor hardware is handled. Each time such an event is signaled, the Fuser component records the sensor reading and posts a task to update the discounted average. The task will then invoke the command in the print interface of the Printer component to display the result. Because

tasks execute atomically with respect to one another, in the order in which they are posted, the only tricky part of this implementation is in recording the sensor data. However, tasks in TinyOS can be passed arguments on the stack, so the sensor data can be recorded there. The management of concurrency becomes extremely simple in this example.

In effect, in nesC/TinyOS, concurrency is much more disciplined than with threads. There is no arbitrary interleaving of code execution, there are no blocking operations to cause deadlock, and there is a very simple mechanism for managing the one nondeterministic preemption that can be caused by interrupts. The price paid for this, however, is that applications must be divided into small, quickly executing procedures to maintain reactivity. Since tasks run to completion, a long-running task will starve all other tasks.

Timed Multitasting (TM)

The TM model is built on top of a component-based approach, where components are called Ptolemy actors [34] [21]. In TM, each task is represented by an actor, which encapsulates a sequence of finite computation. The interface of each actor consists of input/output ports and parameters. An actor communicates with other actors via sending/receiving *events* through its ports. The communication among the actors has an event semantics in the sense that every piece of data will be produced and consumed exactly once [34]. An actor can specify its execution requirements, including priority, execution times, and deadlines, through its parameters. The deadline of a task is specified as a real-time value. The output events are produced if and only if the deadline is reached. In TM, a task is eligible to execute if there is an event that triggers it. By controlling when output events

are produced, TM controls both the starting time and stopping time of each task, thus obtaining deterministic timing properties [34].

TM distinguishes two types of actors: actors that respond to external events and actors that are triggered internally by messages generated by other actors. The first type of actors are called interrupt service routines (ISRs) and the second type of actors are called regular tasks. An ISR actor is represented as a *source actor* – actor that do not have input ports. It converts inputs from the outside world into events that triggers other actors. An ISR actor has no trigger conditions and no deadlines. Its output is made available to downstream actors immediately when the external event occurs. A regular task actor has a much richer interface (including priorities, execution times and deadlines) than an ISR actor and is generally triggered by events at its input ports.

TM assumes there is a single computation resource shared by all the task actors, and the execution of an ISR actor is managed by an independent thread. The execution of the ISR actors is triggered chronologically based on the time stamp of the external events. The execution of other task actors is handled by a priority-based event dispatcher, which sorts and dispatches events based on their priorities. At any given time, only one task actor gets executed. If preemption is allowed, then the execution of a task can by preempted by higher priority tasks. For example, when task A is running, an ISR actor produces an event that triggers task B which has a higher priority than A, then A will be preempted by B. The run time environment of TM tracks when an output event can be produced. If the task can finish its execution before its deadline, the output is made available to its downstream tasks when the deadline is reached. In case a task misses its deadline, TM uses overrun handlers

to complete the current execution of the task quickly to preserve time determinism as much as possible [34].

1.2 Overview of dissertation

This dissertation presents a concurrent model of computation (MoC) for distributed real-time systems called PTIDES (pronounced "tides," for Programming Temporally Integrated Distributed Embedded Systems). Similar to TM, PTIDES is also an event-triggered computation model. But PTIDES achieves deterministic timing behavior for real-time systems in a different way than TM. PTIDES uses a discrete-event (DE) model as the underlying formal semantics to achieve deterministic behavior in both time and value.

In PTIDES, applications are developed as discrete event (DE) models. Whereas traditionally DE has been used as a simulation technology, I am using it as a basis for model-based design of real-time software. The time stamps in conventional discrete-event models are model times just for ordering events and do not need to carry a relationship with real-time. Mapping model time to real time everywhere in a discrete-event model is not efficient as it will lead to total ordering of execution in a distributed system, an unnecessary waste of resources. Based on the observation that in many real-time systems time assurance matters only when they react or act to the physical world, PTIDES relates model time to real time only at sensor and actuator interactions.

Key to making this model effective is to ensure that constraints that guarantee determinacy in the semantics are preserved at runtime. A dependency analysis framework is

presented to allow out of order processing of events without compromising determinism and without requiring backtracking. The basic idea is that if two events have independent effects, then they can be processed in any order. As a result, if the earlier event is delayed due to communication, processing of the later event does not need to be blocked.

Chapter 2 introduces the reader to actor-oriented design and DE models. Chapter 3 presents the dependency analysis framework that allows out of order event processing. Chapter 4 describes the PTIDES model and implementation of a runtime environment for PTIDES models. Chapter 5 studies schedulability of PTIDES systems. Chapter 6 concludes this dissertation.

Chapter 2

Background

This chapter introduces actor-oriented design and the discrete event (DE) model of computation. These form the background knowledge required for understanding the PTIDES model and techniques presented later in this dissertation.

2.1 Actor-Oriented Design

Object-oriented design emphasizes inheritance and procedural interfaces. Actor-oriented design emphasizes concurrency and communication, and admits time as a first-class concept [28]. As a component model, actor is more suitable for concurrent and timed systems. There are many examples of actor-oriented frameworks, including Simulink (from The MathWorks), LabVIEW (from National Instruments) from the industry, Metropolis [18], Giotto [19], Ptolemy and Ptolemy II [21] from the academic community. Hardware design languages, such as VHDL, Verilog, and SystemC, are also actor oriented.

The actor model was first proposed by Carl Hewitt, where actors have their own thread of control and interact via message passing [20]. Gul Agha develops a formal theory for describing concurrent systems that combined Hewitt's message passing with local state update [41]. Agha uses the term "actors," which he defines to extend the concept of objects to concurrent computation. Agha's actors encapsulate a thread of control and have interfaces for interacting with other actors. The protocols used for this interface are called interaction patterns, and are part of the model of computation [28].

Lee generalizes the notion of actors and applies it to software design for concurrent systems [21]. He suggests the term "actor-oriented design" as a refactored software architecture, where instead of objects, *actors* are the primary unit to construct applications. Actors have a well defined interface, which abstracts internal state and execution of an actor and restricts how an actor interacts with its environment. Externally, this interface includes *ports* that represent points of communication for an actor and *parameters* which are used to configure the behavior of an actor [41]. The use of the term actors by Lee is broader in the sense that actors are not required to encapsulate a thread of control [28]. This dissertation uses Lee's concept of actors.

In traditional object-oriented programming, what flows through an object is sequential control. In other words, things happen to objects. In actor-oriented programming, what flows through an actor is evolving data [9]. An actor reacts to input data by performing some computation and producing (possibly) new data to its output port.

Actor-oriented languages can be either self-contained programming languages (e.g. Es-

terel, Lustre) or coordination languages (e.g. Manifold [42], Simulink, Ptolemy II). In the former case, the "atomic actors" are the language primitives. In the latter case, the "atomic actors" are defined in a host language that is typically not actor oriented (but is often object oriented). Actor-oriented design is amenable to both textual syntax, which resemble those of more traditional computer programs, and visual syntax with "boxes" representing actors and "wires" representing connections. The synchronous languages Esterel, Lustre, and Signal, for example, have principally textual syntaxes, although recently visual syntaxes for some of them have started to catch on. Ports and connectors are syntactically represented in these languages by variable names. Using the same variable name in two modules implicitly defines ports for those modules and a connection between those ports. Visual syntaxes are more explicit about this architecture. Examples with visual syntaxes include Simulink, LabVIEW, and Ptolemy II.

Actors can be aggregated to form *composite actors*. A visual syntax for a simple three-actor composition is shown in figure 2.1(a). Here, the actors are rendered as boxes, the ports as triangles, and the connectors as wires between ports. The ports pointing into the boxes are input ports and the ports pointing out of the boxes are output ports. A textual syntax for the same composition might associate a language primitive or a user-defined module with each of the boxes and a variable name with each of the wires.

The semantics of composition, including the communication style, is given by a *model of computation*. A model of computation can be thought of as the "laws of physics" that govern component interactions [28]. Unlike sequential computation where the Von Neumann model provides a wildly successful universal abstraction, no universal model of computation

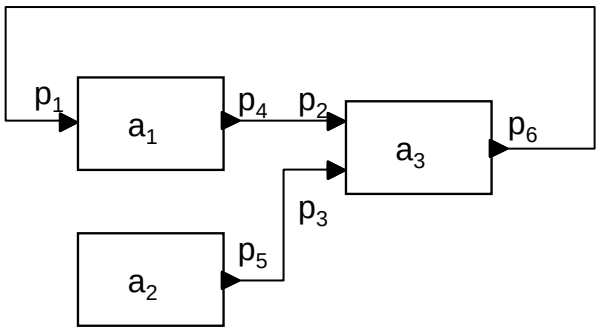


Figure 2.1. composition of actors.

has yet emerged for concurrent computation. There are many models of computation, each dealing with concurrency and time in different ways. A detailed introduction of different models of computations is given in [28], and some examples are briefly described here. In dataflow models, actors are triggered by the availability of input data values (*tokens*), and connections between actors represent the flow of data from a producer actor to a consumer actor. In rendezvous models, the actors are processes, and they communicate in atomic, instantaneous actions called rendezvous. In time triggered models, actors are driven by clocks, and connections between components represent signals with events that are repeated indefinitely with a fixed period. In discrete-event (DE) models of computation, actors are triggered by events , and the connections represent sets of events placed on a time line. I will discuss the DE model of computation in more details in section 2.4.

2.2 Tagged Signal Model

Actors and actor compositions are formally described in the *tagged signal model* [30] by Lee and Sangiovanni-Vincentelli.

2.2.1 Event and Signal

In the tagged signal model, actors interact with each other by tagged signals. Formally, let $\mathcal{T}$ be a poset of tags, V a non-empty set of values. An *event* $e = (t, v)$ is a tag-value pair. That is, e is a member of $\mathcal{T} \times V$. The partial order on the tag set $\mathcal{T}$ specifies ordering of events. Given two events $e_1 = (t_1, v_1), e_2 = (t_2, v_2)$, $e_1 < e_2 \Longleftrightarrow t_1 < t_2$. Two events e_1 and e_2 is said not comparable if neither $e_1 < e_2$ nor $e_2 < e_1$.

Definition 2.1 (Down Set)**.** A down set $D \subseteq \mathcal{T}$ is a subset of $\mathcal{T}$ such that,

$$t \in D \Longrightarrow \forall t' \in \mathcal{T} \text{ where } t' \leq t,\ t' \in D.$$

Definition 2.2 (Signal)**.** A *signal* $s : \mathcal{T} \rightharpoonup V$ is a partial function from $\mathcal{T}$ to V such that $dom(s)$ is a down set of $\mathcal{T}$, where $dom(s)$ denotes the subset of $\mathcal{T}$ on which s is defined.

Let $\mathcal{S}(\mathcal{T}, V)$ denote the set of all signals with tag set $\mathcal{T}$ and value set V. $\mathcal{S}(\mathcal{T}, V)$ is a poset under the *prefix order* [37].

Definition 2.3 (Prefix Order)**.** For any $s_1, s_2 \in \mathcal{S}(\mathcal{T}, V)$, s_1 is a prefix of s_2, denoted by $s_1 \sqsubseteq s_2$, if and only if $\text{dom}(s_1) \subseteq \text{dom}(s_2)$, and $s_1(t) = s_2(t),\ \forall t \in \text{dom}(s_1)$.

It is often useful to form a tuple $\mathbf{s}$ of N signals. The set of all such tuples will be denoted as S^N.

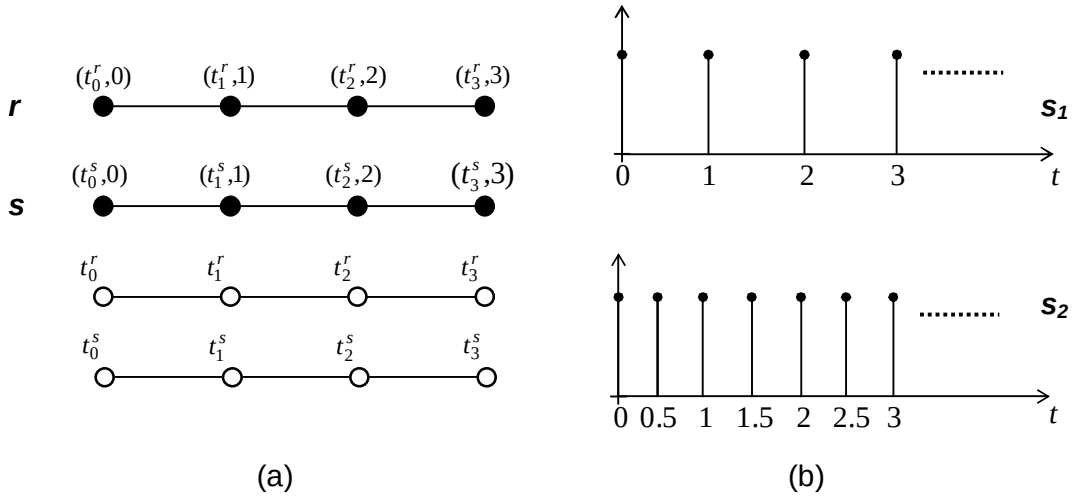


Figure 2.2. (a) two streams r, s and their tag set shown as a *Hasse diagram* with a small variation. Dots represent events, and circles represent tags. (b) two timed signals s_1 and s_2.

The algebraic properties of the tag set are determined by the model of computation. Tags can be used to model time, precedence relationships, synchronization points, and other key properties of a model of computation [30]. We can specify a particular model of computation in the tagged signal model framework by defining the tag set for signals [36].

Figure 2.2 shows some examples of tag sets and signals taken from [36]. Figure 2.2 (a) shows two streams r and s and their tag sets. The tag set of a stream s is $\{t_k^s \mid k \in \mathbb{N}\}$ with the ordering $t_i^s \leq t_j^s$ for all $i, j \in \mathbb{N}$ such that $i \leq j$. The events of a stream are totally ordered. Two tags from different streams are not comparable. Figure 2.2 (b) shows two timed signals, in which the tags mark time. Not only the events of one timed signal are totally ordered, but two events from different signals are also ordered by their tags, i.e. the events of the two timed signals are totally ordered.

2.2.2 Actor

An *actor* is a computational unit that relates a set of signals. An actor a with N ports is a subset of S^N, i.e. $a \subseteq S^N$. Given $\mathbf{s} \in S^N$, $\mathbf{s}$ is called a behavior of a if $\mathbf{s} \in a$. Thus an actor is a set of possible behaviors. Note that we can easily embrace nondeterminism in this model, but we should introduce nondeterminism only when it is needed.

A connector c between M ports P_c is a particularly simple actor $c \subset S^M$ where signals at each port $p \in P_c$ are constrained identical.

In many actor-oriented formalisms, ports are either inputs or outputs to an actor but not both. Consider an actor $a \subseteq S^N$ where $I \subseteq \{1, \cdots, N\}$ denotes the indices of the input ports, and $O \subseteq \{1, \cdots, N\}$ denotes the indices of the output ports. We assume that $I \cup O = \{1, \cdots, N\}$ and $I \cap O = \emptyset$. Given a signal tuple $\mathbf{s} \in a$, we define $\pi_I(\mathbf{s})$ to be the projection of $\mathbf{s}$ on a's input ports, and $\pi_O(\mathbf{s})$ on output ports. The actor is said to be *functional* if,

$$\forall \mathbf{s}, \mathbf{s'} \in a, \pi_I(\mathbf{s}) = \pi_I(\mathbf{s'}) \Longrightarrow \pi_O(\mathbf{s}) = \pi_O(\mathbf{s'})$$

Given a functional actor a with $|I|$ input ports and $|O|$ output ports, we can define an actor function with the following form:

$$F_a : S^{|I|} \longrightarrow S^{|O|}$$

where $|.|$ denotes the size of a set. When it creates no confusion, we make no distinction between the actor a (a set of behaviors) and the actor function F_a [51].

A *source* actor is an actor that only has output ports. It is functional if and only if its

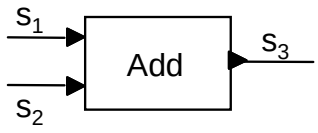


Figure 2.3. an Add actor.

behavior set is a singleton set. That is, it has only one behavior. A *sink* actor is an actor that only has input ports. It is always functional.

Figure 2.3 illustrates an actor that adds two timed signals s_1 and s_2 to produce the signal s_3. Suppose that addition $+\colon V \times V \to V$ is defined on a value set V. Let $+_\varepsilon\colon V_\varepsilon \times V_\varepsilon \to V_\varepsilon$ be defined by

$+_\varepsilon$	$v_2 \in V$	$v_2 = \varepsilon$
$v_1 \in V$	$v_1 + v_2$	v_1
$v_1 = \varepsilon$	v_2	ε

(2.1)

where ε represents the absence of a normal value. For any normal value set V, $V_\varepsilon = V \cup \{\varepsilon\}$.

The *Add* actor $Add \subseteq S^3$ has 3 ports and is functional with $I = \{1, 2\}$ being the indices of the input ports and $O = \{3\}$ being the indices of the output ports. It has the form:

$$Add : S^2 \longrightarrow S^1$$

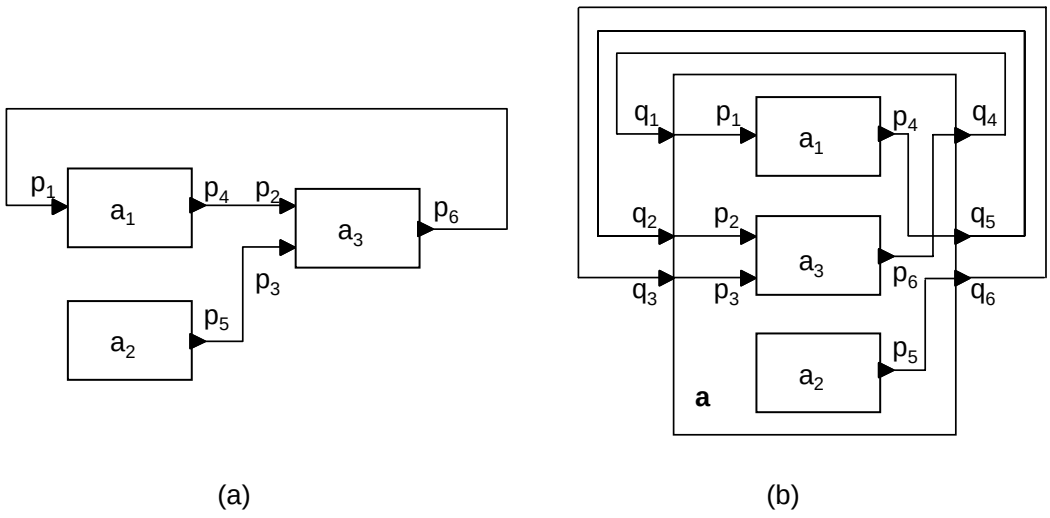


Figure 2.4. composition of functional actors.

Given a signal tuple $\mathbf{s} = (s1, s2) \in S^2$, the *Add* actor adds s_1, s_2 by:

$$Add(s_1, s_2) = s_3 \text{ where}$$

$$\begin{aligned} \mathrm{dom}(s_3) &= \mathrm{dom}(s_1) \cap \mathrm{dom}(s_2), \\ s_3(t) &= s_1(t) +_\varepsilon s_2(t). \end{aligned} \tag{2.2}$$

Actors can be composed to form a network of actors, and from now on we only consider composition of functional actors. The composite actor is the intersection of all possible behaviors of the actors and the connectors in the composition. We have to use some care in forming this intersection. Before we can form such an intersection, each process to be composed must be augmented as a subset of the same set of signals S^N [31]. Note that unlike composition of the objects and threads model, composition of actors does not introduce nondeterminism.

Figure 2.4 (a) shows a composite actor constructed from three actors. We can rearrange

the actors and compose them into one composite actor a as shown in figure 2.4 (b), where internally it has no directed cycles and externally it can be viewed as a feedback system. In fact, any network of actors can be converted to a feedback system, like that in figure 2.4 (b). A constructive procedure that performs this conversion is to create one input port and one output port for each signal in the original network. Then connect the output port providing the signal to the new output port, and connect the new input port to input ports that observe the signal [51].

If the component actors are functional, then the composite actor is functional [30]. Let F_a denote the actor function of a. Assuming a_1, a_2 and a_3 are functional, F_a has the form

$$F_a : S^3 \longrightarrow S^3$$

Given a signal tuple $\mathbf{s} \in a$,

$$F_a(\mathbf{s}) = \mathbf{s} \tag{2.3}$$

A solution that satisfies equation 2.3 is called a *fixed point* of the composition. A key question is whether such a fixed point exists, and whether it is unique. The answer to this question will be discussed in the next section.

The composition in figure 2.4 has no external input ports, i.e. ports that are not connected to the output of any actor in the composition. The composition is called *closed* if it has no external input ports. Otherwise, it is called *open*. We can generalize the fixed point formulation to allow open composition. We partition the input ports of the composition actor into two disjoint indice sets $I = I_1 \cup I_2$, where I_1 is the indices of the external input ports, and I_2 is the indices of the other input ports of the composite actor.

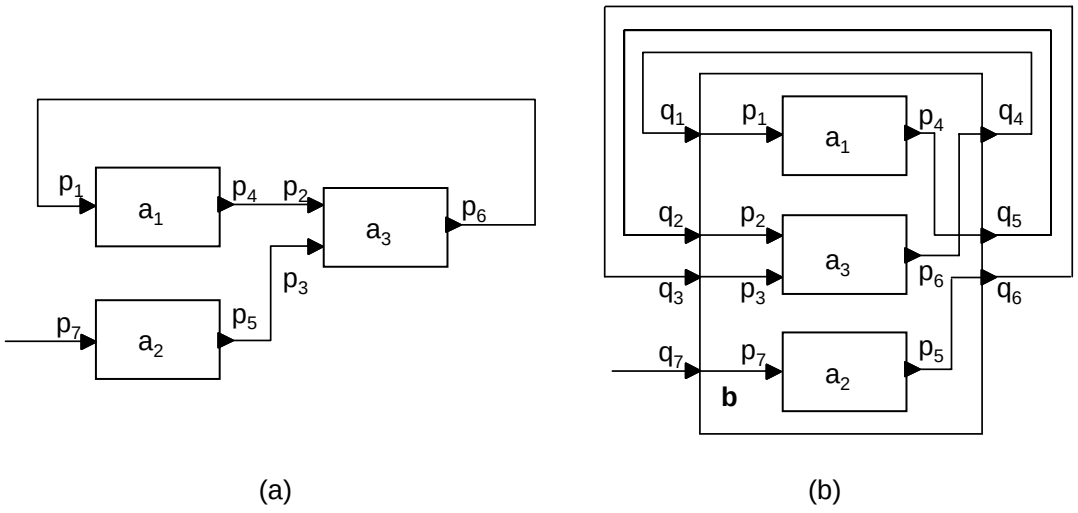


Figure 2.5. open composition of functional actors.

Figure 2.5 shows a composition with one external input port and the rearrangement as a feedback system. For this open composition $b \subseteq S^7$, $I_1 = \{7\}$, and $I_2 = \{1, 2, 3\}$. Let O be the indices of the output ports of b, and $O = \{4, 5, 6\}$ in figure 2.5. Let F_b denote the actor function of b. F_b has the form

$$F_b : S^{|I_1|} \times S^{|I_2|} \longrightarrow S^{|O|}$$

Given a signal tuple $\mathbf{s} \in b$, $\pi_{I_2}(\mathbf{s}) = \pi_O(\mathbf{s})$. The output $\mathbf{s}_2 \in S^{|O|}$ satisfies

$$\forall \mathbf{s}_1 \in S^{|I_1|}, \mathbf{s}_2 = \widetilde{F_b}(\mathbf{s}_2), \text{where} \widetilde{F_b} \colon \mathbf{s}_2 \mapsto F_b(\mathbf{s}_1, \mathbf{s}_2). \tag{2.4}$$

That is, the behavior on the output ports is a fixed point of a function that is parameterized by the input signal.

2.2.3 Fixed Point Semantics

Monotonicity and continuity plays important role in ensuring the existence of the fixed point and finding it constructively. We review some classical results from [10] and apply them to our formulation of actor networks.

Definition 2.4 (monotonic)**.** Let $(S_1, \sqsubseteq)$ and $(S_2, \sqsubseteq)$ be CPOs. A function $F\colon S_1 \to S_2$ is *monotonic* if

$$\forall s, s' \in S_1, s \sqsubseteq s' \implies F(s) \sqsubseteq F(s').$$

Definition 2.5 (continuous)**.** Let $(S_1, \sqsubseteq)$ and $(S_2, \sqsubseteq)$ be CPOs. A function $F\colon S_1 \to S_2$ is (Scott) *continuous* if for all directed sets $D \subseteq S_1$, $F(D)$ is a directed set and

$$F(\bigvee D) = \bigvee \{F(d) | d \in D\}.$$

Continuity implies monotonicity [10]. A well-known fixed point theorem [10] states that a continuous function has a least fixed point and gives a constructive way to find it.

Theorem 2.6 (Fixed Point Theorem)**.** Let S be a CPO with partial order $\leq$ and the least element $\perp$, and $F\colon S \to S$ a continuous function. Then F has a least fixed point given by:

$$fix(F) = \vee \{F^n(\perp) | n \in N\},$$

where $N = \{1, 2, \cdots\}$ is the set of natural numbers.

For closed actor networks like those in figure 2.4, if each component actor is continuous, then the composite actor F_a is a continuous function on a CPO. Thus, it has a least fixed point, and that fixed point is given by (2.3). If a fixed point exists, we define the semantics of the diagram in figure 2.4 to be the least fixed point in the prefix order.

The fixed point theorem 2.6 can also be applied to open systems with a bit more work. Following [37], let $(D, \sqsubseteq)$ and $(E, \sqsubseteq)$ be CPOs, and now we consider a function of the form

$$G\colon D \times E \to E. \tag{2.5}$$

For a given $d \in D$, let $G(d)\colon E \to E$ be the function such that

$$\forall\ e \in E, \quad (G(d))(e) = G(d, e).$$

If G is continuous, then for all $d \in D$, $G(d)$ is continuous (lemma 8.10 in [49]). Hence, $G(d)$ has a unique least fixed point, and that fixed point is

$$\bigvee \{(G(d))^n(\perp_E) \mid n \in \mathbb{N}\},$$

where $\perp_E$ is the least element of E.

We recognize immediately that the feedback system function of (2.4) is a function of form (2.5). Moreover, if the component actors are continuous, then the feedback system function will be continuous, and given an input signal $\mathbf{s}_1 \in S^{|I_1|}$, $\widetilde{F_b}\colon \mathbf{s}_2 \mapsto F_b(\mathbf{s}_1, \mathbf{s}_2)$ is continuous and hence has a least fixed point.

Continuity ensures the existence of a least fixed point. However, even there exist a least fixed point for a continuous actor network, the fixed point can be a trivial signal, i.e. the empty signal $s_\perp$ that is defined nowhere ($dom(s) = \emptyset$). For example, suppose that in figure 2.4 (b) F_a is the identity function, which is continuous. It is easy to see that the least fixed point assigns to each port the empty signal [51]. The next section focus on timed actor networks and study the sufficient condition ensuring a useful fixed point.

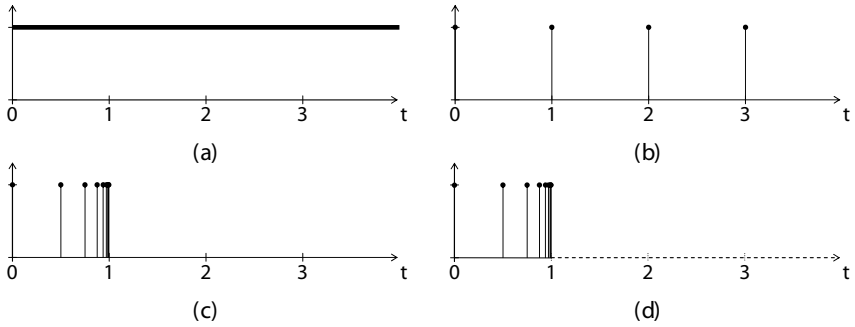


Figure 2.6. Examples of timed signals: (a) $const_1$, (b) $clock_1$, (c) *zeno*, (d) *dzeno*.

2.3 Timed Actor Networks

A subclass of tagged systems are *timed systems*, in which all signals share a common tag set. The tag set is totally ordered, and is a model of global time in the system. Examples of timed systems include synchronous/reactive (SR) models, discrete-event (DE) models, and continuous-time (CT) models. For SR models, the tag set is the set of natural numbers $\mathbb{N}$. For DE and CT models, Any non-empty interval of real numbers may be used as the tag set. In this thesis, the non-negative real numbers $\mathbb{R}_0 = [0, \infty)$ will be used as a representative.

Following [37], we represent a signal s as a tuple $(\text{dom}(s), E)$, where $\text{dom}(s)$ is the domain of the signal (a down set of $\mathcal{T}$), and E is the set of events that are not absent,

$$E = \{(t, s(t)) \,|\, t \in \text{dom}(s),\ s(t) \neq \varepsilon\}.$$

I assume that every value set V contains a special element $\varepsilon \in V$ that represents absence of a value.

The following examples, with $\mathcal{T} = \mathbb{R}_0$ and $V = \{0, 1, \varepsilon\}$, are sketched in figure 2.6:

$$const_1 = (\mathbb{R}_0, \{(t, 1) \,|\, t \in \mathbb{R}_0\}),$$

$$clock_1 = (\mathbb{R}_0, \{(k, 1) \,|\, k \in \mathbb{N}\}),$$

$$zeno = (\mathbb{R}_0, \{(1 - 1/2^k, 1) \,|\, k \in \mathbb{N}\}),$$

$$dzeno = ([0, 1), \{(1 - 1/2^k, 1) \,|\, k \in \mathbb{N}\}).$$

A closed actor network is said to be *live* if the input and output of the network are defined for all tags in $\mathcal{T}$. An open composition is live if if the output of the network are defined at least up to t when all input signals are defined up to t for some $t \in \mathcal{T}$. It is well known that, in general, whether a network of actors is live is undecidable (this is known for Kahn process networks) [37]. For timed systems, a sufficient and checkable condition for a network to be live is that the composite actor is a contraction map, where a metric space of signals is constructed and contraction maps combined with the Banach fixed point theorem yield live systems. Liu and Lee [37] give a sufficient condition for a timed system to be live without requiring a metric space.

Definition 2.7 (Causality)**.** Let $A \subseteq S^N$ be an actor with N ports. Let I and O be the indices of the input and output ports, where $I \cap O = \emptyset$ and $I \cup O = \{1, \cdots, N\}$. Given $\mathbf{s} \in S^N$, and $\forall i \in \{1, \cdots, N\}$, let s_i be the projection of $\mathbf{s}$ on port i. A is causal if it is monotonic, and for all $\mathbf{s} \in A$,

$$\bigcap_{i \in I} dom(s_i) \subseteq \bigcap_{j \in O} dom(s_j)\,. \tag{2.6}$$

Intuitively, this definition says that if the inputs of a causal actor are known up to some

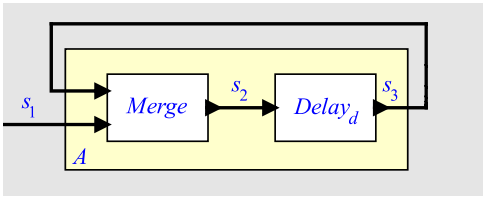


Figure 2.7. A composition that can be shown to be live.

tag $t \in \mathcal{T}$, where $\mathcal{T}$ is a totally ordered tag set, then the outputs are known at least up to tag t.

It is easy to see that any composition of causal actors without directed cycles is itself a causal actor. This is not in general true when there are directed cycles. In this case, the composite actor is causal if at least one actor in the loop is strictly causal [37], as defined next.

Definition 2.8 (Strict Causality)**.** Let $A \subseteq S^N$ be an actor with N ports. Let I and O be the indices of the input and output ports, where $I \cap O = \emptyset$ and $I \cup O = \{1, \cdots, N\}$. Given $\mathbf{s} \in S^N$, and $\forall i \in \{1, \cdots, N\}$, let s_i be the projection of $\mathbf{s}$ on port i. A is strictly causal if it is monotonic, and for all $\mathbf{s} \in A$,

$$\bigcap_{i \in I} dom(s_i) \subset \bigcap_{j \in O} dom(s_j)\,. \tag{2.7}$$

A causal actor is live. Thus, a sufficient condition for a composition of actors to be live is that the composition is causal. The following theorem is from [37].

Theorem 2.9 (Causality of Feedback Compositions)**.** Given a totally ordered tag set and a network of causal and continuous actors where in every dependency loop in the network there is at least one strictly causal actor, then the network is a causal and continuous actor.

Figure 2.7 shows a composition of the $Delay_d$ and *Merge* actor as given in [37]. Let d

be any positive real number. The $Delay_d\colon S \to S$ actor shifts every event in its input signal by d into the future such that if $r = Delay_d(s)$, then

$$\begin{aligned} \mathrm{dom}(r) &= \{t \in \mathcal{T} \mid t - d \in \mathrm{dom}(s) \text{ or } t - d \notin \mathcal{T}\}, \\ r(t) &= \begin{cases} s(t-d) & t - d \in \mathrm{dom}(s), \\ \varepsilon & \text{otherwise.} \end{cases} \end{aligned} \tag{2.8}$$

The $Merge\colon S^2 \to S$ actor combines the present events in its input signals into its output signal, giving precedence to its first input when both input signals are present at the same time. Specifically, if $s = Merge(s_1, s_2)$, then

$$\begin{aligned} \mathrm{dom}(s) &= \mathrm{dom}(s_1) \cap \mathrm{dom}(s_2), \\ s(t) &= \begin{cases} s_1(t) & s_1(t) \neq \varepsilon, \\ s_2(t) & \text{otherwise.} \end{cases} \end{aligned} \tag{2.9}$$

It is easy to prove that $Delay_d$ and $Merge$ are both continuous and causal [36]. $Delay_d$ is also strictly causal for any $d > 0$ with the tag set $\mathcal{T} = \mathbb{R}_0$. The composition of figure 2.7 is continuous and causal when $Delay_d$ is strictly causal, and hence live.

2.4 Discrete-Event Models

The following definitions are from [37]:

Definition 2.10 (Discrete Event Signal)**.** A timed signal $s \in \mathcal{S}(\mathcal{T}, V_\varepsilon)$ is a discrete event (DE) signal if there exists a directed set $D \subseteq \mathcal{S}(\mathcal{T}, V_\varepsilon)$ of finite timed signals such that

$$s = \bigvee D\,.$$

This definition states that DE signals can be approximated by finite signals of $\mathcal{S}(\mathcal{T}, V_\varepsilon)$. Let $\mathcal{S}_{\mathrm{d}}(\mathcal{T}, V_\varepsilon) \subseteq \mathcal{S}(\mathcal{T}, V_\varepsilon)$ denote the set of all DE signals with the same tag and value sets as $\mathcal{S}(\mathcal{T}, V_\varepsilon)$. Among the signals in figure 2.6, $clock_1$ and *dzeno* are DE signals, but not $const_1$ and *zeno*.

Let $D(t) = \{\tau \in \mathcal{T} | \tau \leq t\}$ for some $t \in \mathcal{T}$ denote the smallest down set including t. There are several equivalent definitions of DE signals, as established by the following lemmas [37].

Lemma 2.11. A timed signal s is a DE signal if and only if for all $t \in \mathrm{dom}(s)$, $s \restriction D(t)$ is a finite signal.

Lemma 2.12. A timed signal $s \in \mathcal{S}(\mathcal{T}, V_\varepsilon)$ is a DE signal if and only if $s^{-1}(V \setminus \{\varepsilon\})$ is order-isomorphic to a down set of $\mathbb{N}$, and if $s^{-1}(V \setminus \{\varepsilon\})$ is an infinite set, then

$$\mathrm{dom}(s) = \bigcup_{t \in s^{-1}(V \setminus \{\varepsilon\})} D(t). \tag{2.10}$$

Lemma 2.11 is very useful in proving properties of DE signals. lemma 2.12 is used in [27]. If $s^{-1}(V \setminus \{\varepsilon\})$ is order-isomorphic to a down set of $\mathbb{N}$, then the present events of s can be enumerated in the order of their time. If s is present at an infinite number of times, then equation 2.10 guarantees that for any $t \in \mathrm{dom}(s)$, s is present at a time later than t.

Definition 2.13 (Non-Zeno Signal)**.** A DE signal $s \in \mathcal{S}_{\mathrm{d}}(\mathcal{T}, V_\varepsilon)$ is non-Zeno if either s is a finite signal, or s is a total signal, i.e. $\mathrm{dom}(s) = \mathcal{T}$.

Of the signals in figure 2.6, $clock_1$ is the only non-Zeno DE signal. The only other DE signal, *dzeno*, is a Zeno signal, as $\mathrm{dom}(dzeno) = [0, 1) \neq \mathcal{T} = [0, 1]$. Note the role of the

tag set $\mathcal{T}$ in definition 2.13. If we change the tag set to $\mathcal{T} = [0, 1)$, then the signal

$$([0, 1), \{(1 - \frac{1}{2^k}, 1) \mid k \in \mathbb{N}\})$$

is present at the same set of times as *dzeno*, but it is a non-Zeno signal because its tag set $\mathcal{T}$ is $[0, 1)$ and it is a total signal.

A key property of non-zeno DE signals is that all approximations defined over a subset of $\mathcal{T}$ have a finite number of (non-absent) events. This property is extremely helpful when computing the signals in a composition. It means that a computation can successively approximate signals over down sets of $\mathcal{T}$, iteratively increasing these down sets toward the limit of $\mathcal{T}$, and the computation will never have to represent more than a finite number of events [37].

Definition 2.14 (Discrete-Event Actor)**.** A *discrete event actor* is a function from DE signals to DE signals.

All input and output signals of a DE actor have the same tag set. Among the actors discussed above, $Delay_d$ and *Merge* are DE actors. As an example, the *MaxMerge* actor given in [37] is not a DE actor. The *MaxMerge* actor $s = MaxMerge(s_1, s_2)$ is given by:

$$\mathrm{dom}(s) = \{t \in \mathrm{dom}(s_1) \mid \forall \tau \in D(t) \setminus \mathrm{dom}(s_2),\ s_1(\tau) \neq \varepsilon\}, \tag{2.11}$$

$$s(t) = \begin{cases} s_1(t) & s_1(t) \neq \varepsilon, \\ s_2(t) & \text{otherwise.} \end{cases} \tag{2.12}$$

Intuitively, if the input signal s_1 is continuously present over a time interval beyond $\mathrm{dom}(s_2)$, then those present events are in the output of *MaxMerge*. The "Max" in the name is

suggestive that this actor, unlike *Merge*, produces the maximal output for a given pair of inputs. Consider

$$s_1 = ([0,1], \{(1,1)\}),$$

$$s_2 = dzeno,$$

$$MaxMerge(s_1, s_2) = ([0,1], \{(1 - \frac{1}{2^k}, 1) \mid k \in \mathbb{N}\} \cup \{(1,1)\}).$$

$MaxMerge(s_1, s_2)$ is not a DE signal.

Definition 2.15 (Non-Zeno Actor)**.** A DE actor $A\colon \mathcal{S}_{\mathrm{d}}(\mathcal{T}, V_\varepsilon) \to \mathcal{S}_{\mathrm{d}}(\mathcal{T}, V_\varepsilon)$ is a *non-Zeno actor* if for any non-Zeno signal $s \in \mathcal{S}_{\mathrm{d}}(\mathcal{T}, V_\varepsilon)$, $A(s)$ is a non-Zeno signal.

A causal DE actor is non-Zeno [37]. Combining the previous result from 2.9, we have the following corollary.

Corollary 2.16. If all actors in a DE actor network are causal and continuous, and in every dependency loop in the network there is at least one strictly causal actor, then the network is non-Zeno.

For causal functional actors, if the input is the prefix of the potentially infinite-length input signal up to time t, then the output is the prefix of the final output signal up to at least time t [35]. Furthermore, for non-zeno composition, the prefix of the input or output signals up to time t is finite [36]. This means for discrete event systems, the behavior of the system can be simulated iteratively by computing partial behaviors chronologically. Discrete-event simulators, for example, execute a composition precisely in this manner. A typical Discrete-event simulator operates by keeping a list of events sorted by time stamp. The event with the smallest time stamp is processed and removed from the list. The simulator maintains a

global, monotonically increasing notion of time, and computes the behavior of the system step by step.

2.5 Discrete-Event Simulation

The discrete-event model of computation is frequently used in simulators for such applications as circuit design (e.g. in hardware description languages such as Verilog and VHDL), communication network modeling such as OPNET Modeler[1], Ns-2[2] and VisualSense [4]), etc. Recently, the discrete event model is getting applied to intrinsically distributed systems, such as networked computer games and collaborative virtual environments (e.g. Croquet [46]). In these applications, time is mainly used as a convenient coordination mechanism.

When DE models are executed on distributed platforms, the objective is usually to accelerate simulation [16, 7, 14], not to implement distributed real-time systems. With distributed discrete-event simulation, multiple nodes can process events in parallel and advance time differently. However, every node is required to process events in chronological order. The so-called "conservative" approaches advance model time to t only when each node can be assured that it has seen all events time stamped t or earlier. In the well-known Chandy and Misra technique [8], extra (null) messages are used for one execution node to notify another that there are no such earlier events. This technique binds the execution at the nodes too tightly, making it very difficult to meet realistic real-time constraints. The so-called "optimistic" techniques perform speculative execution and backtrack if and

[1] http://opnet.com/products/modeler/home.html
[2] http://www.isi.edu/nsnam/ns

when the speculation was incorrect [23]. Such optimistic techniques may not be feasible for executing real-time systems, since backtracking physical interactions is usually not possible.

In the next chapter, I develop a dependency analysis framework to allow out of order processing of events without compromising determinism and without requiring backtracking. This approach is conservative, in the sense that events are processed only when it is safe to do so. But it achieves significantly looser coupling than Chandy and Misra's approach.

Chapter 3

Relevant Dependency

As discussed in last chapter, a conventional discrete-event simulator computes the behavior of discrete-event systems chronologically. The simulator maintains a global, monotonically increasing notion of time. Events are sorted in a list by time stamp. Events with the smallest time stamp are processed and removed from the list, and new events may be generated and inserted to the list. In order to process an event e with time stamp t, the simulator needs to know all events before e and make sure these events have been processed. This approach is very restrictive in processing events. In this chapter, I develop a new strategy that preserves the discrete event semantics without requiring to process all events in their time stamp order. In chapter 4, I will show why this new strategy is preferred when DE is used to specify real-time systems.

Consider a very simple example as shown in figure 3.1 where the *Merge* and *Delay* are the same actors as discussed in chapter 2. Let s_i be the signal at port p_i. Given a signal

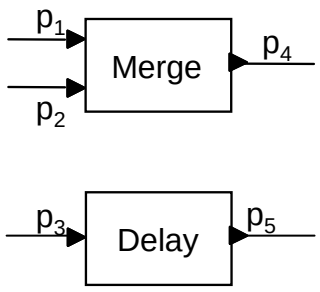


Figure 3.1. A simple example with signals unrelated.

tuple $s = (s_1, s_2, s_3) \in S^3$, the system maps it to a signal tuple $s' = (s_4, s_5) \in S^2$ as:

$$(s_4, s_5) = (Merge(s_1, s_2),\ Delay(s_3))$$

Or we can write the above equation as:

$$s_4 = Merge(s_1, s_2),\ s_5 = Delay(s_3)$$

For causal functional actors, if the input is a prefix of the potentially infinite-length input signal up to time t, then the output is a prefix of the final output signal at least up to time t. Let $s \upharpoonright D(t)$ denote the prefix of s up to t. Recall $D(t) = \{\tau \in \mathcal{T} | \tau \leq t\}$ for some $t \in \mathcal{T}$ is the smallest down set including t. In this dissertation, $s \upharpoonright D(t)$ is often read as s defined on $D(t)$ or s is known up to t. It is easy to see that for this system, given s_1, s_2 and s_3 we can compute signal s_4 and s_5 separately.

$$\forall t \in dom(s_1) \cap dom(s_2),\ t' \in dom(s_3), \quad s_4 \upharpoonright D(t) \subseteq Merge(s_1 \upharpoonright D(t), s_2 \upharpoonright D(t))$$
$$s_5 \upharpoonright D(t') \subseteq Delay(s_3 \upharpoonright D(t'))$$

$\forall t_1, t_2 \in dom(s_1) \cap dom(s_2)$and $t_1 \leq t_2, s_4 \upharpoonright D(t_1) \subseteq s_4 \upharpoonright D(t_2)$. Similarly, $\forall t'_1, t'_2 \in dom(s_3)$and $t'_1 \leq t'_2, s_5 \upharpoonright D(t'_1) \subseteq s_5 \upharpoonright D(t'_2)$. There are many ways to approximate s_4 and

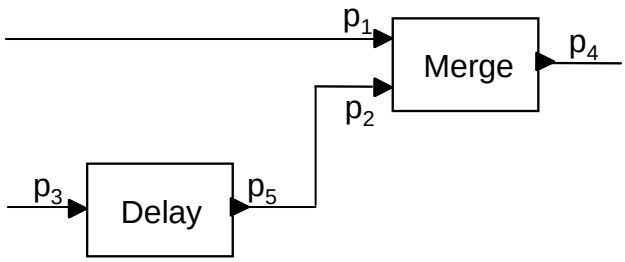


Figure 3.2. An example with signals related but with delay.

s_5. The conventional simulation approach is just one way of them which requires $t = t'$ and $t \in dom(s_1) \cap dom(s_2) \cap dom(s_3)$ when approximating s_4 and s_5. Suppose we know s_1 and s_2 up to t and s_3 up to t' with $t > t'$. The conventional discrete event simulator only computes s_4 and s_5 based on events at s_1, s_2 and s_3 up to t', while as shown above s_4 can be compute based on events at s_1 and s_2 up to t. This chapter will develop a new simulation strategy that allows us to advance time stamp at different signals separately.

Now we consider a slightly different example as shown in figure 3.2 where the output of the $Delay_d$ actor is connected to the second input of the *Merge* actor. For this system, given a signal tuple $s = (s_1, s_2, s_3) \in S^3$, the system maps it to a signal tuple $s' = (s_4, s_5) \in S^2$ as:

$$(s_4, s_5) = (Merge(s_1, s_2),\ Delay(s_3)),\ \text{and}\ s_2 = s_5$$

Knowing that the *Delay* actor shifts every event in its input signal by d into the future, we can compute s_4 and s_5 as follows:

$$\forall t \in dom(s_1),\ t' \in dom(s_3),\ t \leq t' + d, \quad s_4 \upharpoonright D(t) \subseteq Merge(s_1 \upharpoonright D(t), s_2 \upharpoonright D(t))$$

$$s_2 \upharpoonright D(t) = s_5 \upharpoonright D(t) \subseteq (Delay(s_3 \upharpoonright D(t'))$$

That is, we can approximate s_4 at least up to t if we know events of s_1 up to t and

events of s_3 up to $t - d$, while the conventional simulation approach can only compute s_4 up to $t - d$.

Recall that actors respond to events at input ports by producing events at output ports. Knowing which portion of an input signal will affect a given portion of an output signal and whether two input signals will affect the same output signal can help us process events in a discrete-event system more flexibly as shown in the two examples above. In this chapter, I first review a component interface called *causality interface* that captures the causal relationship between an input and an output signal. Then I develop *relevant dependency* that specifies how two input signals affect an output signal. At the end, I show how relevant dependency can be used to facilitate discrete event simulation.

3.1 Causality Interface

Causality interface for actor networks were developed by Zhou et al. in [52]. It is general enough for many models of computation to statically analyze liveness of the network. I review the key results here related to discrete event models.

3.1.1 Dependency Algebra

The *dependency algebra* is a 4-tuple $(D, \leq, \oplus, \otimes)$. The dependency set D contains elements, called *dependencies*, that represent the dependency relations between ports. The

ordering relation $\leq$ is a partial order such that,

$$\begin{aligned}
&\forall\ d \in D, && d \leq d \\
&\forall\ d_1, d_2 \in D, && d_1 \leq d_2 \text{ and } d_2 \leq d_1 \ \Rightarrow\ d_1 = d_2 \\
&\forall\ d_1, d_2, d_3 \in D, && d_1 \leq d_2 \text{ and } d_2 \leq d_3 \ \Rightarrow\ d_1 \leq d_3.
\end{aligned}$$

We use $d_1 < d_2$ to mean $(d_1 \leq d_2) \wedge (d_1 \neq d_2)$.

$\oplus$ and $\otimes$ are two binary operations that satisfy the following axioms.

First, we require that the operators $\oplus$ and $\otimes$ be associative,

$$\forall\ d_1, d_2, d_3 \in D, \quad (d_1 \oplus d_2) \oplus d_3 = d_1 \oplus (d_2 \oplus d_3), \tag{3.1}$$

$$\forall\ d_1, d_2, d_3 \in D, \quad (d_1 \otimes d_2) \otimes d_3 = d_1 \otimes (d_2 \otimes d_3). \tag{3.2}$$

Second, we require that $\oplus$ (but not $\otimes$) be commutative,

$$\forall\ d_1, d_2 \in D, \quad d_1 \oplus d_2 = d_2 \oplus d_1, \tag{3.3}$$

and idempotent,

$$\forall\ d \in D, \quad d \oplus d = d. \tag{3.4}$$

In addition, we require an additive and a multiplicative identity, called $\mathbf{0}$ and $\mathbf{1}$, that satisfy:

$$\begin{aligned}
&\exists\ \mathbf{0} \in D \text{ such that} && \forall\ d \in D, \quad d \oplus \mathbf{0} = d \\
&\exists\ \mathbf{1} \in D \text{ such that} && \forall\ d \in D, \quad d \otimes \mathbf{1} = \mathbf{1} \otimes d = d \\
&\forall\ d \in D, \quad d \otimes \mathbf{0} = \mathbf{0}.
\end{aligned}$$

3.1.2 Causality Interfaces

In timed systems, causality means the time of output events cannot be earlier than the time of input events that caused them. Causality interfaces offer a formalization of this intuition.

A *causality interface* for an actor a with input ports P_i and output ports P_o is a function

$$\delta \colon P_i \times P_o \to D, \tag{3.5}$$

where D is a dependency algebra as defined in the previous section. Ports connected by connectors will always have causality interface $\mathbf{1}$, and lack of dependency between ports will be modeled with causality interface $\mathbf{0}$.

For discrete event models with a totally ordered tag set $\mathcal{T}$, we define the dependency set D to be a set of functions:

$$D = (\mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})), \tag{3.6}$$

where $(X \to Y)$ denotes the set of total functions with domain X and range contained by Y, and $\mathcal{D}(\mathcal{T})$ is the set of down sets of the tag set $\mathcal{T}$. The order relation $\leq$ on the dependency set D is defined as $\forall\ d_1, d_2 \in D$, $d_1 \leq d_2$ if $\forall\ T \in \mathcal{D}(\mathcal{T})$, $d_1(T) \subseteq d_2(T)$.

For input port p and output port p' of an actor a, the causality interface $\delta_a(p, p')$ is interpreted to mean that a signal defined on $T \in \mathcal{D}(\mathcal{T})$ at port p can affect the signal defined on $(\delta_a(p, p'))(T)$ at port p'. That is, there is a causal relationship between the portion of the input signal defined on T and the portion of the output signal defined on $(\delta_a(p, p'))(T)$.

Another way to look at $\delta_a(p, p')$ is from the output signal's point of view. That is, in order to compute the portion of the output signal defined on $(\delta_a(p, p'))(T)$, we need to know the input signal defined on T at port p. In this thesis, we will view causality interface this way since we want to determine what is the least portion of each input signal that is required in order to compute a particular portion of an output signal.

Consider the *Delay* actor with a delay parameter d as an example. The causality interface between its input port p_1 and an output port p_2 is:

$$\forall T \in \mathcal{D}(\mathcal{T}),\ \delta_{\texttt{Delay}}(p_1, p_2)(T) = \{t \in \mathcal{T} | t - d \in T \text{ or } t - d \notin \mathcal{T}\}$$

Note that the causality interface gives us a conservative estimation on which portion of the input signal we need to know to compute the portion of the output signal defined on $(\delta_a(p_1, p_2))(T)$. Given an input event $e_1 = (t_1, v_1)$ at port p_1, the *Delay* actor may produce an event $e_2 = (t_2, v_2)$ where $t_2 \geq t_1 + d$ or no event at all in response to e_1, but it is enough to know the input signal at p_1 up to t_1 to determine the output signal at p_2 up to $t_1 + d$.

A port p' is said to have a *causal* dependency on port p if $d_I \leq \delta(p, p')$. A port p' is said to have a *strict causal* dependency on port p if $d_I \prec \delta(p, p')$, where the relation $\prec$ on D is a strict partial order defined as follows. $\forall\ d_1, d_2 \in D$, $d_1 \prec d_2$ if

1. $d_1 \neq d_2$, and,
2. for each $T \in \mathcal{D}(\mathcal{T})$, $(d_1(T) \subset d_2(T)) \vee (d_1(T) = d_2(T) = \mathcal{T})$, where $\subset$ denotes a strict subset.

A timed actor with at least one input port is said to be causal if every output port has a causal dependency on every input port.

For a functional source actor, we define a fictional *absent input port* ε, and for any output port p_o, $\delta_a(\varepsilon, p_o)$ is given by

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad (\delta_a(\varepsilon, p_o))(T) = \mathrm{dom}(s),$$

where s is the unique signal that satisfies the actor at p_o. If s is complete, $\mathrm{dom}(s) = \mathcal{T}$, then $\delta_a(\varepsilon, p_o) = d_\top$, where $d_\top(T) = \mathcal{T}, \forall\ T \in \mathcal{D}(\mathcal{T})$. A source actor, of course, is always causal.

Similarly, we define the causality interface of a sink actor to be a function that maps an input port p_i of the actor and a fictional *absent output port* to the *bottom function*. I.e.,

$$\delta_a(p_i, \varepsilon) = d_\perp,$$

where $d_\perp(T) = \emptyset, \forall\ T \in \mathcal{D}(\mathcal{T})$.

The causality interface for a connector is simply the multiplicative identity $\mathbf{1} = d_I$.

The $\oplus$ operation computes the greatest lower bound of two elements in D. I.e., $\forall\ d_1, d_2 \in D$, the function $(d_1 \oplus d_2)\colon \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$ is defined by

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad (d_1 \oplus d_2)(T) = d_1(T) \cap d_2(T). \tag{3.7}$$

The $\otimes$ operator is function composition. I.e., $\forall\ d_1, d_2 \in D$, the function $(d_1 \otimes d_2)\colon \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$ is defined by

$$d_1 \otimes d_2 = d_2 \circ d_1$$

or

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad (d_1 \otimes d_2)(T) = d_2(d_1(T)).$$

The additive identity $\mathbf{0}$ is the *top function*, $d_\top \colon \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$, given by

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad d_\top(T) = \mathcal{T}.$$

The multiplicative identity $\mathbf{1}$ is the *identity function*, $d_I \colon \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$, given by

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad d_I(T) = T.$$

With these definitions, the dependency set (3.6) satisfies all of the axioms described in section 3.1.1.

3.1.3 Composition of Causality Interfaces

Given a set A of actors, a set C of connectors, and the causality interfaces for the actors and the connectors, the causality interfaces of the composition is determined by using $\otimes$ for serial composition and $\oplus$ for parallel composition. To do this, we form a weighted, directed graph $G = \{P, E\}$, called the *dependency graph*, where P is the set of ports in the composition and E is the set of edges. If p is an input port and p' is an output port, there is an edge in G between p and p' if p and p' belong to the same actor a and $\delta_a(p, p') \neq \mathbf{0}$. In such a case, the weight of the edge is $\delta_a(p, p')$. If p is an output port and p' is an input port, there is an edge between p and p' if there is a connector between p and p'. In this case, the weight of the edge is $\mathbf{1}$. In all other cases, the weight of an edge would be $\mathbf{0}$, but we do not show such edges. $\forall p, p' \in P$, to compute the value of $\delta(p, p')$, we need to consider

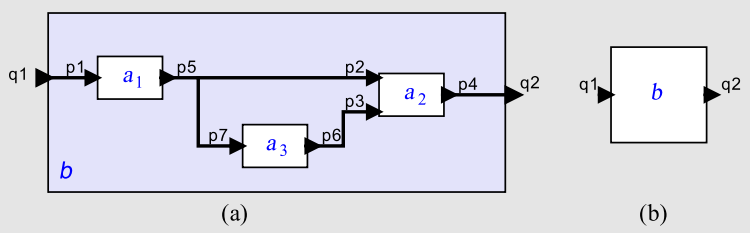


Figure 3.3. A feedforward composition with parallel paths.

all the paths between p and p'. We first discuss feed-forward compositions and then deal with feedback compositions.

A feed-forward system does not have any cycles in its dependency graph. For example, figure 3.3 shows a feed-forward composition, which is abstracted into a single actor b with external input port $q1$ and output port $q2$. To determine the causality interface of actor b, we need to consider all the paths from $q1$ to $q2$, and $\delta_b(q1, q2)$ is given by

$$\begin{aligned}\delta_b(q1,q2) = \quad & \delta_c(q1,p1) \otimes \delta_{a_1}(p1,p5) \otimes [(\delta_c(p5,p2) \otimes \delta_{a_2}(p2,p4)) \oplus \\ & (\delta_c(p5,p7) \otimes \delta_{a_3}(p7,p6) \otimes \delta_c(p6,p3) \otimes \delta_{a_2}(p3,p4))] \otimes \delta_{c3}(p4,q2),\end{aligned}$$

where δ_{a_1}, δ_{a_2} and δ_{a_3} are the causality interfaces for actors a_1, a_2 and a_3, respectively, and δ_c are the causality interfaces for connectors. Since connectors have causality interface $\mathbf{1}$, the above equation simplifies to

$$\delta_b(q1,q2) = \delta_{a_1}(p1,p5) \otimes [\delta_{a_2}(p2,p4) \oplus (\delta_{a_3}(p7,p6) \otimes \delta_{a_2}(p3,p4))]. \tag{3.8}$$

The dependency graph of a feedback system contains cyclic paths. Given a cyclic path $c = (p_1, p_2, ..., p_n, p_1)$, where p_i's $(1 \leq i \leq n)$ are ports of the composition, we define the *gain* of c to be:

$$g_c = \delta(p_1,p_2) \otimes \delta(p_2,p_3) \otimes ... \otimes \delta(p_n,p_1).$$

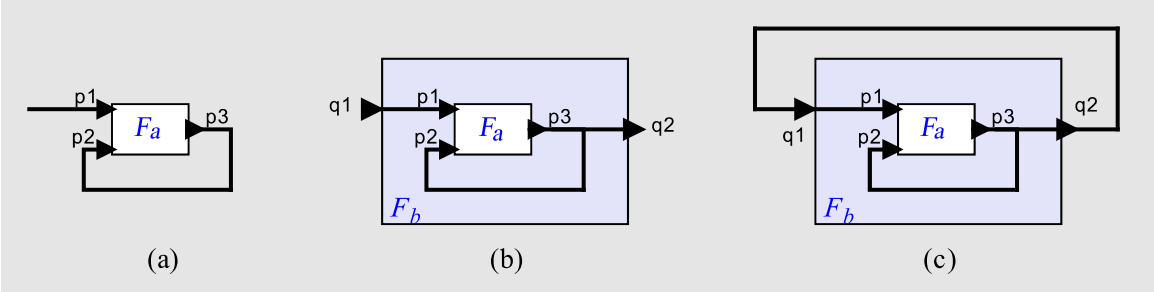


Figure 3.4. An open composition with feedback loops.

Causality interface compositions for feedback systems are more complex and depend on the semantics of the model of computation. For live timed actor networks, [52] shows that the causality interface for systems as shown in figure 3.4 has a much simpler form. The following theorem gives a necessary condition for a timed actor network to be live.

Theorem 1. *A finite network of continuous and live actors where the tag set $\mathcal{T}$ is totally-ordered is continuous and live if and only if for every cyclic path c in the dependency graph, $\mathbf{1} \prec g_c$, where $\mathbf{1} = d_I$ is the multiplicative identity.*

Given the composite actor b as shown in figure 3.4(b), actor b is live if and only if actor a is live and $\mathbf{1} \prec \delta_a(p2, p3)$. Assuming $\mathcal{D}(\mathcal{T})$ is totally-ordered, the causality interface of b is given by

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad \delta_b(q1, q2)(T) = \delta_a(p1, p3)(T) \cap T_0$$

where T_0 is the least fixed point of $\delta_a(p2, p3)$. If $d_I \prec \delta_a(p2, p3)$, then $T_0 = \mathcal{T}$, and therefore $\delta_b(q1, q2) = \delta_a(p1, p3)$.

We can continue to add ports to actor a to construct any actor networks. The above analysis on causal dependencies can be adapted easily.

3.2 Relevant Dependency

As the example shown in figure 3.2, $\forall T \in \mathcal{D}(\mathcal{T})$, to compute an output signal, say s_4, defined on T, we need to know the corresponding portions of the input signals, s_1, s_2 and s_3, that can causally affect s_4. While causality interfaces provide a powerful tool to specify and reason about causal relationships among actors, these dependency values between an input port and an output port do not tell the whole story. Consider the *Merge* actor in figure 3.2, with two input ports. When we construct the dependency graph, it is easy to find that there is no path between the two input ports p_1 and p_2. But, these two ports are not completely independent. In fact, the *Merge* actor cannot react to an event at one port with time stamp t until it is sure it has seen all events at the other port with time stamp less than or equal to t. This fact is not captured in the causal dependencies. To capture it, I define *relevant dependencies*.

A *relevant dependency* for a composition with input ports P_I is a function

$$d\colon P_I \times P_I \to D, \tag{3.9}$$

where $D = (\mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T}))$ as defined in the previous section.

Note that unlike a causality interface that is specified between an input and an output port, relevant dependency is defined on a pair of input ports. The relevant dependency $d(p_1, p_2)(T)$ indicates which portion of signal s_2 can be processed with s_1 defined on $T \in \mathcal{D}(\mathcal{T})$.

The relevant dependency between ports in a composition can be calculated in a way

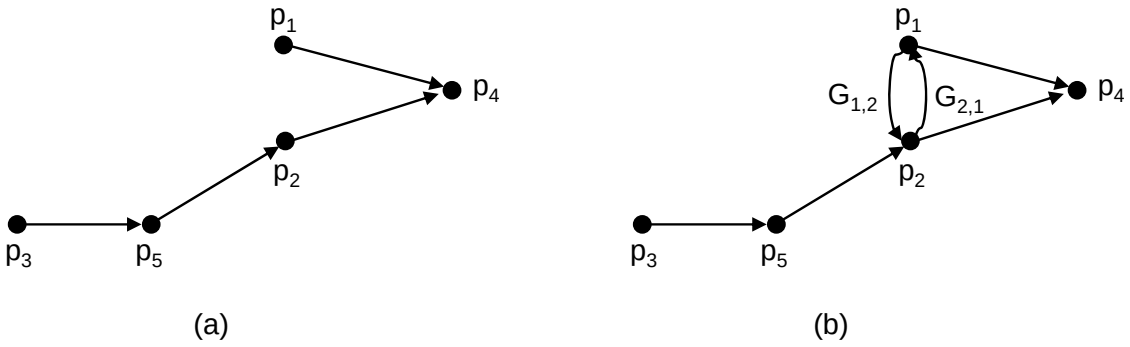


Figure 3.5. The causality and relevant dependency graphs for the model in figure 3.2.

similar to the causal dependency above. We introduce a binary relation $G_{i,j}$ between input port p_i and p_j of the same actor. Specifically, considering an individual actor a, $G_{1,2} = \mathbf{1}$ if two input ports p_1 and p_2 of a will affect the same output port; otherwise $G_{1,2} = \mathbf{0}$. Formally, $\forall p_i, p_j$ of some actor a, $p_i \neq p_j$,

$$G_{i,j} = \begin{cases} \mathbf{1} & \exists p \in P_o, \text{ such that } \delta_a(p_i, p) \neq \mathbf{0} \text{ and } \delta_a(p_j, p) \neq \mathbf{0}, \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

where P_o is the set of output ports of a. Ports p_i and p_j are called *equivalent* if $G_{i,j} = \mathbf{1}$.

For example, in figure 3.2, both input ports of the *Merge* actor affect its output port, i.e. $\delta_{Merge}(p_1, p_4) \neq \mathbf{0}$ and $\delta_{Merge}(p_2, p_4) \neq \mathbf{0}$. Thus $G_{1,2} = \mathbf{1}$.

In addition, we assume that if any actor has state that is modified or used in reacting to events at more than one input port, then that state is treated as a hidden output port. Thus, with the above definition, two input ports are equivalent if they are coupled by the same state variables of the actor.

We next modify the dependency graph by adding edges between equivalent ports, i.e.

there is an edge with weight $\mathbf{1}$ from p_i to p_j if $G_{i,j} = \mathbf{1}$, to create a new graph that we call the *relevant dependency graph*. Similar to causal dependencies, relevant dependencies are calculated by examining weights of the relevant dependency graph. $\forall p_i, p_j \in P$, to compute the value of $d(p_i, p_j)$, we need to consider all the paths between p_i and p_j. We again combine parallel paths using $\oplus$ and serial paths using $\otimes$. It is easy to show that if every actor in a composition is causal, then $d_I \leq d(p_i, p_j)$.

$\forall T = D(t_1) \in \mathcal{T}$, the relevant dependency $d(p_1, p_2)(T) = D(t_2)$ means that any events with a tag $t \in d(p_1, p_2)(T)$ at port p_2 can be processed when events of the signal at port p_1 are known up to t_1. If $t_2 > t_1$, then $d(p_1, p_2)(T) = [0, t_2] \supset [0, t_1]$. This means that we can process events at p_2 up to t_2 with knowing events at p_1 only up to t_1. In particular, $d(p_1, p_2) = d_\top$ indicates that events at port p_2 can be processed without knowing anything about events at port p_1.

Consider the model shown in figure 3.2 as an example. The causality interface for each actor in the model is:

$$\begin{aligned} &\delta_{Delay}(p_3, p_5) = (T \mapsto \{t \in \mathcal{T} | t - d \in T \text{ or } t - d \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T}) \\ &\delta_{Merge}(p_1, p_4) = \mathbf{1}, \quad \delta_{Merge}(p_2, p_4) = \mathbf{1} \end{aligned} \tag{3.10}$$

The causality interface for each connector is $\mathbf{1}$. There is only one connector in this example, i.e. $\delta_c(p5, p2) = \mathbf{1}$.

Figure 3.5(a) shows the causality dependency graph for the example model in figure 3.2. The weights of edges are not explicitly shown in the graph. We can compute the causality

interface for the composition as follows:

$$
\begin{aligned}
&\delta(p_1, p_4) = \delta_{Merge}(p_1, p_4) = \mathbf{1} \\
&\delta(p_2, p_4) = \delta_{Merge}(p_2, p_4) = \mathbf{1} \\
&\delta(p_3, p_4) = \delta_{Delay}(p_3, p_5) \otimes \delta_c(p5, p2) \otimes \delta_{Merge}(p_2, p_4) \\
&\qquad\qquad = (T \mapsto \{t \in \mathcal{T} | t - d \in T \text{ or } t - d \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T}) \\
&\delta(p_1, p_5) = \mathbf{0} \\
&\delta(p_2, p_5) = \mathbf{0} \\
&\delta(p_3, p_5) = \delta_{Delay}(p_3, p_5) \\
&\qquad\qquad = (T \mapsto \{t \in \mathcal{T} | t - d \in T \text{ or } t - d \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T})
\end{aligned}
\tag{3.11}
$$

Based on the causality interface of actors in the model, we can determine that binary relation $G_{i,j}$ between input ports p_i and p_j for each actor.

$$
G_{1,2} = \mathbf{1}, \quad G_{2,1} = \mathbf{1} \tag{3.12}
$$

Figure 3.5(b) shows the relevant dependency graph for this example. It is easy to check

the relevant dependency in this composition.

$$
\begin{aligned}
d(p_1, p_2) &= G_{1,2} = \mathbf{1} \\
d(p_2, p_1) &= G_{2,1} = \mathbf{1} \\
d(p_1, p_3) &= \mathbf{0} \\
d(p_3, p_1) &= \delta(p_3, p_5) \otimes \delta_c(p5, p2) \otimes G_{2,1} \\
&= (T \mapsto \{t \in \mathcal{T} | t - d \in T \text{ or } t - d \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T}) \\
d(p_2, p_3) &= \mathbf{0} \\
d(p_3, p_2) &= \delta(p_3, p_5) \otimes \delta_c(p5, p2) \\
&= (T \mapsto \{t \in \mathcal{T} | t - d \in T \text{ or } t - d \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T})
\end{aligned}
\tag{3.13}
$$

$\forall T \in \mathcal{T}, d(p_3, p_1)(T) = \{t \in \mathcal{T} | t - d \in T \text{ or } t - d \notin \mathcal{T}\})$ means that any events with a tag $t \in d(p_3, p_1)(T)$ at port p_1 can be processed when the signal at port p_3 are defined on T. This flexibility in processing events is precisely the result we were after as discussed at the beginning of this chapter.

Intuitively, relevant dependency indicates which portion of a signal can be processed at a port without knowing all the events on the other ports in a composition. By knowing how signals on different ports depends on each other, we can advance time stamp on ports at different paces, while the conventional discrete event simulation approach advance all ports to the same time stamp each step.

Note that relevant dependency is asymmetric. For example, $d(p_1, p_3)$ does not equal to $d(p_3, p_1)$ since the output that depends on events at p_3 does not depends on events at p_1 while the output that depends on events at p_1 does depends on events at p_3.

From the definition of relevant dependency, we have the following Corollary,

Corollary 1. *Let P_I be the set of input ports of the a composition of causal actors.* $\forall p_1, p_2 \in P_I, \forall T \in \mathcal{D}(\mathcal{T}), d(p_1, p_2)(T) \supseteq T$

That is, $\forall p_i \in P_I$, if the signals on other input ports are defined on $T = D(t)$, then we can process events of s_i at least to t.

3.3 Relevant Order

What we gain from the dependency analysis is that we can process certain events out of their linear chronological order. Consider the model shown in figure 3.2 as an example. An event e at port p_1 with time stamp t_1 can be processed if the events at port p_3 are known up to t_3 and $D(t_1) = d(p_1, p_2)(D(t_3))$. Given $d > 0$, we have $t_1 = t_3 + d$. This means we can process e without knowing any events at p_1 with time stamp in $D(t_1) \setminus D(t_3)$, while the conventional discrete event simulation approach based on chronological order would require we know all events at p_3 in $D(t_1)$ before e can be processed. Essentially, we can introduce a new order, which we call *relevant order* ($\leq_r$), on events based on relevant dependency analysis.

We define the *relevant order* on events as follows,

Definition 3.1 (Relevant Order)**.** Let $e_1 = (t_1, v_1)$ be an event at at port p_1 and $e_2 = (t_2, v_2)$ be an event at at port p_2.

$$e_1 <_r e_2 \Longleftrightarrow d(p_1, p_2)(D(t_1)) \subset D(t_2).$$

Recall $D(t) = \{\tau \in T | \tau \leq t\}$ for some $t \in T$ is the smallest down set including t. That

is, $e_1 <_r e_2$ if processing the signal defined on $D(t_2)$ at port p_2 depends on the signal defined on $D(t_1)$ at port p_1. We interpret $e_1 <_r e_2$ as e_1 needs to be processed before e_2.

Two events e_1 and e_2 are not comparable, denoted as $e_1||_r e_2$, if neither $e_1 <_r e_2$, nor $e_2 <_r e_1$. If $e_1||_r e_2$, then e_1, e_2 can be processed in any order.

We have the following lemma that relates the relevant order and the ordinary time stamp order of events,

Lemma 1. $\forall e_1, e_2, e_1 <_r e_2 \Rightarrow e_1 < e_2$.

Proof. $e_1 <_r e_2 \Rightarrow d(p_1, p_2)D(t_1) \subset D(t_2)$. From corollary 1, we have $d(p_1, p_2)D(t_1) \supset D(t_1)$. Then we have $D(t_1) \subset D(t_2)$. This means $t_1 < t_2$, hence $e_1 < e_2$. □

3.4 Execution Based on Relevant Order

We now design execution strategies based on the relevant order to enable out of order execution without hurting determinism.

Following the concept used in distributed snapshot, we define *cut* and *consistent cut* for discrete event systems.

Definition 3.2 (Cut)**.** Given a composition with input port P_I and output port P_O, a *cut* $c = \bigcup_{\{p_i \in P_I\}} s_i \upharpoonright D(t_i)$. That is, a cut contains a prefix of each input signal.

Definition 3.3 (Consistent Cut)**.** A cut c is called consistent if $\forall e_1 <_r e_2, e_2 \in c \Rightarrow e_1 \in c$.

That is, if a consistent cut contains an event e, it also contains all events that are relevantly ordered smaller than e.

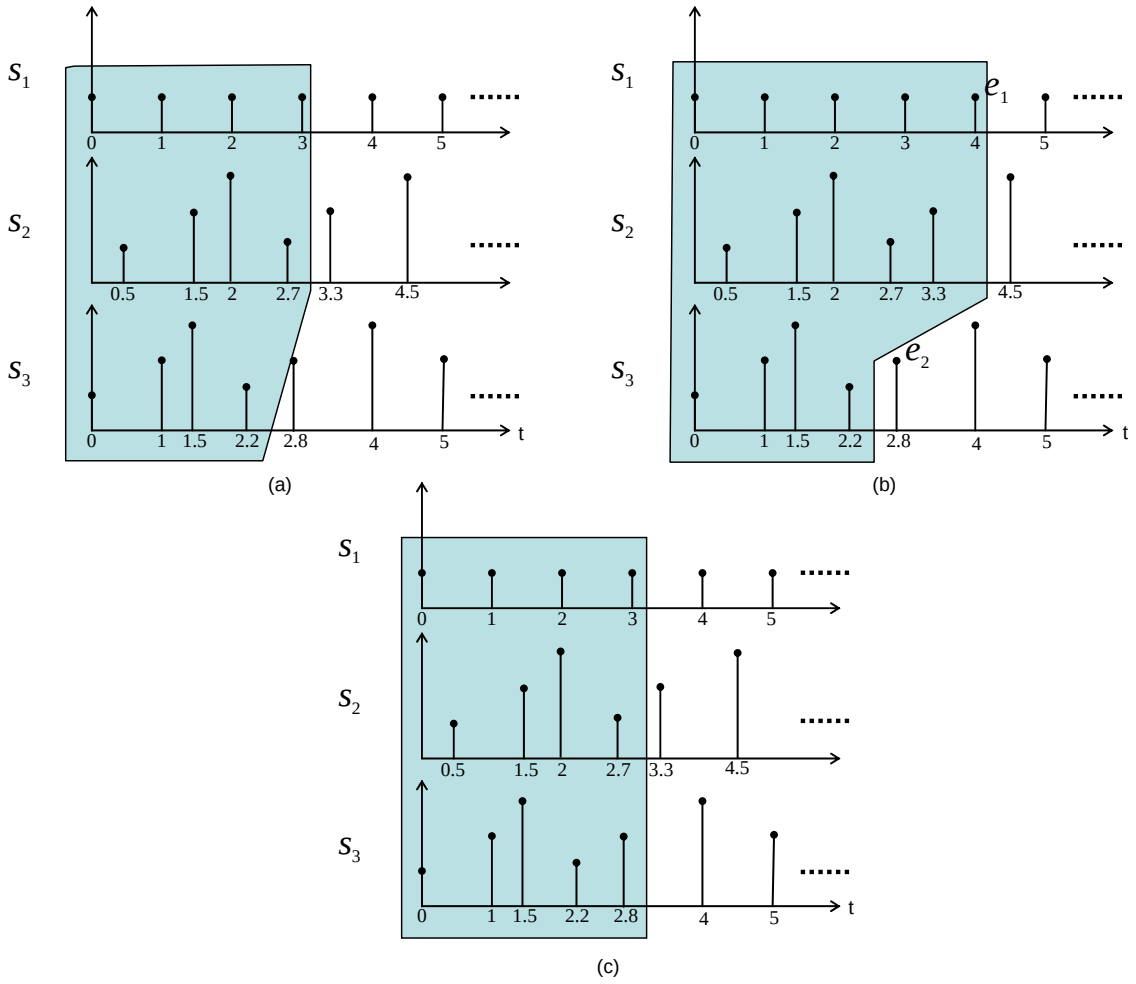


Figure 3.6. Three different cuts.

Figure 3.6 shows three different cuts given that s_1, s_2 and s_3 are the three input signals at port p_1, p_2 and p_3 for the composition shown in figure 3.2. Suppose the *Delay* actor has a delay parameter $d = 0.5$, then the cut in figure 3.6 (a) and (c) are both consistent cuts as we can verify that $\forall e_1 <_r e_2, e_2 \in c \Rightarrow e_1 \in c$, but the cut in 3.6 (b) is not, i.e. $e_1 <_r e_2$ but it is not contained by the cut.

The execution algorithm based on relevant order works as follows:

1. Start with E, a set of events in the event queue.
2. Choose $r \subset E$, s.t. each event in r is *minimal* in E.
3. Process events in r, which may produce a set of new events E'.
4. Update$E$ to $(E \setminus r) \cup E'$.
5. Go to 2.

An event $e = (t, v)$ at port p is *minimal* in E if $\forall e' \in E, e <_r e',$ or $e||_r e'$.

Each iteration i, we process a set of events r. Let $c_i = c_{i-1} \cup r$, and $c_0 = \emptyset$, be the set of events that have been executed. It is easy to see that c_i is a consistent cut.

The progress of a discrete-event simulation can be visualized as the forward movement of the frontier of a consistent cut. Since events are only partially ordered by the relevant order, there may be multiple ways to construct consistent cuts, hence multiple ways to compute the behavior of a discrete-event system. However, they all approximate the same fixed point. The conventional discrete-event simulation based on chronological order is just one way to construct consistent cuts, which requires the cut of all the input signals to be

at the same time tag t and processes all events with time stamp smaller than t before it moves to the next cut.

Definition 3.4 (Synchronized Cut)**.** A cut c is called synchronized if $\forall e_1 \leq e_2, e_2 \in c \Rightarrow e_1 \in c$.

Note $e_2 \leq e_1$ is defined based on their time stamp order. If the latest event contained by a synchronized cut c has time stamp t, then we say c is defined up to t. It is easy to see that a synchronized cut defined up to t contains a prefix of each input signal restricted to the same down set $D(t)$, i.e. $c = \bigcup_{\{p_i \in P_I\}} s_i \upharpoonright D(t)$.

Lemma 2. *If a cut c is synchronized, it is consistent.*

Proof. $\forall e_1, e_2, e_1 <_r e_2 \Rightarrow e_1 < e_2$. Then if $e_2 \in c$, we have $e_1 \in c$. So c is consistent. □

For the two consistent cuts shown in figure 3.6 (a) and (c), the cut in (c) is also a synchronized cut, but (a) is not.

Lemma 2 means given a set E of events, if the conventional discrete-event simulation approach can process events on each input port up to t, then the simulation based on relevant order of events can at least process events on each input port up to t. If $\exists e_1 = (t_1, v_1)$ with $t_1 > t$ at port p such that any event $e_2 <_r e_1$ has time stamp $t_2 < t$, then the set of events can be processed based on relevant order is strictly larger than that can be processed by the conventional approach.

Chapter 4

Application to Real-Time Systems

DE models have primarily been used in performance modeling and simulation, where time stamps are a modeling property bearing no relationship to real time during execution of the model. This chapter extends DE models with the capability of relating certain events to physical time. In this approach, DE is not a simulation technology, but rather an application specification language, which serves as a semantic basis for obtaining determinism in distributed real-time systems. Applications are given as distributed DE models, where for certain events, their modeling time is mapped to physical time. For example, a programmer may specify that an actuator must produce a physical output at the time determined by the time stamp of an event sent to the actuator. When these models are executed in a runtime environment that ensures DE semantics, we know that the applications will have deterministic behaviors regardless of the actual implementations. I call this programming model PTIDES (pronounced "tides"), for Programming Temporally Integrated Distributed Embedded Systems.

Preserving DE semantics at runtime can be challenging, since the global, consistent notion of time may lead to a total ordering of execution in a distributed system, an unnecessary waste of resources. The execution strategy based on *relevant dependencies* and *relevant orders* developed in chapter 3 can be used to build a more efficient run-time environment for distributed real-time systems specified by PTIDES models. The runtime environment is divided into two layers: global coordination, and local resource scheduling. When receiving an event from the network, the global coordination layer determines whether the event can be processed immediately or it needs to wait for other potentially preceding events based on the underlying DE semantics. Once it is sure that the current event can be processed, it hands the event over to local resource scheduler, which may use existing real-time scheduling algorithms, such as earliest deadline first (EDF) to prioritize the processing of all pending events. This chapter focuses on the global coordination layer, which is key to achieving DE semantics in distributed systems. Real-time scheduling and schedulability analysis will be discussed in chapter 5.

Unlike many hard real-time distributed systems that depend on domain specific network architectures, the only assumption of communication behavior in PTIDES is that it delivers packets reliably with a known bounded delay. PTIDES relies on network time synchronization [24], where the computing nodes on the network share a common notion of time to a known precision. This has the potential for being lightweight and delivering repeatable and predictable behaviors at a variety of timing precisions. Network time synchronization is available on a variety of platforms, including standard computers on the Internet (e.g. NTP [39]), time-triggered buses such as TTA or FlexRay [25], TCP/IP over

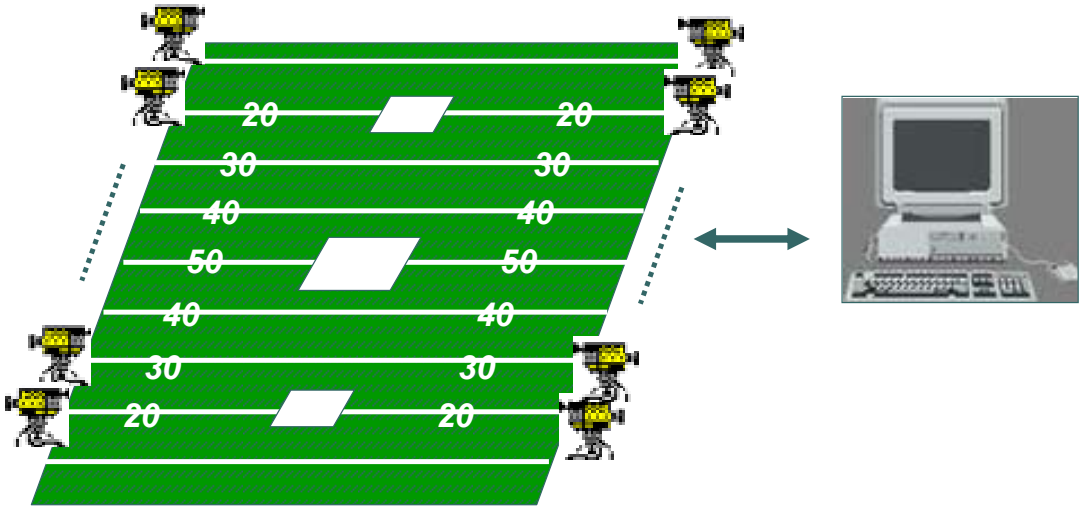


Figure 4.1. Networked camera application.

Ethernet (e.g. IEEE 1588), and wireless networks (e.g. RBS [13]). Implementations of IEEE 1588 have demonstrated time synchronization as precise as tens of nanoseconds over networks that stretch over hundreds of meters, more than adequate for many manufacturing and instrumentation systems.

4.1 Motivating Example

This chapter uses a camera network as a motivating scenario and a running example in later discussions. Throughout this chapter, I use t for model time and τ for physical time.

Consider N cameras connected via Ethernet are distributed over a football field as shown in figure 4.1. Suppose that the clocks at all the cameras and the central computer are precisely synchronized. All the cameras have computer-controlled picture and zoom capabilities. Each camera only has a partial view of the field. We can control a camera

to take a picture or zoom precisely at some physical time. Precise time synchronization allows cameras to take sequences of pictures simultaneously. The images produced by each camera are time stamped and transferred over the network to the central computer, where the images get processed to produce a composite view of the field or a sequence of views for some interesting moment. A user sitting in front of the central computer may issue commands to the cameras to zoom or change the frequency at which images are taken. Suppose that zooming takes time κ to stabilize, and during this period of time no picture should be taken. Given that the commands controlling the cameras are transmitted over the network with a bounded delay, the challenges here are how to coordinate the zooming and picture taking actions properly on each camera so that the retrieved images are precisely synchronized.

This application is inspired by the "eye vision" project[1] at CMU and CBS Television. However, rather than focusing on the challenges in real-time image processing and control, this chapter considers how to program the whole system at a high level and how to realize the timing relations in the system. The principles are general enough to apply in many scenarios that require distributed time-coordinated physical actions.

4.2 PTIDES Programming Concepts

Taking an actor-oriented approach for building discrete-event systems [27], components in PTIDES are actors with further annotations. Borrowing ideas from TinyOS and nesC [17], PTIDES introduces real-time components as thin wrappers around hardware to DE

[1] http://www.ri.cmu.edu/events/sb35/tksuperbowl.html

models. The time stamps in a PTIDES model are values of model time. However, some actors can bind model time to physical time by imposing real-time constraints.

4.2.1 Relating Model time to Real Time

Definition 4.1 (Real-time Constraint)**.** A *real-time constraint* is a function from model time to physical time, $\gamma\colon \mathcal{T} \to \mathbb{R}_0$, where $\mathcal{T}$ is the set of model time and $\mathbb{R}_0$, the non-negative real numbers, is the set of physical time.

That is, a real-time constraint maps a model time to a physical time. In this thesis, $\mathbb{R}_0 = [0, \infty)$ will be used as a representative for the tag set $\mathcal{T}$. The idea can be easily generated to other choices of the tag set.

We say a port is a *real-time port* if there is a real-time constraint associated with it. If an input port is a real-time port with constrain γ, then an event $e = (t, v)$ at this port must be processed before $\gamma(t)$. If an output port is a real-time port with constrain γ, it means an event $e = (t, v)$ will be generated at this port no later than $\gamma(t)$.

Actors that wrap the interaction with some underlying hardware generally introduce real-time constraints and hence have real-time input or output ports. A *Sensor* actor is a thin wrapper for sensors or input devices. It is modeled as a source actor with one or more real-time output ports. The interrupt service routine associated with the hardware would invoke a method in the actor to post events that satisfies the real-time constraints associated with its output ports. An *Actuator* actor is a thin wrapper for actuators or output devices. It is modeled as a sink actor with one or more real-time input ports. Events at its input ports need to be delivered to the hardware driver to fulfill the real-time constraints. We

may also have actors as both sensor and actuator, which have real-time input and output ports. The *Camera* actor discussed in the next section is an example of such an actor.

We limit the relationship of model time to physical time to only those circumstances where this relationship is needed. For other actors in the model, there is no real-time constraint and model time is used to define execution semantics.

4.3 Specification of the Motivating Example

PTIDES uses DE as an application specification language, which serves as a semantic basis for obtaining determinism in distributed real-time systems.

Cameras in this application are both sensors and actuators. We need to generate precisely timed physical actions, like picture taking and zooming, at each camera, and the cameras respond with time stamped images. We model a camera as an actor that has one input port and two output ports, depicted graphically as follows:

This actor is a software component that wraps the interaction with the camera driver. At its input port, it receives a potentially infinite number of events in chronological order. The time stamp of an event specifies when an action should be taken, and the value of the event dictates what kind of action (zooming level or shutter speed) should be taken. Obviously, in order to generate the physical action at the actuator, it needs to receive an input event with time stamp t at some physical time $\tau \leq t$. Hence the input port is a

real-time port with constraint $\gamma(t) = t$. If desired, we can also specify a *setup time* σ for a real-time input port, in which case it requires that each input event with time stamp t be received before physical time reaches $t - \sigma$. That is, the real-time constraint is $\gamma(t) = t - \sigma$. In this example, we simply assume $\sigma = 0$

Assume there is a response delay $\mu \geq 0$ of the digital output device. The first output port of the *Device* actor produces an event for each input event $e = (t, v)$, where the time stamp of the output event $t + \mu$ is strictly greater than that of the input event, to indicate the physical action has completed. The real-time constraint for the first output port is $\gamma(t) = t$. The second output port produces the time-stamped image and sends it to the central computer. The time stamp of the output event on the second output port is the same as the time stamp of the input event. The real-time constraint for the first output port is $\gamma(t) = t + \mu$.

Figure 4.2 shows a distributed DE model to be executed on the cameras and the central computer platform. The dashed boxes divide the model into two parts, the top one to be executed on each camera and the bottom one to be executed on the central computer. The parts communicate via signal s_1 and s_2. We assume that events in these signals are sent over the network as time-stamped values.

The *Command* actor is a Sensor actor. It wraps interactions with the user input device. When a user input comes in, the *Command* actor checks with its synchronized clock for the current time, uses the returned time value to time stamp the input message and sends the time stamped message to all the cameras. Its output port is a real-time port with constraint

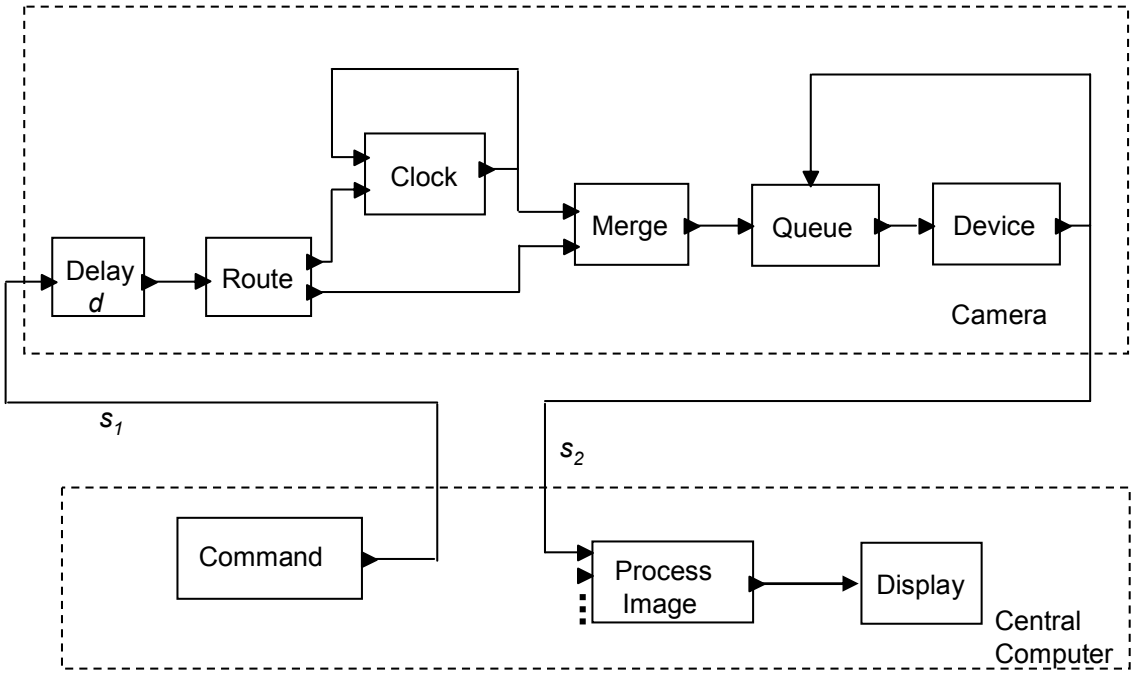


Figure 4.2. Specification of the Networked camera application.

$\gamma(t) = t$, i.e. an event with time stamp t is generated at physical time no later than t. In particular, this actor will output an an event with time stamp t exactly at physical time $\tau = t$.

All other actors in the model shown in figure 4.2 do not impose real-time constraints on their ports. The right part of the model on the central computer processes the images taken at each camera and displays the result.

The *Clock* actor produces time-stamped outputs where the time stamp is some integer multiple of a period p. The time stamps are used to control when the camera takes pictures. The period can be different for each clock and can be changed during run time upon receiving an input on the second input port. If there is an event with value v and time stamp t at

the second input, the *Clock* actor will scale its period from p to $p' = p * v$ and produce an output with time stamp $t_0 + np'$ where t_0 is the time stamp of the last output and n is the smallest integer so that $t_0 + np' \geq t$. We can specify a minimal period P_{min} for the *Clock* actor. The feedback loop around the clock actor is used to trigger the next output, and we assume there is an initial event on the first input at the beginning.

Two kinds of user commands are received at each camera, the *change frequency* command and the *adjust zoom* command. The *Router* actor separates these events and sends the *change frequency* events to the *Clock* actor and the *adjust zoom* events to the Merge actor.

The *Delay* actor is the same actors as defined in chapter 2. The *Delay* actor with a delay parameter d will produce an event with time stamp $t + d$ at its output given an event with time stamp t at its input.

The *Merge* actor merges the events on the two input ports in chronological order. It gives higher priority to the second input port when there are simultaneous events at both of its input ports. That is, we give higher priority to the user to control a camera.

The *Queue* actor buffers its input event until an event is received at the trigger port, which is the one at the top of the actor. The *Device* actor sends a trigger event to the *Queue* actor when a physical action is done, and we assume there is an initial event on the trigger port at the beginning. The feedback loop around the *Queue* and *Device* actor ensures that the *Device* does not get overwhelmed with future requests. It may not be able to buffer those requests, or it may have a finite buffer.

As shown in this example, PTIDES programs are discrete-event models constructed as networks of actors. For each actor, we specify a physical host to execute the actor. To represent the underlying communication explicitly between actors on different hosts, we use a *NetworkOut* actor for sending events over the network and a *NetworkIn* actor for receiving events from the network. Figure 4.3 shows the models running on each camera and the central computer platform with the network communication explicitly modeled and ports named. We assume distributed models interact with each other via "real-time" events, i.e. events send or received by the *NetworkOut* or *NetworkIn* actor have time stamps that are related to physical time. The *NetworkOut* actor is an *Actuator* actor with real-time constraint on its input port. In particular, we specify that the *NetworkOut* actor on the central computer has real-time constraint $\gamma(t) = t$, while the *NetworkOut* actor on each camera has constraint $\gamma(t) = t + \mu$, assuming the image processing part on the central computer can tolerate some latency $\mu + \Delta$ on receiving images. We assume the communication network delivers packets reliably with a known bounded delay Δ. The *NetworkIn* actor in the camera model is a *Sensor* actor with real-time constraint $\gamma(t) = t+\Delta$ on its output port. That is an event with time stamp t will be produced at its output port no later than physical time $t + \Delta$. The *NetworkIn* actor in the central computer model has real-time constraint $\gamma(t) = t + \mu + \Delta$.

We view the model shown in figure 4.3 as a representative scenario for many distributed embedded applications. For example, the *Command* actor can be viewed as an example of sensor components, the *Device* actor is an example of actuator components, and other actors in between are the control or computation part. The problems discussed in this paper

are common to many distributed sensing and actuation systems, such as manufacturing, instrumentation, surveillance, and scientific experiments.

4.4 Run-time Environment

How to build a run-time environment to execute the distributed model shown in figure 4.3 to deliver the correct behavior and meet the real-time constrains is a challenging problem. A first-come-first-serve strategy cannot preserve deterministic DE semantics, since the network may alter the order that events are delivered. A brute-force implementation of a conservative distributed DE execution of this model would stall execution in a camera at some time stamp t until an event with time stamp t or larger has been seen on signal s_1. Were we to use the Chandy and Misra approach [8], we would insert null events into s_1 to minimize the real-time delay of these stalls. However, this brute-force technique will unnecessarily postpone the release time of events even when these events can be safely processed. The so-called "optimistic" techniques for distributed DE execution will also not work in our context. Optimistic approaches perform speculative execution and backtrack if and when the speculation was incorrect [23]. Since we have physical interactions in the system, backtracking is not possible.

In this section, I show how to apply dependency analysis to the model running on each camera and how the execution strategy based on *relevant order* can help to meet real-time constraints in the model.

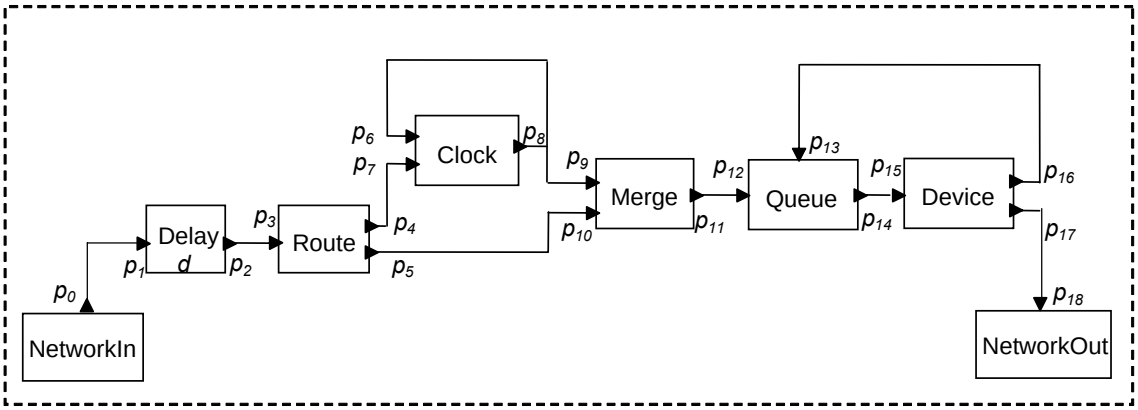
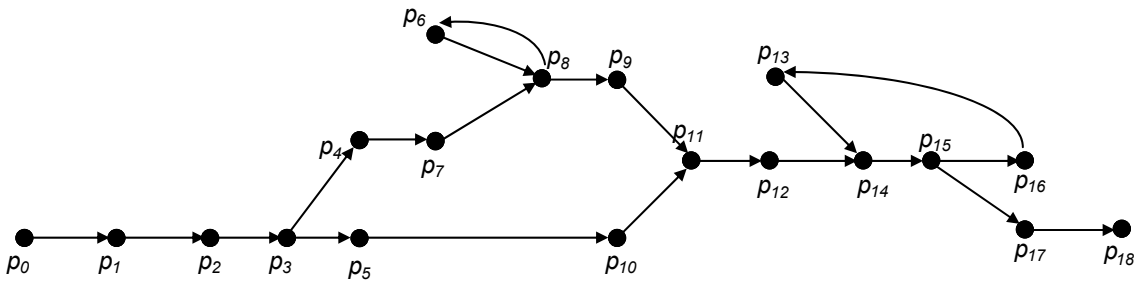


Figure 4.3. The program on the camera.

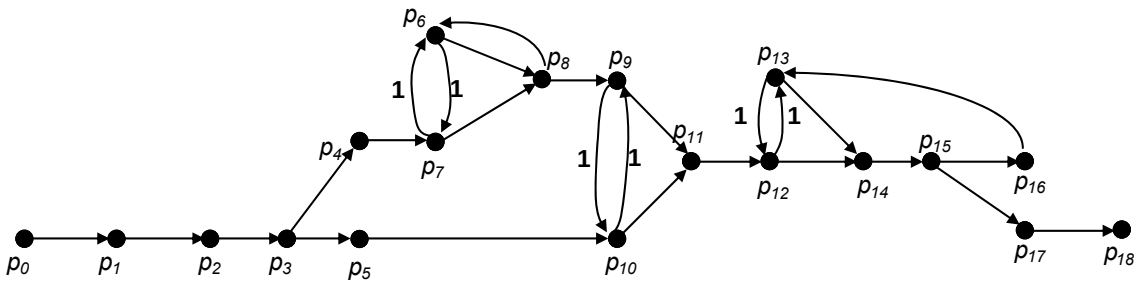
4.4.1 Dependency Analysis

Consider the model shown in figure 4.3 as an example. The causality interface for each actor in the model is:

$$
\begin{aligned}
&\delta_{Delay}(p_1, p_2) = (T \mapsto \{t \in \mathcal{T} | t - d \in T \text{ or } t - d \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T}),\\
&\delta_{Router}(p_3, p_4) = \mathbf{1}, \quad \delta_{Router}(p_3, p_5) = \mathbf{1},\\
&\delta_{Clock}(p_6, p_8) = (T \mapsto \{t \in \mathcal{T} | t - P_{min} \in T \text{ or } t - P_{min} \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T}),\\
&\delta_{Clock}(p_7, p_8) = \mathbf{1},\\
&\delta_{Merge}(p_9, p_{11}) = \mathbf{1}, \quad \delta_{Merge}(p_{10}, p_{11}) = \mathbf{1},\\
&\delta_{Queue}(p_{12}, p_{14}) = \mathbf{1}, \quad \delta_{Queue}(p_{13}, p_{14}) = \mathbf{1},\\
&\delta_{Device}(p_{15}, p_{16}) = (T \mapsto \{t \in \mathcal{T} | t - \mu \in T \text{ or } t - \mu \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T}),\\
&\delta_{Device}(p_{15}, p_{17}) = \mathbf{1}
\end{aligned}
\tag{4.1}
$$



(a)



(b)

Figure 4.4. The causality and relevant dependency graphs for the camera

.

where P_{min} represents the minimum time interval between two consecutive picture taking actions at the camera, and $\mu \geq 0$ is the response delay of the digital output device (i.e. the minimum model-time delay across the input and the first output of the *Device* actor).

Figure 4.4 (a) shows the causality dependency graph for the model in figure 4.3, and (b) shows the relevant dependency graph. The weights of edges are not explicitly shown in the graph except for the binary relations that are equal to **1**. Based on the dependencies specified for each actor, It is easy to check the relevant dependency in this composition. As an example, the relevant dependency between port p_9 and p_1 is $d(p_1, p_9) = (T \mapsto \{t \in$

$\mathcal{T}|t + d \in T\}), \forall T \in \mathcal{D}(\mathcal{T})$. This means that any events with a tag $t \in T$ at port p_9 can be processed when the signal at port p_1 are known up to $t - d$. Assume the network delay is bounded by Δ, at physical time $\tau = t - d + \Delta$ we are sure that we have seen all events with time stamps smaller than $t - d$ at p_1. Hence, an event e at p_9 with time stamp t can be processed at physical time τ or later. Note that although the *Delay* actor has no real-time properties at all (it simply manipulates model time), its presence loosens the constraints on the execution. By choosing d properly, i.e. $d \geq \Delta$, we can deliver e to p_{15} before physical time reaches t and thus satisfy the actuation constraint at p_{15}. This is precisely the result we were after. It would not be achieved with a Chandy and Misra policy. And unlike optimistic policies, there will never be any need to backtrack.

4.4.2 Execution Based on the Relevant Order

We leverage and improve on distributed DE techniques to relax constraints on execution based on relevant dependency analysis. The key idea is that events only need to be processed in time-stamp order when they are causally related. Based on relevant order of events, the global coordination layer can release received events out of their time stamp order while preserving DE semantics and without requiring backtracking. This out-of-order execution also loosens some constraints for the local resource schedulers.

In chapter 3, I discussed an execution strategy based on the relevant order to enable out of order execution. However, it fails when there are events coming from external hardware, i.e. events from a real-time output port. The pitfall here is that it assumes all the events that have been generated in the system are in E (the set of events in the event queue), but

in a distributed system with network delays, this is not true. The execution strategy needs some adjustment to be used in the coordination layer of the run-time environment.

To take account of external events injected into the system via real-time output ports, I define *real-time minimal* for events.

Definition 4.2 (Real-time Minimal)**.** An event $e = (t, v)$ in E is real-time minimal if it satisfies 1 and 2 below,

1. it is minimal in E
2. we are assured that we have seen all events that are less than it in the relevant order.

If e depends on events from a real-time port up to t' with real-time constraint γ, then at physical time $\tau = t' + \gamma(t')$ we are assured that we have seen all events that are less than it in the relevant order. If e depends on events from multiple real-time port, then we need to wait for the maximum of τ of each real-time port. The idea is simple: if there is some event from external hardware that needs to be processed before e, the coordination layer needs to make sure it has received and released all these events before it releases e. Based on this idea, we can adjust the execution strategy as follows to be used in the coordination layer:

1. Start with E, a set of events in the event queue.
2. Choose $r \subset E$, s.t. each event in r is real-time minimal in E.
3. Process events in r, which may produce a set of new events E'.
4. Update E to $(E \setminus r) \cup E'$.
5. Go to 2.

When clocks in the distributed systems are not perfectly synchronized, we also need to take into account the time synchronization errors in the estimated physical time τ. In particular, if the difference of the clock time between any two nodes in the systems is bounded by ξ, we need to wait until the current physical time is $\tau + \xi$ to make sure e is minimal.

Chapter 5

Scheduling Analysis of PTIDES Models

In chapter 4, I introduced a two-layer architecture for the run time environment of PTIDES programs: a global coordination layer and a resource scheduling layer. Assume an action is associated with each input port to process events received at the port. When events come in, the global coordination layer intercepts the events and decides when the corresponding actions can be enabled and posts them to the scheduling layer. Actions may compete for resources, such as CPU, I/O access or network bandwidth. The scheduling layer is responsible for allocating resources and scheduling actions.

In a PTIDES model, events are ordered according to the relevant order ($<_r$), which in turn introduces precedence constraint on action executions. I discuss how the coordination layer translates relevant order and precedence constraints into deadline constrains on actions. By doing so, the complexity of resource scheduling is significantly reduced. The

scheduling layer can follow an EDF execution strategy, invoking actions based on the event deadlines.

A key requirement for preserving runtime determinism of PTIDES programs is to ensure that events at a real-time port meet their real-time constraints. A PTIDES program is said *feasible* if the real-time constraints can be guaranteed. This chapter studies techniques that statically check feasibility for a given PTIDES program with system characteristics such as communication delays and execution time bounds.

This chapter is organized as follows. Section 5.1 discusses related work on real-time scheduling algorithms. Section 5.2 develops algorithms for assigning deadlines to ensure the precedence constraints on actions. Section 5.3 first studies feasibility analysis for PTIDES models where actions have negligible execution time, and then adapts work on feasibility analysis for sporadic task systems [5] to PTIDES models to derive sufficient conditions when the execution time can not be ignored.

5.1 Real-Time Scheduling

5.1.1 Definition and Terminology

The terminology reviewed here is based on [5]. A real-time task system $\tau = \{T_1, ..., T_n\}$ consists a finite set of tasks. Each task can be executed potentially infinite number of times during a run of a real-time system, and each execution of a task is called a *job*. A job j is characterized by three parameters – a release time r which is the time instance when the job is available for execution, an execution time w and a deadline d by which the job shall

be completed. That is, job j requires w of execution time during $[r, d)$. In a periodic task system, each task gets released at regular periodic intervals. In a sporadic task system, tasks are not release at regular intervals but there is a minimum interval between any two consecutive release times of the task. A periodic task T_i is characterized by 4 parameters – an offset O_i which is the release time of the first job of the task, a period P_i that denotes the interval between the release times of two consecutive jobs of the task, a relative deadline D_i that means a job of the task released at t shall be finished by $t + D_i$, and a worst case execution time W_i that specifies the upper limit on the execution time of each job of the task. We use a 4-tuple (O_i, P_i, D_i, W_i) to denote a periodic task T_i. A periodic task system $\tau = \{T_1, ..., T_n\}$ is called a *synchronous task system* if the offsets of all tasks are equal. Similarly, a sporadic task T_i is characterized by 3 parameters – P_i, D_i and W_i with D_i and W_i meaning the same as for periodic tasks, while P_i means that the interval between the release time of successive jobs is at least P_i. Similar to periodic tasks, a 3-tuple (P_i, D_i, W_i) is used to denote a sporadic task T_i.

5.1.2 EDF for Periodic Independent Tasks

Determining whether an arbitrary periodic task system is feasible with EDF is co-NP-complete in the strong sense [5]. However, the following cases are tractable.

- A periodic real-time task system $\tau = \{T_1, ..., T_n\}$ in which every task T_i has relative deadline equal to its period ($D_i = P_i$) is feasible if and only if

$$\sum_{i=1}^{n} \frac{W_i}{P_i} \leq 1 \tag{5.1}$$

Let $U(\tau) = \sum_{i=1}^{n} \frac{W_i}{P_i}$. U is often called the utilization factor. With periodic tasks that have deadlines equal to their periods, EDF has a utilization bound of 100%. That is, all deadlines are met provided that $U(\tau)$ is no more than 100%.

- For synchronous task systems with U bounded by a constant strictly less than one, there is a pseudo-polynomial time algorithm for the feasibility analysis.

For arbitrary periodic task systems even with a utilization bounded by a constant strictly less than one, exact feasibility analysis is also co-NP-complete in the strong sense. However, it is known that there is a pseudo-polynomial algorithm for the sufficient feasibility check of such systems based on the following theorems and lemma from [5].

Theorem 2. *A real-time periodic task system* $\tau = \{T_1 = (O_1, P_1, D_1, W_1),\ ...,\ T_n = (O_n, P_n, D_n, W_n)\}$ *is feasible if the synchronous periodic task system* $\tau' = \{T_1' = (0, P_1, D_1, W_1),\ ...,\ T_n' = (0, P_n, D_n, W_n)\}$ *is feasible.*

Theorem 3. *A synchronous periodic task system is feasible if and only if for all* $t > 0$, $\eta(0, t) \leq t$.

Where $\eta(t_1, t_2)$ is the *total processing time demand* for tasks over the interval $[t_1, t_2]$. $\eta(t_1, t_2)$ is defined to be the sum of the execution time of jobs that have arrival times at or after time-instant t_1 and deadlines at or before time-instant t_2. For synchronous periodic task system $\tau' = \{T_1' = (0, P_1, D_1, W_1),\ ...,\ T_n' = (0, P_n, D_n, W_n)\}$, $\eta(0, t) = \sum_{i=1}^{n} W_i \cdot max\left\{0, \left\lfloor \frac{t - D_i}{P_i} \right\rfloor + 1\right\}$

Despite the simplicity of theorem 3, feasibility analysis for general synchronous periodic task system is as intractable as for arbitrary task systems. However, when the utilization

$U(\tau) = \frac{W_1}{P_1} + ... + \frac{W_n}{P_n}$ of a synchronous periodic task system is bounded by a constant $0 < c < 1$, the following lemma indicates that feasibility analysis is pseudo-polynomial for such systems.

Lemma 3. *Let c be a constant and $0 < c < 1$. If a synchronous periodic task system τ with $U(\tau) \leq c$ is not feasible, then there exists a t_0 such that $0 < t_0 < \frac{c}{1-c} max(P_i - D_i)$ and $\eta(0, t_0) > t_0$.*

Lemma 3 suggests a pseudo-polynomial time feasibility analysis algorithm for a synchronous periodic task system τ with bounded utilization. We can generate the EDF schedule for τ until $\frac{c}{1-c} max(P_i - D_i)$ and declare that τ is feasible if and only if no deadlines are missed in this schedule [5].

5.1.3 EDF for Sporadic Independent Tasks

For periodic task systems the set of jobs to be scheduled is known a priori during feasibility analysis. For sporadic task systems, however, there is no a priori knowledge about which set of jobs will be generated by the task system during run-time. In analyzing sporadic task systems, therefore, every conceivable sequence of possible requests must be considered.

Given a sporadic task system $\tau = \{T_1 = (P_1, D_1, W_1), ..., T_n = (P_n, D_n, W_n)\}$, let (i, t) denote the job of task T_i released at t. A legal sequence of jobs R_τ is a (possibly infinite) list of jobs such that if (i, t_1) and (i, t_2) both belong to R_τ, then $|t_2 - t_1| \geq P_i$.

Given a sporadic task system, there can be infinite number of legal job sequences. Fortunately, it turns out that, at least in the context of dynamic-priority preemptive uniprocessor

scheduling, it is relatively easy to identify a unique worst-case legal sequence of jobs, such that all legal sequences of jobs can be scheduled to meet all deadlines if and only if this worst-case legal sequence can. This particular worst-case legal sequence of jobs is exactly the unique legal sequence of jobs generated by the synchronous periodic task system with exactly the same parameters as the sporadic task system [5]:

Lemma 4. *A sporadic task system* $\tau = \{T_1 = (P_1, D_1, W_1),\ ...,\ T_n = (P_n, D_n, W_n)\}$ *is feasible if and only if the synchronous periodic task system* $\tau' = \{T_1' = (0, P_1, D_1, W_1),\ ...,\ T_n' = (0, P_n, D_n, W_n)\}$ *is feasible.*

The idea of proving lemma 4 is as follows [5]:

- τ is infeasible if and only if there is some legal sequence of jobs R_τ and some interval $[t, t + t_0]$ such that $\eta(t, t + t_0) > t_0$, where $\eta(t, t + t_0)$ is the total processing time demand for R_τ.

- The total processing time demand by jobs generated by τ over an interval of length t_0 is maximized if each task in τ generates a job at the start of the interval, and then generates successive jobs exactly P_i time apart. This is exactly the sequence of jobs that generated by the synchronous periodic task system τ' defined in lemma 4.

Based on lemma 4, to check whether a sporadic task system τ is feasible, we only need to determine whether the corresponding synchronous task system is feasible. Hence, if the utilization $U(\tau)$ of τ is bounded by a constant $0 < c < 1$, the feasibility analysis can be performed in pseudo-polynomial time.

5.2 Precedence Constrains in PTIDES Models

In a PTIDES model, events are ordered according to the relevant order ($<_r$). Assume an action is associated with each input port to process events received at the port. Similar to tasks considered by real-time scheduling algorithms, actions are finite amount of computations that require some resources and take some time to execute. Let P_I be the set of input ports of a PTIDES model and a_i be the action for the input port $p_i \in P_I$. During an execution of the system, p_i may receive a sequence of events. Each event triggers a_i being executed once. Each execution of an action is called a *job*. Let a_i^n denote the job that corresponds to the nth execution of action a_i, where $n \in \mathbb{N}$. Let s_i be the signal at port p_i and $s_i(n)$ be the nth event of s_i, where $n \in \mathbb{N}$. Given two events $s_i(k)$ and $s_j(h)$ with $s_i(k) <_r s_j(h)$, job a_i^k shall be executed before a_j^h.

Assume that the scheduling layer has a job queue where the coordination layer posts actions. Following the terminology discussed in section 5.1, a job a_i^n is characterized by a release time r_i^n, an execution time w_i^n and a deadline d_i^n. The release time r_i^n is the time when the job is posted to the job queue. The execution time w_i^n denotes how long the job takes to execute. The deadline d_i^n is the time when the job shall be completed. Further, I introduce two other quantities: the start time l_i^n is when the job is started to execute, and the finish time f_i^n is when the job ends its execution. These quantities are related as $r_i^n \leq l_i^n \leq f_i^n$.

Precedence constraints can be guaranteed by priority assignment or deadline assignment. In fixed-priority scheduling, job a_i^k will always precede job a_j^h if action a_i has a higher priority

than a_j and $r_i^k \leq r_j^h$. Similarly, with EDF a_i^k will always precede action a_j^h if a_i^k has an earlier deadline and $r_i^k \leq r_j^h$ [38]. For a given scheduling policy, such as fixed-priority or EDF, the problem of preserving precedence constraints then becomes finding an assignment to its parameters, priorities for fixed-priority and deadlines for EDF, that ensures consistency with the precedence constraints. In this thesis, I focus on EDF scheduling and develop an algorithm that assign deadlines for jobs.

Let $a_i^k \prec a_j^h$ denote the precedence constraint between a_i^k and a_j^h. Spuri and Stankovic [47] give a general theorem for EDF-like schedulers to satisfy precedence constraints. The following definition and theorem is from [47] with adaption to the notations used here.

Definition 5.1. Given two jobs $a_i^k \prec a_j^h$, we say the release time and deadline are *consistent* with the precedence constraint if

$$a_i^k \prec a_j^h \Rightarrow r_i^k < r_j^h \text{ and } d_i^k \leq d_j^h$$

where d_i^k and d_j^h represent the deadlines of a_i^k and a_j^h.

Definition 5.2. Given any schedule of a job set, we say it is *quasi-nomal* if

$$r_i^k < r_j^h \text{ and } d_i^k \leq d_j^h \Rightarrow f_i^k < l_j^h.$$

That is, in a quasi-normal schedule, if a_i^k is released before a_j^h and a_i^k has an earlier deadline than a_j^h, then a_j^h can not preempt a_i^k and can only start execution after a_i^k is finished. It is easy to see that schedules based on the EDF scheduling algorithm are quasi-normal.

Theorem 4. *Given a set of jobs with release time and deadlines consistent with their precedence constraints, any feasible schedule (i.e. that satisfy both the release times and deadlines) obeys the precedence constraints if it is quasi-normal.*

The above theorem was stated as a necessary and sufficient condition in [47]. However, Mangeruca et al. [38] shows that it is only a sufficient condition.

What we learn from theorem 4 is that if the coordination layer post jobs to the scheduling layer with release time and deadlines consistent with precedence constraints on the jobs, then any EDF scheduler can be applied in the scheduling layer and the precedence constraints will be guaranteed.

5.2.1 Assigning Deadlines

Recall that in a PTIDES program, events at a real-time input port need to satisfy the real-time constraint associated with that port. If an input port p_i is a real-time port with constrain γ_i, then an event $s_i(k) = (t_i, v)$ at this port must be processed before $\gamma_i(t_i)$. We can view $\gamma(t_i)$ as the deadline for job a_i^k. Next, I consider how to compute the deadlines for jobs triggered by events at non-real-time ports.

Given an event $s_j(h) = (t_j, v)$ at input port p_j, if there exists a real-time input port p_i having relevant dependency on p_j, i.e. $d(p_j, p_i) \neq \mathbf{0}$, then assign deadline $d_j^h = \gamma_i(t)$ for job a_j^h, where $D(t) = d(p_j, p_i)(D(t_j))$. Intuitively, if $s_j(h)$ affects an event at a real-time port that needs to be processed before $\gamma_i(t)$, a_j^h certainly must be processed before $\gamma_i(t)$. In general, p_j may have relevant dependency with multiple real-time input ports, and a_j^h shall be processed before the smallest deadline of the real-time events at these ports. Formally, let I_j denote the set of real-time input ports that have relevant dependency with p_j, then

$$d_j^h = min\{\gamma_m(t_m) | \forall p_m \in I_j, D(t_m) = d(p_j, p_m)(D(t_j))\}. \tag{5.2}$$

If p_j does not have causal dependency with any real-time input port, we assign infinity as the deadline of job a_j^h. This means we do not care when the execution of a_j^h completed.

Lemma 5. *Deadlines assigned by 5.2 are consistent with the precedence constraint, i.e.* $\forall a_i^k \prec a_j^h$, *equation 5.2 assigns* $d_i^k \leq d_j^h$.

Proof. $\forall a_i^k \prec a_j^h$, we have $s_i(k) <_r s_j(h)$ and $d(p_i, p_j)(D(t_i)) \subset D(t_j)$, where $s_i(k) = (t_i, v)$ and $s_j(h) = (t_j, v')$. Let I_j denote the set of real-time input ports that causally depend on p_j. Then $\forall p_m \in I_j$, p_m also depends on p_i, and $d(p_i, p_m) \leq d(p_i, p_j) \otimes d(p_j, p_m)$, i.e. $d(p_i, p_m)(D(t_i)) \subseteq d(p_j, p_m)(d(p_i, p_j)(D(t_i))) \subseteq d(p_j, p_m)(D(t_j))$. Hence $d_i^k \leq d_j^h$. □

Given two jobs $a_i^k \prec a_j^h$, the release times given by the coordination layer satisfy $r_i^k < r_j^h$. The release times and deadlines given by the coordination layer are consistent with the precedent constraints. Thus any feasible EDF schedule (i.e. that satisfies both the release times and deadlines) obeys the precedence constraints. I study when there exists a feasible EDF schedule for a PTIDES program in the next section.

5.3 Feasibility Analysis

A key requirement for preserving runtime determinism of PTIDES programs is that event $e = (t, v)$ at a real-time port p_i must be processed before $\gamma_i(t)$. We call a PTIDES program *feasible* if this requirement can be guaranteed. We are interested in statically checking feasibility for a given PTIDES program and some system characteristics such as communication delay and execution time bounds. In this section, I first study feasibility analysis assuming that the execution time of each action is negligible. Then I describe solution for more general cases which assumes non zero execution times for actions.

5.3.1 Feasibility Analysis with Zero Execution Time

When the execution time is negligible, feasibility check becomes straightforward: the system is feasible if all the events at the real-time input ports of the system can be released by their model time. Whether an event at a real-time input port can be released depends on whether this event has relevant dependency with events produced by real-time output ports, since events from other output ports can be computed in zero time if they do not need real-time information. Note that I am using the "causal dependency" more liberally here in the sense that I am referring to causal dependency from an output port to an input port, which strictly speaking is not defined. However, It is easy to extend the causal dependency in a composition to be defined between output and input ports and there is no fundamental change in the computation of these dependencies.

Let O denote the set of real-time output ports in a PTIDES model. Recall that for a real-time output port $p_i \in O$ with constrain γ_i, an event e with model time stamp t will be generated at this port no later than physical time $\gamma_i(t)$. Given a real-time input port $p_j \in P_I$, where P_I is the set of input ports in the model. if there is a port $p_i \in O$ such that the relevant dependency $d(p_i, p_j) \neq \mathbf{0}$, then there are order constraints on the events at p_i and p_j. That is, an event e_i received at p_i with model time t_i is less than (in the relevant order, $<_r$) an event e_j with model time t_j at p_j , where $d(p_i, p_j)(D(t_i)) = D(t_j)$. Hence the event e_i cannot be processed until we are sure that all events at p_i with time stamp less than t_i have been received. In the worst case, we need to wait until the physical time reaches $\gamma_i(t_i)$ to make sure all events at p_i with model time less than t_i have been

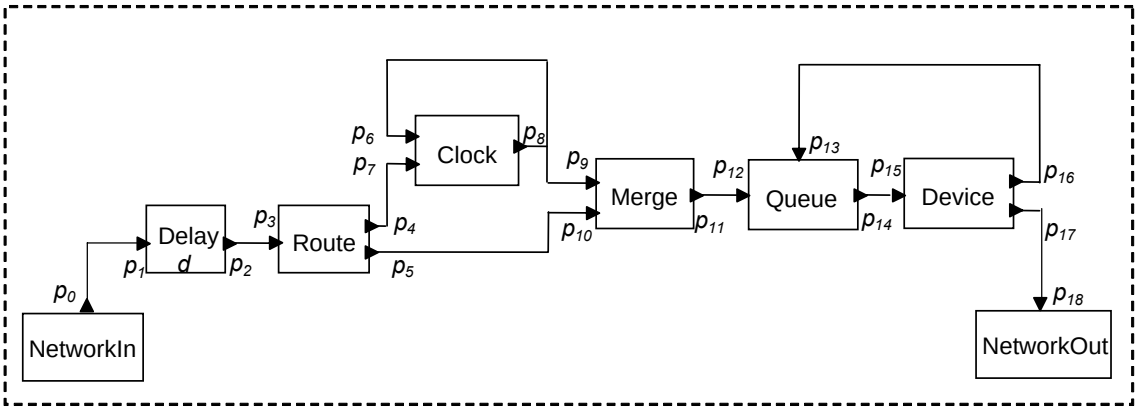


Figure 5.1. PTIDES Model for each camera in the motivating example of chapter 4

.

received. If $\gamma_i(t_i) \leq \gamma_j(t)$, we can guarantee that the event e_j at port p_j is processed before the deadline, and we call port p_j *feasible*. In general, a real time input port p_j can have relevant dependency with multiple output ports in O, then p_j is feasible if $\gamma_i(t_i) \leq \gamma_j(t)$ for all $p_i \in O$ that has relevant dependency with p_j. We call a system *feasible* if all its real-time input ports are feasible.

As an example, consider again the system shown in figure 5.1. The set of real-time output ports is $O = \{p_0, p_{16}, p_{17}\}$, and the set of real-time input ports is $I = \{p_{15}, p_{18}\}$. The causality interface for each actor and the relevant dependencies are the same as given in chapter 4. We can perform the following analysis.

The real-time input port p_{15} have relevant dependency with ports p_0 and p_{16} in I. The relevant dependencies are:

$$\begin{aligned} &d(p_0, p_{16}) = (T \mapsto \{t \in \mathcal{T} | t - d \in T \text{ or } t - d \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T}) \\ &d(p_{16}, p_{15}) = \mathbf{1} \end{aligned} \tag{5.3}$$

That is an event e with model time stamp t at p_{15} depends on events at p_0 up to time $t' = t - d$ and events at p_{16} up to time $t'' = t$. The timing constraints on port p_0 and p_{16} are $\gamma_0(t') = t' + \Delta = t - d + \Delta$ and $\gamma_{16}(t'') = t'' = t$. If $d \geq \Delta$, we have $\gamma_0(t') \leq \gamma(t)$ and $\gamma_{16}(t'') \leq \gamma(t)$ and thus port p_{15} is feasible.

Similarly, one can check the feasibility condition for real-time input port p_{18}, which turns out also requiring $d \geq \Delta$. Recall that Δ represent the upper bound of the network delay. By choosing the delay parameter properly, i.e. $d \geq \Delta$, we can make sure the deadlines on all the real-time input ports in the system are met. Hence the system is feasible if $d \geq \Delta$.

The analysis discussed above is the beginning of what allows us to statically check whether a PTIDES specification is feasible over a network of nodes. A full analysis, when the execution time is not negligible, requires integrating relevant dependency analysis with real-time scheduling and worse case execution time analysis on individual nodes. I study this next.

5.3.2 Feasibility Analysis with Worst Case Execution Time

When execution time for actions are not negligible, general event triggered real-time systems are not amenable to compile-time feasibility analysis [34]. However, when the discrete activities can come in predictable patterns, real-time scheduling theories are applicable to many PTIDES models. Assuming that the triggering signals of a PTIDES model are sporadic, i.e. there is a minimum interval between any two consecutive events of the same signal, the question I try to answer in this section is which kind of PTIDES models is amenable to feasibility analysis and how to apply real-time scheduling theories to these

models. Here the triggering signals of a PTIDES model are signals at its *triggering* ports, where actors in a PTIDES model can declare a output port to be a *triggering* port if events produced on this port is not from any action of processing input events. For example, the triggering signals of a PTIDES model can include output signals of all source actors. Recall that events from sensors or input devices are injected to a PTIDES model via actors with real-time output ports and they are part of the triggering signals of a PTIDES model. Signals as the outputs of source actors without real-time constraint are also part of the triggering signals of a PTIDES model.

Formally, we say a signal s_i is sporadic if $\forall e_1 = (t_1, v_1) \in s_i, e_2 = (t_2, v_2) \in s_i$, $|t_2 - t_1| > P_i$, where $P_i > 0$ is the minimal interval of s_i.

Definition 5.3 (Sporadic Actor)**.** A *source* actor is sporadic if its output signals are sporadic; a *sink* actor is always sporadic; a *transformer* actor, which is actor with both input and output ports, is sporadic if it maps sporadic signals to sporadic signals.

That is, if the input signals of a sporadic actor are sporadic, then the output signals of it are also sporadic. The nice property of sporadic actors is that it preserves the "sporadicness" of signals. Thus if the triggering signals to a PTIDES model with only sporadic actors are sporadic, then all the other signals are also sporadic.

Transformer actors with only one input port is sporadic if the delay from the input port to any of the output ports is constant. For example, the *Delay* actor discussed in chapter 3 with a constant delay parameter d is sporadic. Actors with variable delay from its input port to output port are not sporadic. For example, the *VariableDelay* actor that shifts every event in its input signal by a random variable x into the future is not sporadic since

there is no minimal interval strictly greater than 0 for its output signal even if its input signal is sporadic.

For transformer actors with more than one input ports and constant delay from input to output, whether they are sporadic or not is challenging to decide. Let's look at two actors first. The *Merge* actor is not sporadic. As an example, consider the model shown in figure 5.2. Signal s_1 and s_2 represent the triggering signals of the model and assume they are defined as following:

$$s_1 = (\mathbb{R}_0, \{(k + \frac{1}{k+1}, 1) \,|\, k \in \mathbb{N}\}),$$
$$s_2 = (\mathbb{R}_0, \{(k + \frac{1}{k+2}, 1) \,|\, k \in \mathbb{N}\})$$

Signals s_1 and s_2 are both sporadic as $P_1 = 1/2$ and $P_2 = 5/6$. Signal s_9 is the merge of s_1 and s_2. It is easy to see that s_9 is not sporadic. On the other hand, the *Queue* actor discussed in chapter 4 is sporadic as whether an output event can be produced is only determined by the signal at it triggering port. So if the signal at the triggering port is sporadic, then the output signal of the *Queue* actor is sporadic. In general, for an actor with output port p depends on more than one input port, it is sporadic if only one input port can determine when the output get produced and I call this input port the *control port* of the output port p.

Feasibility Analysis for Sporadic PTIDES Models

This section studies feasibility analysis for PTIDES Models that contain only sporadic actors. For actors with more than one input port, I assume there is only one control input

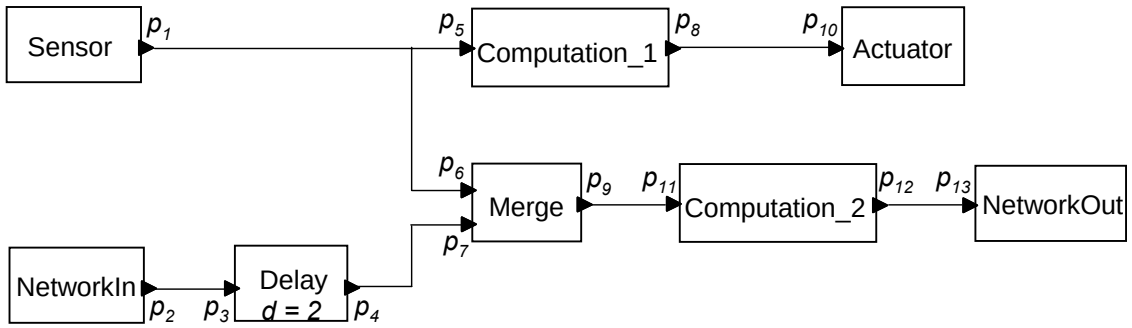


Figure 5.2. An example of sporadic triggering signals result to no sporadic signals

.

port for each output port. Let $a_i = (P_i, D_i, W_i)$ represent the action at input port p_i, where P_i, D_i and W_i have the same meaning as discussed in section 5.1 for tasks. Given that the triggering signals are sporadic and all the actors in a PTIDES model are sporadic, we have all the signals sporadic and hence the actions in a PTIDES model are sporadic. Thus we can use theorem 2 to check the feasibility of the model. Assuming that the worst case execution time for each action is given, this section focuses on how to compute the minimal interval and relative deadline for each action in order to apply theorem 2.

Let O_n be the set of trigger ports. $\forall p_i \in O_n$, let I_i be the set of input ports that have causal dependency with p_i.

$\forall p_j \in I_i$, depends on whether p_j is a control port or not, the minimal interval of the down-stream actions may or may not depends on p_i. To capture this information, we can separate actions into groups, where an action group A_i is a set of actions that have minimal interval depends on $p_i \in O_n$. To compute the action groups, we can transform the *dependency graph* of a PTIDES model by removing the edge between an input port and an

output port of the same actor if the input port is not a control port of the output port. Then $\forall p_j \in I_i$, $a_j \in A_i$ if there is a path from p_i to p_j in the transformed graph.

Computing the minimal interval P_j for action a_j is straightforward. $\forall p_i \in O_n$, all actions in A_i inherit the minimal interval of the signal at port p_i. Doing this for each $p_i \in O_n$, we can compute the minimal interval of all the actions. Note that since we assume that there is only one control input port for each output port, an action can not belong to more than one action groups.

To compute the relative deadline D_j action a_j, we can compute the deadline d_j and release time r_j of the action first and then get D_j by $D_j = r_j - d_j$. First of all, $\forall p_i \in O_n$, if p_i is a real-time output port, then there is a real-time constraint γ_i on when an event $e = (t, v)$ be generated, i.e. e will be generated no later than $\gamma_i(t)$; otherwise we can assume events at p_i are generated at physical time no later than their model time (in theory, these events can be computed when the system just starts since they do not depends on real-time information, and it is up to the implementation of the run-time environment to decide when to generate these events as long as they are generated before their model time).

$\forall a_j \in A_i$, if p_j only has relevant dependency with p_i and no relevant dependency with any other port in O_n, then we can release the action a_j at $\Gamma_i(t)$, where

$$\Gamma_i(t) = \begin{cases} \gamma_i(t) & \text{if } p_i \text{ is a real-time triggering port,} \\ t & \text{otherwise.} \end{cases}$$

If p_j also has relevant dependencies with some other port $p_m \in O_n$, then there are order constraints on the events at p_m and p_j. That is, an event received at p_m with model

time t_m is less than (in the relevant order, $<_r$) an event e' with model time stamp t' at p_j, where $D(t') = \delta(p_i, p_j)D(t)$ and $d(p_m, p_j)(D(t_m)) \subset D(t')$. Hence the action a_j to process e' cannot be released until we are sure that all events at p_m with time stamp less than t_m have been generated. For the worst case, we need to wait until the physical time reaches $max(\Gamma_i(t), \Gamma_m(t_m))$, where $\Gamma_m(t_m)$ is defined in the same way as $\Gamma_i(t)$. In general, p_j can have relevant dependencies with several ports in O_n. We need to compute the generation times for all the events on these ports that are less than e and take the latest time as the release time of action a_j.

Following the idea from section 5.2.1, $\forall a_j \in A_i$, if there exists a real-time input port p_k having relevant dependency on p_j, then assign deadline $d_j = \gamma_k(t')$ for action a_i, where $D(t') = d(p_i, p_k)D(t)$. If p_j has relevant dependency with multiple real-time input ports, then assign the smallest deadline to a_j as:

$$d_i = min\{\gamma_k(t_k) | \forall p_k \in R_j, D(t_k) = d(p_i, p_k)D(t)\}. \tag{5.4}$$

where R_j is the set of real-time input port that depends on p_j.

For each port $p_i \in O_n$, after we determine the release time and the absolute deadline of each action, we can compute the relative deadline of each action by subtracting the release time from the absolute deadline. It is easy to show that when the real-time constraints in the model are all linear functions, the actions are sporadic, i.e. there are minimal intervals between any two release time of the same action. With the minimal interval and relative deadline computed for each action, we can check whether the given PTIDES model is feasible or not based on theorem 2.

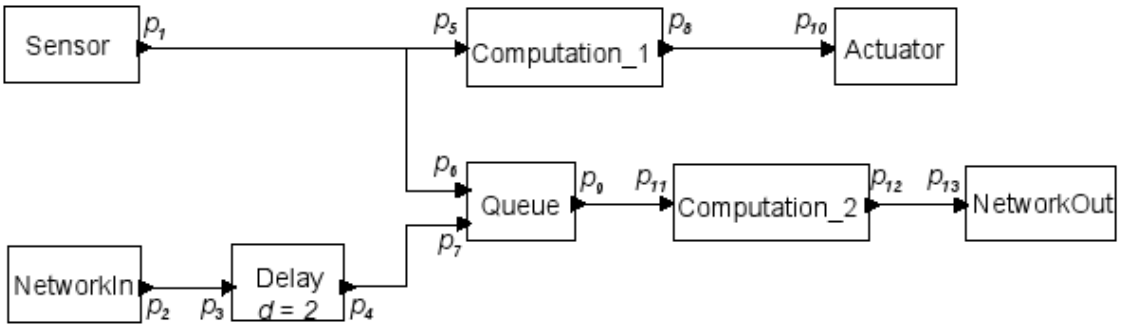


Figure 5.3. A PTIDES model contains only sporadic actors.

.

One thing that needs to be pointed out is that the release time used above is not necessary to be consistent with the precedence constraint on jobs. That is two jobs $a_1(k) \prec a_2(h)$ can have release time $r_1(k) \leq r_2(h)$. This is not a problem when we considering the feasibility problem since if the system is feasible, this means that there is a schedule for every sequence of possible job orders which includes the one that satisfies the precedence constraints between jobs. In a real execution of the system, if $a_1(k) \prec a_2(h)$ and two actions have the same release time, the coordination layer guarantees the precedence constraints between actions by enqueue a_1 to the queue in the scheduling layer before a_2.

Now let's look at an example to apply the algorithm above for computing the minimal intervals and deadlines. Consider the model shown in figure 5.3, which is quite similar to the model shown in 5.2 except that the *Merge* actor is replaced with the *Queue* actor to

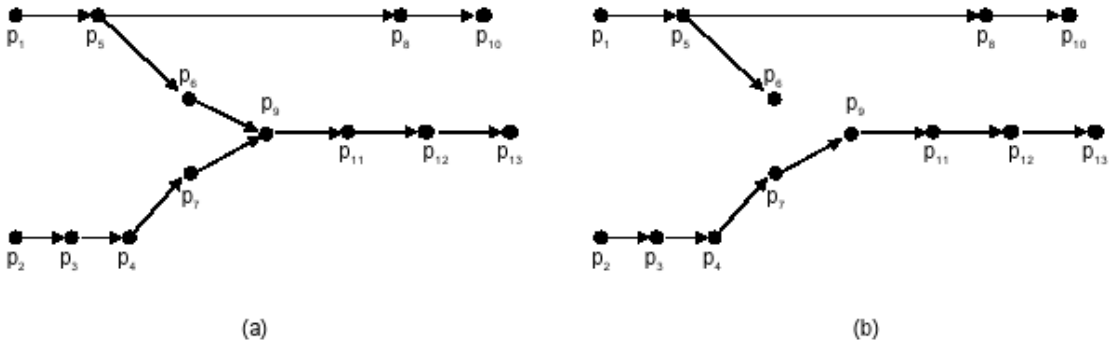


Figure 5.4. The dependency graph for the model in figure 5.3.

.

make the model sporadic. The causality interface for each actor in the model is:

$$\begin{aligned}
&\delta_{Delay}(p_3, p_4) = (T \mapsto \{t \in \mathcal{T} | t - d \in T \text{ or } t - d \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T}),\\
&\delta_{Computation_1}(p_5, p_8) = \mathbf{1},\\
&\delta_{Queue}(p_6, p_9) = \mathbf{1}, \quad \delta_{Queue}(p_7, p_9) = \mathbf{1},\\
&\delta_{Computation_2}(p_{11}, p_{12}) = \mathbf{1},
\end{aligned} \tag{5.5}$$

Output ports p_1 and p_2 have real-time constraints γ_1 and γ_2, and input ports p_{10} and p_{13} have real-time constraints γ_{10} and γ_{13}. The set of triggering ports is $O_n = \{p_1, p_2\}$. And assume p_7 is the control input port for the *Queue* actor. Figure 5.4 (a) shows the *dependency graph* of the model, and (b) shows the transformed graph by removing the edge between not controlling input port and the corresponding output port.

We have $A_1 = \{a_5, a_6, a_{10}\}$ and $A_2 = \{a_3, a_7, a_{11}, a_{13}\}$. All actions in A_1 have the same minimal interval as s_1, and all actions in A_2 have the same minimal interval as s_2.

Given an event $e = (t_1, v_1)$ at p_1, action a_5 and a_{10} can be released at $\gamma_1(t_1)$ since p_5

and p_{10} only have relevant dependency with p_1. That is, $r_5 = \gamma_1(t_1)$ and $r_{10} = \gamma_1(t_1)$. Action a_6 need to wait till $max(\gamma_1(t_1), \gamma_2(t'))$ since p_6 has relevant dependency with p_2, where $t' = t_1 - d$. So $r_6 = max(\gamma_1(t_1), \gamma_2(t_1 - d))$. Now we calculate the deadlines for each actions in A_1. Port p_5 has relevant dependency with both real-time input ports p_{10} and p_{13}, so $d_5 = min(\gamma_{10}(t_1), \gamma_{13}(t_1))$. Port p_6 only has relevant dependency with real-time input ports p_{13}, so $d_6 = \gamma_{13}(t_1)$. Port p_{10} only has relevant dependency with real-time input ports p_{10}, so $d_{10} = \gamma_{10}(t_1)$. After get the release time and deadline of each action in A_1, it is straightforward to compute the relative deadline of each action, and we have:

$$\begin{aligned} D_5 &= min(\gamma_{10}(t_1), \gamma_{13}(t_1)) - \gamma_1(t_1), \\ D_6 &= \gamma_{13}(t_1) - max(\gamma_1(t_1), \gamma_2(t_1 - d)), \\ D_{10} &= \gamma_{10}(t_1) - \gamma_1(t_1), \end{aligned} \tag{5.6}$$

Similarly, we can calculate the deadline for each action in A_2. Given an event $e' = (t_2, v_2)$ at p_2, $r_3 = \gamma_2(t_2)$, $r_7 = r_{11} = r_{13} = max(\gamma_2(t_2), \gamma_2(t_2 + d))$, $d_3 = d_7 = min(\gamma_{10}(t_2 + d), \gamma_{13}(t_2 + d))$, and $d_{11} = d_{13} = \gamma_{13}(t_2 + d)$. So the deadlines for actions in in A_2 are:

$$\begin{aligned} D_3 &= min(\gamma_{10}(t_2 + d), \gamma_{13}(t_2 + d)) - \gamma_2(t_2), \\ D_7 &= min(\gamma_{10}(t_2 + d), \gamma_{13}(t_2 + d)) - max(\gamma_2(t_2), \gamma_2(t_2 + d)), \\ D_{11} &= \gamma_{13}(t_2 + d) - max(\gamma_2(t_2), \gamma_2(t_2 + d)), \\ D_{13} &= \gamma_{13}(t_2 + d) - max(\gamma_2(t_2), \gamma_2(t_2 + d)), \end{aligned} \tag{5.7}$$

Assuming the worst execution time W_i is given for the model. Now we have all the info to check whether the given PTIDES model is feasible or not based on theorem 2.

Feasibility Analysis for PTIDES Models with Non-sporadic Actors

This section extends feasibility analysis to more general PTIDES models where only actors in a feedback loop are required to be sporadic and other actors in the model can be non-sporadic DE actors that have constant delay from its input port to output port.

The challenge to deal with non-sporadic actors is that they can map sporadic signals to non-sporadic signals and making theorem 2 not applicable. One example of such actor is the *Merge* actor shown in figure 5.2, where even both the input signals, s_1 and s_2 are sporadic, the output signal s_9 is not sporadic, which in turn caused the action at port p_{11} that connect to p_9 not sporadic. To address this problem, we can split action a_{11} to two actions, one corresponding to events causally depend on events of s_1 and another one corresponding to events causally depend on events of s_2. Then both actions are sporadic. In general, if an input port p_i of a PTIDES model has causal dependency with m ports in O_n, we need to split action a_i to m actions, with $a_i \restriction p_j$ representing action of a_i that is triggered by events at port $p_j \in O_n$.

To compute the action group A_i for each $p_i \in O_n$, we do the same transformation on the *dependency graph* as discussed in the previous section, i,e., removing the edge between an input port and an output port if the input port is not a control port. There may be multiple control input ports now for an output if the actor containing these ports are not sporadic. For example, both input ports of the *Merge* actor are control ports of its output port. Recall that I_i is the set of input ports that have causal dependency with p_i. Then

$\forall p_j \in I_i$, add $a_j \upharpoonright p_i$ to A_i if there is a path from p_i to p_j in the transformed graph, and All actions in A_i inherit the minimal interval of the signal at p_i.

The algorithm for computing the release time and deadline of each action in an action group is the same as discussed in the previous section as it does not depends on whether an actor is sporadic or not. Now we have all the information to check whether the given PTIDES model is feasible or not based on theorem 2.

As an example, let's compute the minimal intervals and relative deadlines for actions in the PTIDES model shown in 5.1, which is the camera model in the motivating example of chapter 4. As we discussed in last section, the *Queue* actor in one of the feedback loops is sporadic and assume port p_13 is its control port in this example. The *Clock* actor in the other feedback loop is also sporadic with p_6 being its control port. All other actors have constant delay from input port to output port. The causality interface for each actor in the model has been given in chapter 4, and is listed here again for convenience:

$$
\begin{aligned}
&\delta_{Delay}(p_1, p_2) = (T \mapsto \{t \in \mathcal{T} | t - d \in T \text{ or } t - d \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T}),\\
&\delta_{Router}(p_3, p_4) = \mathbf{1}, \quad \delta_{Router}(p_3, p_5) = \mathbf{1},\\
&\delta_{Clock}(p_6, p_8) = (T \mapsto \{t \in \mathcal{T} | t - P_{min} \in T \text{ or } t - P_{min} \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T}),\\
&\delta_{Clock}(p_7, p_8) = \mathbf{1},\\
&\delta_{Merge}(p_9, p_{11}) = \mathbf{1}, \quad \delta_{Merge}(p_{10}, p_{11}) = \mathbf{1},\\
&\delta_{Queue}(p_{12}, p_{14}) = \mathbf{1}, \quad \delta_{Queue}(p_{13}, p_{14}) = \mathbf{1},\\
&\delta_{Device}(p_{15}, p_{16}) = (T \mapsto \{t \in \mathcal{T} | t - \mu \in T \text{ or } t - \mu \notin \mathcal{T}\}), \forall T \in \mathcal{D}(\mathcal{T}),\\
&\delta_{Device}(p_{15}, p_{17}) = \mathbf{1}
\end{aligned}
\tag{5.8}
$$

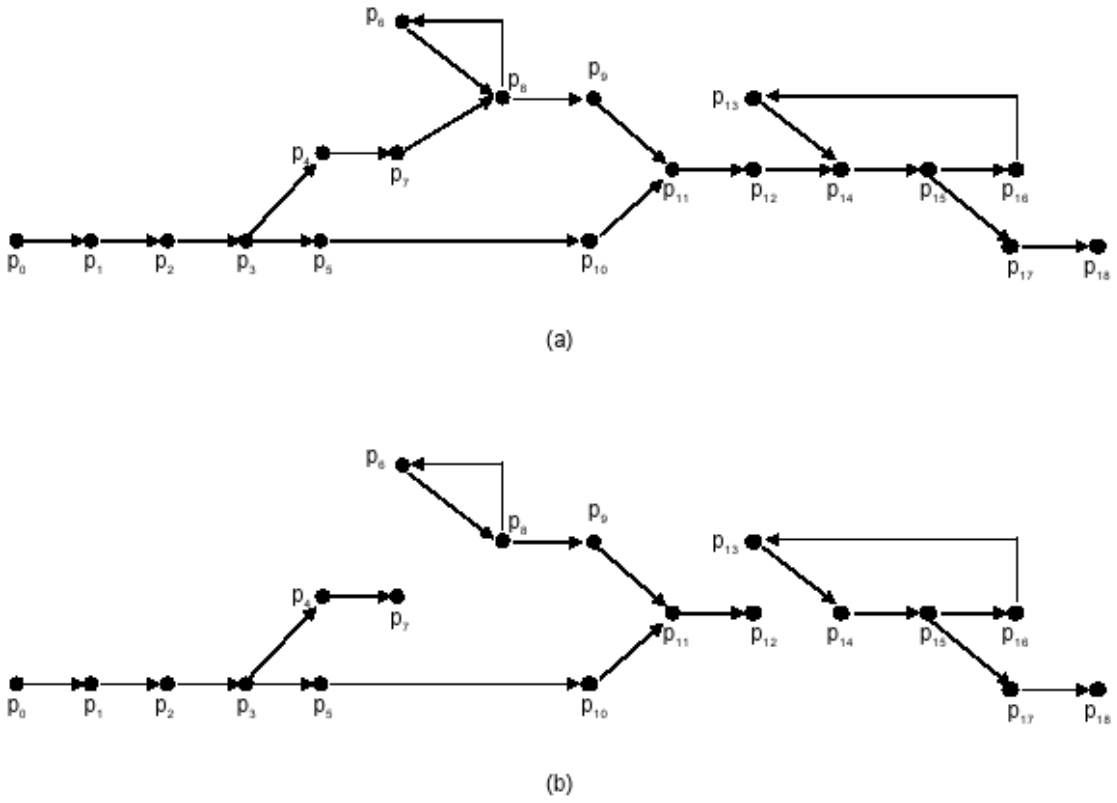


Figure 5.5. The dependency graph for the model in figure 5.1.

.

Output ports p_0 and p_{16} have real-time constraints γ_0 and γ_{16}, and input ports p_{15} and p_{18} have real-time constraints γ_{15} and γ_{18}. The set of triggering ports is $O_n = \{p_0, p_8, p_16\}$. Figure 5.5 (a) shows the *dependency graph* of the model, and (b) shows the transformed graph by removing the edge not controlling input port and the corresponding output port.

We have $A_0 = \{a_1, a_3, a_7, a_{10}, a_{12} \upharpoonright p_0\}$, $A_8 = \{a_6, a_9, a_{12} \upharpoonright p_8\}$ and $A_{16} = \{a_{13}, a_{15}, a_{18}\}$. All actions in A_i have the same minimal interval as s_i, where $i = 0, 8, 16$.

Given an event $e = (t_0, v_0)$ at p_0, action a_1 and a_3 can be released at $\gamma_0(t_0)$. That is, $r_1 = r_3 = \gamma_0(t_0)$. Action a_7 and a_{10} need to wait till $max(\gamma_0(t_0), \Gamma_8(t'))$ since p_7 and

p_{10} have relevant dependency with p_8, where $t' = t_0 - d$ and $\Gamma_8(t') = t'$. So $r_7 = r_{10} = max(\gamma_0(t_0), t_0 - d)$. Now we calculate the deadlines for each actions in A_0. Port p_1, p_3, p_7 and p_{10} all have relevant dependency with both real-time input ports p_{15} and p_{18}, so $d_1 = d_3 = d_7 = d_{10} = min(\gamma_{15}(t_0 + d), \gamma_{18}(t_0 + d))$. So the relative deadline of actions in A_0 are:

$$\begin{aligned} D_1 = D_3 &= min(\gamma_{15}(t_0 + d), \gamma_{18}(t_0 + d)) - \gamma_0(t_0), \\ D_7 = D_{10} &= min(\gamma_{15}(t_0 + d), \gamma_{18}(t_0 + d)) - max(\gamma_0(t_0), t_1 - d), \end{aligned} \tag{5.9}$$

Similarly, one can calculate the deadlines for actions in A_8 and A_{16}. Once we have the minimal intervals and deadlines for all the actions in this model, we can apply theorem 2 to verify whether it is feasible or not.

Chapter 6

Conclusion and Future Work

6.1 Summary of Results

This dissertation addresses several challenging issues of using DE models as a basis for model-based design of distributed real-time systems. DE models have primarily been used in performance modeling and simulation, where time stamps are a modeling property bearing no relationship to real time during execution of the model. When DE is used for specifying real-time systems, a key question we need to answer is how to relate model time to real time. Mapping model time to real time everywhere in a DE model is not efficient and not necessary. I present a programming model called PTIDES that embraces DE semantics but relates model time to real time only at sensor and actuator interactions based on the observation that in many real-time systems time assurance matters only when they react to or act on the physical world.

Besides the programming model, this dissertation studies how to build a run-time en-

vironment to execute PTIDES models. Conventional DE simulation strategy orders events in a DE model totally based on their model times and computes the behavior of the model chronologically. In order to process an event e with time stamp t, the simulator needs to know all events before e and make sure these events have been processed. Although simple, this approach is very restrictive in processing events. I develop a new execution strategy that preserves DE semantics without requiring to process all events in their time stamp order. Based on causality analysis of DE models, I define *relevant dependency* and *relevant orders* to enable out-of-order execution without compromising determinism and without requiring backtracking. A two-layer architecture that includes a global coordination layer and a resource scheduling layer for the run time environment of PTIDES programs is discussed. The new execution strategy is used in the coordination layer to release events to the scheduling layer.

Compile-time schedulability analysis is often important to many real-time systems. Although general event triggered real-time systems are not amenable to compile-time feasibility analysis, this dissertation shows that when the discrete activities can come in predictable patterns, real-time scheduling theories are applicable to many PTIDES models. I studies a sufficient condition for a PTIDES model to be feasible when the inputs of the model are sporadic and the actors in any feedback loop are sporadic actors. The result developed is general enough to apply to a wide range of PTIDES models.

6.2 Future Work

This dissertation focuses on the formal foundation of a new execution strategy that allows us to execute PTIDES programs without the penalty of totally ordered executions. One direction for future work is to make this new execution strategy more practical by implementing it on different platforms. There are several interesting projects going on in the Ptolemy group working in this direction. To name a few, Derler, Feng, et al. [11] introduce the notion of when events are safe to process and discusses a family of strategies that during execution determine events that are safe to process. Derler, Lee and Matic's work on developing a programming and simulation environment [12] for the PTIDES programming model in Ptolemy II [21] can be used as a visual programming environment for PTIDES and is very useful for understanding the behavior of PTIDES programs. Forbes, Zhou, Matic and Lee's work on PtidyOS [15], a novel lightweight embedded operating system based on PTIDES, is important to make PTIDES practical.

Another area that remains much to be explored is schedulability analysis of PTIDES programs. This dissertation studies schedulability analysis for systems with sporadic actors in their feedback loops. Sporadicness of actors is a very useful concept in determining whether a PTIDES program is schedulable or not. How to tell whether an actor is sporadic or not is not addressed by this thesis but opens as an interesting direction to work on. One may also want to generalize the schedulability analysis work introduced in this thesis to a broader set PTIDES models or even other event-triggered models.

Bibliography

[1] B. Andersson and J. Jonsson. Preemptive multiprocessor scheduling anomalies. In *IPDPS '02: Proceedings of the 16th International Symposium on Parallel and Distributed Processing*, page 12, Washington, DC, USA, 2002. IEEE Computer Society.

[2] N. Audsley. Deadline monotonic scheduling. Technical report, Department of Computer Science, University of York, September 1990.

[3] Sanjoy K. B., Aloysius K. M., and Louis E. R. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[4] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao. Modeling of sensor nets in Ptolemy II. In *Information Processing in Sensor Networks (IPSN)*, Berkeley, CA, USA, 2004.

[5] S. Baruah. Scheduling real-time tasks: Algorithms and complexity. Technical report, http://www.ulb.ac.be/di/ssd/goossens/baruahGoossens2003-3.pdf, 2003.

[6] A. Burns. Scheduling hard real-time systems: A review. *Software Engineering Journal*, pages 116+, may 1991.

[7] C. G. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis*. Irwin, 1993.

[8] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5), 1979.

[9] E. Cheong. *Actor-oriented programming for wireless sensor networks*. PhD thesis, EECS Department, University of California, 2007.

[10] B. A. Davey and H. A. Priestly. *Introduction to lattices and order*. Cambridge University Press, 1990.

[11] P. Derler, T. H. Feng, E. A. Lee, S. Matic, H. D. Patel, Y. Zhao, and J. Zou. PTIDES: A programming model for distributed real-time embedded systems. Technical Report UCB/EECS-2008-72, EECS Department, University of California, Berkeley, May 2008.

[12] P. Derler, E. A. Lee, and S. Matic. Simulation and implementation of the PTIDES programming model. In *Proceedings of the 12-th IEEE International Symposium on Distributed Simulation and Real Time Applications*, October 2008.

[13] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA.*, December 2002.

[14] G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis.* Springer-Verlag, 2001.

[15] S. Forbes, J. Zou, S. Matic, and E. A. Lee. PtidyOS: An operating system based on the PTIDES programming model. In *to appear RTAS 2009*, 2009.

[16] R. Fujimoto. *Parallel and Distributed Simulation Systems.* John Wiley and Sons, 2000.

[17] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, 2003.

[18] G. Goessler and A. Sangiovanni-Vincentelli. Compositional modeling in Metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Grenoble, France, 2002. Springer-Verlag.

[19] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, volume LNCS 2211, Tahoe City, CA, 2001. Springer-Verlag.

[20] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323–363, 1977.

[21] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, , Y Xiong, Y. Zhao, and H. Zheng. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, EECS Department, University of California, Berkeley, July 2003.

[22] L. Sha J. Lehoczky and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium, pp. 166-171*, December 1989.

[23] D. Jefferson. Virtual time. *ACM Trans. Programming Languages and Systems*, 7(3):404–425, 1985.

[24] S. Johannessen. Time synchronization in a local area network. *IEEE Control Systems Magazine*, pages 61–69, April 2004.

[25] H. Kopetz. *Real-time systems: design principles for distributed embedded applications.* Kluwer Academic Publishers, 1997.

[26] H. Kopetz and I. G. Bauer. The time-triggered architecture. In *Proceedings of the IEEE*, pages 112–126, 2003.

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

[28] E. A. Lee. Embedded software. *Advances in Computers*, 56, 2002.

[29] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

[30] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[31] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12), 1998.

[32] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[33] C. L. Liu and J. W. Leyland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[34] J. Liu and E. A. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, pages 65–75, 2003.

[35] J. Liu and E. A. Lee. On the causality of mixed-signal and hybrid models. *6th International Workshop on Hybrid Systems: Computation and Control (HSCC '03)*, April 2005.

[36] X. Liu. *Foundation of the tagged signal model*. PhD thesis, EECS Department, University of California, December 2005.

[37] X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. Technical Report UCB/EECS-2006-67, EECS Department, University of California, Berkeley, May 2006.

[38] L. Mangeruca, A. Ferrari, and A. Sangiovanni-Vincentelli. Uniprocessor scheduling under precedence constraints. *RTAS*, 0:157–166, 2006.

[39] D. L. Mills. A brief history of NTP time: Confessions of an internet timekeeper. *ACM Computer Communications Review*, 33, April 2003.

[40] L. D. Molesky, C. Shen, and G. Zlokapa. Predictable synchronization mechanisms for multiprocessor real-time systems. Technical Report UM-CS-1989-106, University of Massachusetts, 1989.

[41] S. Neuendorffer. *Actor-oriented metaprogramming.* PhD thesis, EECS Department, University of California, 2005.

[42] G. A. Papadopoulos, A. Stavrou, and O. Papapetrou. An implementation framework for software architectures based on the coordination paradigm. *Science of Computer Programming*, 60(1):27–67, 2006.

[43] I. Ripoll, A. Crespo, and A. K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Syst.*, 11(1):19–39, 1996.

[44] M. Saksena. Real-time system design: A temporal perspective. In *Proc. of IEEE Canadian Conference on Electrical and Computer Engineering, Waterloo*, May 1998.

[45] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[46] D. A. Smith, A. Kay, A. Raab, and D. P. Reed. Croquet - a collaboration system architecture. *Creating, Connecting and Collaborating through Computing, International Conference on*, 0:2, 2003.

[47] M. Spuri and J. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *IEEE Transactions on Computers*, 43(12):1407–1412, December 1994.

[48] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. Softw. Eng.*, 23(12):759–776, 1997.

[49] G. Winskel. *The formal semantics of programming languages.* MIT Press, Cambridge, MA, USA, 1993.

[50] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of modeling and simulation.* Academic Press, 2nd edition, 2000.

[51] Y. Zhou. *Interface theories for causality analysis in actor networks.* PhD thesis, EECS Department, University of California, Berkeley, May 2007.

[52] Y. Zhou and E. A. Lee. Causality interfaces for actor networks. Technical Report UCB/EECS-2006-148, EECS Department, University of California, Berkeley, Nov 2006. This paper was accepted for publication by ACM Transactions on Embedded Computing Systems on March 22, 2007.