From Ptides to PtidyOS, Designing Distributed Real-Time Embedded Systems

Jia Zou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-53
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-53.html

May 13, 2011

From Ptides to PtidyOS, Designing Distributed Real-Time Embedded Systems

by

Jia Zou

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

in

Electrical Engineering and Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Edward A. Lee, Chair
Professor Alberto L. Sangiovanni-Vincentelli
Professor Lee W. Schruben

Spring 2011

From Ptides to PtidyOS, Designing Distributed Real-Time Embedded Systems

Abstract

From Ptides to PtidyOS, Designing Distributed Real-Time Embedded Systems

by

Jia Zou
Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Edward A. Lee, Chair

Real-time systems are those whose correctness depend not only on logical operations but also on timing delays in response to environment triggers. Thus programs that implement these systems must satisfy constraints on response time. However, most of these systems today are designed using abstractions that do not capture timing properties. For example, a programming language such as C does not provide constructs that specify how long computation takes. Instead, system timing properties are inferred from low-level hardware details. This effectively means conventional programming languages fail as a proper abstraction for real-time systems.

To tackle this problem, a programming model called "Ptides" was first introduced by Yang Zhao. Ptides builds on a solid foundation in discrete-event (DE) model of computation. By leveraging the temporal semantics of DE, Ptides captures both the functional and timing aspects of the system. This thesis extends prior work by providing a set of execution strategies that make efficient use of computation resources and guarantees deterministic functional and timing behaviors. A complete design flow based on these strategies is then presented.

Our workflow starts with a programming environment where a distributed real-time application is expressed as a Ptides model. The model captures both the logical operations of the system and the desired timing of interactions with the environment. The Ptides simulator supports simulation of both of these aspects. If execution times are available, this information can be annotated as a part of the model to show whether desired timing can be achieved in that implementation. Once satisfied with the design, a code generator can be used to glue together the application code with a real-time operating system called PtidyOS. To ensure the responsiveness of the real-time program, PtidyOS's scheduler combines Ptides semantics with earliest-deadline-first (EDF). To minimize scheduling overhead associated with context switching, PtidyOS uses a single stack for event execution, while still enables event preemptions. The first prototype for PtidyOS is implemented on a Luminary microcontroller. We demonstrate the Ptides workflow through a motion control application.

To my parents Zhen Zou and Ling Wang, my wife Eda Huang, and everyone else whom I've had the pleasure of running into for the first twenty-seven years of my life.

Contents

List of Figures

List of Tables

Acknowledgments

I want to thank my parents, Zhen Zou and Ling Wang for bring up me and making me the kind of person I am today. I'll be eternally grateful for all the sacrifices they made to get me to where I am today. I want to thank Eda, my wife for supporting me throughout my 5 years of study. Even though we have our share of fights, I cannot imagine how I could have pulled it off without her.

I want to thank my advisor, Professor Edward A. Lee, for his mentorship through my years at graduate school. Without his suggestions and encouragements, I can't imagine how I could have completed my Ph.D. I'm also grateful for others on my dissertation committee. Professor Alberto Sangiovanni-Vincentelli's charisma and energy for research is always a great source of inspiration for me. Professor Lee Schruben's timely responses even while he is injured is especially heartening.

I want to thank Slobodan Matic, my main collaborator on the Ptides project. Many of the interesting ideas for my thesis stem from our discussions in the (freezing) embedded lab. I would also like to thank John Eidson and Janette Cardoso for reading my thesis and providing me with timely comments. I want to thank Patricia Derler for her first implementation of the Ptides director in Ptolemy II, her hard work is also a great inspiration for me. I also want to thank Jeff Jensen, who put his heart and soul into the creation of the tunneling ball device.

I would also like to thank the rest of the Ptolemy group. Sharing an office with Ben Lickly has been fun. I really enjoyed all the discussions we had on topics both about research, and also about life in general. I want to thank all those who kept attending the code reviews for the Ptides director implementation, including, but not limited to Mike Zimmer, Christos Stergious, Ilge Akkaya, Christopher Brooks and others I've mentioned earlier. I want to thank Christopher Brooks for forcing good coding habits on me. I didn't get it at first, but he really pointed out the importance of having a good habit for software development to me.

Finally, I want to thanks others in the Ptolemy group, especially Issac Liu and Shanna Shaye Forbes for all the fun we had for the last four years. I had a blast!!

Chapter 1

Introduction

This thesis considers distributed real-time embedded systems in applications such as factory automation, large-scale instrumentation, and network supervision and control. Implementations of such systems consist of networked computers ("platforms") with sensors and actuators distributed throughout a network. Orchestrated actions are required from the platforms. Current practices in designing both the real-time and the distributed aspects of these systems are examined. We identify some fundamental flaws in the design process, and motivate our approach for a new design methodology.

1.1 Design of Real-Time Systems

Real-time systems are those whose correctness of operation depend not only on the logical correctness, but also on the amount of time it takes to perform that logic [3]. These systems can be divided into two categories, *hard real-time* and *soft real-time* systems. Hard real-time systems are those where deadline violations result in immediate system failures, while soft real-time systems are able to tolerate deadline violations to some extent. For example, in a vehicle, the subsystems that control the steering and airbags are generally considered hard real-time systems, while audio control is considered a soft real-time system, since missed beats in the sound track are not safety-critical. A data logging subsystem that runs in the background is usually considered a non-real-time system. We focus on hard real-time systems in this work.

Real-time systems are prevalent in embedded environments since interactions with the physical world are generally only meaningful if performed within a bounded period of time. However, to program these real-time systems, engineers use tools that do not offer the necessary abstractions. Specifically, programming languages such as C are often used for embedded system design, yet these languages only provide constructs for the programmer to specify systems' logic, not their timing. Programmers are forced to step outside of the programming language in order to specify real-time behavior. A common practice is to control timing using priorities. A real-time operating system (RTOS) is a prime example of this practice. RTOSs are used to perform resource allocation and schedule real-time programs. Most RTOSs, such as VxWorks [51], RTLinux [16], FreeRTOS [48], allow programmers to assign priorities to system tasks. These priorities are used to formulate a schedule, and the timing of the system is a consequence of this schedule. There are two major disadvantages with this approach.

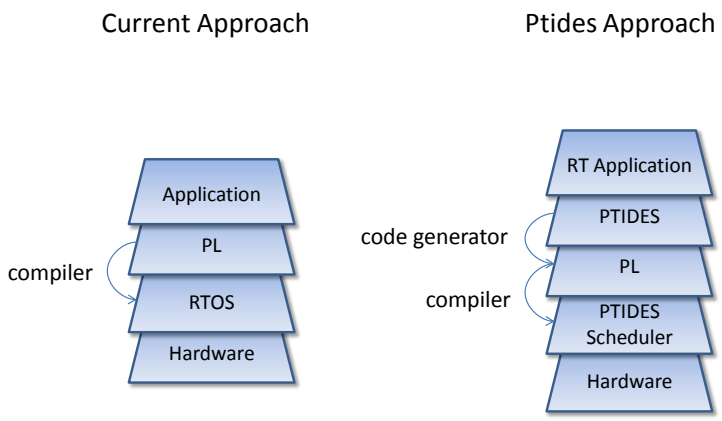

Figure 1.1: Comparison Between Current Approach and the Ptides Approach for the Design of Real-time Systems

1. Along with priorities, execution times are needed to infer the timing delays of a system. However execution time is hardware-dependent information.

2. Sometimes it is unintuitive for programmers to define whether some task is "more important" than another, and it is even more obscure to understand how the priority assignments affect the schedulability of the system.

Since the timing of the system is a consequence of the priority assignments and execution times, the hardware characteristics must be known right from the start of the design process. Another consequence is that once the design is completed for a particular platform, the entire hardware-software stack must be frozen, and cannot be easily upgraded, debugged, or ported to another platform. Any change to even the most minuscule detail of the system may result in timing anomalies. Zhao first introduced a new programming model, called Ptides, which addresses this problem by allowing programmers to fully specify the logic and the timing of the system within the model, without the knowledge of execution times [53]. This approach decouples the software design from the hardware platform. Another way to view Ptides is that it's an additional abstraction layer that captures both the logic and the timing of a system. Fig. 1.1 visualizes and compares these two approaches.

To address the second point, system timing constraints are defined between sensors and actuators. That is, from an environment point of view, the only delays that are of interest are end-to-end delays. For example, a real-time requirement for the safety system of a vehicle would be something in the form of: when the system senses the car has hit a tree, the airbags must pop out (actuate) within n milliseconds. Let us assume multiple tasks are involved to pop out the airbag. Notice the real-time constraint says nothing about the priorities of these tasks, yet the priorities have a direct effect on whether the timing constraint will be satisfied. This can make priority assignments unintuitive and difficult to understand. Ptides takes a different approach. Since the timing requirements are usually defined between sensors and actuators, those are the only timing delays programmers assign in a Ptides model. The Ptides scheduler then uses this information to *infer* event

priorities and perform real-time scheduling. We will explain in more detail in the following chapters about how one specifies timing delays in a Ptides model, but to summarize, conventional RTOS allows programmers to specify task priority. The end-to-end timing behavior is a consequence of these priorities. Ptides on the other hand, allows programmers to explicitly define the timing of the system, and the system event priority is then a consequence of the timing.

1.2 Design of Distributed Systems

Not only is Ptides an abstraction for real-time systems, its scheduling scheme is especially useful in distributed environments. We consider distributed systems consisting of independent computing platform with sensors and actuators, As early as 1978, Lamport pointed out that a "happens-before" relationship between system events could be easily defined if components share a global notion of time [29]. This was the first indication that network time synchronization has the potential to significantly change how we design distributed systems. Clock synchronization protocols such as NTP [40] have been around for a while, however these services only offer time synchronization precision on the order of milliseconds, which is too loose for some applications. However, recently, techniques such as GPS [27] and IEEE 1588 [22] have been developed to deliver synchronization precision on the order of hundreds of nanoseconds, and in certain cases even down to the order of nanoseconds [1]. Such precision offers truly game-changing opportunities for distributed embedded software.

The deterministic semantics of Ptides benefits greatly from a global notion of time. As we will explain in Chap. 3, by relating model time to physical time at specific points in the model, Ptides guarantees deterministic order of event processing in local platforms. By sharing a global notion of physical time across platforms, this deterministic order can be expanded to events arriving through networked components. Thus we envision Ptides to be useful both in single platform and distributed systems that are time synchronized.

1.3 Ptides Model and Design flow

Due to the Ptides programming model's ability to capture system timing properties, as well as the inherent property of maintaining timed semantics in a networked, time-synchronized environment, we argue it is advantageous to program distributed real-time systems as Ptides models. This is contrary to the typical approach where software tasks are defined as threads with periods, priorities, and deadlines.

Ptides (pronounced "tides," where the "P" is silent), is an acronym for *programming temporally integrated distributed embedded systems*. Ptides builds on the solid semantic foundation of Discrete Event (DE) models [8, 19]. DE specifies that each actor should process events in timestamp order, and thus the order of event processing is independent of the physical times at which events are delivered to the actors. The particular variant used in Ptides provides a determinate semantics without requiring actors to introduce time delays between inputs and outputs [33]. Whereas classically DE would be used to construct *simulations* of such systems, in Ptides, the DE model is an executable specification. By leveraging network time synchronization [25, 14], Ptides provides a mechanism for distributed execution of DE models [20, 9, 41, 23] that requires neither backtracking (as in [23]) nor null messages (as in [9, 41]).

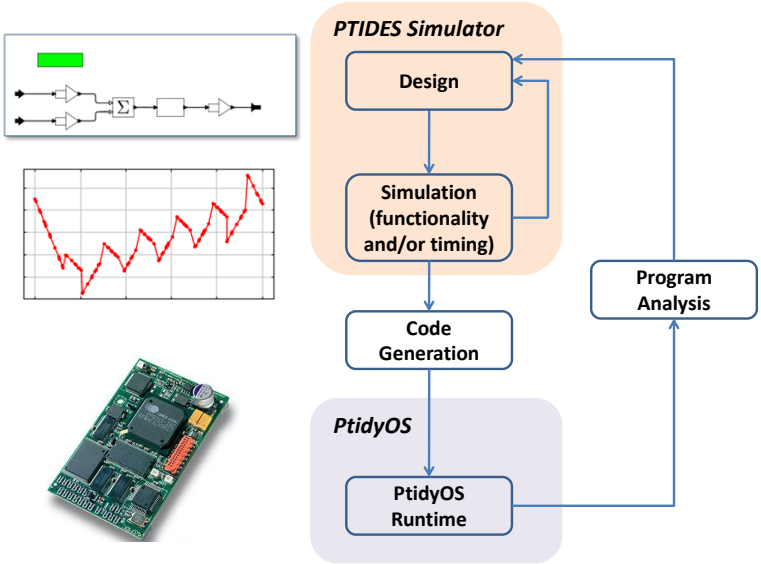

Figure 1.2: Ptides Design Flow Visualization

The objective with Ptides is to compile (or code generate) this specification into a deployable implementation. Hence Ptides follows the principles of model-based design [28]. This approach shortens the design cycle for real-time embedded applications. It also allows the programmer to separate the design from the implementation, where applications can be easily ported from one platform to another while retaining system logical and timing determinacy.

This thesis presents a design flow that encompasses a simulator, a code generator, and an ultra-lightweight real-time operating system, as shown in Fig. 1.2. The simulator is built on the Ptolemy II framework [15]. Since the timing of the program is captured as a part of the Ptides model, distributed real-time applications can be designed and simulated without the knowledge of execution times. Once execution times are known, the Ptides simulator can simulate physical time and event preemption. This in turn allows the programmer to check for scenarios in which deadline misses occur. Once satisfied with the design, a target-specific code generator can be used to transform the model into a C implementation. This implementation runs on a real-time operating system called PtidyOS. In order to ensure logical and timing determinism, as well as responsiveness of the real-time program, PtidyOS's scheduler combines Ptides semantics with traditional scheduling methods such as earliest-deadline-first (EDF). To minimize scheduling overhead associated with context switching, PtidyOS performs all event processing in interrupt service routines, while only using a single stack for event execution.

The remaining chapters are organized as follows. Chapter 2 presents work related to the Ptides programming model. Chapter 3 reviews basic concepts such as notions of time and the relationships between them, as well as a baseline execution strategy. Chapter 4 defines a family of execution strategies for Ptides that assures compliance with DE semantics and support distributed scheduling and control. Each of these strategies is based on different assumptions, and each has different scheduling properties. Chapter 5 then talks about the Ptides design flow, from simulator to code generator to PtidyOS. Chapter 6 applies this design flow on an application called the tunneling

ball device. We conclude and provide pointers for future work in Chapter 7.

Chapter 2

Related Work

This chapter explores previous work that is related to either the Ptides programming model or the real-time operating system PtidyOS.

2.1 Logical and Physical Time in Distributed Systems

There exists history dating back to the 1970's concerning the use of logical (model) and physical time to schedule distributed systems. In [29], Lamport famously introduced the use of logical time to define a "happened-before" relationship between two events. He also presented an algorithm that uses logical time to totally order system events. Lamport then gave an example showing that by maintaining a total order, certain anomalous event processing behaviors can be avoided. Namely, logical determinism can be preserved. However, having totally ordered events may lead to inefficient scheduling, since events that are inherently concurrent are forced to process in order. This makes Lamport's scheme largely inapplicable to real-time system.

In [9], Chandy and Misra introduced an algorithem based on a discrete-event (DE) model of computation, which also uses logical time, but ensures determinacy while allowing inherently concurrent elements to execute independently. Their algorithm is widely used in the area of circuit simulation. However, since their algorithm focuses on simulation, only logical, but not physical time is used.

Lamport on the other hand, defines another "happened-before" relationship between system events by assuming an upper bound on network transmission delay, as well as an upper bound on the time synchronization error between physical clocks on distributed platforms. By utilizing physical clocks as part of the execution semantics, Lamport's scheduling mechanism can be applied to implementations. The Ptides approach is similar to [9] but is used for distributed real-time system scheduling instead of simulation. Moreover, Ptides uses physical clocks in its scheduling scheme, thus eliminating the need for "null messages" [9] or backtracking [23]. Like [29], by assuming bounded network delay and clock synchronization error across distributed physical clocks, and by conforming to DE semantics, Ptides ensures logical determinism.

Ptides also offers interesting fault tolerance properties. Lamport noted in [29] that:

The problem of failure is a difficult one... Without physical time, there is no way to distinguish a failed process from one which is just pausing between events. A user can tell that a system has "crashed" only because he has been waiting too long for a response.

As we will show in later sections, the Ptides framework, by using physical clocks, makes the problem of failure identification a simple one, and we can define formally what constitutes "waiting too long".

2.2 Distributed Discrete Event Simulation

Chandy and Misra started a field of work on distributed DE simulation, with the goal of accelerating simulation by exploiting distributed computing resources. The approach described in [9] is one of the so-called "conservative" techniques, which process timestamped events only when it is sure that no earlier timestamped event will be delivered to the same software component. So-called "optimistic" techniques [23] speculatively process events even when there is no such assurance, and roll back if necessary.

For distributed embedded systems, the potential for roll back is limited by actuators (which cannot be rolled back once they have had an effect on the physical world) [17]. Established conservative techniques, however, also prove inadequate. In the classic Chandy and Misra technique [9], each compute platform in a distributed simulator sends messages even when there are no data to convey in order to provide lower bounds on the timestamps of future messages. The messages that only carry time stamp information and no data are called "null messages." This technique carries an unacceptably high price in our context. In particular, messages need to be frequent enough to prevent violating real-time constraints due to waiting for such messages. Not only do these messages increase networking overhead, but the technique is also not robust. Failure of a single component results in no more such messages, thus blocking progress in other components. Ptides is related to several efforts to reduce the number of null messages, such as [20], but makes much heavier use of static analysis, thus minimizing network communication overhead.

2.3 Synchronous Languages

Considerable research activity has been devoted to exploring high-level MoCs for embedded systems, and Ptides is not the first to incorporate timed semantics into system models. Synchronous languages such as Esterel, Lustre, Signal, SCADE, and various dialects of Statecharts have long been used for the design of embedded systems [4], however, there are major differences between synchronous languages and Ptides. In Synchronous languages, logical time (also called model time) is expressed as a member of the natural numbers. At every tick, all system components execute until a fixed point is reached, and the program steps to the next tick. When implementing applications, it is intuitive to map this logical time (ticks) to the period at which data is sampled. This allows for a mapping from synchronous models to a time-triggered architecture (TTA) or loosely time-triggered architecture (LTTA) [50, 5, 7], where the system is time-triggered at each tick. Ptides on the other hand, expresses logical time as the superdense model of time [39]. i.e., logical time is expressed as a pair of timestamp and microstep, where timestamps are members of the Real number set. Thus it is much more intuitive to map Ptides's timestamps to physical time. The finer granularity of logical time makes Ptides much more suitable for event-triggered applications. Since time-triggered is a special case of event-triggered architecture, we argue Ptides can be used to design a more general class of systems.

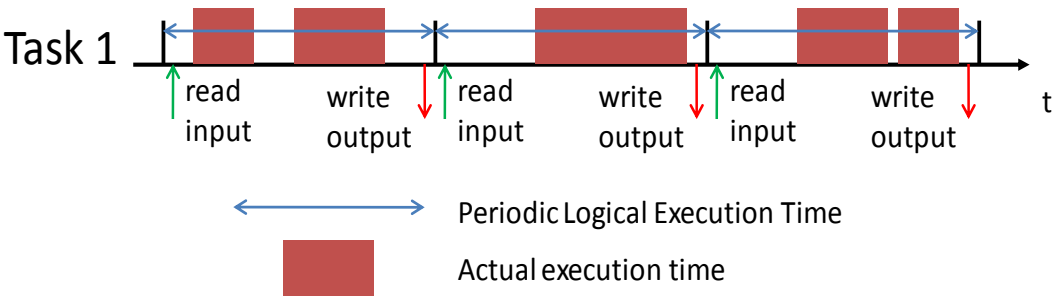

Figure 2.1: Giotto Schema

2.4 Giotto

Another high level MoC used for the design for real-time systems is Giotto. The notion of logical time in Giotto is "logical execution time" (LET) [21]. An example is shown in Fig. 2.1. Each task in Giotto has a LET of a fixed period. Giotto specifies that data reads must happen at the beginning of the LET, while data writes must occur at the end. Task executions must occur during LET, and the end of the LET is effectively the deadline for that task. However, the execution does not need to span the entire LET, i.e., tasks can be preempted during the LET.

Ptides is different from Giotto in two regards: 1) Giotto assumes periodic tasks, while Ptides takes an event trigger approach, and 2) Giotto relates logical time and physical time at points of data transfer between tasks. Ptides relates them at points where communication with the outside world occurs. In other words, Giotto enforces deadlines for every periodic task in the system, including those that do not interact with the physical environment. Ptides takes a different approach, where deadlines are only enforced at points where interactions with the physical environment occur. Our intuition is that since from the environment point of view, the only places where timing is relevant are at sensors and actuators, these are the only points where timing of the interactions with the outside world should be enforced.

2.5 Scheduling Aperiodic Tasks

Finally, we review previous work that deals with real-time scheduling of aperiodic events. Periodic scheduling scheme such as rate-monotonic (RM) has been widely adopted in the industry [45], mainly for its simplicity in implementation. To build on its success, the research community has worked on extending periodic scheduling scheme to allow aperiodic tasks [49, 35, 46]. These approaches use a special purpose process called a "server" to schedule aperiodic tasks. These servers take a "slack-stealing" approach, where aperiodic tasks can "steal" as much processing power as possible, without causing periodic tasks to miss their deadlines. This approach is based on the assumption that periodic tasks have hard deadlines, while aperiodic tasks have soft or "firm" deadlines. Tasks with firm deadlines are those whose executions can be rejected by the scheduler, but the scheduler will meet deadlines for the accepted tasks. However, this assumption is not true in many real-world applications. For example, faults are inherently aperiodic, and they usually re-

quire hard real-time processing in order to guarantee safety of the system. Ptides takes a purely event-triggered approach, and treats all periodic and aperiodic tasks equally as hard real-time tasks. Processing power is allocated purely based on events' deadlines, where these deadlines are inferred from the real-time specifications of the system.

Another difference between classic scheduling approaches and Ptides is that with its timed semantics, Ptides schedulers ensure not only the timeliness of event processing, but also logical determinacy of the system. Ptides defines a "safe-to-process" analysis, which releases events only if they are processed in logical time order. This aspect is absent from the classic scheduling theory, where the deterministic behavior of the system is ignored, even though it can be directly affected by the scheduling scheme used.

Chapter 3

Background

3.1 Ptides Model Formuations

Ptides, an acronym for Programming Temporally Integrated Distributed Embedded Systems, is a programming model based on the discrete-event (DE) model of computation. DE is used for hardware simulation in languages such as VHDL, for network simulation, and for complex system simulations [52]. It has the advantages of simplicity, time-awareness, and determinism. Ptides takes advantage of DE semantics and offers a programming model that ensures logical and timing determinacy. By logical determinacy, we simply mean given the same sequence of timestamped inputs, the same sequence of timestamped outputs will always be produced. In addition, depending on the implementation of the actuators, Ptides models can be specified to guarantee timing determinacy, i.e., the outputs will always be produced after the same physical time delay, assuming no deadline misses occur [54, 17].

Ptides has been formally defined in [56]. Here, we will review the basic formulation as well as a baseline execution strategy. All execution strategies in the following chapter are based on this baseline strategy.

First, we introduce some basic background about the Ptides formulation. The detailed formulations are given in [56] and [17], but here we will try to focus on examples that demonstrate the intuitive ideas behind each formulation.

3.1.1 Actor-Oriented Programming

We present Ptides in the context of actor-oriented programming methodology [31]. In this methodology, the most primitive software component is called an *actor*, which one can interpret as a block of code. Actors communicate with each other through explicit connections. An example is shown in Fig. 3.1.

In this work we assume actors cannot have underlying communications that are not explicitly expressed in the model. Take the example shown in Fig. 3.1, Computation1 and Computation2 cannot access shared variables. By explicitly specifying the points where communication occurs between code blocks, potential data race conditions for shared variables are eliminated (which is a major source of bugs in thread based approaches [30]), and the entire model becomes much easier to analyze.

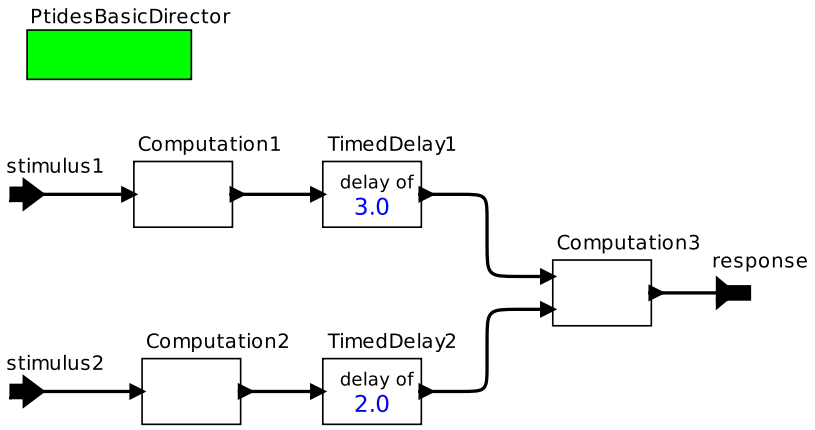

Figure 3.1: Actor Oriented Programming

3.1.2 Notions of Time and Their Relationships

This section explores notions of time in a Ptides model, as well as their relationships with each other.

Physical Times

In a distributed system context, we assume each platform has a local clock that keeps track of the physical time. This clock could be implemented, for example, using a hardware timer in a microcontroller. The time maintained by this clock is called *"platform time"*. In a system that is not time synchronized, the platform times of distributed platforms are completely independent from each other. As mentioned in the introduction, this work is mainly concerned with platforms that share a global notion of time. In other words, we assume platform times across the distributed system to be synchronized within some bounded error.

We also introduce a notion of *"oracle time"*. We assume there exists an all-knowing oracle that keeps track of the "correct" physical time in a distributed system. Special relativity tells us that this notion of oracle time does not exist, but for all practical purposes we assume it exists for the applications we are interested in. Most time-synchronization protocols implement a master-slave scheme. In this scheme, one platform is chosen to be the master, while slaves try to synchronize their clocks with respect to the master clock. Here the oracle time can be interpreted the time of the master clock.

We now define a function $\mathbb{F}_i$, which converts oracle time t_o to platform time t_i for a particular platform i,

$$t_i = \mathbb{F}_i(t_o). \tag{3.1}$$

In other words, t_i is the time of the platform clock at oracle time t_o. We assume $\mathbb{F}_i$ to be a bijection. This means the platform clock can only drift with respect to the oracle clock at a rate greater than zero, where a rate of one would indicate the platform clock drifts at the same rate as the oracle clock. Another consequence of this assumption is that platform clock cannot be reset backward

to an arbitrary value. With this function, we can define the clock synchronization error bound $\epsilon_{i,j}$ between two platforms i, j,

$$\epsilon_{i,j} = \max_{t_o \in \mathbb{R}} \{|\mathbb{F}_i(t_o) - \mathbb{F}_j(t_o)|\}. \tag{3.2}$$

Notice this error bound could be unbounded. For example, if two platforms drift at different rates with respect to the oracle clock, and this rate is never updated, then the error bound would be infinity.

We can then deduce the *maximum clock synchronization error bound* for a distributed system:

$$\epsilon = \max_{i,j \in P} \{\epsilon_{i,j}\}, \tag{3.3}$$

where P is the set of all platforms in this system.

Model Time

In DE and Ptides, actors communicate through explicit links by sending events. An event is a pair with a data value and a tag. In this case, Ptides' tag uses the superdense model of time [34], which means the tag is a tuple of a timestamp and a microstep.

With these tags, we can define an ordering between two events. The order we are interested in is a lexicographical ordering of the following tuple: $< T, I >$, where $T = \mathbb{R}$ represents the set of event timestamps. $I = \mathbb{N}$ represents the set of events' microsteps. For $e_1 =< \tau_1, i_1 >$, $e_2 =< \tau_2, i_2 >$, and $e_1, e_2 \in < T, I >$, the ordering is defined as:

$$\begin{aligned} e_1 \ < \ e_2 \ \Leftrightarrow \ & (\tau_1 < \tau_2) \ \vee \\ & (\tau_1 = \tau_2 \wedge i_1 < i_2) \end{aligned}$$

These tags are also referred to as "*model time*" (also called logical time). Specifically, if an event of a certain tag is currently being processed in a platform, then that tag is the current model time of that platform. Notice here we assume each platform has a single processor, and only one event is actively processed at a time. The Ptides formulation can be extended to work in a multi-core environment, however we will restrict ourselves to single core CPUs in this thesis. Actors can manipulate event tags, with the constraint that all actors must be causal. i.e., the tag of an output event must be larger than or equal to that of the triggering input event.

Each of the actors may also have a state associated with them. DE semantics specifies that each actor state must be accessed by events in tag order. For example, if the processing of an input event results in the read and/or write of the state of the actor, then that actor must process input events in tag order.

Note the notion of model-time is distinctly different from time in the physical world. For example, while time in the physical world has a total order, model time in Ptides does not have to observe a total order. The scheduler may schedule an event of bigger tag before processing one of smaller tag, as long as these events do not access the same actor's state.

3.1.3 Real-Time Ports

Sensor and Actuator Ports

As mentioned before in Chap. 2, there is a long history behind distributed discrete event simulation. In that context DE is used as a simulation tool, the simulation runs entirely in model time. The Ptides model however, relates model time to platform time. These relations are introduced at points where interactions between the computer and the physical environment occur. We call these points *real-time ports*. The definitions for these ports were first introduced in [56], and we review them here.

Sensor ports are inputs to a Ptides platform. When sensing occurs, measurements are taken, and the measured data value is timestamped with the current platform time. In other words, a sensor event is created. Its value is the measured data, and its tag consists of microstep 0, and timestamp equal to the platform time of when sensing occurred. This event is sent to the Ptides scheduler and made visible to the rest of the platform. Let the sensor event's timestamp be τ, and the platform time at which the event is visible to the scheduler is t, then the following equation must be true:

$$t \geq \tau, \tag{3.4}$$

Also if we assume an upper bound on the worst-case-response-time of a sensor actor, then:

$$t \leq \tau + d_s, \tag{3.5}$$

where $d_s \in \mathbb{R}^+ \cup \{\infty\}$ is a parameter of the port called *maximum physical time delay* (also referred to as the *real-time delay*).

On the actuator side, we enforce the following rule: an event e with timestamp τ must be delivered to the environment at platform time equal to τ. This implies e must be sent to the actuator at platform time $t - d_a$, where d_a is again the worst-case-response-time for the system to deliver an actuation signal to the environment. In other words, let t be the delivery time of an event to the actuator, the following constraint must hold:

$$t \leq \tau - d_a. \tag{3.6}$$

This constraint imposes a platform time deadline for each event delivered to an actuator, where the timestamp $\tau - d_a$ serves as the deadline. If there exists multiple actuation events of the same timestamp τ, the microsteps of the tags are used to define the order by which actuations occur.

Notice here, we only defined what each element of the tag means at actuators, but we do not specific how the system enforces these deadlines and actuation orderings defined by the tags. These are application-dependent concepts. For example, if two actuation events have timestamps whose difference is less than d_a, then the second actuation event will not be delivered to the environment on time. We leave it to the application programmer to decide what to do in such a scenario. The programmer might enforce certain rules at other parts of the model to ensure such a scenario does not occur, or he/she might decide the delay is small enough such that the real-time properties of the system is not affected, or he/she might decide to throw an exception. In any case, we only present the semantic meaning of the tag at actuators, but the actual implementation is application-dependent.

Network Input and Output Ports

Since we envision Ptides as a programming model for distributed systems, we must properly define timed semantics for data transmitted across distributed platforms. Two alternatives are first considered:

1. only data is transmitted; data is timestamped with the local platform time at the receiving platform.

2. a data and tag pair is transmitted; the event retains its original tag at the receiving platform.

The first alternative considers network inputs as sensor ports. The advantage of this scheme is that each platform is essentially independent of the other, and the safe-to-process analysis of Ptides (which we will introduce in the next section) can be performed in each platform independently, without the knowledge of execution time in other platforms. The major disadvantage of this scheme is that the model's behavior becomes dependent on the network delay. i.e., a model will only produce deterministic data outputs if the network delay does not change during different runs. Notice most commonly used network transmission schemes do not guarantee such behavior.

The second scheme, on the other hand, preserves the original timestamps as events travel across network boundaries. This means the logical output of the model will always be deterministic regardless of the network delay, provided DE semantics is preserved. However, if the network delay can be arbitrarily large, preserving DE semantics becomes a challenge. The related work section mentioned so-called conservative and optimistic approaches to deal with this problem. Inspired by [29], Ptides steps away from these approaches. Instead, the following assumptions about network and clocks are made:

1. maximum clock synchronization error between platforms is bounded by ϵ.

2. maximum network communication delay is bounded by d_n.

The first assumption has already been described in Sec. 3.1.2. The second assumption simply states that if platform i sends a packet to platform j at oracle time t, then platform j must receive this packet no later than oracle time $t + d_n$.

From these assumptions we infer the following property: If data is transmitted from platform i at its platform time t_1, then it must reach the receiver platform j, at Platform j's time, no later than

$$t_1 + d_n + \epsilon. \tag{3.7}$$

We then make the same requirement at network outputs as actuators: The platform time t at which an event e must be delivered to an network output device must be no later than τ, where τ is the timestamp of e. I.e.,

$$t \leq \tau. \tag{3.8}$$

Combining Eq. 3.7 and 3.8, we have the following property at the sink network input device: when platform j receives a network input event of timestamp τ at platform time t_j:

$$t_j \leq \tau + d_n + \epsilon. \tag{3.9}$$

Notice that Eq. 3.9 is in the same form as Eq. 3.5, with $d_s = d_n + \epsilon$. This effectively allows us to treat network inputs the same way as sensor inputs, thus retaining the advantage of

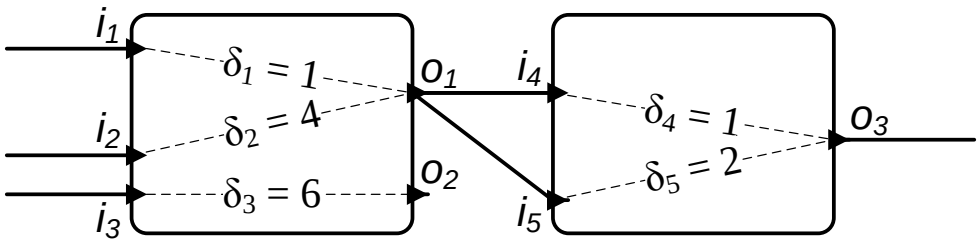

Figure 3.2: Model Time Delay

treating each platform separately when formulating the Ptides execution strategies. The details of this is discussed in Sec. 3.3.

3.2 Causality Relationships Between Ports

Before describing the Ptides execution strategies, we first review the concept of model time delay.

Intuitively, model time delay is the amount by which tag value is incremented by an actor as it consumes and produces events. For example, If an actor provides a model time delay of x, then after consuming an input event of tag $(\tau, 0)$, the actor should produce an output event of tag $(\tau + x, 0)$.

Here we review two model time delay functions δ_0 and δ, which were first defined in [56]. These functions are rooted in causality interface theories [55], and both take a pair of ports as arguments. The first function $\delta_0(i, o)$ takes a pair of ports as argument, and i, o are input and output ports, respectively. δ_0 represent the *minimum* model time delay between this pair of ports. If these ports do not belong to the same actor, or if there is no causality relationship between these ports, then $\delta_0(i, o) = \infty$. For example in Fig. 3.2, the dotted lines within an actor indicate that the pairs of input and output ports are causally related, while δ_i is the minimum timestamp delay between these ports. For simplicity, we ignore the microsteps field of the delays, which are all assumed to be zero. In this example, consumption of an input event at i_1 leads to the production of an event at o_1, and $\delta_0(i_1, o_1) = \delta_1 = 1$, while $\delta_0(i_1, o_2) = \delta_0(i_1, o_3) = \delta_0(i_3, o_1) = \infty$. Since all actors are assumed to be causal, $\delta_0(i, o)$ takes a minimum value of 0.

The second delay function $\delta(p_1, p_2)$ takes a pair of arbitrary ports (can both be input or output ports) in the system. Intuitively, $\delta(p_1, p_2)$ is the minimum model time delay from p_1 to p_2. Note unlike the δ_0 function, ($\delta(p_1, p_2)$ can take a value less than ∞ even if p_1 and p_2 do not reside in the same actor). Again using the example shown in Fig. 3.2, $\delta(i_1, o_3) = min\{\delta_1 + \delta_4, \delta_1 + \delta_5\} = 2$, while $\delta(i_3, o_3) = \infty$.

We also review the definition of input port groups. The formal definition of an input port group is as follows:

Let I be the set of all input ports on all actors in the system, and O be the set of all output ports in the system. $i', i'' \in I$ belong to the same input port group if and only if:

$$\exists o \in O \mid (\delta_0(i', o) < \infty) \wedge (\delta_0(i'', o) < \infty)$$

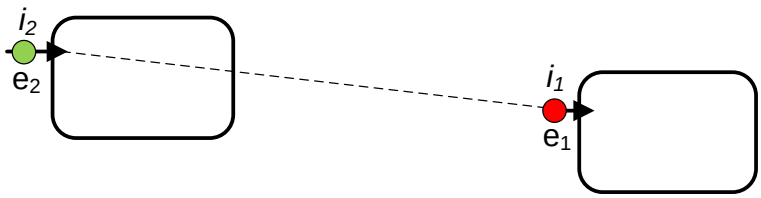

Figure 3.3: Causality Relation Example

Note the causality function used here is δ_0 instead of δ. In other words, two input ports reside in the same input port group if and only if they reside in the same actor, and they are both causally related to the same output port. Again take Fig. 3.2 as example, i_1, i_2 belong in one input port group, while i_3 belongs to another input port group all by itself. We use the notation $\mathbb{G}$ to denote input port groups, where $\mathbb{G}_{i_1} = \mathbb{G}_{i_2} = \{i_1, i_2\}$, and $\mathbb{G}_{i_3} = \{i_3\}$.

Armed with the above definitions, we are ready to define the baseline strategy.

3.3 Baseline Execution Strategy

All Ptides strategies presented in this work take an event (here on referred to as the *event of interest*) in the system as argument. If the strategy returns true, the destination actor can "safely" process this event. By "safe," we mean the same input port group will not receive an event of smaller tag at a later physical time. If an event e could potentially render the event of interest unsafe, then we say e *causally affects* the event of interest. This term is inspired by Lamport in [29]. Take Fig. 3.3 for example, where e_2 causally affects e_1 if the processing of e_2 will produce a future event e_2' such that e_2' and e_1 have destination ports that reside in the input port group, and e_2' may have a tag that is less than or equal to that of e_1. In other words,

e_2 *causally affects* e_1 *if and only if:*

$$\exists i_x \in \mathbb{G}(i_1) \mid \tau(e_2) + \delta(i_2, i_x) \leq \tau(e_1), \tag{3.10}$$

where i_2 and i_1 are the destination ports of e_2 and e_1, respectively, and $\tau(e)$ is a function that maps an event e to its tag.

As Yang Zhao pointed out in [54], our problem formulation of an execution strategy is very much the same as distributed discrete event simulation. Chandy and Misra defined a strategy in [9] that solves this problem using "null messages." In the Ptides formulation however, by relating model time to physical time, and by assuming a global notion of time across all computation platforms, and a bounded network delay, the transmission of "null messages" can be replaced by a comparison between the timestamp of the event and physical time to check whether events are safe [54].

In summary, the strategies presented in this thesis: 1) leverage network time synchronization to eliminate the need for null messages, 2) improve fault isolation in distributed systems by making it impossible for components to block others by failing to send messages, and 3) allow causally independent events to be processed out of tag order, thus increasing concurrency and allowing more models to become schedulable w.r.t real-time constraints.

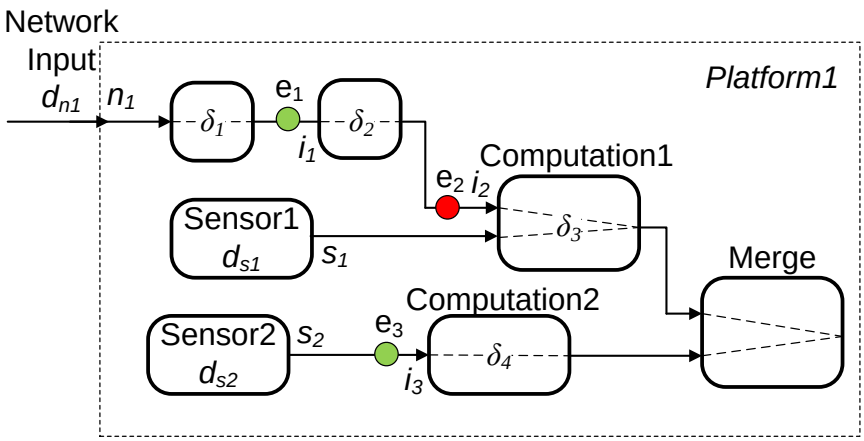

Figure 3.4: Simple Ptides Example

A general execution strategy that is the basis for all execution strategies presented in this thesis was presented in [56]. Instead of reviewing the general execution strategy, a baseline execution strategy is described. This baseline strategy is similar to the general strategy, but it makes an explicit assumption about the selection of the *dependency cut* [17]. The dependency cut is used to separate the general strategy into two parts. Based on where the cut is defined, two different schemes are used to determine whether the event of interest can be causally affected by events inside and outside of the cut. Interesting readers can refer back to [56] for the definition of the dependency cut and its significance, however we only identified a single significant use of the cut, and that is the cut used in this thesis. This cut is defined at the "boundary" of the platform. By "boundary," we mean points at which communications (both input and output) with the outside of the platform occur. This output may include the rest of the distributed system or the physical environment, or both. Since each platform receives inputs from the outside through sensors and network input devices, we assume the cuts are always made at the real-time sensor and network input ports in the platform. Let us denote all these input ports as $\mathbb{I}_p$, where p is the platform in which these ports reside. Using Fig. 3.4 as an example, $\mathbb{I}_p = \{n_1, s_1, s_2\}$. This definition is used to separate the baseline strategy into two parts.

3.3.1 Example Illustrating the Baseline Execution Strategy

Recall that the goal of the baseline strategy is to define whether an event of interest can be causally affected by other events in the system. We separate the baseline strategy into two parts, based on the location of the other events.

The first part checks for events outside of this platform. We again use Fig. 3.4 to demonstrate our strategy. Let e_2 be our event of interest. Let the tag of this event be τ_2. The first part of the strategy should return true if neither Sensor1 nor the Network Input can produce events arriving at the Computation1 with smaller tag.

Let's look at the bottom path from Sensor1 to Computation1, and assume real-time delay $d_{s1} = 0$ for now. Recall that a sensor relates model time and platform time by producing

events with a timestamp that is exactly equal to the platform time at which sensing occurred and a microstep 0. This implies that at a later platform time t, the sensor cannot produce an event of timestamp $\tau < t$. Thus we know *after platform time* τ_2, Computation1 *will not receive an event of timestamp smaller than* τ_2 *from* Sensor1. If $d_{s1} > 0$, then Eq. 3.5 says an event of timestamp τ will be produced at a platform time t no later than $\tau + d_{s1}$ from the sensor. Thus we can modify our formulation to be: *After platform time* $\tau_2 + d_{s1}$, Computation1 *will not receive an event of timestamp smaller than* τ_2 *from* Sensor1.

The same holds for the path from the Network Input; the only difference is that model time delays δ_1 and δ_2 are present. This implies each event received at the Network Input will have its timestamp incremented by at least $\delta_1 + \delta_2$ when it arrives at Computation1. Thus our formulation becomes: *after platform time* $\tau_2 - \delta_1 - \delta_2 + d_n + \epsilon$, Computation1 *will not receive an event of timestamp smaller than* τ_2 *from* Network Input.

Finally, notice we do not need to worry about any events sensed at Sensor2. This is because events produced by Sensor2 do not causally affect e_2; i.e., $\delta(s_2, i_2) = \infty$.

Combining the above formulations, we know events from outside of platform cannot causally affect e_2 when platform time passes $\max\{\tau_2 - \delta_1 - \delta_2 + d_n + \epsilon, \tau_2 + d_{s1}\}$.

The second part of the baseline strategy checks to see whether any event from *inside* of the platform can causally affect the event of interest. Again take Fig. 3.4 for example. To determine whether e_2 is safe-to-process, we need to check whether e_1 will result in future events of smaller timestamp than e_1 at the inputs of the Computation1 actor. Since the minimum model time delay between i_1 and i_2 is δ_2, we can conclude e_1 is safe to process if $\tau_1 + \delta_2 > e_2$, where τ_1 is the timestamp of e_1.

3.3.2 Formal Definition for Baseline Execution Strategy

We now give a formal definition of the baseline strategy, but first we define a *physical-time delay* function $d\colon I \to \mathbb{R}^+ \cup \{-\infty\}$ that maps each port $p \in I$ as follows:

$$d(p) = \begin{cases} d_o & \text{if } p \text{ is a sensor input port,} \\ d_n + \epsilon & \text{if } p \text{ is a network input port,} \\ -\infty & \text{otherwise.} \end{cases}$$

Recall from Sec. 3.2 I is the set of actor input ports in this system. In other words, all inputs that are not sensor or network input ports have $d(p) = -\infty$. Notice the following strategy does not assume events are produced in tag order. In other words, for an event with timestamp τ residing at input port i, there is still the possibility of another event of tag smaller than τ arriving at i at a later platform time. Sec. 4.6 introduces a strategy that assumes events are produced in tag order. A formal definition of the baseline strategy is as follows:

Baseline Strategy. *An event with timestamp τ on platform f and on actor input port $i \in I$ is safe to process when:*
Part 1: platform time exceeds:

$$\tau + \max_{p \in \mathbb{I}_f, i' \in \mathbb{G}_i} \{d(p) - \delta(p, i')\}, \tag{3.11}$$

and
Part 2: for each port $p' \in I$ in platform f, each event at the input queue of p' has time stamp

1. *greater than*

$$\tau + \max_{i' \in \mathbb{G}_i} \{-\delta(p', i')\} \;\; if \;\; p' \notin \mathbb{G}_i. \tag{3.12}$$

2. *greater than or equal to*

$$\tau \;\; if \;\; p' \in \mathbb{G}_i. \tag{3.13}$$

Recall $\mathbb{G}_i$ denotes the input port group for port i, while $\mathbb{I}_f$ denotes the set of sensor and network input ports for platform f. Eq. 3.11 makes sure that no input port $p \in \mathbb{I}_f$ (sensors and network inputs) will produce an event with timestamp greater than τ at a later physical time. Note the second part of baseline strategy is again divided into two sections. Going back to the example in Fig. 3.4, Eq. 3.12 would check to see whether e_1, e_3 causally affects e_2, while Eq. 3.13 would check to see whether there exist events at other input ports of the Computation1 actor that causally affect e_2. If, say, the bottom input of Computation1 has another event of the same tag as e_2, then both events are safe-to-process, and Computation1 would consume both input events during the same firing.

Note this baseline strategy is very similar to **Strategy A** defined in [56]. The only difference is that strategies as presented in [56] ensures events are always *produced* in tag order, however this baseline strategy ensures that all events are *processed* in tag order. In Sec. 4.5, we generalize our strategies such that actors will *both* process inputs as well as produce outputs in tag order.

Finally, note the first part of the strategy (Eq.3.11) is enforced through a check of the tag of the event against the platform time subtracted by some offset, which is calculated statically. From now on we will be referring to this offset as the *delay-offset*. Also notice the inherent fault-tolerant property of Ptides, where the safety of an event execution is checked by comparing against platform time. This implies the execution of an event can never be stalled due to the absence of another event in the system. [13] showed an example that exploits this fault-tolerant property, and implemented a heart-beat detector that identifies system failures by simply observing for missed events. The safe-to-process analysis is used to identify the amount of platform time we need to wait before it is assured that fault has occurred.

Chapter 4

Ptides Execution Strategies

In this chapter a family execution strategies are presented, all of which are based on the baseline strategy as described in Chap. 3. In all cases, the first part of the strategies is the same as Part 1. of the baseline strategy, i.e., to ensure no events from outside of the platform can causally affect the event of interest, a comparison between the tag of the event of interest and platform time is performed. However, all strategies differ in the second part. By making certain assumptions about the system, and by making careful choices about the order of event executions, the second part of the baseline strategy can be simplified. We first describe an event ordering scheme used in discrete event simulation, and discuss how this ordering could simplify the strategies.

4.1 Event Ordering

In Ptides or DE, each platform maintains a pool of events. The schedulers analyze these events and determines which one to process. If these events are sorted in a particular order, the ordering could potentially simplify the scheduling analysis. The order we are interested in is a lexicographical ordering of the following tuple: $< T, I, D >$, where $T = \mathbb{R}$ represents the set of event timestamps. $I = \mathbb{N}$ represents the set of events' microsteps, and $D = \mathbb{N}$ represents the set of events' depths (explained below). For $e_1 =< \tau_1, i_1, d_1 >, e_2 =< \tau_2, i_2, d_2 >$, and $e_1, e_2 \in < T, I, D >$, the ordering is defined as:

$$\begin{aligned} e_1 < e_2 \Leftrightarrow\ & (\tau_1 < \tau_2)\ \vee \\ & (\tau_1 = \tau_2 \wedge i_1 < i_2)\ \vee \\ & (\tau_1 = \tau_2 \wedge i_1 = i_2 \wedge d_1 < d_2) \end{aligned}$$

We have already introduced timestamp and microstep as the tag of an event, but in case two events have the same timestamp and microstep, there still might be a need to define an ordering between these events. An example is shown in Fig. 4.1. In this case, even if e_1 and e_2 have the same timestamps and microsteps, e_1 still causally affects e_2. i.e., e_1 needs to be processed first, which results in an output event, call it e_1'. Computation1 can then process both e_1' and e_2 during the same firing. To capture this property, actor "depths" are assigned through a topological sort algorithm [42].

From here on, we refer to this ordering as the *tag order* or *model time order*, where a tag is expanded to the tuple of $< T, I, D >$.

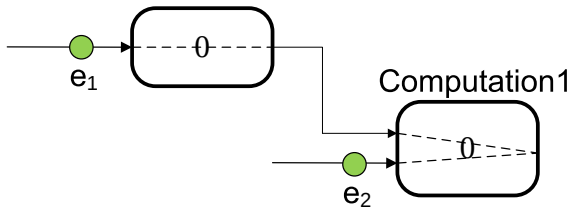

Figure 4.1: Ptides Example Showing Depth

The simple strategy, which we discuss now, assumes the platform maintains the events in tag order.

4.2 Simple Strategy

The simple strategy is based on the observation that the Part 2 (Eq. 3.12 and Eq. 3.13) of the baseline strategy's checks all events in the event queue, and ensures these events do not causally affect the event of interest. This operation is rather expensive. The complexity is of order $O(n)$, where n is the maximum number of events in the event queue at any given time. To simplify this algorithm, we assume the event queue is sorted in the order described earlier. Since there is a total order for events within the system, and since all actors are assumed to be causal (see Sec. 3.1.2), we are assured no other events in the event queue can causally affect the earliest event (event with the smallest tag). Thus this strategy is simplified to:

Simple Strategy. *For a platform f, the earliest event in the queue is safe-to-process when physical time exceeds:*

$$\tau + \max_{p \in \mathbb{I}_f, i' \in \mathbb{E}_i} \{d(p) - \delta(p, i')\}$$

where i is the destination port of this event, and τ is its timestamp.

As the formulation shows, the advantage of this strategy lies in its simplicity, where a single check against physical time is needed. Since the delay-offset can be calculated at compile time, the runtime does not need to access causality information of the model to determine whether an event is safe to process.

The disadvantage of this strategy is that it does not exploit concurrency in the model. Specifically, if the earliest event is not safe to process, the scheduler simply waits until physical time exceeds a value. During this time, the processor remains idle, even though there might be other events in the queue that are safe.

Note this strategy is similar to **Strategy C** presented in [56]. The only difference is that this strategy ensures events are *processed* in tag order, while **Strategy C** ensures events are *produced* in tag order.

4.3 Parallel Strategy

One way to avoid the unnecessary stall of event execution in the Simple Strategy is to not constrain ourselves to analyze the earliest event in the queue. However it would be nice if the Part 2 of the Baseline Strategy (Eq. 3.12 and 3.13) could be avoided as in the Simple Strategy case. Here we present an approach that achieves both goals.

Without loss of generality, we use Fig. 3.4 to demonstrate how one determines when events are safe to process. Let us assume for this case that Sensor1 is broken and we cannot receive any events from it. Then using the baseline execution strategy (Eq. 3.11 to 3.13), we can determine e_1 is safe to process if given a physical time t:

$$t > \tau(e_1) + d(n_1) - \delta_1 \tag{4.1}$$

and there is no other event in the same platform that causally affects e_1.

Following the same reasoning, e_2 is safe to process if given a physical time t:

$$t > \tau(e_2) + d(n_1) - \delta_1 - \delta_2 \tag{4.2}$$

and there is no other event in the same platform that causally affects e_2.

Here $\tau(e)$ gives the timestamp of the event.

For simplicity, let us also assume e_1, e_2 are the only events in this platform. This means we can conclude e_2 is safe to process if Eq. 4.2 is true and

$$\tau(e_2) \geq \tau(e_1) + \delta_2 \quad \text{is false;} \tag{4.3}$$

i.e., e_1 does not causally affect e_2, thus satisfying Part 2 of the baseline strategy.

Here we make a key observation:

Lemma 1. *If* e_1 *causally affects* e_2*, and the first part of the safe-to-process analysis as defined in the baseline strategy returns false for* e_1*, then it must return false for* e_2.

In other words, we want to prove that if Eq. 4.3 is true, then Eq. 4.2 implies Eq. 4.1. The proof is simple:

Lemma 1 Proof. Since Eq. 4.3 is true, we replace $\tau(e_2)$ in Eq. 4.2, and obtain

$$t > \tau(e_1) + \delta_2 - \delta_1 - \delta_2 + d(n_1)$$

which simplifies to Eq. 4.1.

Lemma 1 says if e_1 causally affects e_2, then at all points of physical time, if there are potential events from outside of the platform that can causally affect e_2, then the same events will causally affect e_1. i.e., the first part of the strategy cannot be true for e_2 unless it is true for e_1.

Armed with Lemma 1, we can scan the entire event queue, and a later event will not pass the physical time check unless all earlier events that causally affect it do so first.

Another way to look Lemma 1 is: Given the set of events in the system as the set E, let the set of events that satisfy Part 1 (Eq.3.11) of the baseline strategy be in $E_s \subseteq E$, then Lemma 1

proves that the second part of the strategy must be satisfied for the smallest event in E_s. Thus, the Parallel Strategy simply picks the smallest event in E_s to process.

Based on Lemma 1, we give the operational semantics of the Parallel Strategy. The Parallel Strategy makes two assumptions:

1. each platform has an event queue that is sorted by the order described in Sec. 4.1.
2. the earliest event in the queue that becomes safe-to-process is processed before events of larger tags. In the previous example, this means e_2 is always processed before e_1, since it resides at the front of the event queue.

Parallel Strategy.
Step 1. Let e be the earliest event in the event queue for platform f.
Step 2. Say e resides at port i, and has timestamp τ. If physical time exceeds

$$\tau + \max_{p \in \mathbb{I}_f, i' \in \mathbb{E}_i} \{d(p) - \delta(p, i')\}$$

then e is safe to process, and it is processed. Otherwise, let e be the next sorted event in the queue, and repeat step 2.

Like the Simple Strategy, the Parallel Strategy is simplified to a simple check against physical time. Moreover, Parallel Strategy exploits parallelism inherent in the model, and never wastes processor cycles as long as a safe event exists, while the Simple Strategy may block the execution of other events if the smallest one is not safe to process.

4.4 Integrating Ptides with Earliest-Deadline-First

As we indicated before, the baseline strategy does not enforce any event ordering between system events. Both the Simple and Parallel Strategies optimize the scheduling algorithm by assuming an event ordering by model time. In this section we explore other possible priority orderings.

As Fig. 4.2 shows, we add another scheduling layer in addition to the safe to process analysis as defined in the baseline strategy. Here, the safe to process analysis is referred to as the *coordination layer*. This layer takes all events in the platform as inputs. When an event is declared safe to process, that event is "released" to the *scheduling layer*, which chooses among all released events, and picks one to process based on their priorities. The next question is, what kind of scheme should be implemented in the scheduling layer?

A natural choice is the famous earliest-deadline-first (EDF) scheme, where the scheduling layer would simply pick the event of earliest deadline to process. This makes it possible for Ptides schedulers to take advantage of some of the interesting properties of the previously studied real-time scheduling scheme, such as EDF's optimality with respect to feasibility [47].

The EDF algorithm contains a notion of "release times," which is the physical time at which this event becomes available to process in the system. By integrating Ptides with EDF, we simply define the release time of an event as the physical time at which that event becomes *safe-to-process*. In addition, traditional EDF scheme assumes the programmer would statically annotate

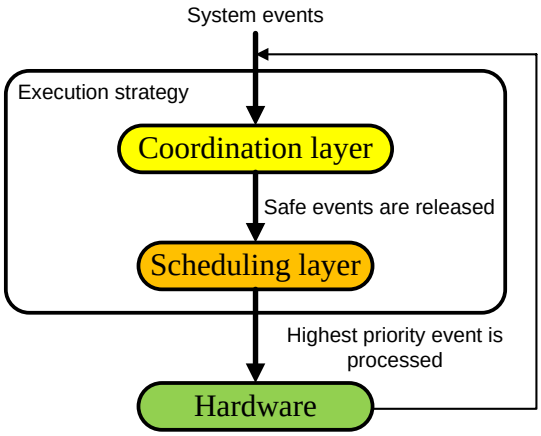

Figure 4.2: Two Layer Execution Strategy

deadlines for each system task. By integrating EDF with Ptides, the deadlines of system events are inferred from timed semantics. We define these deadlines next.

4.4.1 Ptides Execution Strategies with Model-Time-Based Deadline

The execution strategy we present in this section uses model times to calculate deadlines for system events. This carefully chosen set of deadlines allows a simple safe-to-process analysis, while enabling earliest-deadline-first processing, exploiting model parallelism, and guarantying Ptides semantics.

Deadline Calculation

The deadline calculation presented in this subsection is only dependent on model times delays, while making no use of execution times. The deadline of an event is defined by how "far away" the event of interest is from the nearest actuator.

To understand how deadlines are defined, recall that events arriving at actuator and network output ports have deadlines associated with them, where the timestamp of the event is effectively the deadline. Thus for any any event e that is δ model time away from the nearest actuator or network output, the deadline of e is $\tau(e) + \delta$, where $\tau(e)$ is the timestamp of e. Take Fig. 4.3 as an example, the deadline for e_1 is $\tau(e_1) + \delta_1$, where the closest actutor/network output port is i_1, and the delay between i_2 and i_1 is δ_1.

To aid the deadline calculation for events, we define a relative deadline for each actor input port. This relative deadline captures the minimum model time delay between the input port and its reachable actuator/network output ports. In the previous example, $rd(i_1) = 0$, $rd(i_2) = \delta_1$.

The relative deadline for each input port can be calculated with a simple shortest path algorithm starting from each input port of actuators, and traversing backwards until the entire graph is visited. Here ports are nodes, and connections between actors as well as causality relationships within actors are edges. By default, any connection between actors has a weight of zero, while the

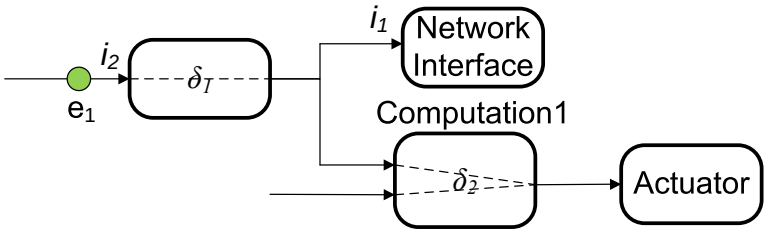

Figure 4.3: Deadline Calculation Through Model Time

model time delays between input and output actor ports are weights on those edges. The shortest path algorithm annotates relative deadlines for each node that is an input port.

With the definition of relative deadlines, the absolute deadline of an event can be written as:

$$AD(e) = \tau(e) + rd(port(e))$$

Here, $AD(e)$ is the absolute deadline of an event e, $\tau(e)$ is the tag of e, $port(e)$ is the destination input port of e, and $rd(port(e))$ is the relative deadline at $port(e)$. Next, we define a new event ordering based on the absolute deadlines.

4.4.2 Event Ordering with Deadlines

The new event ordering is based on the one presented in Sec. 4.1, yet adds one more element into the event tuple. The order we are interested in is a lexicographical ordering of the following tuple: $< AD, T, I, D >$, where again T, I, D are the sets of timestamps, microsteps, and depths. $AD = \mathbb{R}$, represents the set of absolute deadlines. For $e_1 =< ad_1, \tau_1, i_1, d_1 >, e_2 =< ad_2, \tau_2, i_2, d_2 >$, and $e_1, e_2 \in < AD, T, I, D >$, the ordering is defined as:

$$\begin{aligned} e_1 \ < \ e_2 \ \Leftrightarrow \ & (ad_1 < ad_2) \ \vee \\ & (ad_1 = ad_2 \wedge \tau_1 < \tau_2) \ \vee \\ & (ad_1 = ad_2 \wedge \tau_1 = \tau_2 \wedge i_1 < i_2) \ \vee \\ & (ad_1 = ad_2 \wedge \tau_1 = \tau_2 \wedge i_1 = i_2 \wedge d_1 < d_2). \end{aligned}$$

This event ordering is used in our later strategies.

Strategy using Model Time Deadlines

Earlier in this section we introduced a two-layer scheduling strategy. If applied to this case, the baseline strategy can be used as the coordination layer and the event ordering with deadlines can be used to select the earliest deadline event among all those that are safe. However this approach is relatively expensive. Our goal in this section is to simplify this algorithm.

Recall from Sec. 4.3, we proved that if both e_2 and e_1 reside in the same platform, and if e_2 causally affects e_1, e_1 will not pass the physical time check for safe-to-process at an earlier platform time than e_2. Moreover, e_2 is placed at an earlier position than e_1 in an event queue sorted

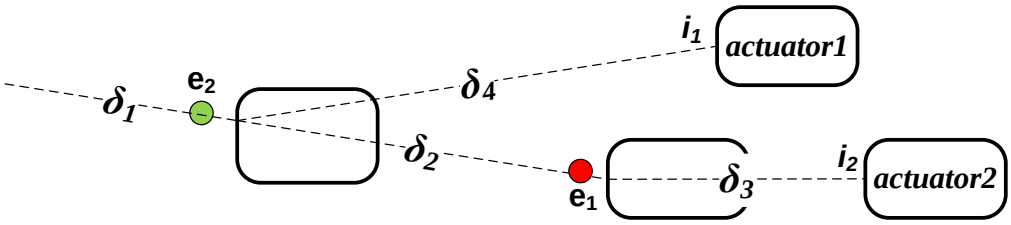

Figure 4.4: Deadline Example

by tag order. In this section, our goal is to prove that if e_2 causally affects e_1, then e_2 will be placed at an earlier position than e_1 in an event queue sorted by the absolute deadline order. Once proven, then maintaining a sorted deadline queue would ensure e_2 is always processed before e_1. We provide the proof below.

Lemma 2. *If e_2 causally affects e_1, then e_2 will be placed at an earlier position than e_1 in an event queue sorted by absolute deadlines, as defined in Sec. 4.4.2.*

Lemma 2. Proof.

By definition, if e_2 causally affects e_1, then the following equation is true:

$$\tau(e_2) \leq \tau(e_1) - \delta(port(e_2), port(e_1)) \tag{4.4}$$

where $port(e)$ is the destination port of event e, and thus $\delta(port(e_2), port(e_1))$ denotes the minimum model time delay between destination ports of e_2 and e_1. Notice this equation can be rewritten as:

$$\tau(e_2) + \delta(port(e_2), port(e_1)) - \tau(e_1) \leq 0 \tag{4.5}$$

The definition of relative deadline and deadline are defined in Sec. 4.4.1, applying these definitions on e_1 and e_2, we have:

$$D(e_1) = \tau(e_1) + \delta(e_1) \tag{4.6}$$

and

$$D(e_2) = \tau(e_2) + \delta(e_2) \tag{4.7}$$

where we let $\delta(e)$ denote the minimum model time delay from $port(e)$ to the nearest actuator.

Take the example shown in Fig. 4.4, $\delta(e_1) = \delta_3$, while $\delta(e_2) = min\{\delta_2 + \delta_3, \delta_4\}$. There are two possible cases: 1, e_2's path to the nearest actuator does not pass through the destination port of e_1; and 2, e_2's path to the nearest actuator does pass through the destination port of e_1. Note the first case implies:

$$\delta(e_2) \leq \delta(port(e_2), port(e_1)) + \delta(e_1), \tag{4.8}$$

while the second case implies:

$$\delta(e_2) = \delta(port(e_2), port(e_1)) + \delta(e_1), \tag{4.9}$$

Again in the example shown in Fig. 4.4, first case implies $\delta(e_2) = \delta_4 \leq \delta_2 + \delta_3$, and the second case implies $\delta(e_2) = \delta_2 + \delta_3 \leq \delta_4$. We observe in both cases Eq. 4.8 must hold. Now substitution $\delta(e_2)$ from Eq. 4.8 into Eq. 4.7, we obtain:

$$D(e_2) \leq \tau(e_2) + \delta(port(e_2), port(e_1)) + \delta(e_1). \tag{4.10}$$

Again by substitute $\delta(e_1)$ from 4.6 into Eq. 4.10, we obtain:

$$D(e_2) \leq \tau(e_2) + \delta(port(e_2), port(e_1)) + D(e_1) - \tau(e_1), \tag{4.11}$$

or,

$$D(e_2) - D(e_1) \leq \tau(e_2) + \delta(port(e_2), port(e_1)) - \tau(e_1), \tag{4.12}$$

Notice the right side of Eq. 4.12 is less than or equal to 0 given Eq. 4.5.

Thus we have two cases. $D(e_2) < D(e_1)$, and $D(e_2) = D(e_1)$. In the first case, obviously e_2 is placed at an earlier position than e_1 in the event queue. If $D(e_2) = D(e_1)$ is true, then Sec. 4.4.2 specifies that we need to compare timestamps, microsteps, and the depths of the events in lexicographical order. Since we assumed actors are causal, e_2 cannot have larger timestamp or microstep than e_1. In the case where $\delta(port(e_2), port(e_1)) = 0$, e_2 will still have smaller depth than e_1. Thus we can conclude e_2 will be placed earlier in the event queue than e_1.

With Lemma 2, both the simple and parallel strategies can be used with a queue that is sorted by deadline order. The same assumptions of the Parallel Strategy also applies to this case, where a single processing unit is used, and an event residing at the front of the event queue must be processed before events with larger deadlines.

Notice that, in essence, this strategy takes the two-level scheduling approach as mentioned in Sec. 4.4, but it is more efficient since only events that are of the higher priorities are analyzed, and the first safe event is processed. This avoids analyzing all events in the event queue every time the scheduler executes.

Also note, our proof only works for Ptides models where events are "trigger events," meaning events with destination ports. In later sections, "pure events" will be introduced, and we will show deadline sorted queues will only work with certain types of actors that produce pure events. We will define the actor types and limitations with a deadline sort queue next.

4.5 Generalization of Ptides Strategies for Pure Events

Strategies in the previous sections only dealt with "trigger events" in the system, which are ones whose values are sent to destination input ports when processed. However, actors in the Ptolemy II environment can also produce so called *pure events*. The motivation for pure events is detailed in [34]. To summarize, in a hierarchical and heterogeneous modeling environment like Ptolemy II , an actor (which may be governed by a different kind of director inside) should be able to request firing at a future model time. The way it does so is through a Director method called fireAt, which produces a *pure event* at the outside director. These pure events do not have destination ports, since they are simply requests to fire the actor at future times.

Moreover, pure events allow actors to not only process, but also produce events in tag order. The motivation is illustrated in Fig. 4.5a. The actor shown in figure consumes input events at port p_i while produce output events at port o. The minimum model time delay between the two ports is δ_1. This means the actual model time delay of the actor must be greater than or equal to δ_1.

Given the Simple and Parallel strategies introduced earlier, the following event trace is possible:

1. actor consumes event e_1 at p_i with timestamp τ_1,

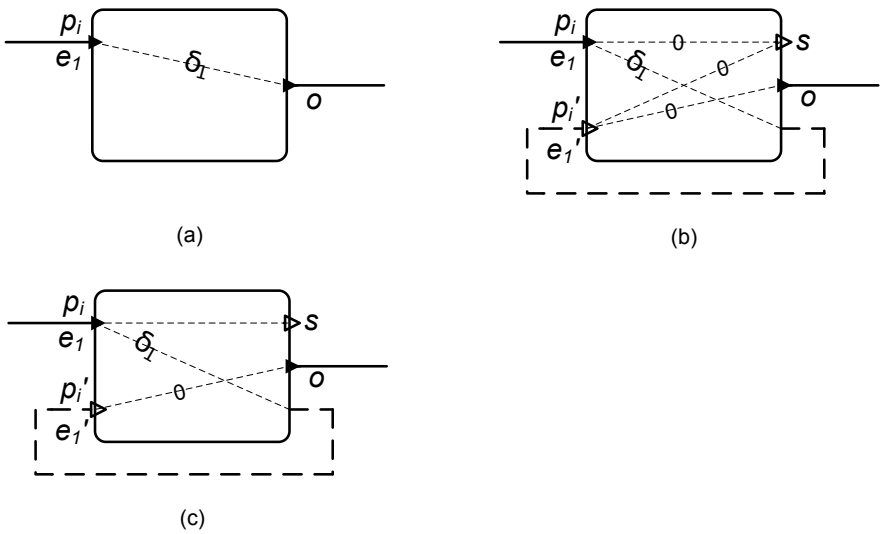

Figure 4.5: Actor with Causality Interface Decomposed

2. actor produces an output event of timestamp $\tau_1 + \delta_2$, where $\delta_2 \geq \delta_1$.

3. actor consumes event e_2 arriving at p_i with timestamp τ_2,

4. actor produces an output event of timestamp $\tau_2 + \delta_1$.

Let us assume for this case $\tau_1 + \delta_2 > \tau_2 + \delta_1$, i.e., the actor produces events out of timestamp order. This identifies a problem with our earlier strategies: they assume all actors in the model only access states during the processing of trigger events, but not the production of output events. However, what if actor accesses its states both during the consumption and production of events?

To be more concrete, let us suppose the actor in Fig. 4.5a is a special accumulator, where its accumulated value is saved in a private variable. This variable can be thought of as the state of the actor. Let us then suppose the accumulated value is incremented every time the actor consumes input and produces an output. In particular, when this actor is ready to produce an output, it first increments the accumulated value before producing an event with that value. Thus the above event trace would result in the output event of timestamp $\tau_1 + \delta_2$ taking value 2, as shown in Fig. 4.6. i.e., consumption of e_1 increments accumulated value to 1, production of event with timestamp $\tau_1 + \delta_2$ increments it to 2, and 2 is produced.

However the correct event trace would be: 1. 3. 4. 2, where the output event should not be produced until an input of smaller timestamp than the output is consumed. This would result a value 3 produced for the output event with timestamp $\tau_1 + \delta_2$. This behavior is achieved through the use of *pure events*. After actor consumes input e_1, instead of immediately producing an output event, it calls the *fireAt* of the director, and produces a pure event e_1' of timestamp $\tau_1 + \delta_2$. This signals to the Ptides director that this actor wishes to fire again at model time $\tau_1 + \delta_2$. The next question is: How do we know when pure event e_1' is safe to process?

To answer this question, the actor in the previous example is decomposed using its causality interface, as shown in Fig. 4.5b. We can imagine for each trigger input p_i, there exists a virtual

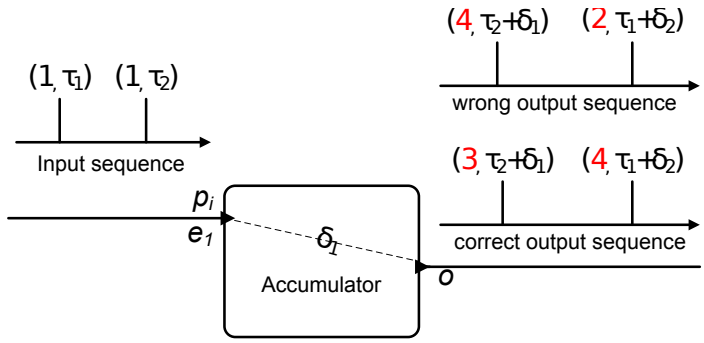

Figure 4.6: Event Trace for Accumulator Actor

input port p'_i. Each time a pure event is produced as a consequence of consuming a trigger event at p_i, that pure event is delivered to p'_i. The delay between p_i and p'_i is δ_1. When the pure event is processed, an output event is produced at o. Since the state of the actor (abstracted as an output port s) is updated both when input is consumed and when output is produced, there are delays of 0 between pairs (p_i, s) and (p'_i, s). This causality also implies p_i and p'_i reside in the same input port group. Recall from Chap. 3.3.2 that the safe-to-process analysis of Ptides calculates a delay-offset parameter for all input ports. Let the offset for p_i be Ω_i, it is easy to see the offset for p'_i is $max\{\Omega_i, \Omega_i - \delta_1\} = \Omega_i$. i.e., the offsets for virtual ports simply take the same values as the trigger input ports. Given this formulation, both the Simple and the Parallel strategies can be applied to models that include actors with pure events.

4.5.1 Causality Marker

To guarantee DE semantics behavior for pure events, the Ptolemy II Ptides simulator uses a public class called CausalityMarker. The scheduler uses this class to determine the causality relationship between trigger and virtual ports. By default, trigger port p_i and virtual port p'_i are placed in the same input port group. Though this assumption ensures the correct order of event processing for the example shown in Fig. 4.5b, this can be overly conservative in certain cases, and more eager scheduling can be achieved. Fig. 4.5c shows an example. If actor in Fig. 4.5a does not update its state while producing output events, then that actor can be decomposed to Fig. 4.5c. In this case, it's clear that ports p_i and p'_i do not reside in the same input port group.

The VariableDelay illustrated in Fig. 4.7 is an example of an actor that produces pure events. Events received at the delay port updates a delay parameter in the actor. Whenever an event is received at its input port, this actor's tag is delayed (incremented) by the previously saved value in the delay parameter. Notice however, unlike the previous Accumulator example, this actor does not have to produce events in tag order. Take the event sequence for example: $[e_1, e_3, e_2, e_4]$. Each event $e_i =< v_i, \tau_i >$, where v_i and τ_i are value and tag of the event, respectively. Here, delay is the destination port for e_1, e_2, and input is the destination port for e_3, e_4. Event e_3 is delayed by the value v_1, while e_4 is delayed by v_2. When e_3 is processed, a pure event with tag $\tau_3 + v_1$ is first produced. Given the previous conservative strategy, this pure event is not processed until neither input will receive an event of smaller tag than $\tau_i + v_1$ at a later platform time. However, since the

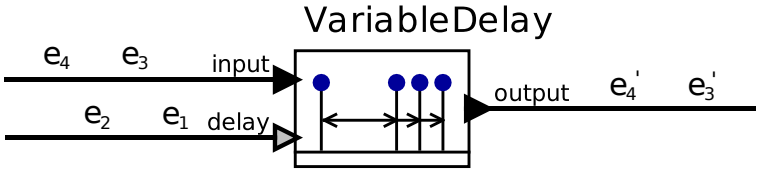

Figure 4.7: Variable Delay Actor With Event Sequences

only functionality of the variable delay actor is to delay the input's timestamp, whether the pure event is processed in the same order as the other input events does not affect the state of the actor nor output events. i.e., it does not affect the determinacy of the system. Thus the pure events can be immediately processed.

To achieve the "eager" processing of actors such as VariableDelay, the causality marker API allows actor designers to specify that certain virtual ports do not belong to the same input port group as the trigger input. In those cases the pure event would be immediately safe to process once it becomes the highest priority event in the system.

4.5.2 Limitations with Models with Pure Events

A consequence of introducing pure events in the system is that the deadline strategy introduced in Sec. 4.4.2 cannot work with actors whose states are updated when pure events are processed. We prove this by giving a counterexample, again using Fig. 4.5b. If e_1 has timestamp τ_1, and let's say e_1', the pure event as a result of process e_1 takes timestamp $\tau_1 + \delta_1$. If the absolute deadline, as defined in Sec. 4.4.1 for e_1 is e_{1ad}, then the deadline for e_1' must take the same value e_{1ad}. This is because while e_1''s timestamp is δ_1 larger than e_1, the shortest delay distance from p_i' to actuators is δ_1 smaller than from p_i to actuators. Thus the deadlines is the same for e_1 and e_1'. Now if an event e_2 arrives at p_i with timestamp $\tau_1 + \epsilon$, where $\epsilon < \delta_1$, then e_2 should be processed ahead of e_1'. i.e., e_2 must be placed in front of e_1 in order to ensure the correct order of event processing. This is indeed the case if the event queue is sorted by tags. However e_2 will be placed *behind* e_1' if the queue is sorted by deadlines, since e_2 will have an absolute deadline of value $e_{1ad} + \epsilon$. This counterexample showed that a deadline sorted queue can no longer ensure event processing in timestamp order.

To summarize, by allowing actors that access states at a later model time than the tag of the trigger event, the deadline ordered queue can no longer ensure the current order of event processing. Note the parallel strategy, which maintains the event queue in tag order, still works with actors that produce pure events.

4.6 Execution Strategies with Additional Assumptions

4.6.1 Assumptions

Other than the execution strategies described above, one could make the following assumption in order to simplify the strategies: *actors receive events in tag order.* This assumption is based on three different assumptions:

1. actors produce events in tag order.
2. sensors produce events in tag order.
3. platforms receive network data in tag order.

The first assumption was already addressed in the previous subsection. By simply using the default causality marker annotations, one could easily ensure events are always produced in tag order,

At first glance, the second assumption is guaranteed to be true. As we described before, sensors tag events at the physical time the events are received, and produce events at a physical time that is less than the tag of the event plus the real-time delay. However notice these specifications of the sensors do not guarantee events are produced in tag order. One could implement a sensor that satisfies the requirements yet still produce events out of tag order, as long as the worst-case-response-time for sensors are bounded by d_s. In this section we do not consider sensors with this particular property.

The third assumption is more problematic, and is not be true for many network implementations. Even if network transmissions are done in tag order, network switches could potentially re-order the transmitted messages. To make the problem more complex, in distributed systems it is common to have multiple transmitters connected to multiple receivers across the same network. In this case, events may be delivered in tag order by each transmitter, but if they have the same destination receiver, there is no guarantee that events are delivered in tag order to the receiver. Though difficult, it is not impossible to solve this problem using the current technology. Along with the previous two assumptions, if an order-preserving network protocol such as Transmission Control Protocol (TCP) is used, and we enforce a one-to-one transmitter-receiver pairs, then the network can be guaranteed to deliver events in tag order.

Assuming all three assumptions are true, then actors can be guaranteed to receive events in tag order.

4.6.2 Strategy Assuming Events Arrive in Tag Order

By making the above assumption, we can introduce yet another execution strategy. This strategy is called **Strategy B** in [56]. Instead of presenting the formal definition, we will only give the intuition behind this strategy through an example.

Like all previous strategies, this strategy is also divided into two parts to deal with potential events arriving from outside and inside of the platform separately. Again, a check against physical time is performed for events from outside of the platform. By assuming events are received in tag order, this physical time check can be simplified. Take Fig. 4.8 for example. Any trigger event

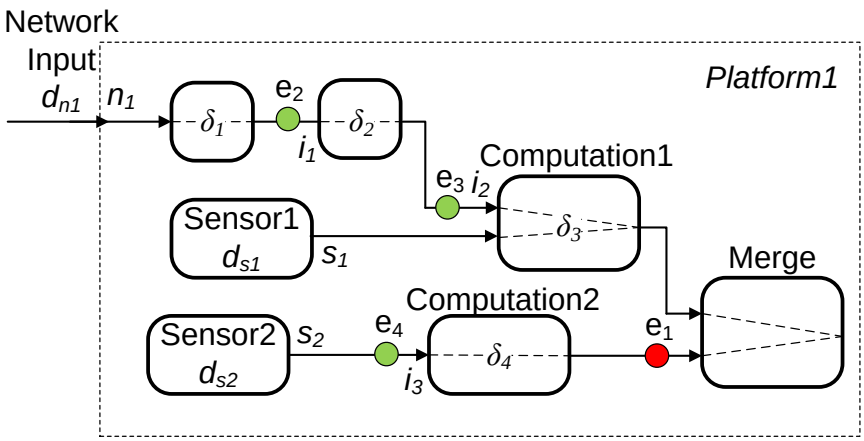

Figure 4.8: Example Illustrating Strategy with Events Arrive in Tag Order

arriving at the input port of Computation2 can be immediately processed. This is because Computation2 only has a single input port i_3, which receives events from Sensor2. When trigger event e_4 arrives at i_3, it already implies that the upstream actor will not produce any event of smaller timestamp, and thus e_4 can be immediately processed. A delay offset of $POSITIVE\_INFINITY$ is used to indicate that the safe-to-process check will always return true for events arriving at i_3.

The second part of this strategy then relies on the causality relationships between the event of interest and a subset of all other system events to determine whether the event of interest is safe. We again take Fig. 4.8 for example. There are a total of three paths from sensor/network inputs to the inputs of the Merge actor: from Network Input, Sensor1, and Sensor2. Let e_1 be the event of interest; we need to ensure events from *each* of these paths do not causally affect e_1. In this example, the only causality calculation we need to perform is for e_3 (but not for e_4 and e_2). Since e_4 comes from the same source as e_1, per our assumption, we are assured Computation2 will only produce in future events with larger tag than e_1. For the same reason, there is no need to check the causality relationship between e_2 and e_1, since if e_2 causally affects e_1, e_3 must also causally affect e_1. In other words, this strategy allows us to check for events that are "closest" to the event of interest for each path that ends at the destination input port group. In the case where events do not exist on that path, the physical time check is performed. In this example the path from Sensor1 to Merge is such a path, thus we need to make sure physical time has exceeded $\tau(e_1) + d_{o1} - \delta_3$.

Notice, the unique advantage of this strategy is that like the baseline strategy, it does not require any particular sorting of the event queue. However, there are also some disadvantages:

1. The assumptions listed in Sec. 4.6 are relatively difficult to achieve, especially since all three assumptions need to be satisfied all at the same time.

2. Unlike previous strategies, where the scheduler only performs a comparison of an event's timestamp with the current platform time, this strategy uses causality relationships between events residing in the same platform. Such a scheme requires additional information about the model structure at runtime, and incurs additional scheduling overhead. This is in generally not desirable in the scheduling of real-time systems.

4.7 Summary of Ptides Execution Strategies

A total of seven Ptides execution strategies have been introduced, not counting the baseline execution strategy. The following criteria highlight the differences between these strategies:

1. The event queue can be either sorted by tags (see Sec. 4.1), by model time based deadlines (see Sec. 4.4.1), or not sorted.

2. The safe-to-process analysis is either performed only on the smallest event in the queue, or on all events in the queue (see Sec. 4.3).

3. To check whether an event is safe-to-process, in some cases a simple check against physical time is performed; while in others, model time comparisons are needed as well. (see Sec. 4.6.2).

4. Actor states can be accessed while only processing trigger events, or they can be accessed while processing both trigger and pure events. Notice here access includes both reads and writes to state variables. Depending on the kind of actors that are present in a Ptides model, different event orderings can be used (see Sec. 4.5).

The table below summarizes the different Ptides strategies as presented in earlier sections. Notice δ is the model time delay of the actor under consideration.

Among all these strategies, the most interesting strategies are: Parallel Strategy with Pure Events and Parallel Deadline Strategy. Obviously Parallel Strategy is more concurrent than Simple Strategy, while Parallel Strategy with Pure Events works with a larger set of actors than the rudimentary Parallel Strategy. The Parallel Deadline Strategy does not work with actors that have a non-zero model time delay or ones that access states while producing output events. However, it is interesting due to its integration with EDF, which is a well-studied scheduling algorithm. Finally, Strategy Assuming Events Arrive in Tag Order is relatively impractical for real-time systems due to its relatively large scheduling overhead.

Table 4.1: Summary of Ptides Execution Strategies

Name	Event(s) Analyzed For Safe-To-Process	Safe-To-Process Check	Actor Assumptions
Simple Strategy	smallest event in queue sorted by tags	physical time check	$\delta = 0$, or actor states are accessed only when trigger input events are processed
Parallel Strategy	all events in queue sorted by tags	physical time check	$\delta = 0$, or actor states are accessed only when trigger input events are processed
Simple Deadline Strategy	smallest event in queue sorted by deadlines	physical time check	$\delta = 0$, or actor states are accessed only when trigger events is processed
Parallel Deadline Strategy	all events in queue sorted by tags	physical time check	$\delta = 0$, or actor states are accessed only when trigger events is processed
Simple Strategy with Pure Events	smallest event in queue sorted by tags	physical time check	Actor states are accessed when processing both trigger and pure events
Parallel Strategy with Pure Events	all events in queue sorted by tags	physical time check	Actor states are accessed when processing both trigger and pure events
Strategy Assuming Events Arrive in Tag Order	all events in queue, not sorted	physical time and model time comparisons	Actor states are accessed when processing both trigger and pure events

Chapter 5

Ptides Design Flow

In the earlier chapters we have described the Ptides execution strategies. This chapter talks about software implementations of these strategies, and how they are used implement real-world applications.

The goal of the Ptides framework is to provide distributed real-time application programmers proper abstractions and tools to design their system. To achieve this goal, a complete design flow built in the Ptolemy II framework [6] is introduced. The design flow includes a mixed simulation environment for the design of Ptides models. Once satisfied with the design, the programmer can leverage the code generation infrastructure in Ptolemy II to generate a C implementation running on top of a lightweight real-time operating system we call PtidyOS. The scheduler in PtidyOS as well as the Ptides simulator implements Ptides strategies mentioned in the last chapter. Fig. 1.2 shows the complete design flow of a distributed real-time embedded system using the Ptides programming model. We first explore the Ptides simulator built in Ptolemy II.

5.1 Ptides Simulator in Ptolemy II

Ptolemy II [6] is a framework to study models of computation. Like Simulink [11] or LabVIEW [26], Ptolemy II uses an actor-oriented [31] programming methodology. It allows programmers to implement applications as models, and the functionality of the system can be simulated. Due to Ptides' emphasis on real-time systems, this functionality includes not only logical operations, but also the passage of physical time; i.e., the programmer can simulate real-time behaviors such as event preemption, deadline misses, etc. The Ptides simulator is implemented as one of the *domains* in Ptolemy II.

Also, the Ptides simulator takes advantage of Ptolemy II's support for heterogeneous domain simulation. There is a large number of simulation domains available, most notably Discrete-Event(DE), Continuous-Time, etc., each with its own specialties. This allows programmers to choose the most intuitive domain to model different parts of the system. But first, we dive into the details of Ptides simulation environment.

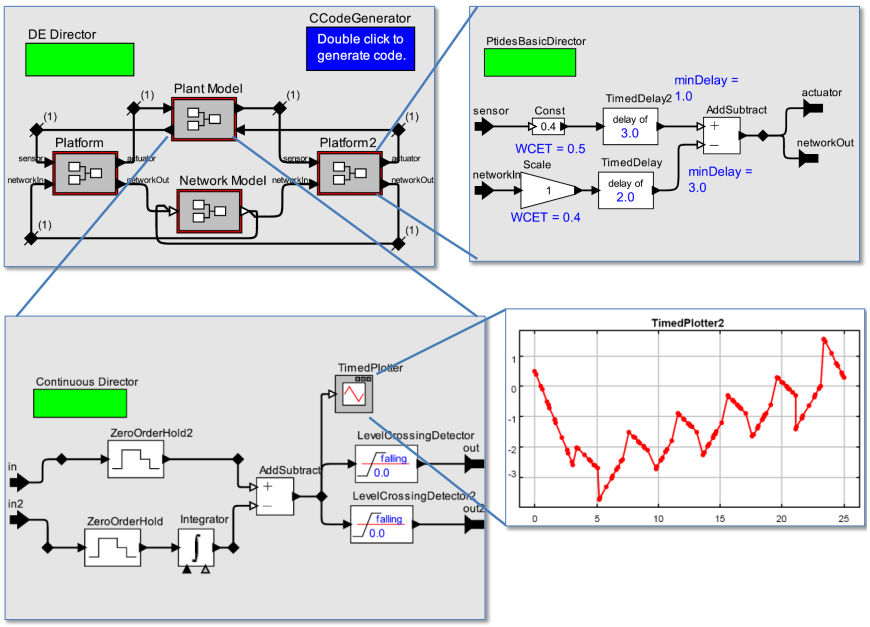

Figure 5.1: Ptides Design Flow Example

5.2 Background

An example model in Ptolemy II is shown in Fig. 5.1, Ptides models mainly include the following components: a Director, Actors (composite and atomic), Ports, and connections.

An *actor* can be thought of as a piece of code. When it fires (executes), it simply consumes inputs, produce outputs, and update its states. Actors can be *atomic*, or many actors can grouped to create a *composite actor*. A composite actor can be thought of as a sub-model.

Actors can have input and output *ports*. Actors consume data tokens sent to its input ports, and produce data tokens to the output ports. The tokens are then sent through *connections* to downstream input ports.

Data transmission between ports, as well as the order in which actors are fired is dictated by the *director*. In other words, the director is the system scheduler.

An example of a Ptides model built in Ptolemy II is shown in Fig. 5.1. This model is hierarchical. The top level has a DE director, which simulates the physical world. In particular, it simulates oracle physical time. We will elaborate on its functionality later. The composite actors Platform and Platform2 each simulates a computation platform (e.g., a micro-controller). These platforms are governed by a Ptides director. These platforms implement some kind of control algorithm, and they communicate across a network. This network usually has some kind of network delay associated with it, and the property of the delay is captured in the network model, which is in another composite actor. Finally, the control platforms communicate with the physical plant, which is yet another composite actor governed by a Continuous-Time director. This example illustrates the mixed simulation made possible by the heterogeneous modeling environment of Ptolemy II. First,

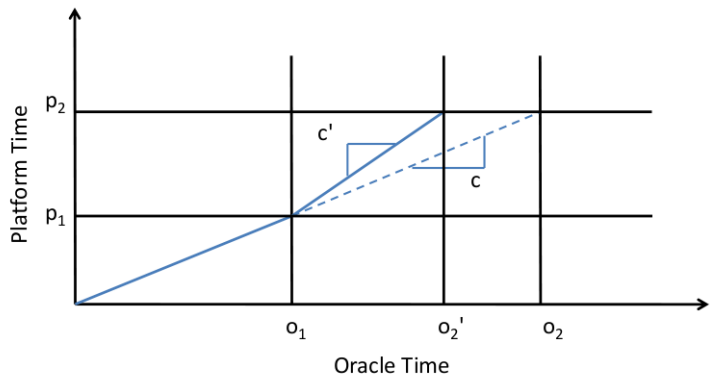

Figure 5.2: Clock Drift and the Relationship Between Oracle Time and Platform Time

we discuss features of the Ptides director.

5.3 Models of Time

In Sec. 3.1.2, we introduced model time, oracle time, and platform time. These times are simulated in the Ptides simulator.

All directors in Ptolemy II have a notion of model time, which aids system scheduling. In addition, both Ptides and DE's director use the superdense model of time. To take advantage of this similarity, the Ptides director subclasses the DE director.

To simulate physical time, we enforce the following condition: a composite actor with a Ptides director must be enclosed by a DE director, and the model time of the enclosing DE director is used to simulate oracle time. DE is a natural choice because model time has a total order in DE.

Finally, with oracle time as the reference, each Ptides director then keeps track of its platform time as a function of its clock drift with respect to oracle time. Specifically, each Ptides director includes a RealTimeClock class. The RealTimeClock has three parameters: lastOracleTime, lastPlatformTime, and clockDrift.

Fig. 5.2 plots platform time (y-axis) vs. oracle time (x-axis) for one particular example. Platform time may drift at a different rate from oracle time, and that rate is captured in the clockDrift parameter. The pair of lastOracleTime and lastPlatformTime indicate a point where these two times are "synchronous", i.e., these two times form a coordination that lies on the time drift line plotted on Fig. 5.2. (o_1, p_1) and (o_2', p_2) are such pairs. Now say at (o_1, p_1), the slave platform receives a time synchronization packet from the master, and updates the speed of its platform clock, then clockDrift is updated from c to the new clock drift c', and (o_1, p_1) are saved in lastOracleTime and lastPlatformTime, respectively. With o_1, p_1, and c', given a future oracle time, the corresponding future platform time can be calculated, and vice versa. The Ptides director offers the following set of protected API's to convert between platform time and oracle time: platformToOracleTime(), oracleTimeToPlatformTime(), getOracleTime(). These methods are accessible by Ptides directors for simulation purposes only, but are not accessible by other classes. The only public API

that deals with physical time is the getPlatformTime() method. This is because code running on the platform should only have access to the local platform clock, but not the oracle clock in the distributed environment.

We have talked about how oracle time and platform time are related. Next we will discuss the relationship between platform time and model time.

5.4 Special Actors and Ports in Ptides Simulator

Sec. 3.1.2 mentioned that model times and platform times are related at sensors, actuators and network input/output devices. Now we examine how these devices are modeled in the Ptides environment.

As Fig. 5.1 shows, a controller modeled in Ptides communicates with the physical plant (implemented as a Continuous-Time model) through sensors and actuators, which communicate with each other through network devices. Thus the input and output ports of the composite actor governed by a Ptides director can only be one of the above types. By default, input ports of composite actors are assumed to be sensors while output ports of composite actors are actuators, however these ports can be annotated to become network ports.

When a composite actor fires (executes), three steps are performed: transferInputs(), fire(), and transferOutputs(). Sensor/network input events are transferred into the composite actor at transferInputs(), while actuator/network output events are transferred out at transferOutputs().

In Sec. 3.1.3, we've described real-time delays d_s at sensors. A parameter called realTimeDelay is annotated on input ports to capture this delay. When transferInputs() is called, sensor inputs are transferred into the platform. The transferred value tokens are timestamped with the local platform time, managed by the Ptides director. However, notice that the platform should not make this event visible to the rest of the platform until platform time $\tau + d_s$ at the latest, where τ is the timestamp of the sensor event. Even though d_s is the maximum bound on the sensor delay, the Ptides simulator models the worst case, and instead of immediately inserting the input event into the event queue, that event is stored in a separate queue, and the Ptides director indicates to the enclosing DE director (by calling fireAt()) that it wants to wake up and fire again at platform time $\tau + d_s$. The sensor event is inserted into the queue at that time. Again recall DE director only keeps track of the oracle time; thus the API platformToOracleTime() of the Ptides director is used to convert the platform time to oracle time first before calling fireAt(). Recall from Sec. 4.5 that fireAt() takes a Time input, which is the requested time at which the enclosing director should fire it.

On the opposite side, before an event is transmitted out of the actuator output port, its timestamp subtracted by d_a (recall from Eq. 3.6: $\tau - d_a$ is the deadline of the actuation event) is compared to the platform time, and an exception is thrown if the deadline is missed. If deadline is not missed, the output event is also put into a separate event queue. Then platformToOracleTime() and fireAt() is called to ensure the actuator event is output to the outside at platform time equal to the event's timestamp.

Input and output ports that model network devices however, do not have the *realTimeDelay* parameter. Instead, recall the relationship $t \leq \tau + d_n + \epsilon$. Thus, network components are annotated with parameter *networkDelay* to model d_n instead. The clock synchronization error bound ϵ on the other hand, is not annotated as a parameter of the port. This is because we assume a single

platform clock on each platform, and thus it does not make sense for different network interfaces to have different ϵ values. Instead, all Ptides directors have a shared parameter call *assumedPlatform-SynchronizationErrorBound*. By implementing it as a shared parameter, once a change is made to this parameter at one of the director, this change would propagate through all Ptides directors. This parameter, together with *networkDelay*, is used in the safe-to-process analysis.

Also recall, unlike sensors and actuators, which only communicate with the outside by sending data tokens, network devices transmit not only the data, but also the timestamp associated with that data. However, while the timestamp of an event in the Ptides director is the model time, the timestamp of the enclosing DE director simulates oracle time. Thus, the timestamp of a network packet must be bundled together with the data before it is transferred to the outside. This action is performed through a NewtworkTransmitter actor. When this packet is received at the sink platform, a NetworkReceiver actor must be present to unpack this packet, and creates an event in the sink platform of timestamp equal to the value stored in the packet.

5.4.1 Ptides Execution Time Simulation

The key idea behind the timed semantics of Ptides is to capture system timing properties. However, even though Ptides semantics captures system timing requirements, there is no guarantee that the generated code running on the target platform will be able to meet these requirements. To tackle this issue, the Ptides simulator supports simulation of physical time.

Approach for Worst-Case-Response-Time Analysis

As Fig. 1.2 shows, once the system is implemented, its real-time properties needs to be tested and confirmed. In other words, a schedulability test needs to be performed. There are generally two approaches to performing such a test: static analysis and dynamic testing. Static analysis generally uses the following parameters to calculate the worst-case-response-time bound (WCRT) for a system:

1. The worst case interval time (WCIT) of system input events.
2. The worst case execution time (WCET) of the computation blocks.
3. The scheduling scheme used.

Results from such an analysis do not require the actual target platform, provided a software tool such as those described in [18] is used to find the WCET of the component blocks. Moreover, since the software tools produce absolute worst-case bounds, the WCRT bound from the static analysis are also guaranteed to be correct. Some disadvantages of the this approach are: the resulting WCRT bound might be too pessimistic at times [43]; Also most analyses today do not take the scheduling overhead into account. On the other hand, dynamic testing methods (which are commonly used in industry) simply take a full implementation, run the system in many different conditions, and check whether all system timing properties are met. This approach requires the full implementation, including both software and hardware, and a good (usually hardware-in-the-loop) testing infrastructure. However, the bounds from dynamic testing usually do not give the actual worst-case bounds, since it's very difficult to find the exact execution trace that results in the worst case.

The Ptides simulator provides a WCRT analysis that combines both approaches. Like static analysis, it uses the same three parameters as input, and does a simulation to test whether deadlines are met. In addition, the scheduling overhead can be simulated, provided the WCET of the scheduler is given. This solves a common problem of static analyses, and provides a WCRT bound that takes into account the scheduling overhead. Also, the Ptides simulator allows the programmer to check for deadline misses without requiring the actual hardware. Like static analysis, if a software WCET analysis tool exists, then the WCRT can be found without the physical hardware platform.

One disadvantage of the Ptides simulator is that even though the WCET of each of the computation blocks and the scheduler can be obtained individually, the combined WCRT may be larger than the each of the elements combined. For instance, whenever an interrupt occurs, the cache behavior of the executing events changes, which may result in longer WCRT than the case where no preemption occurs. Currently this is an open problem that is shared by all event-triggered scheduling schemes, and we do not present a solution here.

Execution Clock

To achieve the aforementioned WCRT simulation, we must first support the simulation of execution time. But first, we must clarify the notion of time we should use to simulate execution time. It's obviously not model time, since execution time is a physical time property. Platform time does not seem like the right choice, because the clock of a platform could reside in a separate hardware processing unit (e.g. a PHY chip), and the rate of this clock is updated as the clock synchronization protocols performs their duty. However, the speed of the CPU clock is in most cases not affected by the change in synchronization error. Finally oracle time does not seem like the right answer either, because the same code running on different platforms may result in different execution times.

Thus we introduce a new clock just for the simulation of execution time. We assume *WCET* parameters can be annotated on actors, and their meaning is the follows: If a parameter is annotated with WCET of value x, then x is the WCET of a piece of code running on a platform with a CPU clock that that runs at the rate of the oracle clock. Then, we define an executionTime-Clock, which is an instance of the RealTimeClock class. Recall the platformTimeClock is also an instance of the same type. Thus the executionTimeClock also has a clock drift component that allows the clock to run at a different rate from the oracle clock. Also, like the platform clock, the clock drift cannot take a value less than or equal to zero; i.e., execution time cannot decrement or stay static as oracle time increases. Also like the platform clock, the same set of API methods allow the conversion between execution clock time and oracle time.

The main reason to model the above behavior is that if the same piece of code runs on different instances of the same hardware platforms, the CPU clocks might be running at slightly different rates. This would result in different execution times.

With this new clock, the execution time of an actor can be simulated. When an event is determined safe to process, the *executionTime* of the destination actor is retrieved. If this parameter is non-zero, then the event is not processed immediately. Instead, platformToOracleTime() and fireAt() are called to ensure that the Ptides director is fired again when the execution time finishes. At that point the event is processed.

Event Preemption

Also notice, with execution time simulation, it is now possible to simulate event preemptions. When a new event arrives at the platform or becomes safe to process, we could either wait until the current processing event to finish, or preempt it and start processing that new event. In DE simulation, the first approach is taken, due to the fact that in a simulation environment, there is no incentive for the simulator to stop its current executing event and preempt it with another one. However since Ptides focuses on real-time systems, event priorities will affect whether deadlines are met at actuators. Allowing preemption could potentially render previously unschedulable models schedulable.

Notice this adds another dimension into the Ptides execution strategies. Each of the previously discussed strategy in Chap. 4 can now be divided into two versions: 1). Preemption is not allowed, and an event is only analyzed for safe to process when the processor is not processing an event, or 2). Preemption is allowed. The second approach implies every time a new event arrives at the platform, or when an event passes the check of physical time becomes greater than timestamp + offset, that event should be analyzed for safe-to-process. If it is safe, and if it is of higher priority than the previously executing event, then that previous event should be preempted.

Now we need to ensure the aforementioned Ptides execution strategies are still valid with preemption.

In fact, the strategies are valid as long as the following requirement is satisfied:

Ptides Strategies Amendment to allow Preemption:
Let event e be an event that is safe to process according to the strategies defined in Chap. 4. If for each event e' that is currently processing or already preempted, e' is of lower priority than e, and e and e' do not share the same destination actor, then e is safe-to-process.

Recall the safe-to-process analyses are divided into two parts (Part 1 and Part 2 of the baseline strategy):

1. Comparison against physical time, which ensures events from outside of the platform do not causally affect event of interest, and

2. Causality analysis using graph structure and sorted event queue to ensure events from inside of the platform do not causally affect the event of interest.

Obviously, the first part would not change once we allow preemption. The event that is currently executing cannot be affected by new incoming events, and whether the new incoming event will be affected by events coming from outside of the platform is independent from an executing event (since it already resides in the platform.)

The second part of the safe-to-process analysis is more interesting. We need assurance that the new incoming event will not be causally affected by the currently executing event. Our workaround is simple, we simply act as if the executing event has not been removed from the event queue, and we compare the priorities of these events. As we proved in Sec. 4.3 and 4.4.2, if the executing event causally affect the new event, then the new event is not safe-to-process.

There is another corner case however, where the new event may be of higher priority, but there could be an older event with lower priority, but share the destination actor with the new event.

This corner case is taken care of by not allowing two events with the same destination to process at the same time.

Notice however, our above strategy has opened the door for priority inversion problems [44]. Luckily, protocols such as priority ceiling [10] resolves these problems.

5.5 Ptides Strategy Implementations in Ptolemy II

We now introduce a number of Ptides directors implemented in Ptolemy II. All of these directors are subclasses of the DE director. The most basic one is the PtidesBasicDirector. This director implements the *Simple Strategy* as defined in Sec. 4.2. However, preemption is disabled, which means if an event is currently executing, and another event with a smaller timestamp arrives at the platform, the first event will finish its execution before the second one starts. This is different from the behavior as defined in 4.2, where the event of smallest timestamp is always processed immediately. Another frequently used director is the PtidesPreemptiveEDFDirector. This director implements the model timed-based deadline strategy as defined in Sec. 4.4.2. Finally, there is the PtidesPreemptiveUserEDFDirector, which implements the baseline strategy. It is called the *Preemptive-User-EDF-Director* because like the deadline-based strategy, a deadline-ordered queue is used to sort system events. The difference here is that it allows programmers to define arbitrary deadlines for these events. Once the subset of all safe events is found using the baseline strategy, the event with the earliest deadline in this subset is processed.

Also notice even though we did not explicitly have a director that implements the Parallel Strategy as defined in The second part of the safe-to-process analysis is more interesting. We need Sec. 4.3, the PtidesPreemptiveUserEDFDirector can achieve the same functionality by using PositiveInfinity as the parameter value for all relative deadlines. This results in all events having the same absolute deadlines, and the event queue will be sorted by the lexicographical order of timestamp, microstep, and depth.

Aside from the aforementioned schedulers, a number of other schedulers are also available. First, there exists a PtidesBasicPreemptiveEDFDirector. This director is a subclass of the PtidesBasicDirector, but it allows for preemption. The exact semantics of this scheduler is not well understood, thus it is generally not used. Also, Patricia Derler has implemented two previous versions of Ptides directors [12], however they are also currently not in use.

Finally, to help a programmer visualize the execution of a system, each Ptides director can be annotated with the parameter animateExecution. When checked, the currently executing actor is highlighted for the duration of its execution time. A parameter commonly used with animateExecution is the synchronizeToRealTime parameter of the enclosing DE director. When checked, the simulation time is slowed down to match oracle time (the model time of the enclosing DE director). To be more concrete, if an actor A is annotated with *executingTime* of 1 second, then the simulation will stall for 1 second to simulate that execution time, before advancing to process the next event. The first parameter, on the other hand, will highlight A for the duration of that 1 second, thus visualizing actor firings in the Ptides model.

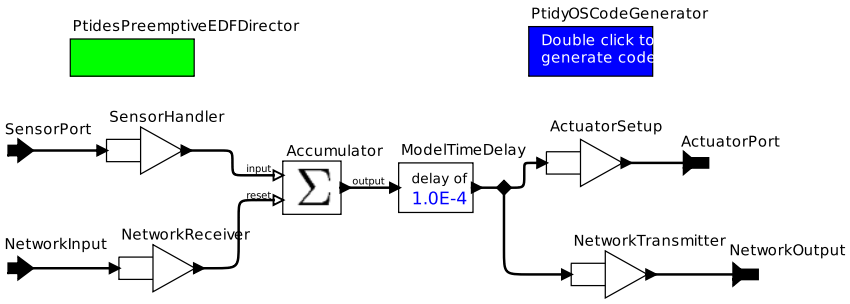

Figure 5.3: Sensor, Actuator and Network Ports

5.6 Code Generation Into PtidyOS

The Ptides simulator in Ptolemy II allows programmers to design and simulate their design using the Ptides programming model. When he/she is satisfied with the design, a code generator is provided to transform the design into a target-specific implementation. The Ptides code generator takes advantage of Ptolemy II's code generation framework. The code generator views all actors as software component libraries, and simply links them together with a Ptides scheduler written in C. The result is a single target-specific C file. Note that the resulting executable, which we call PtidyOS, is very different from conventional real-time operating systems (RTOS) such as OpenRTOS [48], VXWorks [51], and RTLinux [16], which are operating systems that run applications as clients. Instead, our approach is similar to TinyOS (its namesake) [37], an operating system widely used in sensor networks, where the application is statically linked to the scheduler, and starts automatically after system initialization.

5.6.1 Sensors and Actuators

In the Ptides model, all hardware devices are abstracted as input and output ports. Parameters associated with these devices such as d_s, d_a and d_n are captured in the realTimeDelay or networkDelay parameter of these ports, depending on the type of the port. An input port can be either a sensor port or a network input port, while an output port can be either an actuator port or a network output port. The type of the port is determined by the actor that connects to it. If an input port is connected to SensorHandler, that port is a sensor port; otherwise that port must be connected to NetworkReceiver. Take Fig. **??** for example, SensorPort is a sensor port, while NetworkInput is a network port. Note we do not allow inputs that are connected to both SensorHandler and NetworkReceiver. The same is true for output ports: If the output port is connected to an ActuatorSetup actor, that port is an actuator port, otherwise that port must be connected to a NetworkTransmitter. Again in the previous example, ActuatorPort is an actuator port, while NetworkOutput is a network port.

During code generation, subclasses of SensorHandler and ActuatorSetup are used to generate interrupt handlers and set up actuation signals that interacts with physical sensors and actuators.

In Ptides, since we assume sensors are abstractions of a physical device, and since Ptides

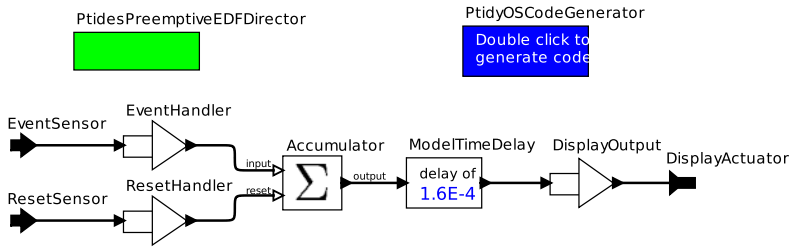

Figure 5.4: Simple Ptides Example

takes an event triggered approach, all sensors are assumed to trigger interrupts. Thus, the fire function generated for SensorHandlers are essentially ISR's. The details of the code generated for an ISR will be talked about in the next section, but first, this ISR is inserted into the vector table. A vector table is usually a list of registers, each associated with a particular interrupt, and each saves an address for an ISR. When an interrupt occurs, that address of the corresponding register is loaded into the PC, thus starting the ISR. The Luminary microcontroller comes with an interactive design environment (IDE) called Keil, which provides software support for accessing the vector table. The vector table is defined in an assembly file with extension .s. During code generation, along with the .c file, this .s file is also generated, with the fire methods for sensor devices inserted into the vector table.

The ActuatorSetup actor, on the other hand, generates two separate functions: an actuator fire function and an actuation function. The actuator fire function is called when input event to the ActuatorSetup actor is processed. This function calculates the data to be actuated, and checks the input events' timestamp against the current platform time. If the current platform time has exceeded its deadline, then an exception is thrown. Otherwise a timer interrupt register is loaded with the value of the timestamp. When the actuation timer interrupt occurs, the actuation function is called. This function takes the previously calculated actuation value and outputs it to the physical environment.

We use Fig. 5.4 as an example. Here, EventHandler, ResetHandler and DisplayOutput actors are created to abstract GPIO ports. They feature specialized parameters to indicate which pad and pin these GPIO devices are connected to. A template file is created for each actor. This template consists of code snippets that initialize system hardware, handle interrupts, create events and insert events into the event queue. During code generation, these code snippets are fetched and modified based on actor parameters in order to produce functional code that runs on a target-specific platform.

5.6.2 PtidyOS Scheduler

The director also serves as a placeholder for code snippets that deals with system scheduling. This mainly includes a while loop that performs the safe-to-process analysis as well as the main function that is the entrance of the program and starts the initialization process. Other system utility functions such as event queue manipulations, etc. are also reported in the adapter [36] for the director. The details of these functions are discussed in Sec. 5.7.

Currently, our code generator uses the Luminary microcontroller LM3S8962 as our prototype target platform. It is based on the 32-bit ARM Cortex-M3 controller (50 MHz, 256 kB flash, 64 kB SRAM), which is equipped with a series of standard peripheral devices. As indicated in earlier sections, most of our strategies only work for systems that run on a single processor, which the Luminary micro provides.

5.7 PtidyOS

The generated PtidyOS is unlike a conventional RTOS. Instead, its software architecture is similar to TinyOS [37]. Its micro-kernel includes a scheduler that performs interrupt handling and context switching and a primitive memory management units. However other conventional OS feature such as a file system is not implemented. When compiled, this kernel links against application code and produces one executable. Since applications are statically linked, dynamic instantiation of applications during runtime is not allowed.

5.7.1 Design Requirements

The primary goal of PtidyOS (or any RTOS) is to support the timely execution of applications. To achieve this goal, a deterministic scheduling scheme is required. Our goal for PtidyOS is to meet deadlines "optimally", where optimality is defined with respect to feasibility. In other words, if there exists a scheduler that is able to meet deadlines for a model, the PtidyOS scheduler should be able to meet these deadlines. At the same time, the scheduler should not be overly complicated, in order to minimize and bound scheduling overhead.

Also depending on the application, hardware resources may be constrained. The particular microcontroller platform for our prototyping purposes only allows for implementations (PtidyOS + application) of size 16kB or smaller. Even if more powerful computation platforms are available, to avoid delays in the operating system layer and minimize scheduling overhead, PtidyOS's application programming interfaces (API) methods should be as lightweight as possible. We discuss our approaches to meet these requirements.

5.7.2 Memory Management

Like in many conventional RTOSs, to ensure real-time system behavior, and to meet the limited memory requirements, memory management in PtidyOS is kept as simple as possible. Specifically, application code is not allowed to dynamically allocate memory on the heap, but instead must allocate memory either on the stack or as static variables. Moreover, recall that actors in a Ptides model communicate through timestamped events. These events are also allocated as static variables at compile time. The number of events allocated must be the maximum number of events running in the system at any time. However this number is difficult to determine. It depends on factors such as the rate of incoming events in the system, worst-case-execution-times, as well as scheduling decisions. We do not attempt to tackle this problem here. Instead, we throw a runtime exception if the pool of free events is exhausted.

The next question is, what kind of data structure should be used to maintain the pool of free events? This data structure must support these basic API methods: allocEvent() and freeEvent(), which allocate and free event structures, respectively. Two alternative designs are

examined. One is to simply create an array of free events, each one with a field indicating whether that event is free. A major disadvantage of this design is that it takes in the worst case $O(n)$ time to find a free event. e.g., if allocEvent() is called when there is only one free event left, residing at index i, and we start to scan the array at index $i + 1$, then we must scan through the entire array before finding this event. An alternative design is to use a linked list. At system initialization, all events are put into this linked list, pointed to by a global variable freeEventsHead. When an event is allocated, the event pointed by freeEventsHead is returned and freeEventsHead is updated. When an event is freed, it is simply inserted at the head of the free list again. Both event allocation and free take $O(1)$ time in this design. The only potential disadvantage with the linked list approach is that it requires more memory to implement. In particular, each Event requires an additional 32 bit value to point to the next free event (assuming a 32bit machine). However, as we will discuss in the next subsection, the event queue is also implemented as a linked list, i.e., a nextEvent pointer is needed for the Event structure anyway. Thus, the same nextEvent pointer is reused to link events in the free list. The only difference then, between an allocated event and a free event is that the free one is only reachable by traversing from freeEventsHead, while the allocated event is only reachable from the head of the event queue.

5.7.3 Ptides Scheduler

The scheduler in PtidyOS implements the deadline ordered scheduler, as presented in Sec. 4.4.2. This scheduler is chosen because it is most applicable as a real-time scheduler. First, it integrates Ptides with EDF, thus allowing us to leverage EDF's optimality with respect to feasibility. Also, it analyzes all events for safe-to-process; i.e., if there exists an event that is safe, then the CPU will not be idle.

As mentioned before, PtidyOS is a C library that links against application code. Most API methods in PtidyOS are for scheduling purposes, such as event allocation, event insertion into the event queue, etc., as well as the main method that determines the order of event processes, called processEvents(). We examine these functions in detail below.

Event Queue Implementation

One critical implementation detail associated with the scheduler is the event queue. Recall event ordering plays an important part in ensuring the correct safe-to-process behavior. Efficiency of queue accesses also plays a critical role in minimizing scheduling overhead. The following API methods are used to access the event queue: peekNextEvent(Event*), addEvent(Event*), removeEvent(Event*).

The event queue is sorted by deadline order to conform with the deadline ordered strategy. peekNextEvent(Event*) takes a pointer to Event as an input. If this input argument is null, then the earliest deadline event from queue is returned. If the input is not null, then the event of next earliest deadline is returned. If the input is already the latest deadline event in the queue, then a null pointer is returned. addEvent(Event*) takes a pointer to Event as input, and inserts this event into the event queue, while obeying the deadline order. Finally, removeEvent(Event*) also takes a pointer to Event as input, and removes this event from the queue. The need for API addEvent() and removeEvent() are intuitive, while the user may find the definition of peekNextEvent() esoteric. It turns out peekNextEvent() is an efficient method because the Ptides scheduler always traverses

the queue in deadline order, analyzing each event for safe-to-process until one is found. The details of how this method is used is discussed in the next subsection.

Given these three functions, the following candidate data structures were considered: binary search tree (BST), heap, and linked list. For BST, all three functions would take $O(\log n)$ to process. For heap, peekNextEvent() could take $O(n)$ time, while addEvent() and removeEvent() would take $O(\log n)$ time. For a singly linked list, peekNextEvent() would take $O(1)$ time, while removeEvent(), addEvent() would take $O(n)$ time. If a doubly linked list is used, however, the complexity for removeEvent() would be reduced to $O(1)$. Also, for a Ptides (or DE) event queues, most events are inserted either at the head or the tail of the event queue. A doubly linked list with both head and tail pointers can take advantage of this by comparing to both the head and the tail first during the insertion, and only traversing the queue if the event cannot be inserted right at the head or the tail. Finally, as mentioned earlier, the pool of free events is also stored as a linked list, and thus the nextEvent pointer of the Event structure can be reused for both lists. Due to these reasons, a doubly linked list is implemented in PtidyOS.

Finally, note that all event access functions are made atomic by disabling interrupts during the procedure, in order to ensure correct concurrent behavior, which we discuss next.

5.7.4 Concurrency Management

Since the PtidyOS scheduler integrates Ptides with EDF, and since schedulability results for EDF only apply under the condition that preemption allowed [38], it is crucial that preemption be supported by PtidyOS. In this subsection, we examine the implementation of a preemptive scheme in PtidyOS. This scheme includes two parts: interrupt handling followed by event processing. We examine these separately.

Interrupt Handling

There are two kinds of interrupts that are of interest: safe-to-process interrupts and external interrupts. Safe-to-process interrupts are essentially timer interrupts, which are set up to indicate at that physical time, some event(s) will become safe to process. External interrupts, on the other hand, are triggered through hardware peripherals such as GPIO or network devices. These interrupts could potentially post new events into the event queue.

To achieve preemptive behavior, the system should not simply return and keep processing the previous event when an interrupt occurs. Instead, if the interrupt results in a earlier deadline event inserted into the event queue, the state of previous processing event should be saved, and the event of earlier deadline should be processed.

PtidyOS achieves this behavior by running the scheduler when timer interrupts occur and when new events arrive at the system through external interrupts. The method to identify new event arrivals is simple. Recall allocEvent() allocates new event from the free event list, starting from the head of the queue. Thus we can simply check at the beginning and end of the ISR to see whether the head of the free event list has changed. If so, then the main scheduling routine processEvents() is called. Otherwise, we return to process the preempted event.

The stripped down implementation of the interrupt service routines is shown below:

```
// ISR for a sensor interrupt.
externalInterrupt() {
    fireActor(thisSensor);
    permitAllInterrupts();
    if (addedNewEvent()) {
        processEvents(1);
    }
}

// ISR for the safe to process timer interrupt
safeToProcessInterrupt() {
    permitAllInterrupts();
    processEvents(1);
}
```

These ISR's are almost identical. For a sensor ISR, since we have an actor that abstracts the sensor hardware device, that actor is fired during the ISR. However, since the safeToProcessInterrupt() is simply a timer interrupt to indicate an event has become safe to process, no actor is fired during this ISR. Next, we notify the rest of the system that new interrupts can be handled. This is the most important function that ensures the responsiveness of PtidyOS, where we assume the hardware supports a permitAllInterrupts() function, which allows all peripherals (including timer) to preempted the currently executing stack once it is called, even if those interrupts are of *lower* priority than the currently executing interrupt. Another way to interpret permitAllInterrupts() is that it tricks the system into thinking that there are no interrupts running in the system. This function is needed because we start the event processing routine next (in processEvents()) and we want to be able to handle all system interrupts during that routine.

Note in the PtidyOS implementation, the interrupt priorities are not important because these interrupts are only assumed to insert new events into the queue. Instead, the event deadlines dictate how computation resources are allocated. With the permitAllInterrupts() function, all interrupts can be given arbitrary priorities, as long as permitAllInterrupts() allows all other interrupts to preempt the currently executing one.

Also note with this interrupt behavior, we can determine the sensor real-time delay d_s. Recall that for every sensor actor, the d_s parameter is defined to be the difference between the sensor event's timestamp (when the sensor event occurred) and the physical time when this event is presented to the rest of the platform. If we take this definition literally, then d_s can be dependent on the number of interrupt instances that are currently present in the system. In particular, if multiple interrupts occur at the same physical time, then one of these events will have to wait until the end of the triggered ISR's to insert events into the event queue. Fig. 5.5 illustrates this scenario.

However, note given the interrupt algorithm described earlier, no scheduling decision is made through the chain of ISR's. This is because permitAllInterrupts() always runs right before processEvents(), thus before processEvents() runs, other ISR would be triggered first. By the time we actually enter processEvents(), all interrupts would have inserted events into the event queue. This means d_s is independent of the existence of other interrupt instances in the system. Thus we can define the sensor real-time delay as follows:

$d_s =$ *platform time at the start of the execution of ISR assuming there are no other interrupt instances minus the event's timestamp*

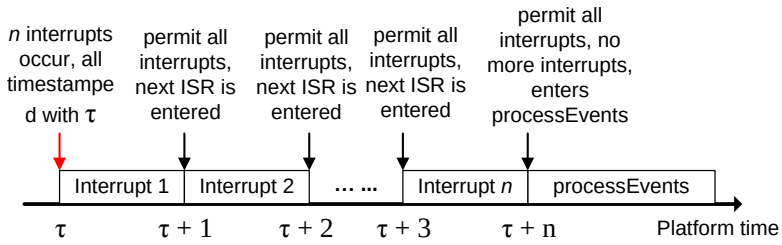

Figure 5.5: Interrupt Chaining

According to this definition, in the previous example, even though interrupt n did not produce an output event until platform time $\tau + n$, the event produced by this interrupt was already safe-to-process at platform time $\tau + 1$. This will not result in a violation of DE semantics because processEvents() does not run between $\tau + 1$ and $\tau + n$.

Notice that d_s could be negative given the above definition. If "software-timestamping" is performed, i.e., the timestamp of sensor event is retrieved *during* the ISR, then d_s would be of a negative value. This would violate the assumption stated in Eq. 3.4: $(t \geq \tau)$. To take care of this corner case, we modify the definition of d_s to be:

$d_s = max$ {*platform time at the start of the execution of the ISR assuming there are no other system interrupt instances minus the event's timestamp,* 0}

Given this definition, any software-timestamped input event will effectively have $d_s = 0$.

Finally, note since we assume that no event processing can occur during an ISR, ISR's must be very lightweight. Otherwise a large execution latency can prevent system events from processing, which can be detrimental to system's responsiveness. In other words, sensor actors are assumed to perform a minimum amount of computation and simply indicate to the system that a sensor event has occurred (e.g., by calling allocEvent() followed by addEvent()).

Event Processing

When new events are created, or when a safe-to-process timer interrupt occurs, the ISRs' in PtidyOS end with a call to processEvents(). As mentioned before, this is the main scheduling function of the Ptides scheduler, and is the second part of the context switching algorithm. This function is rather complex, and we will go through the entire algorithm in detail. The pseudo-code is shown below:

```
processEvents(int restoreStateFlag){
    platformTime = getPlatformTime();
    disableInterrupts();
    Event* event = peekNextEvent(NULL);
    while (event && hasEarlierDeadline(event)) {
        earliestTime = safeToProcessTime(event);
        if (platformTime >= earliestTime) {
            events = removeAllSameTagPortGroupEvents(event);
            pushDeadline(events);
```

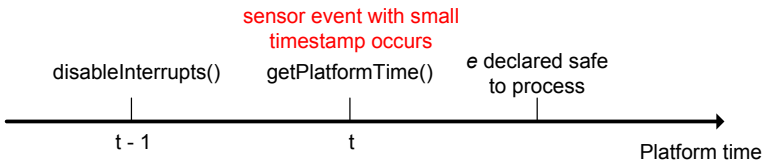

Figure 5.6: Counter Example Illustrating Possible Violation of DE Semantics

```
            enableInterrupts();
            fireActor(events);
            platformTime = getPlatformTime();
            disableInterrupts();
            popDeadline(events);
            freeEvents(events);
            event = NULL;
        } else {
            setTimerInterrupt(earliestTime);
        }
        event = peekNextEvent(event);
    }
    enableInterrupts();
    if (restoreStateFlag) {
        restoreState();
    }
}
```

The first function performed in processEvents is fetching the current platform time. This platform time is used later for the safe-to-process analysis. Recall the safe-to-process analysis checks whether the platform time is larger than the sum of an event's timestamp and a delay offset to determine whether that event is safe. Notice that getPlatformTime() is called *before* disableInterrupts(). This is done purposefully in order to ensure DE semantics, We will diverge here and present a counter example as shown in Fig. 5.6.

In this example, there exists an event e with timestamp τ in the event queue, and e is safe to process at platform time t. Let's say at platform time $t-1$, disableInterrupts() is called, and at platform time t, getPlatformTime() is called, which loads platformTime with the value t. Also at platform time t, a sensor event becomes present, with timestamp less than τ, and shares the same destination actor as e. However this event will not be inserted into the event queue since interrupts are disabled. The safe-to-process analysis later will determine that the current platform time has reached t, and declare e safe to process, thus violating DE semantics.

The downside of calling getPlatformTime() before disableInterrupts() is that interrupts may occur between these functions, and when safeToProcess() function is called, an event that is in fact safe may be determined unsafe. This will result in setting a timer interrupt, and trigger another call to processEvents() later. In other words, an additional call of processEvents() may be sacrificed in order to guarantee DE semantics.

Notice that an alternative is to assume the sensor delay d_s includes the physical time delay to perform getPlatformTime(). In the previous example, this translates to: $d_s = 1+$ worst-case-response-time of the sensor. This ensures the sensor event will be inserted into the event queue

before interrupts are disabled, and thus getPlatformTime() can be placed after disableInterrupts(). However this in effect asks the application programmer to take into account execution time delay of the scheduler, which is independent from the application, and he/she should not worry about such details. Thus we do not choose that implementation.

Going back to the pseudo-code. Since the rest of the scheduling logic should be atomic, getPlatformTime() is followed by disabling all external as well as safe-to-process timer interrupts. Then peekNextEvent() is called. By providing an argument of null, peekNextEvent() will return the event of earliest deadline from the queue. Next, we decide whether to enter a while loop. This loop is entered if the head of the queue is not null, and if the returned event has an earlier deadline than any preempted event. As line 9 shows, when we decide on an event to processed, pushDeadline() is called to store the deadline of that event. hasEarlierDeadline() in line 5 compares the deadline of previous stored event with that of the current event. If a preempted event is of earlier deadline, then we go to line 22 and get ready to return from this method. Otherwise, we proceed to perform the safe-to-process analysis.

Recall that the safe-to-process analysis is simply a check against physical time. The function safeToProcessTime() performs the calculation as shown in Part 1 of the baseline strategy (right side of Eq. 3.11). The returned value is stored in earliestTime. We then compare the previously retrieved platform time to earliestTime. If the platform time is larger or equal to earliestTime, then this event is safe, and we are ready to process this event. If this event is not safe, however, we set up a safe-to-process interrupt to wakeup the system at the earliestTime by calling setTimerInterrupt(), as shown in line 18.

If the event is safe to process, then removeAllSameTagPortGroupEvents() is called. This method takes the safe event as input, scans all events in the queue, and returns all events with the same tag that are in the same destination input port group. We do this because all of these events should be processed during the same firing of the actor [32]. Notice that since the input event is safe, all events returned by this method are also safe. These events are also removed from the event queue in this function. Next, pushDeadline() is called. As mentioned before, the deadline associated with this event is used to compare with future preemptive events in hasEarlierDeadline(). Finally, we are ready to enable interrupts. This allows for new events to enter the system during the firing of the destination actor.

After the destination actor finishes firing, getPlatformTime() is called again to prepare for the firing of the next event. Just as getPlatformTime() is called before disableInterrupt() at the start of this function, these two functions are again called in this order to ensure correct concurrent behavior. After interrupts are disabled, popDeadline() is called to mirror pushDeadline() before the actor firing. The processed events are also freed and made available for future event allocations; i.e., they are put back into the free list of events, and can be reused later by calling allocEvent(). Then, event is reset to null to indicate we should start to examine events from the start of the event queue in the next firing.

Regardless of whether the previous event is safe to process or not, we get the next event to process by calling peekNextEvent() at line 20. The only difference here is that if an event was processed, then we start analyzing the event queue starting from the head of the queue. If no event was processed, then we peek the next event in queue.

When there are no more events to process, or when the preempted event has an earlier deadline, we go to line 22, which enables all system interrupts. This is followed by calling re-

storeState() if restoreStateFlag is true. We will talk about why restoreStateFlag is needed later, but recall saveState() is called at beginning of ISR's to save the original machine execution state. restoreState() simply restores the last saved state. Notice like saveState(), restoreState() also needs to be an atomic operation, even though interrupts are enabled right before the function call. In PtidyOS, this is achieved by implementing restoreState() as a software-triggered interrupt. This software interrupt is given the same hardware priority as all other interrupts, and thus it cannot be preempted during its execution. Even though this may seem like an overkill, there are more important reasons why restoreState() must be implemented as a software interrupt, due to the kind of hardware available to us. We will discuss thos reasons in the Stack Manipulation subsection.

Event Processing to Optimize for the Highest Priority Event

There is however, a major inefficiency in this Ptides scheduler. Suppose there are a total of n events in the system, and none of these events is safe to process. The above algorithm would traverse through the entire event queue, setting timer interrupt for each event, before exiting processEvents(). Notice interrupts are not enabled during this entire time period, and if an interrupt of the earlier deadline comes in during this time, it will not be processed until we finish traversing the event queue. In other words, the "reactiveness" worsens as the number of events in the event queue increases.

To resolve this problem, we modify the previous pseudo-code to the following:

```
processEvents(int restoreStateFlag) {
    processVersion++;
    platformTime = getPlatformTime();
    disableInterrupts();
    Event* event = peekNextEvent(NULL);
    while (event && hasEarlierDeadline(event)) {
        earliestTime = safeToProcessTime(event);
        if (platformTime >= earliestTime) {
            events = removeAllSameTagPortGroupEvents(event);
            pushDeadline(events);
            enableInterrupts();
            fireActor(events);
            platformTime = getPlatformTime();
            disableInterrupts();
            popDeadline(events);
            freeEvents(events);
            event = NULL;
        } else {
            setTimerInterrupt(earliestTime);
            localProcessVersion = processVersion;
            enableInterrupt();
            disableInterrupt();
            if (localProcessVersion != processVersion) {
                break;
            }
        }
        event = peekNextEvent(event);
    }
```

```
29      enableInterrupts();
30      if (restoreStateFlag) {
31          restoreState();
32      }
33  }
```

Lines 2, and 20-25 are added in this new version. The main idea behind this algorithm is to enable interrupts for a short period every time an unsafe event was found. This decouples the context switch time for the earliest deadline event from the number of events currently residing in the event queue. Interrupt enabling and disabling are done in lines 21 and 22.

If an interrupt indeed occurs between these lines, the new interrupt may modify events in the event queue; i.e., the event queue could be corrupted by the time we return from interrupt. This means, we must stop *this* instance of processEvents() if an interrupt occurred. This is accomplished with the global variable processVersion. This global variable is updated each time processEvents() is entered (line 2). On line 20, a local version number is saved, and the local version is compared to the global version after interrupts are disabled again. If the global version is updated, then we simply break out of the while loop and return. On the Luminary implementation, a 32 bit unsigned int is used for this version number. This means line 23 would return the correct answer as long as no more than 2^{32} instances of processEvents() run at the same time, which we assume is the case.

Finally, it should be noted that all functions that actors use to access the event queue, such as allocEvent() and addEvent() are all interrupt protected.

The only other function linked during code generation is the main entry function, main(), which first initializes all actors. During the initialization, actors may post initial events onto the event queue. Following initialization, processEvents() is called with restoreStateFlag set to 0 to process any of the initial events. Recall that restoreState() is only called if restoreStateFlag is true, and since no interrupt has occurred, there is no need to restore the original state of execution. Thus we pass in 0 to simply return when processEvents() finishes. When processEvents() returns, it means PtidyOS is only waiting for external or timer interrupts to trigger itself again. Thus at this point, the machine can go into hibernation mode if there is hardware support for it. If not, we simply loop forever in while (TRUE) to wait for future interrupts.

```
1  void main() {
2      initializeAllActors();
3      processEvents(0);
4      while (TRUE) {
5          hibernate();
6      }
7  }
```

Single Stack for Storing Execution Context

We have finished our discussion on the event processing aspect of the context switch algorithm. There is however, yet another subtlety one must examine. As mentioned before, the functions saveState() and restoreState() save the context in which the preempted event was executing, and

restore the state when we wish to go back to executing the preempted event, respectively. So where is the context stored?

A conventional RTOS usually implements context switching with the help of threads. Each thread has a separate context, and when preemption occurs, the current execution state is saved in the original thread, and the processor switches its execution context to the states saved in another thread. However, there are many problems with using threads [30] in a real-time environment, not the least of which is the scheduling overhead associated with context switching. The thread contexts are usually saved in some data structure, and this data structure is traversed when context switching is needed. The process of searching through the threads to find the corresponding context can lead to unpredictable timing behaviors. PtidyOS steps away from that approach while still allowing context switches.

There are two important properties with our scheduler that allow us to step away from threads:

1. The deadline of an event do not change throughout the lifespan of the event.

2. PtidyOS schedules events according to the monotonic ordering of the deadlines.

These properties ensure if an event e_1 is preempted by a earlier deadline event e_2, then e_1 will never be processed before e_2 finishes processing; i.e., the saved state for e_1 will not be accessed before the state of e_2 is accessed. This in effect allows the use of a stack to save the state for all preempted events. Each time an event e_1 is preempted, its state can be simply pushed onto the stack. The only time this state is accessed again is when e_1 becomes the earliest deadline event in the system. As more events are preempted, the stack grows, and the priorities of the events on the stack grows monotonically as well. Due to this property, PtidyOS does not need to keep a separate data structure that stores all the currently preempted events. Instead, the event states are all pushed onto the stack. Thus, the only context switch operations are to push the current state in order to execute a new event, or pop the last preempted processing state. Note these are $O(1)$ operations, which does not depend on the number of preempted events in the system. Thus, it is easy to bound the context switching time in the PtidyOS environment.

The only caveat with this approach is that we must ensure that priority inversion problems do not occur. If two events with different deadlines share the same destination actor, and the event of later deadline is preempted, then the event with earlier deadline will not be able to execute. Fortunately, [2] presents a resource allocation policy which permits processes with different priorities to share a common stack, thus averting priority inversion problems.

In summary, by choosing proper deadlines, and by defining a scheduler whose ordering is monotonic to these deadlines, PtidyOS is able to constrain the context switching behavior such that only a single stack is needed to keep track of preempted states. This means saveState() is a simple operation of pushing the current registers onto the stack, while restoreState() simply pops the stack.

Stack Manipulation

As we mentioned before, permitAllInterrupts() method assumes there is hardware support (usually in the form of writing a register bit) for the user to signal the end of this ISR, and to allow the CPU to process other interrupts. Unfortunately, not all microcontrollers provide this

support. The ARM Cortex-M microcontroller (which was used for prototyping) for example does not provide such support for security reasons; i.e., the interrupt registers that signal the entering and exiting of an ISR are not writable by a user program. In fact, special instructions for entering and exiting ISR's are hard-coded in a particular memory address. When an interrupt occurs (exits), the interrupt controller in the CPU updates the Program Counter (PC) to point to that address, which then executes a special PUSH (POP) instruction. During the execution of these instructions, the system automatically sets or clears the interrupt registers to indicate the start or end of an ISR. At the end of these instructions, the PC is updated again to jump into or out of an ISR.

In summary, ARM-7 processors do not allow writes directly to the Program Counter (PC), interrupt status registers, as well as program state registers (xPSR). However, they do include an assembly instruction to write to the Stack Pointer (SP). In other words, the only way to write to the PC and xPSR as well as updating the interrupt registers, is to write the PC and xPSR values to the corresponding addresses on the stack, update the SP, and pop the values from the stack onto the PC and xPSR registers through the special POP instruction. This is the approach taken by PtidyOS, in order to jump to processEvents() instead of returning to the previous executing event at the end of ISR.

To take advantage of this scheme, the ISR's are modified, and we show the modified sensor ISR below:

```
externalInterrupt() {
    saveState();
    fireActor(thisSensor);
    if (addedNewEvent()) {
        addStack();
    }
}
```

First, a saveState() function is performed when we enter the ISR. This function saves the registers such as the PC, xPSR, and other registers used to execute the preempted event. Normally these registers are saved when we first enter the ISR. However, as we will explain later, addStack() is a stack manipulation scheme which updates the PC to a place where the compiler does not expect, thus the compiler may not push the necessary registers. Instead, we need to manually save the previously executing registers onto the stack. Next, the functions permitAllInterrupts() and processEvents() are replaced by addStack(), which is a stack manipulation procedure written in assembly. A visualization of the whole process is shown in Fig. 5.7.

Stage 1 shows the program executing at some state, with the SP pointed to the top of the stack. At this point an interrupt occurs, and the special PUSH runs. This PUSH instruction pushes 8 registers onto the stack, including Program Counter(PC), program state register (xPSR), etc. However, the ARM-7 processor has a total of 16 registers. Since processEvents() could potentially use all the registers, the other 8 registers are also pushed onto the stack. This is done through saveState(), which is called at the very beginning of the ISR. At the beginning of state 3, the previous executing state has been saved on the stack. During stage 3, The sensor actor fires, which may or may not insert new events into the event queue. Here we are only interested in the former case, where addStack() is called next.

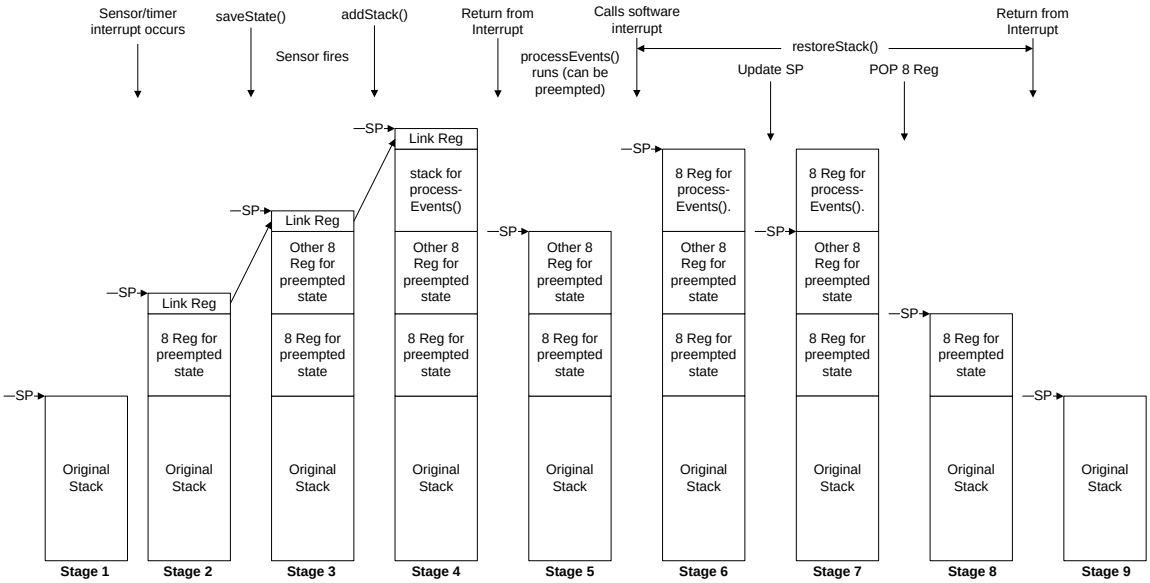

Figure 5.7: Stack Manipulation Visualization

The assembly code for addStack() for an ARM-7 architecture is as follows:

```
addStack
    ; disable interrupts
    CPSID       I
    ; Save stackpointer in R1
    MRS         R1, MSP
    ; Copy previous R4, LR to very top of future stack
    LDRD        R2, R3, [R1, #0]
    STRD        R2, R3, [R1, #-32]
    ; Load R3 with the desired xPSR
    MOV         R3, 0x1000000
    ; Store desired xPSR onto the stack
    STR         R3, [R1, #4]
    ; Load R3 with address at "processEvents + 2".
    ; The first instruction of processEvents stores the original stackframe.
    ; This was done manually at saveState, thus it is skipped here.
    LDR         R3, =processEvents
    ADD         R3, R3, #2
    ; Store desired PC onto the stack
    STR         R3, [R1, #0]
    ; Load stackpointer with updated value
    SUB         R1, R1, #32
    MSR         MSP, R1
    ; enable interrupts
    CPSIE       I
    ; Return to the end of ISR, which will perform the special PUSH
```

```
26      BX          LR
```

This assembly is described visually as the transition between stage 3 and 4. This method increments the SP, and zeros all values between the new SP and the original SP in step 2, except for two 8 byte addresses. These two addresses correspond to the PC and xPSR registers. The address corresponding to the PC is written with the address of the processEvents() function, while the one corresponding to xPSR is written to indicate the normal execution state of the machine. Notice that since this method messes with values on the stack as well as the SP itself, interrupts must be disabled during this function call. Otherwise if an interrupt were to occur during this function, the machine may be left in an inconsistent state. At the end of stage 4, the external/timer interrupt returns, and the special POP instruction runs. This instruction will pop the previously stored values onto the system registers, with the PC now pointed at the start of processEvents().

During processEvents(), we can have additional interrupts occuring, in which case stage 5 can become the new stage 1. At the end of processEvents, we wish to return to the previous executing event by popping the previous executing state, which is stored in 16 registers. Unfortunately we cannot simply return from processEvents(), because the ARM architecture prevents the user from directly popping into the PC and xPSR registers. Instead, only the special POP instruction at the end of ISR's are allowed to do that. We solve this problem by calling a software interrupt. A software interrupt is like any other interrupt, only it is triggered just like any other function call. This interrupt, restoreState() is called at the end of stage 5. It (like all other interrupts) starts with the special PUSH instruction, and pushes the currently executing state onto the stack. Notice at this point processEvents() has finished processing, and the current machine state is longer useful. Instead, at the end of stage 6, the SP is updated to point to the location before software interrupt, and the top 8 memory addresses are popped into the registers (these registers were saved at stage 3). This is followed by the special POP instruction at the end of the software interrupt. This instruction will restore the original machine execution state at stage 1.

Note, we have mentioned before that saveState() and restoreState() functions must be atomic. saveState() is atomic because it runs during an ISR, and all ISR's are given the same priority, and the hardware support ensures it cannot be preempted. The advantage of using the software interrupt to perform restoreState() is that we can simply set the software interrupt to have the same priority as all other external and timer interrupt, then it too will not be preempted.

This sums up the context switching overhead of PtidyOS. We discuss the performance, including latency of the of the context switching algorithm in the next subsection.

5.7.5 Meeting of the Key Constraints

We now examine the key constraints we set out to satisfy for PtidyOS.

Limited Resources

The base size of the generated PtidyOS is 16.18kB. This includes all utility functions in the PtidyOS API. A single Event was statically allocated as the available free pool of events. In other words, 16.18kB is the absolute minimal amount of memory needed to run PtidyOS. As we will talk about in more detail in the next chapter, the tunneling ball device demo was implemented

using the Ptides design flow. After code generation, the entire PtidyOS file is of size 22.32kB, which includes PtidyOS, 5 free event structures, as well as the application code.

Reactive Concurrency

PtidyOS's support for concurrency is evaluated based on the following two metrics: 1) The concurrency exhibited by application and 2) scheduling overhead.

Exhibited Concurrency:

A common metric to test the concurrency exhibited by an application is to look at the percentage of code that is reachable through interrupt contexts [37]. However, this also depends on the definition of "reachable through interrupts." One interpretation would mean the amount of code that runs in the ISR. Recall from Sec. 5.7.4 we assume ISRs only perform event creation and insertion into the event queue. This means only a small amount of code is executed in the ISR.

Another interpretation of code "reachable through interrupts" would be the amount of code that runs between when the interrupt occurs and when we return to the previously executing state. Given this definition, Fig. 5.7 shows clearly that any code run between stage 2 - 8 would be defined as "reachable," which includes a call to processEvents(). Since processEvent() can access the all the actor processing code (through fireActor()), this means the entire application code base is reachable through interrupts except for initialization functions. This constitutes 83.9% of the code base.

Though the second interpretation indicates PtidyOS is highly concurrent, our implementation runs the scheduler at the end of ISR in order to achieve this behavior. Next we analyze the scheduling overhead at the end of ISR.

Context Switching Overhead:

Fig. 5.8 shows a visualization of the context switching overhead. The measurements are taken on a Luminary controller with the help of an oscilloscope. The microcontroller runs at 50MHz. The x-axis of timing diagram is platform time, and the number annotated on each execution trace indicates the amount of time (in microseconds) it takes to run the procedures. The line numbers on top of the traces correspond to the line numbers shown earlier in the pseudo-code for processEvents(), and indicate the procedures ran during that period of time. Since the interrupt handling routines are short, they are not annotated with line numbers. At some points of the timeline, there are multiple procedure running. e.g., at platform time $16\mu s$, two procedures are active. This indicates a possible branching condition. We will explain these branching conditions in more detail below.

As mentioned before, the context switching procedure can be broken down into two parts: interrupt handling and scheduling. From the graph, we see that interrupt handling overhead takes either $7.14\mu s$ or $7.74\mu s$ to complete. If the interrupt did not insert new events into the event queue, then the system simply restores its original state, and goes back to the preempted event. In this case, the total latency for this interrupt is $7.74\mu s$, which translates to 387 cycles given a 50MHz clock. If however, new events have been inserted into the event queue, then it takes $7.14\mu s$ before addStack() finishes, and processEvents() starts.

It might seem unintuitive that the execution trace where an event is inserted takes longer than the execution trace where no event was inserted. This is because restoring the previous execution state requires writing more registers than addStack(). Also notice, interrupt handing includes not only the context switching overhead such as saveState() and addStack(), but also the time it

takes to produce sensor events (call this time sensor firing time). This time varies depending on the application, but is generally assumed to be lightweight. For this measurement, we assume a sensor firing time (execution time of fireActor(thisSensor)) of $2.7\mu s$, based on the tunneling ball device example discussed in the next section.

Recall that addStack() leads to the running of the scheduler, which performs processEvents(). There are four possible scenarios:

1. the event queue is empty,
2. the queue is not empty, but the preempted event is of earlier deadline
3. the queue is not empty, the preempted event is of later deadline, but the preempting event is not safe to process, or
4. the queue is not empty, the preempted event is of later deadline, the preempting event is safe to process.

In Fig. 5.8, the first two cases are visualized by a single execution trace. As mentioned before, higherPriority() is an $O(1)$ operation that takes $< 1\mu s$ to process, and thus is omitted for visualization purposes. These are also the two cases where context switching time is the smallest. The scheduler would simply call restoreStack() to return to the previously processing event. In this case processEvents() take a total of $11.75\mu s$ to complete.

If the event is safe, then interrupts are enabled after $15.01\mu s$. Right after enabling interrupts, that event is processed. In other words, the context switch time to process the earliest deadline event in the queue (assuming that event is safe) takes $7.14 + 15.01 = 22.15\mu s$, or roughly 1100 cycles.

Finally, if the event is not safe, processEvents takes $32.46\mu s$ to finish, which includes setting up a timer interrupt.

To put these numbers into perspective, assume there are n events in the event queue, and none of these events are safe except for the last one. Then it would take $7.14 + 6.35 + (3.38 + 17.38) \times (n - 1) + 3.38 + 5.28 = 22.15 + 20.76 \times (n - 1)\mu s$ until the nth event is processed. Assuming there are no other interrupts occurring during this time, this is the worst case in our context switching algorithm.

Also just to emphasize, as we have discussed earlier, since interrupts are enabled after the safe-to-process analysis of each event, if earlier deadline events are inserted into the event queue during context switching, we can switch to those events and immediately process them. In other words, the PtidyOS scheduler is optimized for the earliest deadline events (the essence of EDF).

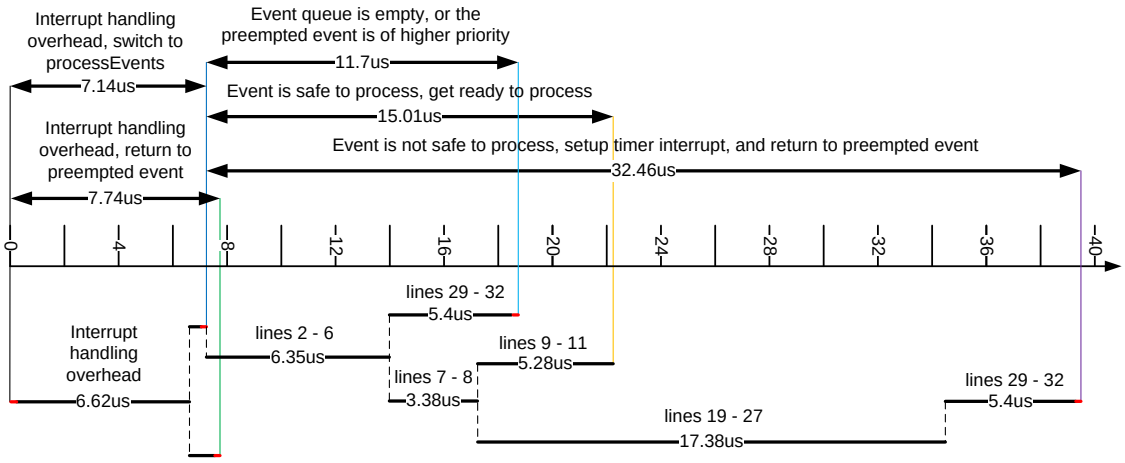

Figure 5.8: Scheduling Overhead Timing Diagram

Chapter 6

Design Flow Experimentation

This chapter discusses two examples implemented using the Ptides design flow.

6.1 Execution Time Simulation

The first is the simple Accumulator example, where the Ptides model is shown in Fig. 5.4. C code is generated using the Luminary microcontroller as target platform. The main functionality we wish to illustrate with this example is the ability of the Ptides simulator to simulate execution time.

Recall the last part of the Ptides workflow as shown in Fig. 1.2 includes a physical time simulator. This simulator takes execution times for computation blocks and the scheduler as inputs, and simulates the physical time delay for the model. Physical time simulation is an essential part of the Ptides workflow, since it enables checks against deadline misses.

Here, we again refer back to the example shown in Fig. 5.4 and demonstrate timing delays associated with that model.

Individually, the execution times for each of the actors running on the Luminary is measured. This information is annotated using an executionTime parameter at each actor's input port. Notice that in this model, the execution time of the SensorHandler includes the context switching time as well as the time to produce a sensor event.

Next, we need to annotate the model with scheduler overhead execution time. This overhead can be captured using the schedulerExecutionTime parameter of the Ptides director. We have measured a set of execution times for functions in processEvents, but which one should we use? As shown in the last chapter, this really depends on whether the currently analyzed event is safe to process. The scheduling overhead is much smaller if it is safe. By default, we take the conservative approach and assume the event is never safe. As shown in the last section, the scheduling overhead is measured to be $32.46\mu s$. We will now illustrate how these values can be used to simulate end-to-end delays in different scenarios.

The first scenario (case **a**) we consider is one where only the EventSensor is connected, while the ResetSensor is idle. We also assume software timestamping is performed at sensors, i.e., $d_s = 0$. By using the previous calculated values, the end-to-end delay from EventSensor to DisplayOutput is simulated to be $175.69\mu s$, while the actually measured delay is $110\mu s$. The reason our simulated delay does not provide a tight bound is due to the fact that we assumed events

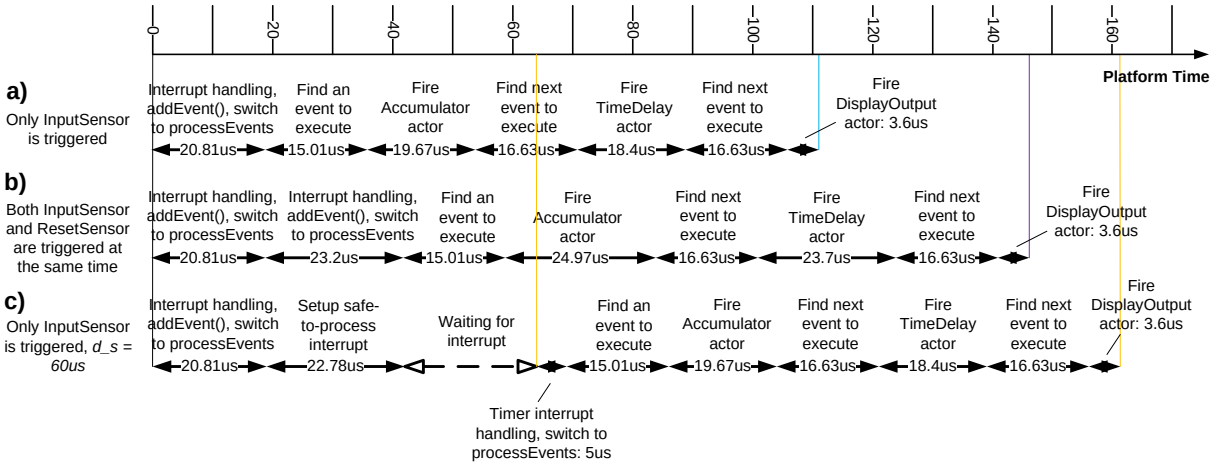

Figure 6.1: Event Trace Timing Diagram

are always *not* safe to process, in order to obtain a conservative estimate of the scheduling overhead. However in this case, due to software-timestamp, input events are actually *always* safe, which lead to over-estimation of the scheduling overhead. Since by analyzing the model statically we already know events are always safe, we can use the previously measured overhead for that case. This value is $16.63\mu s$. By using this value instead of $32.46\mu s$, the end-to-end delay reduces to $112.37\mu s$, which is a bound close to the actually measured delay of $110\mu s$. Notice that in the previous chapter we used the value $15.01\mu s$ for the scheduling overhead for the case where the event is safe. However $15.01\mu s$ only applies when processEvents() is first entered. $16.63\mu s$, on the other hand, represents the delay starting at the end of fireActor(), until the next fireActor() is called. The difference here is that both popDeadline() and freeEvent() are included in this scheduling overhead, The timeline as shown in Fig. 6.1a exhibits the measured execution times at each actor and the scheduling overhead.

Next, we measured the end-to-end delay for the case when both inputs are triggered by the same input signal. Here both sensor interrupts would run one after the other. This adds additional end-to-end latency into the system. The delay from EventSensor to DisplayOutput is simulated to be $162.8\mu s$ in this setup. The actual measured delay is $144\mu s$. Again, the optimistic overhead of $16.63\mu s$ is used. Notice that this bound is not as close as what we obtained the last time. This is due to an imperfection of the current simulator implementation, which assumes a scheduling overhead after each actor firing, including ISR executions. However, in the actual implementation, since EventSensor and ResetSensor are triggered at the same time, the sensor ISRs execute one after another, and only one context switch to processEvents() is needed to find the safe event (the second call to processEvents() is stacked, and will be called after all safe events are processed). Indeed, if the simulator supported this simulation strategy, then a tighter bound of $162.8 - 16.63 = 146.17\mu s$ would be simulated, which is closer to the measured value.

Notice that from the Fig. 6.1, the execution times for Accumulator and TimeDelay are larger in the latter scenario. Recall from Sec. 5.7.2, addEvent() is an $O(n)$ procedure, where n is the number of events in the queue. Since addEvent() is typically called as a part of the actor firing, to correctly simulate the end-to-end delay, the execution time of the actors must be modified.

In both of the previous cases, we assumed a sensor real-time delay $d_s = 0$. In the latter case, we measured the end-to-end delay assuming real-time delay $d_s = 60\mu s$. This delay translates to a non-zero delay offset (defined in Sec. 3.3) at the input ports of the Accumulator. Here the overhead to execute the safe-to-process interrupt is $10\mu s$ (an event becomes safe at the $60\mu s$ mark, but processEvents() does not start until the $70\mu s$ mark). This delay can be attributed to two sources. First, even though we would expect the interrupt to occur at platform time $60\mu s$, it actually happens at $65\mu s$. This delay is related to the hardware timer support of the Luminary microcontroller.

The functionality of the Luminary microcontroller's timer interrupt can be summarized as follows: a value is loaded into a timer register. As soon as this timer is enabled, it starts counting down. When it counts down to zero, an interrupt occurs. Thus, this timer is perfect if we want to wakeup after a specific period of time. However, this timer is less optimal when we wish to wake up *at* a specific time. Notice the subtle difference here, where we need to first get the current platform time, and then perform a subtraction in order to find the value that should be loaded into the timer register. Both of these actions together with the interrupt latency contribute to the $5\mu s$ delay before the safeToProcessInterrupt is handled. In addition, interrupt handling of the safeToProcess interrupt plus the time to context switch to processEvents takes another $5\mu s$, thus the total overhead for the safe-to-process interrupt is $10\mu s$.

A more ideal microcontroller platform for the purpose of Ptides would be one which provides hardware support for interrupt trigger at a specific time. However, even then the interrupt handling overhead may not be negligible. Thus the Ptides simulator provides another parameter called safeToProcessTimerHandlingOverhead to capture this latency. In the above scenario $10\mu s$ is entered for this parameter. The simulated delay is predicted to be $161.56\mu s$, while the actual measured delay is $161\mu s$.

6.2 Application Example

6.2.1 Application Setup

A more real-world example application was built using the Ptides design flow. This application, called the Tunneling Ball Device (TBD), was first developed by Jeff Jensen [24].

The physical device is shown in Fig. 6.2. This device consists of a spinning disc with two holes on opposite ends, and the disc is controlled with a motor. The motor is controlled by a microcontroller, which interacts with the environment through GPIO and a pulse-width-modulator (PWM). Every revolution, the motor generates encoder pulses to a GPIO pin of the microcontroller. These pulses are interpreted as the current position of the disc. The controller outputs a periodic pulse through the PWM. The width of the pulse dictates the power output of the motor, and thus the speed of the disc. Aside from the motor-disc setup, there is also a shaft on the side, with two optical sensors mounted on top. Metal balls are dropped through these sensors. When drop events are detected, pulses are sent to the microcontroller through another GPIO. According to these pulses, the speed of the ball is calculated. Using this information, along with the vertical distance from the sensors to the disk (which is measured statically), and the current position of the disc (which is captured by the encoder ticks), the control algorithm calculates the change in disc speed to ensure one of the holes intersect the trajectory of the ball so the ball can pass through it.

This application is chosen for our design purposes because it is an exemplary cyber-

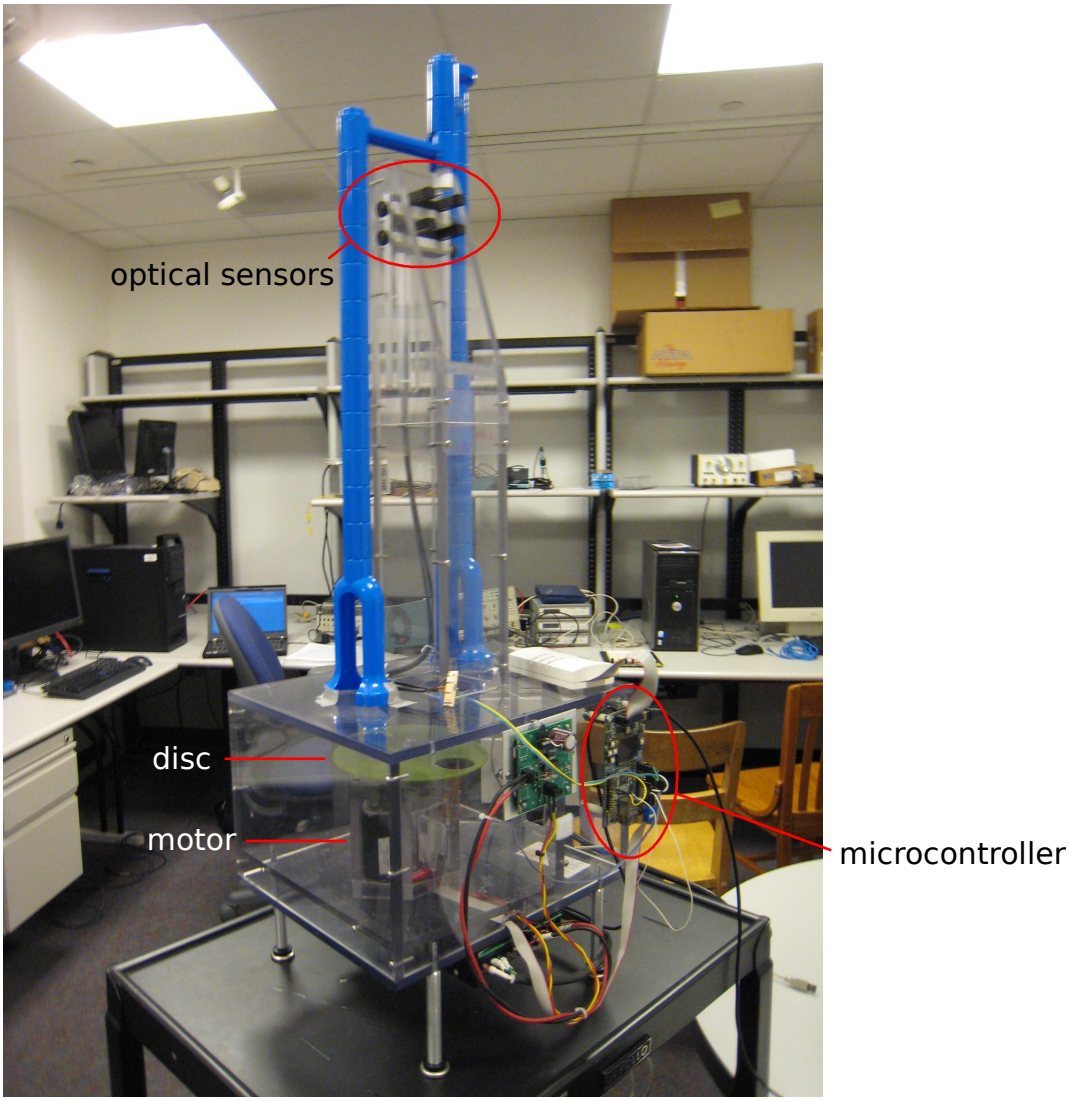

Figure 6.2: Tunneling Ball Device

physical system (CPS). CPS is a category of systems that stress the integration and interaction between the physical plant and a set of networked computing platforms.

In particular, a subcategory of interesting CPS are motion control or automobile applications. In these applications, the physical plant usually communicates the computers through three kinds of signals:

1. periodic. e.g., signals that periodically trigger a control loop.
2. quasi-periodic. e.g., signals that are tied to the rotation of the crankshaft. As motor speeds increase and decrease, the periodicity of the signal changes.
3. sporadic. e.g., signals that indicate abnormal events.

The tunneling ball device encompasses all three kinds of signals. The motor's speed changes, which results in quasi-periodic encoder pulses; The control loop runs periodically; And the drop sensor produces sporadic signals. Because of its similarity to motion control applications, this platform was chosen to test the Ptides design flow.

6.2.2 Implementations

Manual Implementation

Jeff Jensen, the designer of this device, first developed a C program to control it. This program runs on bare silicon. The C program consist of four external interrupt handlers:

1. encoder interrupt: updates the current position of the disk from the motor encoder ticks,
2. desired position interrupt: updates the desired position of the disk,
3. ball drop interrupt: indicates whether a ball drop event has occurred, and
4. user startup interrupt: starts rotating the disc at a nominal speed.

The main procedure of the C program runs in an infinite loop. As control theory dictates, the controller must run periodically according to the control period. Thus the loop starts by busy waiting until the current platform time is equal to a multiple of that period. This is followed by executing the control routine, which implements a proportional-derivative (PD) controller. The calculated output is sent to the PWM, followed by returning to the start of the loop. This loop runs indefinitely until a system reset signal is received.

Ptides Model

A Ptides model for the TBD is shown in Fig. 6.3, which is a modified version from a model first developed by Jeff Jensen.

This model is governed by a DE director on the outside to model the passage of physical time, and it includes two main sub-models. The physical kinematics of the ball, the motor and the disc are modeled in DE. The controller is modeled in Ptides. This model abstracts each of the previously mentioned sensors as sensor actors, excluding the planned position interrupt. Instead of

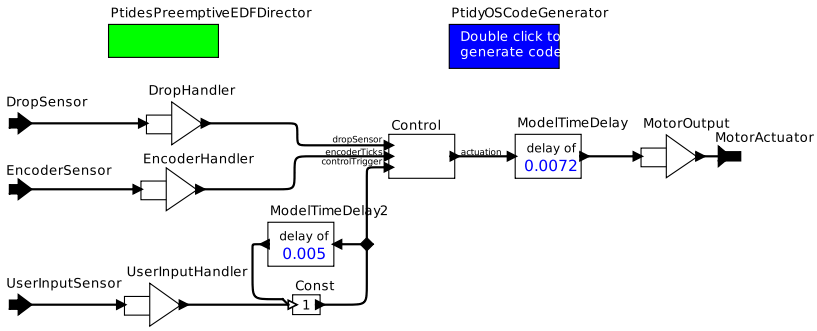

Figure 6.3: Ptides Model For Tunneling Ball Device

having an interrupt that updates the planned position, a local variable in the Control actor keeps track of this position. This variable is updated every time the control loop runs. An ISR is generated for each of these sensors. The main control logic resides in the Control actor.

The drop sensor handles ball drops through the optical sensors. If the sensors detect signals in a short interval, it is assumed a ball drop event has occurred. This triggers the drop sensor interrupt service routine, and an output event is produced by this actor. The payload of this event is the time interval between the two drop events. From this interval, the velocity of the ball drop is calculated in the control routine. Together with the physical distance from the bottom sensor to the disc, one can calculate the platform time at which the ball will strike the disc.

The encoder sensor is connected to the motor encoder. Each time a encoder tick is received, an output event is produced. The accumulation of these encoder ticks allows the controller to keep track of the current position of the disc.

Finally, the user input sensor is connected to a button. Once triggered, an output event is produced to indicate system startup, at which point the controller starts rotating the disc at a nominal speed.

6.2.3 Application Analysis

The key Ptides feature we will demonstrate with this application focuses on the use of timed semantics to ensure deterministic event execution. In particular, the control routine keeps track of a planned disc position. This position is updated at a rate that mirrors the desired disc speed, which is set by the user. The difference between the planned and actual disc position (updated by the encoder sensor) is used to calculate the motor power output. When a ball drop event occurs, the disc speed is adjusted by updating the rate of planned position. This new rate is calculated using the time to impact for the ball (timeToImpact) and the current actual position actualPosition of the disc. The basic assumption here is that when the new rate is calculated, timeToImpact and actualPosition must refer to the state of the disc at the same instance in physical time. However this assumption may not be true given certain constraints. Our goal is to compare the manual and the Ptides implementations in situations where this assumption does not hold.

Fig. 6.4 provides a visual illustration of this problem. The processing of both disc position signals (planned and actual) and drop data takes a negligible amount of physical time, since they

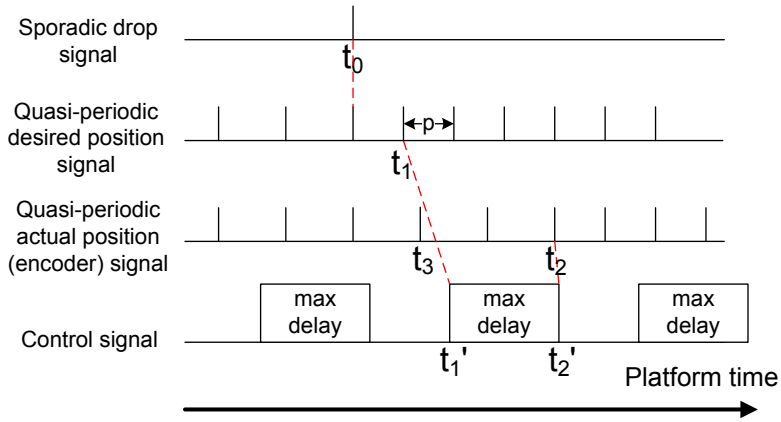

Figure 6.4: Control Signals For the Tunneling Ball Device

are simply ISR's that increment one memory location. Processing the control signal however, can take a large amount of time. In fact, we assume the *max delay* parameter (as shown in Fig. 6.4) captures the maximum amount of time it takes to read both the drop signal and the encoder signal. Notice that this time includes not only the time to perform the reads, but also the time to process other preemptive events that may occur between reads.

The drop signal, which updates the period for which the planned position signal is triggered, is detected at platform time t_0. The planned position is read at platform time t_1', at the beginning of the *max delay* interval. On the other hand, the encoder signal is read at t_2, but represents the state of the disc at platform time t_2'. We observe a large discrepancy between what the signals actually correspond to in terms of the state of the disc (we will refer to this phenomenon as the "desynchronization" of input events). From Fig. 6.4, it is easy to see the largest possible difference between the platform times of these signals is $t_2 - t_1 = p + max\ delay$, where p is the maximum period of the quasi-periodic planned position signal.

However, if the Ptides strategy is used, then in the case represented in Fig. 6.4, we can expect the controller to use the position signals sensed at times t_1 and t_3 for planned and actual positions, respectively. This is because Ptides processes events in timestamp order, and thus the scheduler would process the trajectory planning event with timestamp t_1, followed by encoder event with timestamp t_3, and finally followed by the control event, which reads trajectory data as input values.

Even though the desynchronization of input events can be clearly indicated with this example, this does not directly result in missed ball drops. The mechanics of the system can tolerate whatever error introduced from this desynchronization. However, we note that real-life scenario exists that would result in failed ball drops. Next, we give an example showing a relatively long timing delay between the detection of ball drops and production of the drop event that leads to failed ball drops.

This scenario could happen if the controller is physically far away from the sensor. Let us assume our setup is a small sub-component of a bigger system, then in order to reduce the amount of wiring in the distributed system, a separate platform would be used to parse drop sensor data. This drop sensor controller would then communicate with the controller through a shared bus such

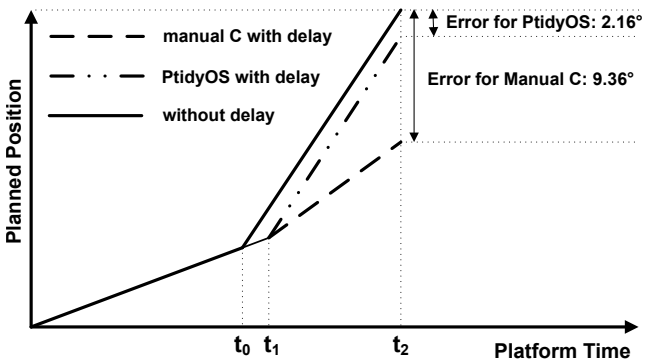

Figure 6.5: Position Errors For PtidyOS vs. Manual C Implementation

as CAN or FlexRay. This communication delay can range from microseconds to low millisecond. To reiterate, our goal here is to demonstrate if one were to design in the Ptides environment, the deterministic logical and timing properties of Ptides can ensure the original behavior of the system is maintained, even with the additional network delay. This is compared with the manual approach, where a performance degradation is observed. Note that we have not implemented a distributed control of the TBD. Instead an artificial delay of $5ms$ is introduced at the input of drop sensors using timer interrupts. I.e., when a drop event occurs at platform time t, a timer interrupt is set up to occur at $t + 5ms$.

As shown in Fig. 6.5, if there is no delay at the sensor, the new rate for updating the planned position is calculated immediately at t_0. If the drop data is delivered at t_1, the Ptides implementation would still acknowledge that the drop occurred at t_0 due to timestamping at the drop sensor platform. Also, Ptides semantics still ensures the timeToImpact and actualPosition parameters are processed in tag order. This means the calculated output will have both variables referencing the same state of disc, which allows for the correct calculation of a new disc rate (the slopes for Ptides implementation with delay is the same as the no delay case). Thus even though the Ptides implementation experiences a position error as a consequence of the sensor delay, the rate calculation is not affected, and the position error at impact time is minimized.

For the manual C implementation however, the drop data still indicates ball drop occurred at t_0, but since the manual implementation has been polling encoder inputs during the $5ms$, it would use the current position of the disc at t_1 instead of t_0. This leads to a wrong rate calculated for the manual implementation, which leads to a much larger position error.

We downloaded both the PtidyOS and manual C implementations to control the TBD, first without then with the simulated $5ms$ network delay. The nominal speed of the disc is set to 3 revolution per second (RPS).

For the Ptides implementation, a delayed sensor input results in a planned position error of 3 encoder ticks, which translates to $2.16°$. In other words, at the impact time of the ball, the planned center of the disc is $2.16°$ away from the ideal position, where a ball drop is guaranteed to be successful. This is compared to a position error of 13 encoder ticks, or $9.36°$ for the manual C implementation. This difference is large enough to cause ball drop failures. A total of 50 ball drops are performed for each implementation. The success rates for different cases of ball drops as shown

below:

Table 6.1: Ball Drop Success Rates

Manual Without Delay		PtidyOS Without Delay		Manual With Delay		PtidyOS With Delay	
96%	100%	100%	100%	40%	62%	98%	100%

Two percentages are shown for each case. The one on the left denotes percentages of ball drops that went through the hole without touching the disc, while the one on the right denotes the percentages of drops that touched the disc, but still went through the hole. i.e., a drop that touched the side of the hole but still went through would count as a success in the first case, while failure in the second. As we can see, with the added delay, the success rate of the manual implementation dropped significantly for both cases, while the success rate of the PtidyOS implementation remained virtually unchanged.

It should be noted however, that the deterministic behavior of the Ptides model is not obtained without a price. One limitation of the Ptides implementation is that each time an encoder interrupt is triggered, an event is created and inserted into the event queue. This event is then analyzed for safe-to-process. Since we expect around (500 ticks / revolution $\times$ 3 RPS =) 1500 encoder events per second, this process results in much larger scheduling overhead than the manual C implementation (which simply polls the updated encoder variable). The side effect is that the PtidyOS implementation can only handle lower disc rates. If the nominal rate is set to 8 RPS for example, the large number of encoder interrupt preemptions would result in stack overflow.

Finally, we should note that it is of course possible to modify the manual implementation such that the calculated planned position takes into account of the delay experienced by the drop sensor. Just like the PtidyOS implementation, this would minimize the position error and ensure successful ball drops. However, Ptides guarantees deterministic functional behavior regardless of the sensor delay. Once the sensor delay d_s's are known, the programmer only needs to update the Ptides model with this value, and the working implementation is generated with the click of a button. This contrasts with the manual implementation, where the programmer has to update the control code base manually in order to take this delay into account.

Chapter 7

Conclusion and Future work

7.0.4 Summary of Results

This thesis focuses on the Ptides programming model first introduced by Yang Zhao. A set of execution strategies that implements Ptides semantics are introduced and summarized. By assuming bounds on synchronization error and network delay, and by defining a relationship between model time and physical time, our strategies ensure event processing that results in deterministic logical and timing behavior, while allowing aggressive out-of-order execution without backtracking. These strategies are based on different assumptions, use different event queue orderings, and result in different complexities for their safe-to-process analyses.

The second part of this thesis focuses on the implementation of these strategies as a part of the Ptides design flow. This design flow includes a simulator, a code generator, and a real-time operating system PtidyOS. The simulator allows an application programmer to capture both the logical operation and the timing aspects of the system, without the knowledge of execution time, which is dependent on low level hardware details. The programmer can then generate a runtime implementation onto a target-specific hardware platform.

A motion control application called the tunneling ball device is implemented using the Ptides design flow. The generated PtidyOS runtime is compared to a manual C implementation. We showed that the Ptides semantics ensures the correct order of event execution, which prevents performance degradation even with a large delay introduced at the sensor. This contrasts the manual implementation, whose performance degraded significantly when the same delay is introduced.

7.0.5 Future Work

One possible future direction is to expand the Ptides programming model and enable it for soft real-time systems. Currently we have focused solely on hard real-time system, where deadline misses at the actuators are interpreted as system faults. However, real-world applications usually consist of an integration of both hard and soft real-time sub-systems. To address this issue, instead of enforcing the timestamps of actuation events as a hard deadlines, these timestamps can be mapped to an utility function, and the goal of the scheduler would be to maximize this utility function.

On the implementation side, as mentioned in Chap. 6, a distributed Ptides has not been developed. This would require an integration of a precision timing protocol such as IEEE1588 into the PtidyOS framework. Since the Ptides strategies were formulated with distributed application in

mind, it would be interesting to test a true distributed application in the Ptides environment.

Bibliography

[1] DP83640 Precision PHYTER - IEEE 1588 Precision Time Protocol Transceiver data sheet.

[2] T. Baker. A stack-based resource allocation policy for realtime processes. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 191–200. IEEE, 2002.

[3] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall, 2nd edition, 1990.

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

[5] A. Benveniste, B. Caillaud, L. Carloni, P. Caspi, A. Sangiovanni-Vincentelli, and S. Tripakis. Communication by sampling in time-sensitive distributed systems. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 152–160. ACM, 2006.

[6] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java. Technical Report Technical Memorandum UCB/ERL M04/27, University of California, July 29 2004.

[7] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 153–162. ACM, 2003.

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

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

[10] M. Chen and K. Lin. A priority ceiling protocol for multiple-instance resources. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 140–149. IEEE, 2002.

[11] J. Dabney and T. Harman. *Mastering Simulink*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.

[12] P. Derler, E. A. Lee, and S. Matic. Simulation and implementation of the Ptides programming model. In *DS-RT '08: Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 330–333, Washington, DC, USA, 2008. IEEE Computer Society.

[13] J. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou. A time-centric model for cyber-physical applications. In *Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB)*, pages 21–35, 2010.

[14] J. C. Eidson. *Measurement, Control, and Communication Using IEEE 1588*. Springer, 2006.

[15] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[16] eWind River: RTLInuxFree.

[17] T. H. Feng and E. A. Lee. Real-time distributed discrete-event execution with fault tolerance. In *Proceedings of RTAS*, St. Louis, MO, USA, April 2008.

[18] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. *EMSOFT 2001: Proceedings*, 2001.

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

[20] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[21] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, volume LNCS 2211, pages 166–184. Springer-Verlag, 2001.

[22] I. Instrumentation and M. Society. 1588: Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. Standard specification, IEEE, November 8 2002.

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

[24] J. Jensen. Elements of Model-Based Design. *University of California, Berkeley, Technical Memorandum. UCB/EECS-2010-19, February*, pages 2010–19, 2010.

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

[26] G. Johnson. *LabVIEW graphical programming: practical applications in instrumentation and control*. McGraw-Hill School Education Group, 1997.

[27] E. Kaplan and C. Hegarty. Understanding GPS: Principles and Applications Second Edition. *Artech House*, 2006.

[28] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[29] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[30] E. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

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

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

[33] E. A. Lee. Discrete event models: Getting the semantics right. In *WSC '06: Proceedings*, pages 1–1. Winter Simulation Conference, 2006.

[34] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems, October 2007.

[35] J. Lehoczky, L. Sha, J. Strosnider, et al. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[36] M. Leung. An Extensible and Retargetable Code Generation Framework for Actor Models. 2009.

[37] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. TinyOs: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005.

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

[39] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-time, theory in practice: REX Workshop, Mook, the Netherlands, June 3-7, 1991: proceedings*, page 447. Springer, 1992.

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

[41] J. Misra. Distributed discrete event simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[42] L. Muliadi. Discrete event modeling in Ptolemy II. Technical Report UCB/ERL M99/29, EECS Department, University of California, Berkeley, 1999.

[43] P. Puschner and A. Burns. A review of worst-case execution-time analyses. *REAL TIME SYSTEMS-AVENEI NJ-*, 18(2/3):115–128, 2000.

[44] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. *Readings in multimedia computing and networking*, pages 476–490, 2001.

[45] L. Sha, M. Klein, and J. Goodenough. *Rate monotonic analysis for real-time systems*. Kluwer Academic Publishers, 1991.

[46] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 2–11. IEEE, 1994.

[47] J. Stankovic. *Deadline scheduling for real-time systems: EDF and related algorithms*. Springer, 1998.

[48] The FreeRTOS Project. OpenRTOS. http://www.freertos.org.

[49] S. Thuel and J. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 22–33. IEEE, 1994.

[50] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale. Implementing synchronous models on loosely time triggered architectures. *Computers, IEEE Transactions on*, 57(10):1300–1314, 2008.

[51] Wind River. Vxworks: Embedded RTOS with suport for POSIX and SMP. http://www.windriver.com/products/vxworks/.

[52] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.

[53] Y. Zhao. *On the Design of Concurrent, Distributed Real-Time Systems*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2009.

[54] Y. Zhao, J. Liu, and E. A. Lee. A programming model for time-synchronized distributed real-time systems. In *Proceedings of RTAS 07*, pages 259–268, 2007.

[55] Y. Zhou and E. Lee. Causality interfaces for actor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–35, 2008.

[56] J. Zou, S. Matic, E. Lee, T. Feng, and P. Derler. Execution strategies for ptides, a programming model for distributed embedded systems. In *15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 77–86. IEEE, 2009.