Schedulability Analysis and Verification of Real-Time Discrete-Event Systems

Christos Stergiou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-163
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-163.html

September 30, 2013

Schedulability Analysis and Verification of Real-Time Discrete-Event Systems

by

Christos Stergiou

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

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Edward A. Lee, Chair
Associate Professor Koushik Sen
Professor David L. Wessel

Fall 2013

Schedulability Analysis and Verification of Real-Time Discrete-Event Systems

Abstract

Schedulability Analysis and Verification of Real-Time Discrete-Event Systems

by

Christos Stergiou

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Edward A. Lee, Chair

Cyber-physical systems are systems where there is a tight interaction between the computing world and the physical world. In spite of the significance of time in the dynamics of the physical world, real-time embedded software today is commonly built using programming abstractions with little or no temporal semantics. PTIDES (Programming Temporally Integrated Distributed Embedded Systems) is a programming model whose goal is to address this problem. It proposes a model-based design approach for the programming of distributed real-time embedded systems, in which the timing of real-time operations is specified as part of the model. This is accomplished by using as an underlying formalism a discrete-event (DE) model of computation, which is extended with real-time semantics.

We address the schedulability question for PTIDES programs and uniprocessor platforms. A PTIDES program is schedulable if for all legal sensor inputs, there exists a scheduling of the PTIDES components that meets all the specified deadlines. The timing specifications allowed in the discrete-event formalism can be seen as a generalization of end-to-end latencies which are usually studied in the hard real-time computing literature. This results in a rather idiosyncratic schedulability problem. We first show that for a large subset of discrete-event models, the earliest-deadline-first scheduling policy is optimal. Second, we show that all but a finite part of the infinite state space of a PTIDES execution results in demonstrably unschedulable behaviors. As a result the schedulability problem can be reduced to a finite-state reachability problem. We describe this reduction using timed automata.

We next turn to the verification problem for DE systems themselves. We study a basic deterministic DE model, where actors are simple constant-delay components, and two extensions of it: first, one where actors are not constant but introduce non-deterministic delays, and second, one where actors are either deterministic delays or are specified using timed automata. We investigate verification questions on DE models and examine expressiveness relationships between the DE models and timed automata.

Finally, we discuss extensions of this work to cover a wider range of discrete-event actors, and propose an approach to design more efficient and practical analyses for schedulability testing.

To my parents.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisors Edward A. Lee and Koushik Sen. They have both been great mentors and teachers and I am grateful for their guidance during my studies. I would also like to thank Sanjit Seshia and David Wessel for serving in my Quals and dissertation committee.

I am indebted to all my collaborators and coauthors: Chang-Seo Park, Jacob Burnim, Nick Jalbert, Roberto Lublinerman, Stavros Tripakis, Chris Shaver, Eleftherios Matsikoudis, and Manfred Broy. I learned a lot by working with all of them. Special thanks go to Eleftherios Matsikoudis, with whom I worked during the last years of my PhD on most of the topics presented in this thesis, and Stavros Tripakis, who has not only been a great collaborator but also a friend and mentor.

I am grateful to all the members of the PTIDES project and specifically Patricia Derler, John Eidson, Slobodan Matic, and Jia Zou for all their help in understanding PTIDES during my first days in Edward's group and throughout my graduate studies.

I have also been a proud member of the Warm Lab. I was lucky to share office space with Dai Bui, Patricia Derler, and Chris Shaver, and I will miss our long discussions. I was also lucky to have lived for most of my graduate studies at 1780 Spruce with George, Kostas, and Anthony.

Last but not least, I would like to thank my father, my mother, and my brother for all their love and support.

Chapter 1

Introduction

One of the defining features of embedded software, and one setting it apart from general-purpose computing, is its intimate relationship with the notion of time. Embedded programs are expected to interact with an inherently timed physical world, and are thus required to embrace time as part of their semantics. This is perhaps best exemplified by computer-control systems, which are required to sense the physical world and act upon it, all in a timely manner.

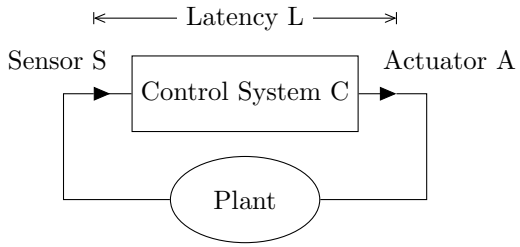


Figure 1.1: Simple control system.

A computer control system can be typically represented as a diagram of the form depicted in Figure 1.1. A sensor is producing measurements of the environment at certain instances of time. Each measurement is processed, and in response, an action on the environment is generated, and handed to an actuator responsible for carrying it out. What is often critical is that each action be delivered to the actuator within a certain time interval beginning from the time of the corresponding measurement. This interval is often chosen based on the control algorithm in use, and the control designer will often try to enforce it by specifying a certain desired latency from a measurement at the sensor to a response to it from the actuator. The system engineer is then responsible for implementing the system in such a way that the latency requirement is met. Assuming a uniprocessor platform, a minimum separation time between measurements S, a latency specification L, and a worst-case computation time W required for processing a measurement, the latency requirement can be met if $W \leq L$ and $W \leq S$.

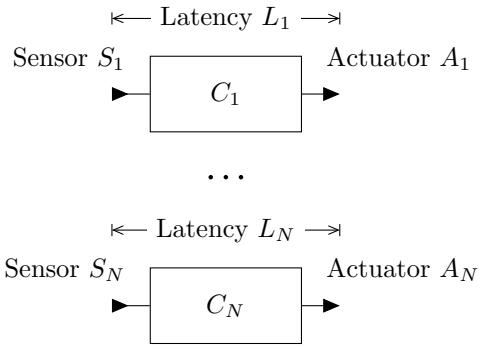


Figure 1.2: Multiple sensor-actuator paths.

A control system can include multiple sensors and actuator paths. In keeping the structure simple, we extend the previous system with more sensor and actuator pairs, each of which is assigned a separate control law, so the system looks like Figure 1.2. In the case of a uniprocessor platform, the job of the engineer is a little harder now, because the computing resources of the system are shared among the different sensor-actuator paths, which must be scheduled in a way that ensures that all different latency requirements are satisfied. If the measurements are periodic or sporadic, i.e., there is a minimum interval between two successive measurements, one can rely on classical hard real-time scheduling theory (e.g., see [4]).

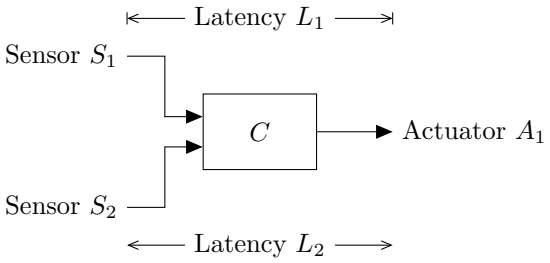


Figure 1.3: Merging paths.

Our control system examples are still, however, not very realistic. In a real system, measurements from different sensors are likely to be combined to control a single or multiple actuators. A simple instance of that pattern is shown in 1.3. Here measurements taken by sensors S_1 and S_2 are processed by computing element C which produces commands for actuator A while respecting the individual latencies L_1 and L_2. To be concrete, we assume that every measurement results in an actuator command, with the exception of measurements taken at the same time from both sensors, in which case we assume that the measurements are combined and result in a single command. To make this example interesting, we also consider element C to carry state.

Assume without loss of generality that $L_1 > L_2$. Suppose that S_1 produces a measurement at time t_1 and that component C eagerly starts processing that measurement. Now, suppose

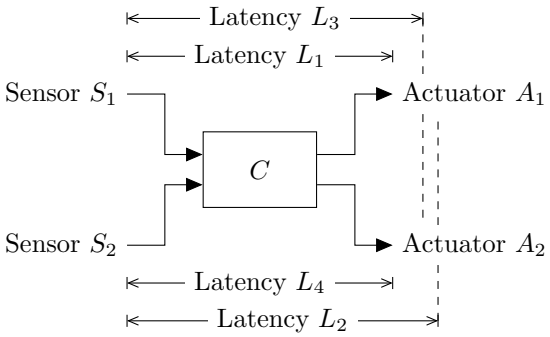
that S_2 later produces a measurement m_2 at a time t_2 such that $t_2 < t_1 + L_1 - L_2$. In other words, suppose that S_2 produces a measurement that according to the latency specifications will result in an earlier actuation of A. The decision to process m_1 eagerly can have several unwanted consequences at this point. If the processing of m_1 has finished or it continues to completion, then the order of the actuation times of the two measurements and the order of their processing by C, and hence the order with which they affect the state of C, will be reversed. We argue that this inversion is wrong. The effect on the state of C should obey the same order as the effect that the measurements have on the physical world. Note here that on a uniprocessor platform and assuming that the execution of C includes state mutation, it is not trivial to preempt the execution of m_1 when m_2 becomes available. If no concurrency control is used, then the state of C might be in an inconsistent state. If locks or some other mechanism are used, then processing m_2 cannot start until the processing of m_1 reaches a preemption point, and any state snapshot and rollback mechanism used to immediately start anew with processing m_2 is likely to be expensive. Therefore, if m_1 is still being processed at time t_2, the system will be in a situation where it is processing a measurement with a larger deadline, $t_1 + L_1$, than another available measurement.

In summary, the relative order of actuation times should translate into a processing order on shared components. And therefore, assuming that m_2 can be produced at any time t_2 between t_1 and $t_1 + (L_1 - L_2)$, m_1 cannot be processed until it is possible to guarantee that $t_1 + L_1 \leq t_2 + L_2$, which is not until $t_1 + (L_1 - L_2)$.

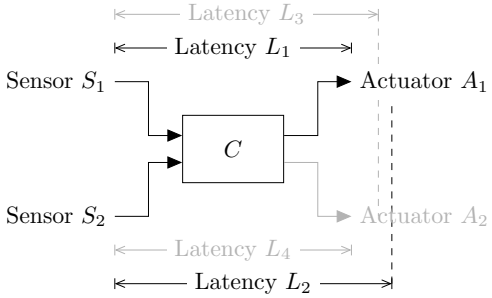
What if we try to extend this line of reasoning to more complex structures, such as the system with two sensors and two actuators as shown in Figure 1.4a? Four latency specifications are available in this system, one for each pair of sensor and actuator. However, not all possible latency valuations are sensible. For instance, assume that $L_1 > L_2$ and that sensors S_1 and S_2 each take a measurement, m_1 and m_2, at times t_1 and t_2 such that $t_1 < t_2 < t_1 + (L_1 - L_2)$. In which order should C process the two measurements? Because $t_2 < t_1 + (L_1 - L_2)$, from the perspective of actuator A_1, see Figure 1.4b, m_2 should be processed before m_1. But then if $t_2 > t_1 + (L_3 - L_4)$, or if $L_1 - L_2 > L_3 - L_4$, from the perspective of actuator A_2, see Figure 1.4c, m_1 should be processed before m_2. So, it has to be $L_1 - L_2 \leq L_3 - L_4$ for the two actuators to agree. The last inequality together with the initial assumption that $L_1 > L_2$ imply further that $L_3 - L_4 > 0$ or $L_3 > L_4$. Applying the symmetric argument we get that it has to be $L_3 - L_4 \leq L_1 - L_2$. Hence, for a latency assignment to define a consistent ordering, it has to satisfy the equation $L_1 - L_2 = L_3 - L_4$.

What this means is that the designer is expected to either come up with a real-time specification in terms of a number of dependent variables or accept non-deterministic behavior. Both are rather unappealing and impractical ways to design systems that are unlikely to scale well on yet more complex structures.

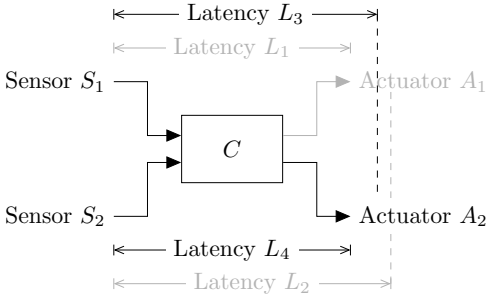
Luckily, there is a very natural solution to this problem. The way to go from dependent to independent variables is to forgo the notion of a path latency in favour of a notion of a link delay. For example, to return to the system in Figure 1.3, one may specify a delay D_1 on the link from S_1 to the processing component C, a delay D_2 on the link from S_2 to C, and a delay D_3 on the link from C to A, as in Figure 1.5a, and by setting $D_1 = L_1 - L_2$, $D_2 = 0$, and



(a) Sensor and actuator latency specifications.

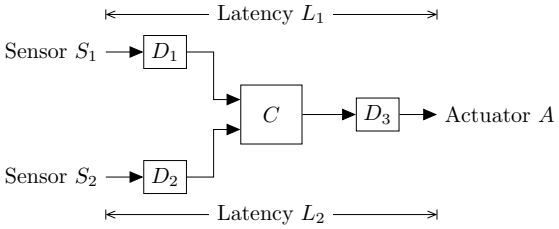


(b) Actuator A_1 perspective.

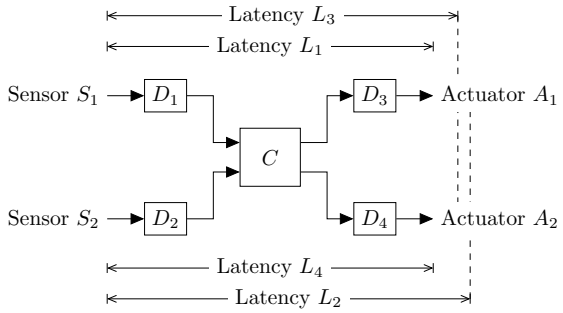


(c) Actuator A_2 perspective.

Figure 1.4: Multiple sensors and actuators.



(a) Merging paths with delays.



(b) Multiple sensors and actuators with delays.

Figure 1.5: Link delay specifications.

$D_3 = L_2$, one can prescribe the same end-to-end latencies as before. And one can populate the links of the system in Figure 1.4a with any arbitrary choice of delays without any fear of ending up with nonsensical specifications. Indeed, according to the link-delay specification of Figure 1.5b, the difference between L_1 and L_2 will be $(D_1 + D_3) - (D_2 + D_3) = D_1 - D_2$, and that between L_3 and L_4, $(D_1 + D_4) - (D_2 + D_4) = D_1 - D_2$. Hence, by construction, the end-to-end latencies will satisfy the consistency constraint.

Once liberated from the notion of path latency, the designer may even begin to think of

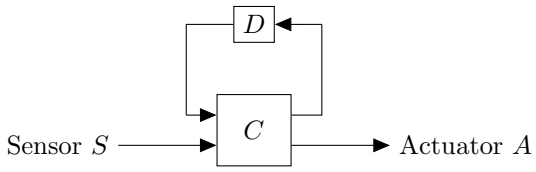


Figure 1.6: Feeback loop.

systems with feedback loops, as in Figure 1.6, where end-to-end latency specifications are at the very least incomplete, if not ambiguous.

Interestingly, it is now possible to come up with the same latency specification in more than one way. For example, in Figure 1.5a, one may specify the same end-to-end latencies as before by setting $D_1 = L_1$, $D_2 = L_2$, and $D_3 = 0$. This raises the question of what the semantics of a link-delay specification is, and whether different delay specifications corresponding to identical latency specifications should be implemented differently by the system engineer.

One way to address this question is by making use of the above observation that the relative order of actuation times translates into a processing order on shared components. Of course, in order to resolve processing decisions at shared components deep inside the system, one would need to keep track of the paths traversed by the arriving tokens. And especially in the case of loops, this can become quite complicated. But there is an equivalent, and we believe, more natural interpretation.

We may think of processing as a logical operation that takes no time, and transmission over a link as an operation that takes time equal to the delay associated with that link. For example, in the system of Figure 1.5a, a measurement at S_1 at time t is not available for processing at C until $t + D_1$, at which time it can be safely processed, along with any measurement arriving at that time through the link from S_2. This leads naturally to a programming model according to which programs are represented as block diagrams, and blocks correspond to components, or so-called "actors", consuming and producing timestamped tokens, or so-called "events", conceptually ordered according to their timestamps. What we end up with is a discrete-event model of computation that is not used for modeling and simulation (e.g., see [9]), but for real-time programming; this real-time discrete-event model of computation is called PTIDES, introduced by Zhao et al. [33]. Semantically then, different delay specifications correspond to different programs. However, the notion of time imparted by the timestamps is not a physical one, but a logical one. And while the designer is free to think of this as a timed programming model, and think of the components of the control algorithm and the physical world as living in the same space, and sharing the same temporal semantics, this is not required for the engineer. A measurement is initially timestamped by the sensor with the actual time at which it was taken. It is then communicated between actors, which may increase the timestamp by some arbitrary but fixed amount before forwarding to the next actor, until it reaches an actuator, where the timestamp is interpreted as the actual time when actuation is to occur. It does not really matter when events are

processed, as long as they are processed in timestamp order. And the job of the engineer is to make sure that every event reaching an actuator is delivered before the time corresponding to its timestamp.

This amounts to a rather complex schedulability problem that has not been considered before in the hard real-time computing literature. A typical approach to schedulability problems is to reduce the problem of scheduling every possible scenario to the problem of scheduling a single "worst-case" scenario. And that "worst-case" scenario typically corresponds to a job-arrival pattern that maximizes processor demand over a certain time window. But in our case, there is another factor that may contribute to a bad scenario, i.e., the requirement to satisfy the timestamp order of processing. And the two pull in opposite directions. In fact, in section 2.5 we will show, using two example PTIDES programs, that there are cases where the first strategy is successful in producing a deadline violation whereas the second is not, and vice versa.

Another source of complexity is the expressiveness of the proposed programming model. The ability to design systems with feedback loops comes with a cost, namely the possibility of unbounded accumulation of events with unbounded timestamps circulating in these feedback loops, and thus, it is not immediately obvious that the schedulability problem is even decidable. In fact, the discrete-event model of computation is Turing-complete.

The purpose of this work is to address this schedulability problem for a uniprocessor system [26]. Because of the above complications, it is unlikely that an analytical solution exists. Here, we consider an algorithmic solution. We introduce a detailed formalization of the programming model, prove that the earliest-deadline-first scheduling policy is optimal, and show that the schedulability problem can be reduced to a finite-state reachability problem. Finally, we describe how to carry out this reduction using timed automata. The formalism of timed automata is not intrinsic to our solution, but rather a convenient tool to seamlessly integrate system abstraction with different, and complex, input-event arrival models.

Finally, we study the verification problem of traditional discrete-event programs by applying similar techniques as the ones used for bounding the state-space of PTIDES executions [30]. Starting from a model with constant-delay actors, we explore two different extensions, one where the delay is non-deterministic, and one where actors can also be described using timed-automata. We discuss the differences in expressiveness of the above models and compare them to timed automata.

1.1 Related work

The use of time as a programming abstraction is not a new idea. Lee et al. [22] motivated the incorporation of timing constraints into real-time programs and provided a taxonomy of features that need to be integrated into a high-level languages that supports time as a first class citizen. Real-Time Java [6], Ada [28], and TimeC [23] are examples of existing languages that have been extended to allow for the specification of real-time constraints.

Furthermore, Simulink and LabVIEW are programming frameworks for embedded systems that also support the development of timed systems.

Synchronous languages (e.g, see [5]) have also been used for programming real-time systems. However, their approach is different than the one presented here, in that latencies arise from the implementation instead of being part of the programming abstraction.

Our programming model belongs in the family of logically execution time based models that was pioneered by Giotto [17]. A significant difference between Giotto and our case is the fact that the former is time-triggered. That distinction is moderated with xGiotto [16], which is an event-triggered extension of Giotto that, however, does not allow for the specification of relative deadlines on events.

There has been a vast amount of work on the topic of the schedulability of real-time systems. In a seminal paper [25] for the field, Liu and Layland study systems of periodic tasks. They prove the optimality of the rate monotonic scheduling policy among fixed priority policies, and introduce the notions of the utilization factor of a task and utilization of a system. They provide a simple schedulability test for the case of the EDF scheduling policy. Their task model restricts tasks to have deadlines equal to their period. Leung and Whitehead [24] lift that constraint and extend the model of Liu and Layland by allowing tasks to have deadlines that are independent of their periods. They show that the deadline monotonic scheduling policy is optimal over fixed priority policies. In the mean time, Dertouzos [12] shows that the EDF algorithm is optimal in terms of feasibility: if a task system is schedulable, then it will be schedulable under EDF. Mok [27] introduces sporadic tasks to model external interrupts of a system. Baruah et al. [4] give a pseudo-polynomial sufficient and necessary test for the schedulability of sporadic task systems with task deadlines independent of their periods. To do that they introduce the concept of the demand bound function.

More recently, we can identify two directions in the work on schedulability that is relevant to PTIDES. The split is related on how much focus is placed on the efficiency of the resulting feasibility test. On the one hand, there is work that focuses on extending the task model while insisting on pseudo-polynomial feasibility tests and in most cases sporadic inputs. The other direction explores the expressiveness of both the input patterns and the task model and is based on timed automata.

In the first direction, the digraph model [31], generalized multiframe tasks [2], and the recurring real-time task model [3] are pushing the boundary of expressiveness with the ability to model different job types, conditional execution, and looping structures. However, all these models rely on the task independence assumption in order to offer tractability (see Note 1 in [3] and pages 8-10 in [2]). Specifically, they cannot accurately model systems where the execution of different tasks depend on each other, which is the case in PTIDES programs.

In the other direction, the use of timed automata for schedulability of real-time task systems begins by Norström et al. [29]. Timed automata allow for more expressive task arrival patterns that go beyond the previously studied sporadic and periodic task systems. [29] is restricted in that it does not allow for preemption. Fersman et al. [15, 14] extend the previous results to preemptive task systems. Schedulability there is checked algorithmically

via reduction to a decidable subclass of updatable automata. Asynchronous processes are bound to timed automata locations thereby allowing for considerable expressiveness in the task arrival and dependency patterns.

Finally, Yang Zhao in her thesis [32] also investigated the schedulability problem of PTIDES systems. She distinguished between actors that have a sporadic output given sporadic inputs to the model and those that might not. For example, an actor that merges to sporadic input streams is not sporadic since there can be no lower bound on the distance between two successive events on the output of such an actor. The outcome of her investigation was a sufficient schedulability test for PTIDES programs that are scheduled using EDF and in which actor loops contain solely sporadic actors.

Chapter 2

Real-Time Discrete-Event Systems

We have introduced PTIDES as a programming model for real-time systems. We saw that it relies on a discrete-event formalism which we motivated as a means to generalize end-to-end latency specifications in real-time applications. In this chapter, we will delve into more details on how the discrete-event model of computation is extended with real-time semantics, the mechanisms with which the PTIDES framework stays faithful to discrete-event semantics, and the real-time features of it. This introduction to PTIDES will be kept in an informal level and guided by examples, so that the main ideas are fully understood before we delve into the formalization of the programming model in the next chapter. Finally, in the last section, we will provide more intuition on the schedulability problem through examples.

Note to reader: The PTIDES programming model was introduced in 2007 by Zhao et al. [33]. Since then, two theses and several papers have been written on it. We find that the formalism as well as the manner in which the ideas are introduced have evolved as well. Initial work on PTIDES, e.g., [33] and [32], follows a denotational approach, closer to the work of denotational semantics for discrete-event systems, whereas, later work, e.g., [34], [35], and [13] follows a more operational approach. The introduction of PTIDES of this chapter is similar to the later work, in that there is a focus on events rather than signals.

2.1 Discrete-Event Programs

The discrete-event model of computation is an actor oriented model. Components are actors that communicate by events, which are timestamped values. Actors can fire in response to a stimulus or can be source actors with no input channels. For example, in Figure 2.1, the clock actor, at the top left, is a source actor that generates events with period P, starting with timestamp t_0. The value of those events could vary as well, however, in this example it is constant and equal to v.

A common type of discrete-event actor is one that only introduces a time delay. Actors D_1 and D_2 in Figure 2.1 are time-delay actors. When a time-delay actor processes an event

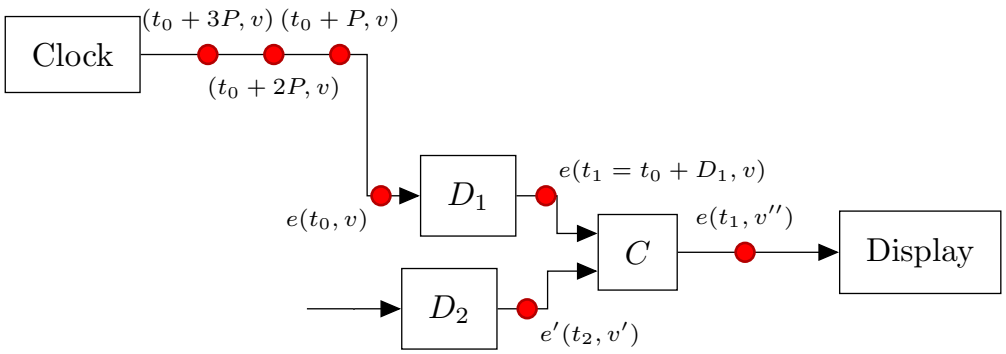


Figure 2.1: Example discrete-event program.

with timestamp t and value v, it produces an event with timestamp $t + D$, where D is a parameter of the actor, and value v. So a time-delay actor adds a delay on the timestamp of its input events without changing their values. In the following, we will use D_i to name time-delay actors, and interchangeably use D_i to refer to both the actor and its delay parameter. Note that in the example, in response to input $e(t_0, v)$, actor D_1 produces event $e(t_0 + D_1, v)$.

Actor C in the example represents is a generic computation actor. This actor processes its inputs and does not modify their timestamp. A constraint enforced by the discrete-event model of computation is that actors process their inputs in timestamp order. Actor C, in this example, is presented with events in both input channels, events e and e' with timestamps t_1 and t_2 respectively. If $t_1 < t_2$ then C has to process e first and then e'. The timestamp order requirement is important for guaranteeing that the execution of a discrete-event model is deterministic in the context of stateful (but always deterministic) actors where the processing order could result in different outputs. Furthermore, it agrees with the idea that timestamps represent a logical notion of time. In order to guarantee the timestamp order requirement, discrete-event frameworks usually maintain an ordered global event-queue, which collects all events in an execution. If at every step, the event with the smallest timestamp in the queue is chosen to process, then the timestamp order is guaranteed across all actors in the program. Note that this is more than what is required for determinism, which per-actor timestamp order execution.

Last, there are also actors that have no output channels. These consume events and do not produce any events as a result. An example is the Display actor shown in Figure 2.1.

2.2 Sensors and Actuators

PTIDES is based on the discrete-event model of computation to provide a framework for programming real-time systems. The inputs in a real-time platform are provided through sensors. The outputs of a real-time system are actuators. Sensors take measurements of the physical world or the plant that is being controlled. The real-time system then processes those measurements and produces commands to affect the physical world or control the

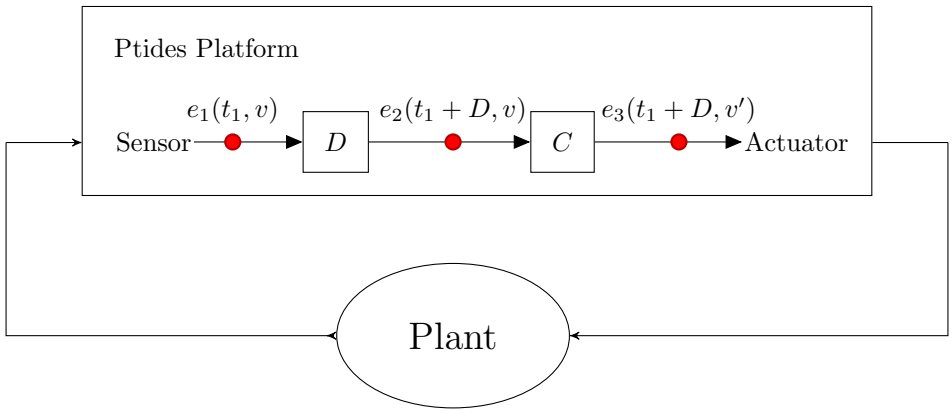


Figure 2.2: Sensors and actuators in PTIDES.

plant. PTIDES assigns real-time semantics on a discrete-event program on the boundaries of the program with the physical world, or in the sensors and actuators.

Therefore, the inputs of a PTIDES platform will be provided by sensors: each measurement a sensor takes will result in a new event at a channel of the platform dedicated to that sensor. The value of that event will be equal to the measurement. The question that arises is what should its timestamp be? Again a natural answer is that the timestamp of a new sensor event should be equal to the time that the measurement was taken. So we assume that if a sensor takes a measurement at time t, an event with timestamp t is produced and appears inside the platform at the same time. Note that this is obviously a simplification. Between the time that the sensor takes the measurement and the time that the event appears in the platform there will be some delay. If τ is the timestamp of the new event and t the time that it enters the ptides platform we can be sure that $t > \tau$. This delay is principally due to the execution time of the sensor device driver, therefore a bound on the difference $t - \tau$ can be computed and in fact the PTIDES simulation environment, see [11], allows the designer to provide such a bound.

Symmetrically, the outputs of a PTIDES program will be inputs to the actuator devices of the platform. The timestamp that an event has when it reaches an actuator is interpreted as a specification of when the actuator should react or actuate. This might be less intuitive than the assignment of real-time semantics to timestamps at sensors: another natural interpretation would be that the timestamp of an event at an actuator is treated solely as a deadline, but the actuation occurs as soon as possible. The benefit of the PTIDES approach is the guarantee that if the inputs to a PTIDES platform are the same and all the deadlines are met, because of the determinism of the underlying discrete-event formalism, the outputs of the platform will also be the same, both in their values and in the time that they are produced. This determinism is one of the most important features of PTIDES: it allows for real-time systems to be programmed in modular way and independently of the hardware platform.

Figure 2.2 shows a simple PTIDES program used as a platform to control a plant. The

sensor takes a measurement at time t_1 with value v, and the event $e_1(t_1, v)$ encapsulates that measurement and appears in the platform at the same time. Next, the time-delay actor D processes event e_1, increases its timestamp by D, and produces event $e_2(t_1 + D, v)$. Next, the computation actor process e_2 and produces $e_3(t_1 + D, v')$ at its output channel which is the actuator input. At this point, real-time will be equal to t_1, the time the measurement is taken, plus the time that the processing of actors D and C required. If the real-time when e_3 is produced is less than or equal to $t_1 + D$ then when real-time becomes equal to $t_1 + D$ the actuator will produce the effect described by the value of e_3, v', to the plant. If the real-time when e_3 is produced is greater than $t_1 + D$, then that "command" is no longer valid, its time has passed, or a deadline has been missed.

2.3 Safe-to-process

We mentioned in section 2.1 that it is fairly easy to guarantee that actors fire in timestamp order in an execution of a discrete-event program. Specifically, one solution is to maintain a global event queue and always choose as the next event to process the smallest event in that queue. Can we adapt this solution to also guarantee timestamp order in a PTIDES program? The algorithm relies on the fact that in a discrete-event program all sources of events are visible and their events are available in the queue. However, in a PTIDES program, the source of events is the environment of the platform, and hence it is not known in advance when new events will be produced.

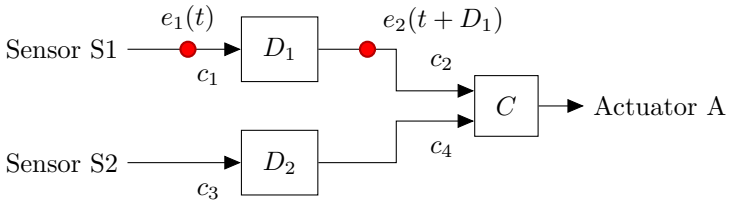


Figure 2.3: Safe-to-process example.

We look at the example program in Figure 2.3. Assume that sensor S1 takes a measurement at time t and produces an event e_1 with timestamp t at the input channel of the delay actor D_1. First, note that after time t no event with timestamp smaller than t will ever appear that the input of D_1, since events at sensor channels are timestamped with the real-time that the corresponding measurement is taken. Therefore, assuming that no other events are pending at the input of D_1, we are sure that if D_1 immediately processes e, timestamp order for D_1 will not be violated.

Event e_1 is processed by delay actor D_1 which produces a new event e_2 with timestamp $t + D_1$ at the top input channel of the computation actor C. Now, even though we are sure that no event with smaller timestamp than e_2 will appear at channel c_2, can we say the same about channel c_4, the other input channel of C? Two checks are necessary. First, we need to

check if any events currently in the system can have a smaller timestamp when they arrive at C and, second, we need to check if any events that might arrive in the platform in the future can have a smaller timestamp when they arrive at C. The first question translates to inspecting channel c_4 for events with timestamp less than $t + D_1$ and channel c_3 for events with timestamp less than $t + D_1 - D_2$. To answer the second check, we need to wait for real-time to be larger than $t + D_1 - D_2$. Only then we can be sure that any new event that appears at c_3 from sensor S2 will have timestamp larger than $t + D_1 - D_2$ and hence timestamp larger than $t + D_1$ when it arrives to c_4.

This analysis that figures out if processing an event could lead to violation of the timestamp order property is called *safe-to-process analysis*. A more elaborate treatment of the ideas discussed here can be found in [35].

Lastly, note that the way we described it here, the safe-to-process analysis is input agnostic, meaning it does not use any information on the input pattern of sensors. If more information about the way inputs arrive at sensors is available, then the analysis can be made more efficient. For example, in the case of Figure 2.3, if we knew that sensor S2 is a sporadic with minimum interarrival time greater than D_2 and moreover that it last fired at time t, then the second check would not be necessary.

2.4 Scheduling Policy

We continue the introduction of the PTIDES programming model by discussing how the next event to process is chosen. Whereas, in a discrete-event execution, it was sufficient to always choose the event with the smallest timestamp, the real-time semantics of PTIDES add an extra constraint. Since the timestamp of an event at an actuator is treated as a specification of when the actuation should occur, we would like every event to reach an actuator earlier than the time specified in its timestamp. Otherwise, because we cannot actuate in the past, the event will have missed its deadline.

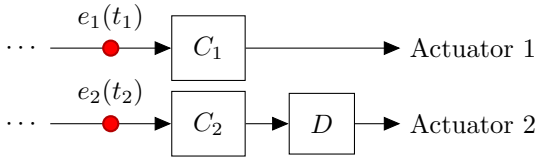


Figure 2.4: Scheduling policy example.

Assume for example that the execution of a PTIDES program is in the state shown in Figure 2.4. The fraction of the program shown contains two actor paths to actuators 1 and 2. The first consists of a single actor, C_1, that does not change the timestamp of the events it processes. Actors C_1 and C_2 are computation actors that do not change the timestamps of the events they process. Actor D adds delay D to the timestamps of the events it processes. Event e_1 in the Figure, which has timestamp t_1, will be processed by C_1, and when it reaches

Actuator 1, its timestamp will still be equal to t_1. On the other hand, event e_2, which has timestamp t_2, will be processed by actors C_2 and D, and when it reaches Actuator 2, its timestamp will be $t_2 + D$. Therefore, if our goal is to produce valid actuation commands at the actuators, in contrast to a discrete-event execution, the real comparison we should look at is not that between t_1 and t_2, but rather the comparison between t_1 and $t_2 + D$.

In summary, in PTIDES we associate with each event a notion of deadline, equal to the timestamp of the event when it reaches an actuator. If the event might reach multiple actuators, the minimum of the corresponding timestamps is chosen. For example, in Figure 2.5, the deadline of e_1 is t_1 and the deadline of e_2 is t_2.

Despite the fact that we have hinted at an earliest-deadline-first as being the natural scheduling policy, other policies might be appropriate according to the application. For instance, it might be the case that we know that it is fine for some of the actuators to miss their deadlines, while for others it is not. In that case, a fixed priority policy might be more fitting. In Chapter 4 we will actually prove that the earliest-deadline-first policy with preemption is optimal with regards to feasibility for PTIDES programs executing on a uniprocessor platform. Preemption here refers to the mechanism that suspends the execution of an actor if an event that has a smaller deadline becomes available during its execution.

In general, the algorithm for scheduling events works as follows: first, we take out events that are not safe to process, since processing those could lead to a violation of the timestamp order property, and second, choose an event based on the chosen scheduling policy. For example, in Figure 2.5, there are three events in the system, e_1, e_2, and e_3. Assuming that e_1 is not safe to process, the first step of the process would remove e_1 from consideration, and the second step would choose the next event to process between e_2 and e_3.

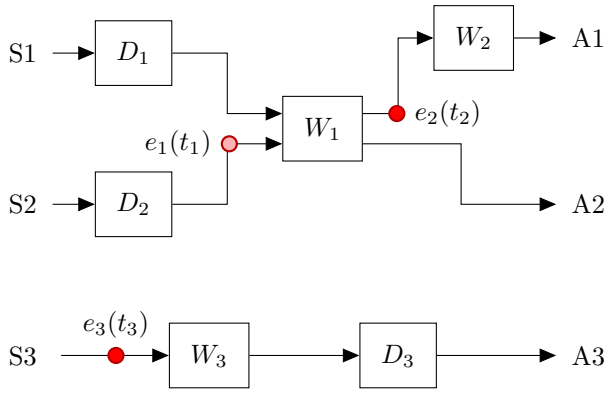


Figure 2.5: Scheduling pending events.

Note that the definition of deadline relies on two simplifications. First, it assumes that every event will eventually reach the actuator. However, an actor is not required to always produce an output in response to processing an event. It could simply consume the event and update its state accordingly. Second, the effective use of this deadline notion further requires to be able to predict for every event what its timestamp will be when it reaches an

actuator. This might not be trivial in the case that there are actors whose modification on the timestamp of the events they process depends on their state or on the value of those events.

2.5 Schedulability

As discussed earlier, a PTIDES program can be compiled and executed on various platforms. In fact, the PTIDES toolchain includes a code generation framework that has been used to compile and execute programs on various microprocessors. The details of the framework and experiments can be found in previous work [10].

The execution of an actor on any specific platform will take up processor time. A worst-case execution time can be computed for a combination of microprocessor and actor. Another characteristic of a computing platform is the input model that describes the behavior of the sensors. A sensor could take measurements periodically, it could sense events sporadically, which means there is a minimum interarrival time between measurements, or it could allow for possible bursts of measurements. Roughly, the schedulability problem amounts to, given that platform specification, whether it can be guaranteed that the every event in a program execution will meet its deadline.

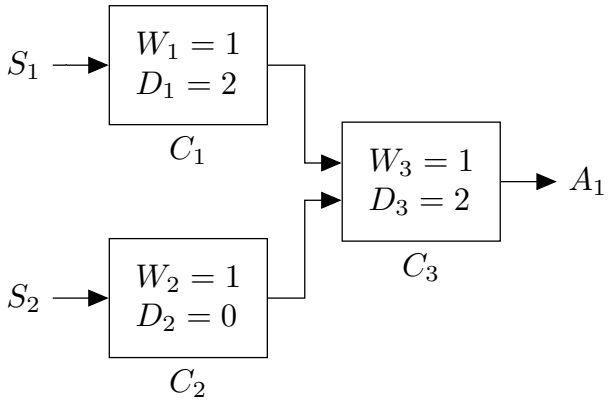


Figure 2.6: Program that misses deadline due to safe-to-process delay.

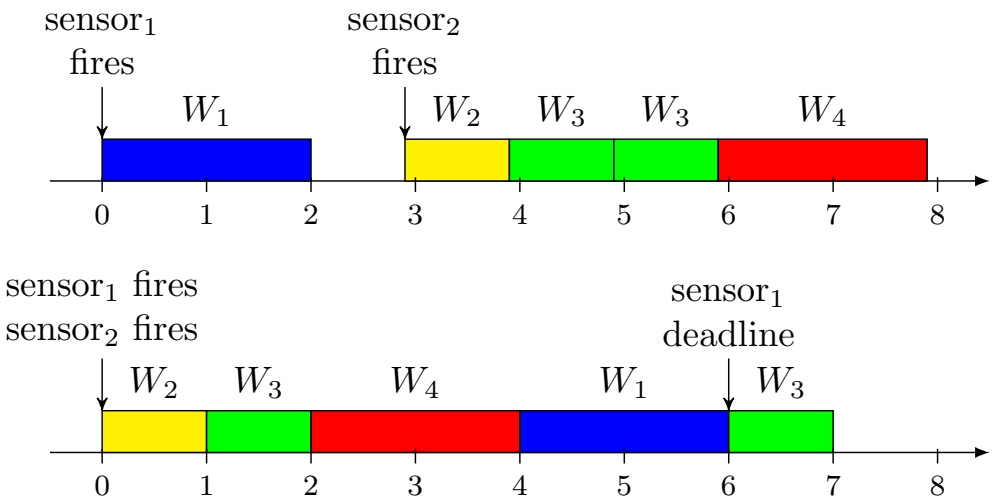


Figure 2.7: Schedule of program in Figure 2.6 with deadline miss due to safe-to-process delay.

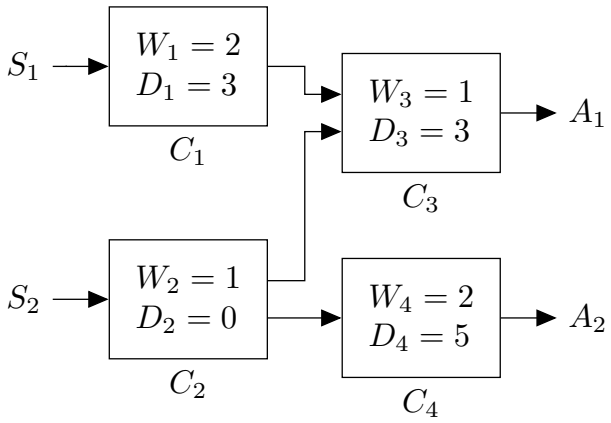


Figure 2.8: Program that misses deadline due to computation overlap.

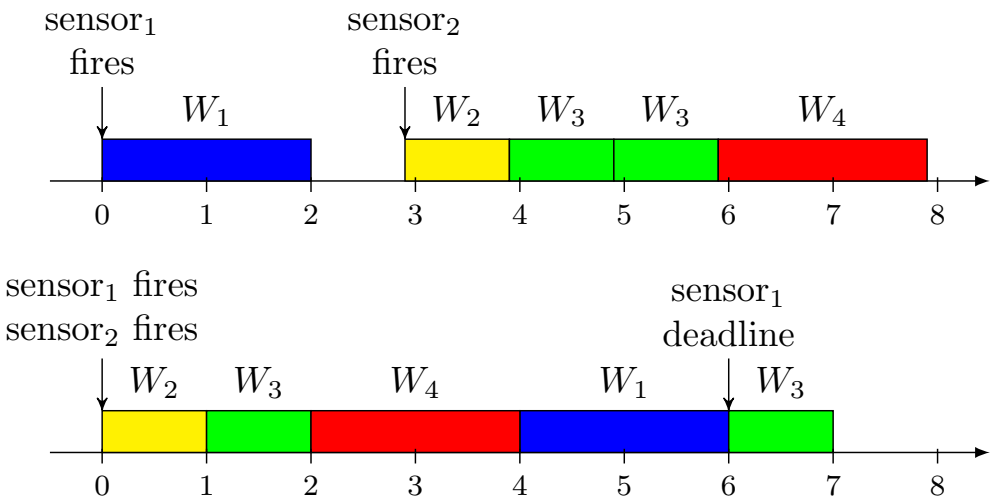


Figure 2.9: Schedule of program in Figure 2.8 with deadline miss due to computation overlap.

A typical approach to schedulability problems in hard real-time theory, is to reduce the problem of scheduling every possible scenario to the problem of scheduling a single "worst-case" scenario. Usually this worst-case scenario corresponds to a job-arrival pattern that maximizes processor demand over a certain time window. However, in our case, another factor that may contribute to a bad scenario, is the delay for events to become safe to process.

For example, consider the system in Figure 2.6, where each actor is annotated with a worst-case execution time and a delay added to the timestamps of processed events. Suppose that both sensors produce events at time 0, as in the schedule in top part of Figure 2.7. This is a scenario that maximizes processor demand, and yet is perfectly schedulable. Now, suppose that, instead, S_1 produces an event e_1 at time 0, whereas S_2 produces an event e_2 at time $2 - \epsilon$ for some small ϵ, as in the bottom part of Figure 2.7. Then C_1 will begin processing e_1 at time 0, and having a worst-case execution time $W = 1$ and a delay $D = 2$, will produce an event e_1' at time 1 with timestamp 2. But since C_2 has a delay 0, e_1' cannot be processed by C_3 until real time reaches the timestamp of that event, namely 2, lest there be another event produced by S_2 at some time t between 1 and 2, and thus, another one by C_2 with timestamp t. Thus, the system remains idle for almost a unit of time, in order to make sure that events are safely processed in timestamp order. And after C_2 produces e_2' in response to e_2, C_3 will have to process e_2' before e_1', since e_2' has a timestamp of $2-\epsilon$ whereas e_1' a timestamp of 2. This causes the deadline of e_1' to be missed. The reverse situation is exhibited by the system of Figure 2.8, where instead, trying to maximize the time wasted waiting for events to become safe to process does not yield the worst-case scenario.

Chapter 3

PTIDES Formalization

In this chapter we provide a formalization for the PTIDES programming model. Specifically, we formally define *programs* that model discrete-event programs and *systems* that model programs executed on a specific platform. We also formally define execution of programs and executions of systems, and describe how the two are related.

We will focus on discrete-event programs with actors that introduce a constant delay and always produce outputs in response to processing an event. For such discrete-event programs, the schedulability analysis presented in Chapter 4 will be necessary and sufficient. If actors are allowed to not produce outputs, then the analysis remains sufficient, however, we will not discuss its accuracy.

3.1 Events and signals

We use $\mathbb{T}$ to represent our time domain. Even though we could generalize $\mathbb{T}$ to some algebraic structure with at least an addition operation and a total ordering relation, for simplicity, we fix it here to be the set of positive real numbers $\mathbb{R}_{\geq 0}$. This choice is motivated by the reduction to timed automata in 4. There, clock variables that are real valued are used to track the passage of time. In order to formalize our notion of event, we further postulate an infinite set $\mathbb{C}$ of *channels*, and an infinite set $\mathbb{V}$ of values.

Definition 3.1.1. A *sort* is a nonempty finite subset of $\mathbb{C}$.

Sorts will serve to represent the interfaces of actors, and facilitate their composition into programs. We will use C to range over sorts.

Definition 3.1.2. A *program event* of sort C is an ordered triple $\langle c, t, v\rangle \in C \times \mathbb{T} \times \mathbb{V}$.

We write $\mathrm{E}^{\mathrm{prog}}(C)$ for the set of all program events of sort C.
Assume a program event $e = \langle c, t, v\rangle$.
We write $\mathsf{chan}(e)$ for c, $\mathsf{time}(e)$ for t, and $\mathsf{val}(e)$ for v.

Definition 3.1.3. A subset of $\mathrm{E}^{\mathrm{prog}}(C)$, s, is a *program signal* if for every $t \in \mathbb{T}$ and every $c \in C$ there is at most one $e \in s$ such that $\mathsf{time}(e) = t$ and $\mathsf{chan}(e) = c$.

Our program signals are related to the signals of the tagged signal model [19]. There, a signal is a subset of $T \times V$,where T is a set of tags and V a set of values. We can view a program signal of sort C as a signal in the tagged signal model if we set the set of tags T to be our time domain $\mathbb{T}$, and the set of values V to be the set of nonempty functions from a subset of C to $\mathbb{V}$. In other words, the value of our signals at a time instant where they are defined, if they interpreted in the tagged signal model, is a map from channels to values.

We write $\mathrm{Sig}(C)$ for the set of all program signals of sort C, and $\mathrm{Sig}_{\mathrm{fin}}(C)$ for the set of all finite program signals of sort C.

We will use signals to represent inputs and outputs of programs, as well as to store the events circulating inside programs.

Assume $s \in \mathrm{Sig}(C)$.

We say that s is *discrete-event* if and only if there is an order-embedding[1] from $\langle \{\mathsf{time}(e) \mid e \in s\}, \leq \rangle$ to $\langle \mathbb{N}, \leq \rangle$ where $\mathbb{N}$ is the set of natural numbers.

We write $\mathrm{Sig}_{\mathrm{DE}}(C)$ for the set of all discrete-event program signals of sort C.

We say that s is *non-Zeno* if and only if for every $t \in \mathbb{T}$, the set $\{e \in s \mid \mathsf{time}(e) \leq t\}$ is finite.

We write $\mathrm{Sig}_{\mathrm{NZ}}(C)$ for the set of all non-Zeno program signals of sort C.

Here, we will assume that inputs to programs and systems are non-Zeno signals. This is a realistic assumption, since ultimately, in any implementation, the source of those inputs will be sensors taking measurements of the physical world.

Notice that $\mathrm{Sig}_{\mathrm{fin}}(C) \subseteq \mathrm{Sig}_{\mathrm{NZ}}(C) \subseteq \mathrm{Sig}_{\mathrm{DE}}(C) \subseteq \mathrm{Sig}(C)$.

3.2 Actors

Actors are the basic blocks of computation in our programs. An actor is a stateful, in general, component that interfaces with its environment through a set of input and a set of output channels, what we call its input and output sort respectively. Each time it fires, it consumes a set of events from its input sort, and produces a set of events at its output sort, possibly updating its state in the process.

Definition 3.2.1. An *input action* of sort C is a nonempty subset α of $\mathrm{E}^{\mathrm{prog}}(C)$ such that for every $c \in C$, $|\{e \in \alpha \mid \mathsf{chan}(e) = c\}| \leq 1$, and for every $e_1, e_2 \in \alpha$, $\mathsf{time}(e_1) = \mathsf{time}(e_2)$.

For example, if $C = \{c_1, c_2\}$ is a sort consisting of channels c_1 and c_2, and $\tau_1, \tau_2 \in \mathbb{T}$ such that $\tau_1 \neq \tau_2$, then the following are input actions of sort C: $\alpha_1 = \{(c_1, \tau_1, v_1), (c_2, \tau_1, v_2)\}$, $\alpha_2 = \{(c_1, \tau_1, v_1)\}$, $\alpha_3 = \{(c_2, \tau_2, v_2)\}$, and the following are not input actions of sort C: $\emptyset$, $\{(c_1, \tau_1, v_1), (c_2, \tau_2, v_2)\}$, $\{(c_1, \tau_1, v_1), (c_1, \tau_2, v_2)\}$.

[1] Given two partially ordered sets A and B, an order-embedding is a function $f : A \to B$ such that for all $a, a' \in A$, $a \leq a'$ if and only if $f(a) \leq f(a')$.

We write $\mathsf{InputActions}(C)$ for the set of all input actions of sort C.

Assume $\alpha \in \mathsf{InputActions}(C)$.

We write $\mathsf{chan}(\alpha)$ for $\{\mathsf{chan}(e) \mid e \in \alpha\}$, and $\mathsf{time}(\alpha)$ for the unique $t \in \mathbb{T}$ such that for every $e \in \alpha$, $\mathsf{time}(e) = t$. So, in the previous example we have $\mathsf{chan}(\alpha_1) = \{c_1, c_2\}$ and $\mathsf{time}(\alpha_1) = \tau_1$.

Input actions restrict the possible input behaviors of actors. When actors fire, they process each time a set of events that share the same timestamp. This is central to our perception of a timestamp as a logical time instant: an actor fires at the unique logical time instant $\mathsf{time}(\alpha)$ associated with the corresponding input action α.

Definition 3.2.2. An *output action* of sort C is a nonempty subset α of $\mathrm{E}^{\mathrm{prog}}(C)$ such that for every $c \in C$,

$$|\{e \in \alpha \mid \mathsf{chan}(e) = c\}| \leq 1.$$

We write $\mathsf{OutputActions}(C)$ for the set of all output actions of sort C.

Unlike input actions, the events of an output action need not share the same timestamp. This is not inconsistent with our perception of timestamps as logical time instants. Output actions can be understood operationally: they schedule events in logical time according to their timestamps. We have constrained output actions to contain at most one event per output channel. If an actor could output multiple events on a channel, as a result of processing an input action, then it would also be possible for different firings of the actor to result in events with the same channel and timestamp. While there are approaches for handling such cases, e.g., the use of superdense time [21], [20], we will not consider them in this work.

Definition 3.2.3. An *actor* is an ordered sextuple $\langle S, s_{\mathrm{init}}, C_{\mathrm{in}}, C_{\mathrm{out}}, f, u\rangle$ such that the following are true:

1. S is a nonempty set of states;
2. $s_{\mathrm{init}} \in S$ is the initial state;
3. C_{in} is a sort;
4. C_{out} is a sort;
5. f is a function from $S \times \mathsf{InputActions}(C_{\mathrm{in}})$ to $\mathsf{OutputActions}(C_{\mathrm{out}})$;
6. u is a function from $S \times \mathsf{InputActions}(C_{\mathrm{in}})$ to S.

Assume an actor $A = \langle S, s_{\mathrm{init}}, C_{\mathrm{in}}, C_{\mathrm{out}}, f, u\rangle$.

We write $\mathsf{states}(A)$ for S, $\mathsf{state}_{\mathrm{init}}(A)$ for s_{init}, $\mathrm{C}_{\mathrm{in}}^A$ for C_{in}, $\mathrm{C}_{\mathrm{out}}^A$ for C_{out}, f^A for f, and u^A for u.

We now identify the class of actors that we will consider in this work.

We say that A is *output-homogeneous* if and only if for every $\langle s, \alpha \rangle \in \mathsf{states}(A) \times \mathsf{InputActions}(\mathrm{C}_{\mathrm{in}}^A)$,

$$\{\mathsf{chan}(e) \mid e \in \mathrm{f}^A(\langle s, \alpha \rangle)\} = \mathrm{C}_{\mathrm{out}}^A.$$

An output-homogeneous actor is simply an actor that, when fired, produces a single event at each channel in its output sort.

Assume an output-homogeneous actor A.

We say that A is *constant-delay* if and only if for every $c \in \mathrm{C}_{\mathrm{out}}^A$, there is $\delta \in \mathbb{Q}_{\geq 0}$, where $\mathbb{Q}_{\geq 0}$ are the positive rational numbers, such that for every $\langle s, \alpha \rangle \in \mathsf{states}(A) \times \mathsf{InputActions}(\mathrm{C}_{\mathrm{in}}^A)$, and every $e \in \mathrm{f}^A(\langle s, \alpha \rangle)$ such that $\mathsf{chan}(e) = c$,

$$\mathsf{time}(e) = \mathsf{time}(\alpha) + \delta.$$

Perhaps, one would expect the domain of δ to be $\mathbb{T}$ here. The reason we choose $Q_{\geq 0}$ instead, is that in Chapter 4 we will use the discrete state of a timed automaton to model actor delays.

Constant-delay output-homogeneous actors are *causal* actors, i.e., their output depends on past and present inputs but not on future ones, that produce events at fixed, non-negative logical time distances from the logical time instant associated with the input events.

As mentioned in the beginning of the chapter, we need to focus on constant-delay output-homogeneous actors for the analysis in Chapter 4 to be both necessary and sufficient. Furthermore, note that for this class of actors, we can statically associate an accurate deadline with each event in the program. If actors are constant-delay but not output-homogeneous then the analysis will be sufficient. However, for that case, coming up with a necessary and sufficient schedulability test does not appear to be easy. Assume that an actor, while it has outputs that lead to actuators, does not produce any outputs when it processes some of its inputs. Deadline assignment for the events that are on those input channels becomes a difficult problem and EDF ceases to be an optimal scheduling policy. We defer that discussion for Section 6.1.

Assume a constant-delay output-homogeneous actor A.

We write $\mathsf{delay}\, A$ for a function from $\mathrm{C}_{\mathrm{out}}^A$ to $\mathbb{Q}_{\geq 0}$ such that for every $c \in \mathrm{C}_{\mathrm{out}}^A$, every $\langle s, \alpha \rangle \in \mathsf{states}(A) \times \mathsf{InputActions}(\mathrm{C}_{\mathrm{in}}^A)$, and every $e \in \mathrm{f}^A(\langle s, \alpha \rangle)$ such that $\mathsf{chan}(e) = c$,

$$\mathsf{time}(e) = \mathsf{time}(\alpha) + (\mathsf{delay}\, A)(c).$$

3.3 Programs

For every actor A_1 and A_2, we say that A_1 and A_2 are *compatible* if and only if $\mathrm{C}_{\mathrm{in}}^{A_1} \cap \mathrm{C}_{\mathrm{in}}^{A_2} = \emptyset$ and $\mathrm{C}_{\mathrm{out}}^{A_1} \cap \mathrm{C}_{\mathrm{out}}^{A_2} = \emptyset$. Every channel in a program will have at most a single writer and a single reader. Any other behavior can be modeled using simple actors, that either merge the events of two channels into a single one, or copy the events of a channel to multiple ones.

Definition 3.3.1. A *program* is a nonempty finite set of pairwise compatible constant-delay output-homogeneous actors.

Assume a program P.

We write $\mathsf{chan}(P)$ for $\bigcup\{\mathrm{C}^A_{\mathrm{in}}, \mathrm{C}^A_{\mathrm{out}} \mid A \in P\}$, $\mathsf{sort}_{\mathrm{in}}\, P$ for $\bigcup\{\mathrm{C}^A_{\mathrm{in}} \mid A \in P\} \setminus \bigcup\{\mathrm{C}^A_{\mathrm{out}} \mid A \in P\}$,and $\mathrm{C}^P_{\mathrm{out}}$ for $\bigcup\{\mathrm{C}^A_{\mathrm{out}} \mid A \in P\} \setminus \bigcup\{\mathrm{C}^A_{\mathrm{in}} \mid A \in P\}$.

$\mathsf{chan}(P)$ is the set of all channels appearing in a program, whereas $\mathrm{C}^P_{\mathrm{in}}$ and $\mathrm{C}^P_{\mathrm{out}}$ are the sets of all unconnected input and output actor channels respectively.

We use a labeled transition system to formalize all possible executions of P.

Definition 3.3.2. A *state* of P is an ordered triple $\langle Q, \iota, \varepsilon\rangle$ such that the following are true:

1. $Q \in \mathrm{Sig}_{\mathrm{fin}}(\mathsf{chan}(P))$;
2. there is a partition $\{P_{\mathrm{idle}}, P_{\mathrm{exec}}\}$ of P such that the following are true:
   a) ι is a function from P_{idle} such that for any $A \in P_{\mathrm{idle}}$, $\iota(A) \in \mathsf{states}(A)$;
   b) ε is a function from P_{exec} such that for any $A \in P_{\mathrm{exec}}$, $\varepsilon(A) \in \mathsf{states}(A) \times \mathsf{InputActions}(\mathrm{C}^A_{\mathrm{in}})$.

We write $\mathsf{states}(P)$ for the set of all states of P.

A state $\langle Q, \iota, \varepsilon\rangle$ of P captures the composite state of P at some particular time instant during its execution: Q represents the set of all events circulating inside the program, ι the set of all idle actors, along with their state, and ε the set of all executing actors, along with their state and input events processed. Note that it is required that at every state the domain of ι and the domain of ε form a partition of the set of actors in the program, or that an actor must either be idle or executing, but not both.

We write $\mathsf{state}_{\mathrm{init}}(P)$ for a state $\langle Q, \iota, \varepsilon\rangle$ of P such that the following are true:

1. Q is the empty program signal;
2. ι is a function from P such that for every $A \in P$,
$$\iota(A) = \mathsf{state}_{\mathrm{init}}(A);$$
3. ε is the empty function.

The labels of the transition system that we will associate with P represent the possible actions of P. P can either receive an event from its environment, have an idle actor start processing an input action, have a processing actor finish its execution and produce an output action, or send an event to its environment.

We write $\mathsf{label}_{\mathrm{in}}\, P$ for $\mathrm{E}^{\mathrm{prog}}(\mathrm{C}^P_{\mathrm{in}})$, $\mathsf{label}_{\mathrm{start}}\, P$ for

$$\{\langle A, \langle s, \alpha\rangle\rangle \mid A \in P,\ s \in \mathsf{states}(A), \text{ and } \alpha \in \mathsf{InputActions}(\mathrm{C}^A_{\mathrm{in}})\},$$

$\mathsf{label}_{\mathrm{finish}}\, P$ for

$$\{\langle A, \alpha, s\rangle \mid A \in P,\ \alpha \in \mathsf{OutputActions}(\mathrm{C}^A_{\mathrm{out}}), \text{ and } s \in \mathsf{states}(A)\},$$

Table 3.1: Program transition rules.

$$\text{input}\ \frac{l \in \mathsf{label}_{\text{in}}\, P}{\langle Q, \iota, \varepsilon\rangle \xrightarrow{l}_P \langle Q \cup \{l\}, \iota, \varepsilon\rangle}$$

$$\text{output}\ \frac{l \in \mathsf{label}_{\text{out}}\, P \qquad l \in Q}{\langle Q, \iota, \varepsilon\rangle \xrightarrow{l}_P \langle Q \setminus \{l\}, \iota, \varepsilon\rangle}$$

$$\text{start}\ \frac{\langle A, \langle s, \alpha\rangle\rangle \in \mathsf{label}_{\text{start}}\, P \qquad \alpha \subseteq Q \qquad \iota(A) = s}{\langle Q, \iota, \varepsilon\rangle \xrightarrow{\langle A, \langle s, \alpha\rangle\rangle}_P \langle Q \setminus \alpha, \iota \setminus \{\langle A, s\rangle\}, \varepsilon \cup \{\langle A, \langle s, \alpha\rangle\rangle\}\rangle}$$

$$\text{finish}\ \frac{\begin{array}{c}\langle A, \alpha, s\rangle \in \mathsf{label}_{\text{finish}}\, P \\ \mathrm{f}^A(\varepsilon(A)) = \alpha \qquad \mathrm{u}^A(\varepsilon(A)) = s\end{array}}{\langle Q, \iota, \varepsilon\rangle \xrightarrow{\langle A, \alpha, s\rangle}_P \langle Q \cup \alpha, \iota \cup \{\langle A, s\rangle\}, \varepsilon \setminus \{\langle A, \varepsilon(A)\rangle\}\rangle}$$

$\mathsf{label}_{\text{out}}\, P$ for $\mathrm{E}^{\text{prog}}(\mathrm{C}^P_{\text{out}})$, and $\mathsf{label}\, P$ for

$$\mathsf{label}_{\text{in}}\, P \cup \mathsf{label}_{\text{start}}\, P \cup \mathsf{label}_{\text{finish}}\, P \cup \mathsf{label}_{\text{out}}\, P.$$

We write $\longrightarrow_P$ for a ternary relation between $\mathsf{states}(P)$, $\mathsf{label}\, P$, and $\mathsf{states}(P)$ defined by the rules in Table 3.1.

We write $\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow{l}_P \langle Q_2, \iota_2, \varepsilon_2\rangle$ if and only if $\longrightarrow_P(\langle Q_1, \iota_1, \varepsilon_1\rangle, l, \langle Q_2, \iota_2, \varepsilon_2\rangle)$.

We write $\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow{l}{}^{\text{in}}_P \langle Q_2, \iota_2, \varepsilon_2\rangle$ if and only if $\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow{l}_P \langle Q_2, \iota_2, \varepsilon_2\rangle$ and $l \in \mathsf{label}_{\text{in}}\, P$.

We write $\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow{l}{}^{\text{start}}_P \langle Q_2, \iota_2, \varepsilon_2\rangle$ if and only if $\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow{l}_P \langle Q_2, \iota_2, \varepsilon_2\rangle$ and $l \in \mathsf{label}_{\text{start}}\, P$.

We write $\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow{l}{}^{\text{finish}}_P \langle Q_2, \iota_2, \varepsilon_2\rangle$ if and only if $\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow{l}_P \langle Q_2, \iota_2, \varepsilon_2\rangle$ and $l \in \mathsf{label}_{\text{finish}}\, P$.

We write $\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow{l}{}^{\text{out}}_P \langle Q_2, \iota_2, \varepsilon_2\rangle$ if and only if $\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow{l}_P \langle Q_2, \iota_2, \varepsilon_2\rangle$ and $l \in \mathsf{label}_{\text{out}}\, P$.

Definition 3.3.3. An *execution* of P is a nonempty sequence $\mathrm{Exec}^{\text{prog}}$ such that the following are true:

1. $\mathrm{Exec}^{\text{prog}}(0) = \mathsf{state}_{\text{init}}(P)$;

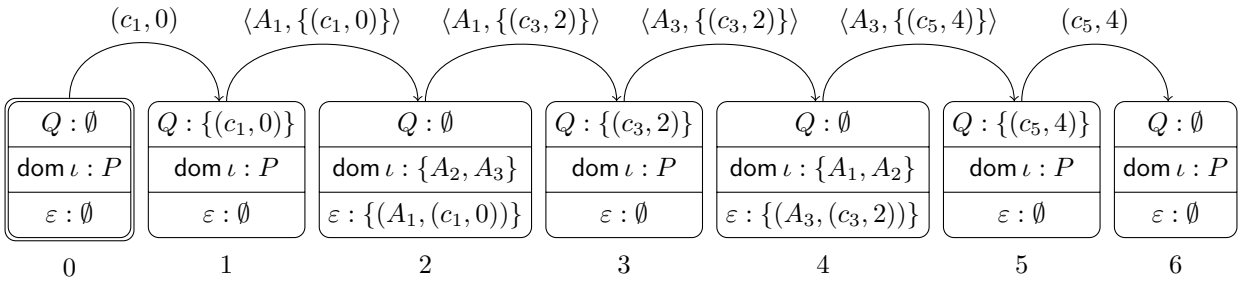


Figure 3.1: An example of a finite execution of the program of Figure 2.6, where event values and actor states have been omitted.

2. one of the following is true:

   a) $\text{Exec}^{\text{prog}}$ is finite, and the following are true:

      i. for any $n < (|\text{Exec}^{\text{prog}}| - 1)/2$, $\text{Exec}^{\text{prog}}_{2n} \xrightarrow{\text{Exec}^{\text{prog}}_{2n+1}}_P \text{Exec}^{\text{prog}}_{2n+2}$;

      ii. for every l and $\langle Q, \iota, \varepsilon\rangle$, if $\text{Exec}^{\text{prog}}(|\text{Exec}^{\text{prog}}| - 1) \xrightarrow{l}_P \langle Q, \iota, \varepsilon\rangle$, then $l \in \mathsf{label}_{\text{in}}\, P$;

   b) $\text{Exec}^{\text{prog}}$ is infinite, and for every $n \in \mathbb{N}$, $\text{Exec}^{\text{prog}}_{2n} \xrightarrow{\text{Exec}^{\text{prog}}_{2n+1}}_P \text{Exec}^{\text{prog}}_{2n+2}$.

Condition 2(a)ii states that an execution can be finite only if at the last state no transitions other than input ones are enabled, which further implies that both Q and $\mathsf{dom}\,\varepsilon$ must be empty.

Figure 3.1 displays an example of a finite execution of the program of Figure 2.6, where event values and actor states have been omitted.

In the following we will borrow from functional programming languages some notation and vocabulary for manipulating sequences. Specifically, we define a filter operation that takes as arguments a set and a sequence and returns a new sequence which contains all the elements of the original sequence that appear in the set. Formally, if $\mathsf{head}(s)$ is the first element of sequence s, $\mathsf{tail}(s)$ is the suffix sequence of s after removing the first element of s, and $\cdot$ is the concatenation of two sequences, then filter is defined as follows:

$$\mathsf{filter}(L, s) = \begin{cases} \langle\rangle & \text{if } s = \langle\rangle \\ \langle \mathsf{head}(s)\rangle \cdot \mathsf{filter}(L, \mathsf{tail}(s)) & \text{if } \mathsf{head}(s) \in L \\ \mathsf{filter}(F, \mathsf{tail}(s)) & \text{if } \mathsf{head}(s) \notin L \end{cases}$$

Assume an execution $\text{Exec}^{\text{prog}}$ of P.
Assume $L \subseteq \mathsf{label}\, P$.
We write $\mathsf{trace}_{\text{in}}\, \text{Exec}^{\text{prog}}$ for $\mathsf{filter}(\mathsf{label}_{\text{in}}\, P, \text{Exec}^{\text{prog}})$.
For every $A \in P$, we write $\mathsf{trace}^A_{\text{start}}\, \text{Exec}^{\text{prog}}$ for

$$\mathsf{filter}(\{\langle A, \langle s, \alpha\rangle\rangle \mid s \in \mathsf{states}(A) \text{ and } \alpha \in \mathsf{InputActions}(\mathrm{C}^A_{\text{in}})\}, \text{Exec}^{\text{prog}}).$$

For every $A \in P$, we write $\mathsf{trace}^{A}_{\text{finish}}\,\text{Exec}^{\text{prog}}$ for

$$\mathsf{filter}(\{\langle A, \alpha, s\rangle \mid \alpha \in \mathsf{OutputActions}(\text{C}^{A}_{\text{out}}) \text{ and } s \in \mathsf{states}(A)\}, \text{Exec}^{\text{prog}}).$$

For every $c \in \text{C}^{P}_{\text{out}}$, we write $\mathsf{trace}^{c}_{\text{out}}\,\text{Exec}^{\text{prog}}$ for $\mathsf{filter}(\text{E}^{\text{prog}}(\{c\}), \text{Exec}^{\text{prog}})$.
We write $\mathsf{in}\,\text{Exec}^{\text{prog}}$ for $\{\text{Exec}^{\text{prog}}_{2n+1} \mid n \in \mathbb{N} \text{ and } \text{Exec}^{\text{prog}}_{2n+1} \in \mathsf{label}_{\text{in}}\, P\}$.
We write $\mathsf{out}\,\text{Exec}^{\text{prog}}$ for $\{\text{Exec}^{\text{prog}}_{2n+1} \mid n \in \mathbb{N} \text{ and } \text{Exec}^{\text{prog}}_{2n+1} \in \mathsf{label}_{\text{out}}\, P\}$.
Notice that $\mathsf{in}\,\text{Exec}^{\text{prog}} \in \text{Sig}(\text{C}^{P}_{\text{in}})$ and $\mathsf{out}\,\text{Exec}^{\text{prog}} \in \text{Sig}(\text{C}^{P}_{\text{out}})$.

Clearly, our definition of execution is too liberal. In particular, it allows for executions where an actor processes its inputs out of timestamp order, what is at odds with the intended role of timestamps as a logical notion of time.

We say that $\text{Exec}^{\text{prog}}$ is *actor-safe* if and only if for every $A \in P$, and any n_1, $\langle Q_1, \iota_1, \varepsilon_1\rangle$, and $\langle s_1, \alpha_1\rangle$ such that

$$\text{Exec}^{\text{prog}}_{2n_1} = \langle Q_1, \iota_1, \varepsilon_1\rangle$$

and

$$\varepsilon_1(A) = \langle s_1, \alpha_1\rangle,$$

and any n_2, $\langle Q_2, \iota_2, \varepsilon_2\rangle$, and $\langle s_2, \alpha_2\rangle$ such that

$$\text{Exec}^{\text{prog}}_{2n_2} = \langle Q_2, \iota_2, \varepsilon_2\rangle$$

and

$$\varepsilon_2(A) = \langle s_2, \alpha_2\rangle,$$

if $n_1 < n_2$, then $\mathsf{time}(\alpha_1) < \mathsf{time}(\alpha_2)$.

We say that $\text{Exec}^{\text{prog}}$ is *output-safe* if and only if for every $c \in \text{C}^{P}_{\text{out}}$, and any n_1 and n_2 such that $\text{Exec}^{\text{prog}}_{2n_1+1} \in \text{E}^{\text{prog}}(\{c\})$ and $\text{Exec}^{\text{prog}}_{2n_2+1} \in \text{E}^{\text{prog}}(\{c\})$, if $n_1 < n_2$, then $\mathsf{time}(\text{Exec}^{\text{prog}}_{2n_1+1}) < \mathsf{time}(\text{Exec}^{\text{prog}}_{2n_2+1})$.

We say that $\text{Exec}^{\text{prog}}$ is *safe* if and only if $\text{Exec}^{\text{prog}}$ is actor-safe and output-safe.

Informally, an execution is safe just as long as every actor processes its input events in timestamp order. Notice that safety does not constrain a program to process every single event in timestamp order, only that each actor does so.

Another worrisome type of execution allowed by our definition is that of one where either an event in the program remains unprocessed indefinitely, or an actor is not given the opportunity to finish processing, or an output event of the program is never sent to the environment.

We say that $\text{Exec}^{\text{prog}}$ is *actor-fair* if and only if for every $A \in P$, the following are true:

1. for any n and $\langle Q, \iota, \varepsilon\rangle$ such that

   $$\text{Exec}^{\text{prog}}_{2n} = \langle Q, \iota, \varepsilon\rangle,$$

   if $A \in \mathsf{dom}\,\iota$, and there is $e \in Q$ such that $\mathsf{chan}(e) \in \text{C}^{A}_{\text{in}}$, then there is n', $\langle Q', \iota', \varepsilon'\rangle$, s', and α' such that $n < n'$,

   $$\text{Exec}^{\text{prog}}_{2n'} = \langle Q', \iota', \varepsilon'\rangle,$$

$A \in \mathsf{dom}\, \varepsilon'$,

$$\varepsilon'(A) = \langle s', \alpha' \rangle,$$

and $e \in \alpha'$;

2. for any n and $\langle Q, \iota, \varepsilon \rangle$ such that

$$\text{Exec}_{2n}^{\text{prog}} = \langle Q, \iota, \varepsilon \rangle,$$

if $A \in \mathsf{dom}\, \varepsilon$, then there is n' and $\langle Q', \iota', \varepsilon' \rangle$ such that $n < n'$,

$$\text{Exec}_{2n'}^{\text{prog}} = \langle Q', \iota', \varepsilon' \rangle,$$

and $A \in \mathsf{dom}\, \iota'$.

We say that $\text{Exec}^{\text{prog}}$ is *output-fair* if and only if for every $c \in \text{C}_{\text{out}}^{P}$, and any n and $\langle Q, \iota, \varepsilon \rangle$ such that

$$\text{Exec}_{2n}^{\text{prog}} = \langle Q, \iota, \varepsilon \rangle,$$

if there is $e \in Q$ such that

$$\mathsf{chan}(e) = c,$$

then there is n' such that $n \leq n'$ and

$$\text{Exec}_{2n'+1}^{\text{prog}} = e.$$

We say that $\text{Exec}^{\text{prog}}$ is *fair* if and only if $\text{Exec}^{\text{prog}}$ is actor-fair and output-fair.

Fairness is a well-studied notion in reactive systems (see for example Chapter 3 in [1] for an introduction). Fairness assumptions have been used to rule out unrealistic executions. The general form that those assumptions take, is that if an execution of a system visits a state infinitely often, then all transitions leaving that state should be taken in that execution. Actor-safety and output-safety have the same form. The transitions in the case of actor-safety are start and finish transitions, and in the case of output-safety, output transitions.

We say that $\text{Exec}^{\text{prog}}$ is *correct* if and only if $\text{Exec}^{\text{prog}}$ is safe and fair.

Correct executions of a program constitute a semantic specification of valid implementations of that program. It has been used to rule out unrealistic

Theorem 3.3.4. *For every correct execution* $\text{Exec}_1^{\text{prog}}$ *and* $\text{Exec}_2^{\text{prog}}$ *of* P*, if*

$$\mathsf{in}\, \text{Exec}_1^{\text{prog}} = \mathsf{in}\, \text{Exec}_2^{\text{prog}},$$

then the following are true:

1. *for every* $A \in P$*, the following are true:*

   a) $\mathsf{trace}_{\text{start}}^{A}\, \text{Exec}_1^{\text{prog}} = \mathsf{trace}_{\text{start}}^{A}\, \text{Exec}_2^{\text{prog}}$*;*

   b) $\mathsf{trace}_{\text{finish}}^{A}\, \text{Exec}_1^{\text{prog}} = \mathsf{trace}_{\text{finish}}^{A}\, \text{Exec}_2^{\text{prog}}$*;*

2. for every $c \in \mathrm{C}_{\mathrm{out}}^{P}$,

$$\mathsf{trace}_{\mathrm{out}}^{c}\,\mathrm{Exec}_1^{\mathrm{prog}} = \mathsf{trace}_{\mathrm{out}}^{c}\,\mathrm{Exec}_2^{\mathrm{prog}}.$$

Proof. We will show that for every $A \in P$, $\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_1^{\mathrm{prog}} = \mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_2^{\mathrm{prog}}$. The other statements follow easily.

We write $\alpha_i^{A,j}$ for the input action of the j^{th} start transition of actor A in execution E_i.

We write $s_i^{A,j}$ for the state of actor A at the j^{th} start transition of A in execution E_i.

Let n be the largest n such that for all $A \in P$:

$$\langle(\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_1^{\mathrm{prog}})(0), \ldots, (\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_1^{\mathrm{prog}})(n-1)\rangle = \langle(\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_2^{\mathrm{prog}})(0), \ldots, (\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_2^{\mathrm{prog}})(n-1)\rangle$$

This implies that there is $A \in P$ such that either $(\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_1^{\mathrm{prog}}(n) \neq (\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_2^{\mathrm{prog}}(n)$ or without loss of generality $|(\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_1^{\mathrm{prog}}| > n$ and $|(\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_2^{\mathrm{prog}}| = n$.

We examine those cases separately, first assume that $(\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_1^{\mathrm{prog}})(n)$ exists but $(\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_2^{\mathrm{prog}})(n)$ does not.

For any $e \in \alpha_1^{A,n}$, if $\mathsf{chan}(e) \in \mathrm{C}_{\mathrm{in}}^{P}$ or there is $i < n$ and A' such that e is produced by $(\mathsf{trace}_{\mathrm{finish}}^{A'}\,\mathrm{Exec}_1^{\mathrm{prog}})(i)$ then e is produced in $\mathrm{Exec}_2^{\mathrm{prog}}$ as well, and by fairness $(\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_1^{\mathrm{prog}})(n)$ exists.

Therefore for every $e \in \alpha_1^{A,n}$, there is A' such that $\mathsf{chan}(e) \in \mathrm{C}_{\mathrm{out}}^{A'}$, $(\mathsf{trace}_{\mathrm{start}}^{A'}\,\mathrm{Exec}_1^{\mathrm{prog}})(n)$ precedes $(\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_1^{\mathrm{prog}})(n)$ in $\mathrm{Exec}_1^{\mathrm{prog}}$, and $|(\mathsf{trace}_{\mathrm{start}}^{A'}\,\mathrm{Exec}_2^{\mathrm{prog}})| < n$.

If we apply the same reasoning as we did for A in the case of A' and so on, we will either find actor A'' such that $\mathrm{C}_{\mathrm{in}}^{A''} \subseteq \mathrm{C}_{\mathrm{in}}^{P}$ or reach actor A again. In the first case, since the input signals of A'' are the same in $\mathrm{Exec}_1^{\mathrm{prog}}$ and $\mathrm{Exec}_2^{\mathrm{prog}}$, the start transitions cannot differ. In the second case, we will have concluded that the n^{th} start transition of A precedes the n^{th} start transition of A which is also a contradiction.

Therefore it must be $|(\mathsf{trace}_{\mathrm{start}}\,\mathrm{Exec}_2^{\mathrm{prog}})| > n$ and $(\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_1^{\mathrm{prog}})(n) \neq (\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_2^{\mathrm{prog}})(n)$.

Note that the value of $s_i^{A,n}$ depends on $\langle(\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_1^{\mathrm{prog}})(0), \ldots, (\mathsf{trace}_{\mathrm{start}}^{A}\,\mathrm{Exec}_1^{\mathrm{prog}})(n-1)\rangle$ therefore $s_1^{A,n} = s_2^{A,n}$, and it has to be that $\alpha_1^{A,n} \neq \alpha_2^{A,n}$.

One of the following is true:

1. $\mathsf{time}(\alpha_1^{A,n}) > \mathsf{time}(\alpha_2^{A,n})$:

   Note that because $\mathrm{Exec}_1^{\mathrm{prog}}$ is actor-safe, there cannot be any start transitions in $\mathrm{Exec}_1^{\mathrm{prog}}$ that process the events in $\alpha_2^{A,n}$. Furthermore, because $\mathrm{Exec}_1^{\mathrm{prog}}$ is actor-fair, we conclude that the events $\alpha_2^{A,n}$ are never produced in $\mathrm{Exec}_1^{\mathrm{prog}}$.

   Let e be an event in $\alpha_2^{A,n}$, and A' the actor such that $\mathsf{chan}(e) \in \mathrm{C}_{\mathrm{out}}^{A'}$. Such an actor exists since if $\mathsf{chan}(e) \in \mathrm{C}_{\mathrm{in}}^{P}$ it would also exist in $\mathrm{Exec}_1^{\mathrm{prog}}$.

Since $\text{Exec}_2^{\text{prog}}$ is actor-safe and constant-delay, A' produces events in timestamp order. Furthermore, the first n start and finish transitions of A' are the same in $\text{Exec}_1^{\text{prog}}$ and $\text{Exec}_2^{\text{prog}}$.

Hence, we conclude that $(\mathsf{trace}^{A'}_{\text{start}}\,\text{Exec}_1^{\text{prog}})(n) \neq (\mathsf{trace}^{A'}_{\text{start}}\,\text{Exec}_2^{\text{prog}})(n)$, $\mathsf{time}(\alpha_1^{A',n}) > \mathsf{time}(\alpha_2^{A',n})$, and $(\mathsf{trace}^{A'}_{\text{start}}\,\text{Exec}_2^{\text{prog}})(n)$ precedes $(\mathsf{trace}^{A}_{\text{start}}\,\text{Exec}_2^{\text{prog}})(n)$ in $\text{Exec}_2^{\text{prog}}$.

2. $\mathsf{time}(\alpha_1^{A,n}) = \mathsf{time}(\alpha_2^{A,n})$:

   One of the following is true:

   - $\mathsf{chan}(\alpha_1^{A,n}) = \mathsf{chan}(\alpha_2^{A,n})$: there is an upstream actor A' such that $(\mathsf{trace}^{A'}_{\text{start}}\,\text{Exec}_1^{\text{prog}})(n) \neq (\mathsf{trace}^{A'}_{\text{start}}\,\text{Exec}_2^{\text{prog}})(n)$, $(\mathsf{trace}^{A'}_{\text{start}}\,\text{Exec}_1^{\text{prog}})(n)$ precedes $(\mathsf{trace}^{A}_{\text{start}}\,\text{Exec}_1^{\text{prog}})(n)$ in $\text{Exec}_1^{\text{prog}}$, $(\mathsf{trace}^{A'}_{\text{start}}\,\text{Exec}_2^{\text{prog}})(n)$ precedes $(\mathsf{trace}^{A}_{\text{start}}\,\text{Exec}_2^{\text{prog}})(n)$ in $\text{Exec}_2^{\text{prog}}$, and $\mathsf{time}(\alpha_1^{A',n}) = \mathsf{time}(\alpha_2^{A',n})$.
   - $\mathsf{chan}(\alpha_1^{A,n}) \neq \mathsf{chan}(\alpha_2^{A,n})$: without loss of generality there is $e \in \alpha_1^{A,n}$ such that $\mathsf{chan}(e) \notin \mathsf{chan}(\alpha_2^{A,n})$ and with the same reasoning as in case 1, we can conclude that there is A' such that $(\mathsf{trace}^{A'}_{\text{start}}\,\text{Exec}_1^{\text{prog}})(n) \neq (\mathsf{trace}^{A'}_{\text{start}}\,\text{Exec}_2^{\text{prog}})(n)$, $\mathsf{time}(\alpha_1^{A',n}) < \mathsf{time}(\alpha_2^{A',n})$, $(\mathsf{trace}^{A'}_{\text{start}}\,\text{Exec}_1^{\text{prog}})(n)$ precedes $(\mathsf{trace}^{A}_{\text{start}}\,\text{Exec}_1^{\text{prog}})(n)$ in $\text{Exec}_1^{\text{prog}}$, and $(\mathsf{trace}^{A'}_{\text{start}}\,\text{Exec}_2^{\text{prog}})(n)$ precedes $(\mathsf{trace}^{A}_{\text{start}}\,\text{Exec}_2^{\text{prog}})(n)$.

In all cases, we can repeat the process for the upstream actor A' in the same way as when it was showed that both $(\mathsf{trace}^{A}_{\text{start}}\,\text{Exec}_1^{\text{prog}})(n)$ and $(\mathsf{trace}^{A}_{\text{start}}\,\text{Exec}_2^{\text{prog}})(n)$ must exist, and reach a contradiction. □

The following is immediate:

Corollary 3.3.5. *For every correct execution* $\text{Exec}_1^{\text{prog}}$ *and* $\text{Exec}_2^{\text{prog}}$ *of* P*, if*

$$\mathsf{in}\,\text{Exec}_1^{\text{prog}} = \mathsf{in}\,\text{Exec}_2^{\text{prog}},$$

then

$$\mathsf{out}\,\text{Exec}_1^{\text{prog}} = \mathsf{out}\,\text{Exec}_2^{\text{prog}}.$$

Corollary 3.3.5 formally characterizes the determinacy of discrete-event programs. It asserts that as long as the logical notion of time in a program is not violated, in the rather loose sense of the safety property, and each party involved is given the opportunity to make progress, the behaviour of the program is determinate with respect to input and output signals.

The question remains whether a program has correct executions at all, that is, other than the trivial one. A program that contains cycles of zero logical-time delay cannot possibly have an execution where an event enters the cycle without violating either safety or fairness. Here, we exclude such programs from consideration.

We say that P is *well defined* if and only if for every nonempty $X \subseteq P$, there is $A \in X$ such that for every $A' \in X$,

$$(\mathsf{delay}\,A')^{-1}(0) \cap \mathrm{C}^A_{\text{in}} = \emptyset.$$

The condition above implies that there cannot be a cycle of actors, such that all of them have delay equal to zero.

Assume a well defined program P.

Theorem 3.3.6. *For any $s \in \mathrm{Sig}_{\mathrm{NZ}}(\mathrm{C}_{\mathrm{in}}^{P})$, there is a correct execution $\mathrm{Exec}^{\mathrm{prog}}$ of P such that*

$$\mathsf{in}\,\mathrm{Exec}^{\mathrm{prog}} = s.$$

Proof. We describe how to construct given a program P and an input signal s a correct program execution $\mathrm{Exec}^{\mathrm{prog}}$ such that $\mathsf{in}\,\mathrm{Exec}^{\mathrm{prog}} = s$.

In every step, we describe how to extend a prefix of the program execution, in a way that guarantees that the execution is safe and fair. The prefix is extended with either an input transition, a start transition followed by the corresponding finish transition, or an output transition.

The initial prefix of the program execution is

$$\mathrm{Exec}^{\mathrm{prog}}(0) = \mathsf{state}_{\mathrm{init}}(P).$$

At the beginning of each iteration, we calculate the next event to be processed.

In the following we assume that the last state of the prefix is $\langle Q, \iota, \varepsilon \rangle$ and the remaining input signal is s.

The source of the next event might be the input signal s, or the program signal Q.

We define a function $(\mathsf{min\text{-}time}^{\mathrm{prog}} P)$ from $\mathrm{Sig}(\mathsf{chan}(P))$ to $\mathbb{T}$ such that for every $s' \in \mathrm{Sig}(\mathsf{chan}(P))$ $(\mathsf{min\text{-}time}^{\mathrm{prog}} P)(s') = \min\{\mathsf{time}(e) \mid e \in s'\}$.

If we choose the event that has the smallest timestamp, then the resulting execution is guaranteed to be safe.

Let $\tau_i = (\mathsf{min\text{-}time}^{\mathrm{prog}} P)(s)$ and $\tau_q = (\mathsf{min\text{-}time}^{\mathrm{prog}} P)(Q)$.

If $\tau_i \leq \tau_q$, let e_i be an event in $\{e \mid e \in s \text{ and } \mathsf{time}(e) = \tau_i\}$ and $s' = s \setminus \{e_i\}$. We extend the program execution prefix with an input transition with label e_i and the new state $\langle Q \cup \{e_i\}, \iota, \varepsilon \rangle$

If $\tau_i > \tau_q$, we will append a start and a finish transition, or an output transition.

Let $S = \{e \in Q \mid \mathsf{time}(e) = \tau_q\}$

We define an order between channels in $\mathsf{chan}(P)$ in the following way: $c_1 \prec c_2$ if and only if there is a sequence of actors $A_1, \ldots, A_N$ such that $c_1 \in \mathrm{C}_{\mathrm{in}}^{A_1}$, $c_2 \in \mathrm{C}_{\mathrm{in}}^{A_N}$, and for every i such that $1 \leq i < N$, $(\mathsf{delay}\, A_i)^{-1}(0) \cap \mathrm{C}_{\mathrm{in}}^{A_{i+1}} \neq \emptyset$.

We lift that order to program events and say that $e_1 \prec e_2$ if and only if $\mathsf{chan}(e_1) \prec \mathsf{chan}(e_2)$.

Let S' be the set of events that are minimal elements of S according to $\prec$.

We prove now that S' cannot nonempty because P is well-defined.

By definition of τ_q, S is not empty. S' is empty if and only if S has no $\prec$-minimal elements. Note that S is finite since Q is single-valued and thus for every $c \in \mathsf{chan}(P)$, $|\{e \in S \mid \mathsf{chan}(e) = c\}| \leq 1$. Assume that S' is empty. The absence of a $\prec$-minimal element implies that there is a sequence $e_1, \ldots, e_N$ such that for every i such that $1 \leq i < N$,

$e_i \prec e_{i+1}$, and $e_1 = e_N$. By the definition of $\prec$, given cycle $e_1 \prec \ldots \prec e_{N-1} \prec e_1$, we can construct a set of actors $X \subseteq P$ such that for every $A \in X$, there is $A' \in X$ such that $(\mathsf{delay}\, A')^{-1}(0) \cap \mathrm{C}_{\mathrm{in}}^{A} \neq \emptyset$. That implies that P is not well-defined, therefore S' cannot be empty.

If the set $\{A \in P \mid \text{there is } e \in S' \text{ such that } \mathsf{chan}(e) \in \mathrm{C}_{\mathrm{in}}^{A}\}$ is empty, then it must be the case that for every $e \in S'$, $\mathsf{chan}(e) \in \mathrm{C}_{\mathrm{out}}^{P}$. In that case we pick an event e from S' and add an output transition with label e, and a state $\langle Q \setminus \{e\}, \iota, \varepsilon \rangle$.

If the set is not empty, then let B be an actor in the set $\{A \in P \mid \text{there is } e \in S' \text{ such that } \mathsf{chan}(e) \in \mathrm{C}_{\mathrm{in}}^{A}\}$, and $\alpha_{\mathrm{input}} \in \mathsf{InputActions}(B)$ such that $\alpha_{\mathrm{input}} = \{e \in S' \mid \mathsf{chan}(e) \in \mathrm{C}_{\mathrm{in}}^{B}\}$.

We extend the current program execution prefix with a start transition for actor B with label l_{start} and input action α_{input}, a state $\langle Q', \iota', \varepsilon' \rangle$, a finish transition for actor B with label l_{finish}, and a final state $\langle Q'', \iota'', \varepsilon'' \rangle$.

Let s_B be the state of actor B at the beginning of the iteration, or $s_B = \iota(A_B)$, s''_B the state of B after the finish transition, or $s''_B = \mathrm{u}^B(\langle s_B, \alpha_{\mathrm{input}} \rangle)$, and α_{output} the output action of the finish transition, or $\alpha_{\mathrm{output}} = \mathrm{f}^B(\langle s_B, \alpha_{\mathrm{input}} \rangle)$.

The label of the start transition will then be

$$l_{\mathrm{start}} = \langle B, \langle s_B, \alpha_{\mathrm{input}} \rangle \rangle,$$

the intermediate state

$$\langle Q', \iota', \varepsilon' \rangle = \langle Q \setminus \alpha_{\mathrm{input}}, \iota \setminus \{\langle B, s_B \rangle\}, \varepsilon \cup \{l_{\mathrm{start}}\} \rangle,$$

the label of the finish transition

$$l_{\mathrm{finish}} = \langle B, \alpha_{\mathrm{output}}, s''_B \rangle,$$

and the final state

$$\langle Q'', \iota'', \varepsilon'' \rangle = \langle Q' \cup \alpha_{\mathrm{output}}, \iota' \cup \{\langle B, s''_B \rangle\}, \varepsilon' \setminus \{l_{\mathrm{start}}\} \rangle.$$

The resulting execution is safe.

We know argue about its fairness.

First, the second part of the actor-fairness definition is satisfied because we explicitly accompany each start transition with the corresponding finish transition.

Second, for both the first part of actor-fairness and output-fairness we argue that in the resulting execution $\mathrm{Exec}^{\mathrm{prog}}$ if there is n, $\langle Q, \iota, \varepsilon \rangle$, and e such that $\mathrm{Exec}^{\mathrm{prog}}(n) = \langle Q, \iota, \varepsilon \rangle$, and $e \in Q$, then there is n' and $\langle Q', \iota', \varepsilon' \rangle$ such that $(\mathsf{min\text{-}time}^{\mathrm{prog}} P)(Q') > \mathsf{time}(e)$.

Let s be the unprocessed input signal when $\mathrm{Exec}^{\mathrm{prog}}(n)$ was added in the program execution. At that point it could either be the case that $(\mathsf{min\text{-}time}^{\mathrm{prog}} P)(s) \leq (\mathsf{min\text{-}time}^{\mathrm{prog}} P)(Q)$ or not. In the former case, because s in non-Zeno, after a finite number of steps that add input transitions, the latter case will hold. Without loss of generality, we assume that it holds when the input is s and $\mathrm{Exec}^{\mathrm{prog}}(n)$ is added.

The process described above will then add a pair of a start and a finish transition such that the program time of the input action is $(\mathsf{min\text{-}time}^{\text{prog}} P)(Q)$ and the program time of the events in the output action is $\geq (\mathsf{min\text{-}time}^{\text{prog}} P)(Q)$. Such start and finish transition pairs with an input action of program time equal to $(\mathsf{min\text{-}time}^{\text{prog}} P)(Q)$ will continue to be added as long as there are events in the program with program time equal to $(\mathsf{min\text{-}time}^{\text{prog}} P)(Q)$. Because P is well-defined, in a bounded number of transitions, the state of the program will be $\langle Q', \iota', \varepsilon' \rangle$ where $(\mathsf{min\text{-}time}^{\text{prog}} P)(Q') \geq (\mathsf{min\text{-}time}^{\text{prog}} P)(Q) + \delta$ and $\delta \geq \min \{(\mathsf{delay}\, A)(c) \mid A \in P,\, c \in \mathrm{C}^A_{\text{out}}, \text{ and } (\mathsf{delay}\, A)(c) > 0\}$.

The program time of the input action of the next start transition will not be greater or equal than $(\mathsf{min\text{-}time}^{\text{prog}} P)(Q) + \delta$ if the next input event has program time smaller than that. Note, however, that because the number of input events with program time between $(\mathsf{min\text{-}time}^{\text{prog}} P)(Q)$ and $(\mathsf{min\text{-}time}^{\text{prog}} P)(Q) + \delta$ is finite, since the input signal is non-Zeno, it is still guaranteed that in a finite number of steps there will be a start transition with input action with program time larger than or equal to $(\mathsf{min\text{-}time}^{\text{prog}} P)(Q) + \delta$.

We can now repeat the reasoning above $\left\lceil \frac{\mathsf{time}(e) - (\mathsf{min\text{-}time}^{\text{prog}} P)(Q)}{\delta} \right\rceil$ number of times. □

Theorem 3.3.6 guarantees that any program free of zero logical-time delay cycles will have a correct execution for every possible discrete-event and non-Zeno input signal it is presented with.

We write $\mathsf{delay}\, P$ for a function from $\mathsf{chan}(P) \times \mathsf{chan}(P)$ to $\mathbb{Q}_{\geq 0} \cup \{\infty\}$ such that for every $c_1, c_2 \in \mathsf{chan}(P)$,

$$(\mathsf{delay}\, P)(c_1, c_2) = \begin{cases} 0 & \text{if } c_1 = c_2; \\ \infty & \text{if } c_1 \neq c_2 \text{ and } c_1 \in \mathrm{C}^P_{\text{out}}; \\ \min\{(\mathsf{delay}\, A)(c) + (\mathsf{delay}\, P)(c, c_2) \mid A \in P, \\ \qquad c_1 \in \mathrm{C}^A_{\text{in}}, \\ \qquad \text{and } c \in \mathrm{C}^A_{\text{out}}\} & \text{otherwise.} \end{cases}$$

We will frequently "overload" $\mathsf{delay}\, P$ and apply to sets of channels as well. For $C_1, C_2 \subseteq \mathsf{chan}(P)$,

$$(\mathsf{delay}\, P)(C_1, C_2) = \min\{(\mathsf{delay}\, P)(c_1, c_2) \mid c_1 \in C_1 \text{ and } c_2 \in C_2\}.$$

3.4 Systems

Programs deal exclusively in logical time. There is nothing in their semantics that bears any relevance to physical time at all. But if we are to use them as executable real-time specifications, we need to determine how exactly these two different notions of time relate to one another.

Definition 3.4.1. A *system* is an ordered pair $\langle P, R \rangle$ such that the following are true:

1. P is a well-defined program;
2. R is a function from P to $\mathscr{P}_{\geq 1}\,\mathbb{T}$,[2] such that for every $A \in P$, $0 < \inf R(A) \leq \sup R(A)$ and $\sup R(A) \in \mathbb{Q}_{\geq 0}$.

Assume a system $\langle P, R\rangle$.

$\langle P, R\rangle$ is understood as a program P running on a given uniprocessor platform. For each A, the set $R(A)$ represents the possible computation times that A can have on that platform. We use a timed labeled transition system to formalize all possible executions of $\langle P, R\rangle$.

Definition 3.4.2. A *state* of $\langle P, R\rangle$ is an ordered sextuple $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$ such that the following are true:

1. $\langle Q, \iota, \varepsilon\rangle \in \mathsf{states}(P)$;
2. ρ is a function from $\mathsf{dom}\,\varepsilon$ to $\mathbb{T}$ such that for any $A \in \mathsf{dom}\,\varepsilon$, there is $r \in R(A)$ such that
$$\rho(A) \leq r;$$
3. $\pi \in \{\texttt{NULL}\} \cup \{A \mid A \in \mathsf{dom}\,\varepsilon \text{ and } 0 < \rho(A)\}$;
4. $t^{\text{sys}} \in \mathbb{T}$.

A state $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$ of $\langle P, R\rangle$ augments the state of P with information on the actual state of the platform at some particular time instant during the execution of P. Specifically, ρ captures the remaining computation time of any processing actor, π represents the actor running on the processor at that time instant, and t^{sys} stands for that time instant.

We write $\mathsf{states}(\langle P, R\rangle)$ for the set of all states of $\langle P, R\rangle$.

We write $\mathsf{state}_{\text{init}}(\langle P, R\rangle)$ for a state $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$ of $\langle P, R\rangle$ such that the following are true:

1. $\langle Q, \iota, \varepsilon\rangle = \mathsf{state}_{\text{init}}(P)$;
2. ρ is the empty function;
3. $\pi = \texttt{NULL}$;
4. $t^{\text{sys}} = 0$.

Assume $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle \in \mathsf{states}(\langle P, R\rangle)$.

We write $\mathsf{prog}\,\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$ for $\langle Q, \iota, \varepsilon\rangle$.

We write $\mathsf{time}^{\text{sys}}\,\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$ for t^{sys}.

We write $\mathsf{label}_{\text{in}}\,\langle P, R\rangle$ for $\mathsf{label}_{\text{in}}\,P$.

We write $\mathsf{label}_{\text{start}}\,\langle P, R\rangle$ for $\{\langle\langle A, \langle s, \alpha\rangle\rangle, r\rangle \mid \langle A, \langle s, \alpha\rangle\rangle \in \mathsf{label}_{\text{start}}\,P \text{ and } r \in R(A)\}$.

We write $\mathsf{label}_{\text{finish}}\,\langle P, R\rangle$ for $\mathsf{label}_{\text{finish}}\,P$.

[2] For every set A, we write $\mathscr{P}_{\geq 1}\,A$ for the set of all nonempty subsets of A.

Table 3.2: System transition rules.

$$\text{input}\ \frac{\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow[P]{l}{}^{\text{in}} \langle Q_2, \iota_2, \varepsilon_2\rangle \qquad \mathsf{time}(l) = t^{\text{sys}}}{\langle Q_1, \iota_1, \varepsilon_1, \rho, \pi, t^{\text{sys}}\rangle \xrightarrow{l}_{\langle P,R\rangle} \langle Q_2, \iota_2, \varepsilon_2, \rho, \pi, t^{\text{sys}}\rangle}$$

$$\text{output}\ \frac{\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow[P]{l}{}^{\text{out}} \langle Q_2, \iota_2, \varepsilon_2\rangle \qquad \mathsf{time}(l) = t^{\text{sys}}}{\langle Q_1, \iota_1, \varepsilon_1, \rho, \pi, t^{\text{sys}}\rangle \xrightarrow{l}_{\langle P,R\rangle} \langle Q_2, \iota_2, \varepsilon_2, \rho, \pi, t^{\text{sys}}\rangle}$$

$$\text{start}\ \frac{\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow[P]{\langle A,\langle s,\alpha\rangle\rangle}{}^{\text{start}} \langle Q_2, \iota_2, \varepsilon_2\rangle \qquad r \in R(A) \qquad \rho_2 = \rho_1 \cup \{\langle A, r\rangle\}}{\langle Q_1, \iota_1, \varepsilon_1, \rho_1, \pi, t^{\text{sys}}\rangle \xrightarrow{\langle\langle A,\langle s,\alpha\rangle\rangle,r\rangle}_{\langle P,R\rangle} \langle Q_2, \iota_2, \varepsilon_2, \rho_2, \pi, t^{\text{sys}}\rangle}$$

$$\text{finish}\ \frac{\langle Q_1, \iota_1, \varepsilon_1\rangle \xrightarrow[P]{\langle A,\alpha,s\rangle}{}^{\text{finish}} \langle Q_2, \iota_2, \varepsilon_2\rangle \qquad \rho_1(A) = 0 \qquad \rho_2 = \rho_1 \setminus \{\langle A, 0\rangle\}}{\langle Q_1, \iota_1, \varepsilon_1, \rho_1, A, t^{\text{sys}}\rangle \xrightarrow{\langle A,\alpha,s\rangle}_{\langle P,R\rangle} \langle Q_2, \iota_2, \varepsilon_2, \rho_2, \texttt{NULL}, t^{\text{sys}}\rangle}$$

$$\text{context-switch}\ \frac{l \in \{\texttt{NULL}\} \cup \mathsf{dom}\,\varepsilon}{\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle \xrightarrow{l}_{\langle P,R\rangle} \langle Q, \iota, \varepsilon, \rho, l, t^{\text{sys}}\rangle}$$

We write $\mathsf{label}_{\text{out}}\,\langle P, R\rangle$ for $\mathsf{label}_{\text{out}}\,P$.
We write $\mathsf{label}_{\text{prog}}\,\langle P, R\rangle$ for $\mathsf{label}_{\text{in}}\,\langle P, R\rangle \cup \mathsf{label}_{\text{start}}\,\langle P, R\rangle \cup \mathsf{label}_{\text{finish}}\,\langle P, R\rangle \cup \mathsf{label}_{\text{out}}\,\langle P, R\rangle$.
For every $l \in \mathsf{label}_{\text{prog}}\,\langle P, R\rangle$, we write $\mathsf{prog}\,l$ for

$$\begin{cases} \langle A, \langle s, \alpha\rangle\rangle & \text{if } l \in \mathsf{label}_{\text{start}}\,\langle P, R\rangle, \text{ and there is } r \text{ such that} \\ & l = \langle\langle A, \langle s, \alpha\rangle\rangle, r\rangle; \\ l & \text{otherwise.} \end{cases}$$

We write $\mathsf{label}_{\text{sch}}\,\langle P, R\rangle$ for $\{\texttt{NULL}\} \cup P$.
We write $\mathsf{label}\,\langle P, R\rangle$ for $\mathsf{label}_{\text{prog}}\,\langle P, R\rangle \cup \mathsf{label}_{\text{sch}}\,\langle P, R\rangle$.

Notice that the labels related to an actor starting to process an input action are augmented with a rational number representing the amount of computation time that will be required for processing that input action.

We write $\longrightarrow_{\langle P,R\rangle}$ for a ternary relation between $\mathsf{states}(\langle P, R\rangle)$, $\mathsf{label}\,\langle P, R\rangle$, and $\mathsf{states}(\langle P, R\rangle)$ defined by the rules in Figure 3.2.

Table 3.3: System time transition rules.

$$\text{idle}\ \frac{\pi = \texttt{NULL}}{\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle \xrightarrow{d}_{\langle P,R \rangle} \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} + d \rangle}$$

$$\text{busy}\ \frac{\pi \neq \texttt{NULL} \qquad d \leq \rho_1(\pi) \qquad \rho_2(A) = \begin{cases} \rho_1(A) - d & \text{if } A = \pi; \\ \rho_1(A) & \text{otherwise;} \end{cases}}{\langle Q, \iota, \varepsilon, \rho_1, \pi, t^{\text{sys}} \rangle \xrightarrow{d}_{\langle P,R \rangle} \langle Q, \iota, \varepsilon, \rho_2, \pi, t^{\text{sys}} + d \rangle}$$

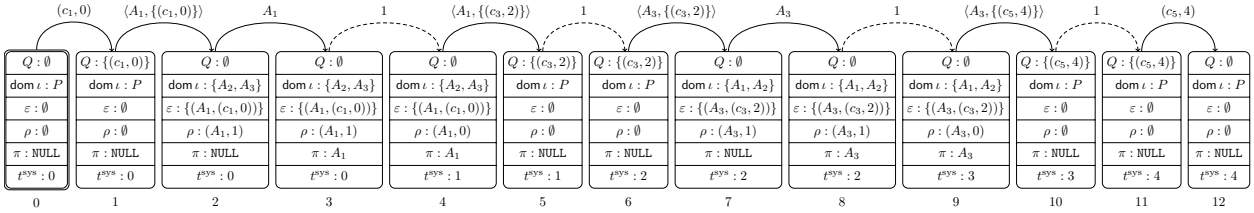


Figure 3.2: A prefix of an execution of the system of Figure 2.6 corresponding to the program execution of Figure 3.1, where event values and actor states have been omitted.

There are a few observations that need to be made here.

First, input and output actions of a system bind program time to system time and system time to program time respectively. One may think of an input channel of a program as connected to an ideal sensor that will timestamp its measurements with the exact physical time at which they were made, and instantaneously deliver them to the program, and an output channel as connected to an ideal actuator that will actuate the environment instantaneously at the exact physical time dictated by the timestamp of the corresponding output event.

Second, every time an actor starts processing, a requirement for computation-time necessary to perform its processing is chosen nondeterministically, and the actor becomes available for execution. But it is not executed until the system decides to allocate the platform's processor to it. Once it starts executing, it can be preempted and resumed arbitrarily according to the scheduling decisions of the system.

And third, a processing actor finishes exactly when it has been allocated processor time equal to its corresponding computation-time requirement, at which point it is taken off the processor.

System-time progress is modeled using a different type of transition labeled with the amount of physical time elapsed.

We write $\langle Q_1, \iota_1, \varepsilon_1, \rho_1, \pi_1, t_1^{\text{sys}} \rangle \xrightarrow{l}_{\langle P,R \rangle} \langle Q_2, \iota_2, \varepsilon_2, \rho_2, \pi_2, t_2^{\text{sys}} \rangle$ if and only if

$\longrightarrow_{\langle P,R\rangle}(\langle Q_1, \iota_1, \varepsilon_1, \rho_1, \pi_1, t_1^{\text{sys}}\rangle, l, \langle Q_2, \iota_2, \varepsilon_2, \rho_2, \pi_2, t_2^{\text{sys}}\rangle)$.

We write $\dashrightarrow_{\langle P,R\rangle}$ for a ternary relation between $\mathsf{states}(\langle P, R\rangle)$, $\mathbb{T}$, and $\mathsf{states}(\langle P, R\rangle)$ defined by the rules in Figure 3.3.

We write $\langle Q_1, \iota_1, \varepsilon_1, \rho_1, \pi_1, t_1^{\text{sys}}\rangle \overset{d}{\dashrightarrow}_{\langle P,R\rangle} \langle Q_2, \iota_2, \varepsilon_2, \rho_2, \pi_2, t_2^{\text{sys}}\rangle$ if and only if $\dashrightarrow_{\langle P,R\rangle}(\langle Q_1, \iota_1, \varepsilon_1, \rho_1, \pi_1, t_1^{\text{sys}}\rangle, d, \langle Q_2, \iota_2, \varepsilon_2, \rho_2, \pi_2, t_2^{\text{sys}}\rangle)$.

When the processor of the platform is free, there is no actor executing, and unless there is an actor available for processing that the system decides to execute, system time may progress arbitrarily. But when the processor is allocated to a processing actor, system time cannot progress more than the remaining computation time of that actor, for at that point, a discrete transition corresponding to that actor finishing must occur.

Definition 3.4.3. An *execution* of $\langle P, R\rangle$ is an infinite sequence Exec^{sys} such that the following are true:

1. $\text{Exec}^{\text{sys}}(0) = \mathsf{state}_{\text{init}}(\langle P, R\rangle)$;
2. for every $n \in \mathbb{N}$, one of the following is true:
   a) $\text{Exec}^{\text{sys}}_{2n} \xrightarrow{\text{Exec}^{\text{sys}}_{2n+1}}_{\langle P,R\rangle} \text{Exec}^{\text{sys}}_{2n+2}$;
   b) $\text{Exec}^{\text{sys}}_{2n} \overset{\text{Exec}^{\text{sys}}_{2n+1}}{\dashrightarrow}_{\langle P,R\rangle} \text{Exec}^{\text{sys}}_{2n+2}$;
3. for every $t^{\text{sys}} \in \mathbb{T}$, there is n such that $t^{\text{sys}} \leq \mathsf{time}^{\text{sys}}\, \text{Exec}^{\text{sys}}_{2n}$.
4. for any n such that
$$\mathsf{time}^{\text{sys}}\, \text{Exec}^{\text{sys}}_{2n} = \mathsf{time}^{\text{sys}}\, \text{Exec}^{\text{sys}}_{2n+2},$$
if $\text{Exec}^{\text{sys}}_{2n+3} \in \mathsf{label}_{\text{in}}\, \langle P, R\rangle$, then $\text{Exec}^{\text{sys}}_{2n+1} \in \mathsf{label}_{\text{in}}\, \langle P, R\rangle$;

A system execution is always infinite, for even if the environment ceases to produce input stimuli, system time will continue to progress, and in fact, diverge, as required by the third clause of the definition. What the fourth clause amounts to is giving input transitions a higher priority, and is necessitated by the idealization choices of our formalization, as will soon become clear.

We define a map operation, borrowed again from functional languages, that takes as arguments a function and a sequence, applies the function to every element in the sequence, and returns a sequence of the results:

$$\mathsf{map}(f, s) = \begin{cases} \langle\rangle & \text{if } s = \langle\rangle, \\ \langle f(\mathsf{head}(s))\rangle \cdot \mathsf{map}(f, \mathsf{tail}(s)) & \text{otherwise.} \end{cases}$$

Assume an execution Exec^{sys} of $\langle P, R\rangle$.

We write $\mathsf{prog}\, \text{Exec}^{\text{sys}}$ for a sequence over $\mathsf{states}(P) \cup \mathsf{label}\, P$ such that for every $n \in \mathbb{N}$,

$$\mathsf{prog}\, \text{Exec}^{\text{sys}} = \mathsf{map}(\mathsf{prog}, \mathsf{filter}(L_{\text{prog}} \cup \{\text{Exec}^{\text{sys}}(n) \mid \text{Exec}^{\text{sys}}(n+1) \in L_{\text{prog}}\}, \text{Exec}^{\text{sys}})).$$

where $L_{\text{prog}} = \mathsf{label}_{\text{prog}}\, \langle P, R\rangle$.

Proposition 3.4.4. $\mathsf{prog}\,\mathrm{Exec}^{\mathrm{sys}}$ *is an execution of* P.

Figure 3.2 shows a prefix of an execution of the system of Figure 2.6, whose underlying program execution is that of Figure 3.1, where, again, event values and actor states have been omitted.

Just as was the case with program executions, system executions are too general. What we want is that the logical-time specification of our programs prescribe the physical-time behaviour of our systems. And for that to be the case, we need to make sure that program time, as expressed through timestamps of events, maintains its role as a logical notion of time. To do so we have to limit ourselves to system executions whose underlying program executions are safe. But here we must further guarantee that the resulting system specification will exclude non-causal implementations that clairvoyantly execute actors without ever violating program safety, correctly guessing the environment's future behaviour. And having bound program time to system time at the input edges of a program, we can use the structure and state of the program to determine at any given time, whether it is safe for an actor to process an input event or not.

We say that A is *safe to process* in $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\mathrm{sys}}\rangle$ if and only if $A \in \mathsf{dom}\,\iota$, there is $\alpha \in \mathsf{InputActions}(\mathrm{C}^A_{\mathrm{in}})$ such that $\alpha \subseteq Q$, and the following are true:

1. $t^{\mathrm{sys}} \geq \mathsf{time}(\alpha) - (\mathsf{delay}\,P)(\mathrm{C}^P_{\mathrm{in}}, \mathrm{C}^A_{\mathrm{in}})$;
2. for any $e \in Q \setminus \alpha$,
$$\mathsf{time}(e) > \mathsf{time}(\alpha) - (\mathsf{delay}\,P)(\mathsf{chan}(e), \mathrm{C}^A_{\mathrm{in}});$$
3. for any A' and $\langle s', \alpha'\rangle$ such that $\varepsilon(A') = \langle s', \alpha'\rangle$,
$$\mathsf{time}(\alpha') > \mathsf{time}(\alpha) - (\mathsf{delay}\,P)(\mathrm{C}^{A'}_{\mathrm{in}}, \mathrm{C}^A_{\mathrm{in}}).$$

Informally, an actor is safe to process just as long as there is an event to process, it is impossible for a new event arriving at a sensor to eventually cause an event of smaller or equal timestamp as the event to be processed, and the same is true for any other event already circulating or being processed inside the program. Notice that the inequality in the first clause is non-strict, as opposed to those in the second and third one. This reflects our idealization that allows a system to instantaneously make a scheduling decision that takes into account the input status at that same time instant, and is the reason for giving input transitions a higher priority in system executions.

We say that $\mathrm{Exec}^{\mathrm{sys}}$ is *actor-safe* if and only if for every $A \in P$, and any n, $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\mathrm{sys}}\rangle$, $\langle s, \alpha\rangle$, and r such that

$$\mathrm{Exec}^{\mathrm{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\mathrm{sys}}\rangle$$

and

$$\mathrm{Exec}^{\mathrm{sys}}_{2n+1} = \langle\langle A, \langle s, \alpha\rangle\rangle, r\rangle,$$

A is safe to process in $\text{Exec}^{\text{sys}}_{2n}$.

Of course, we are interested in system executions that meet the physical-time constraints implied by the logical-time specification of our programs.

We say that Exec^{sys} is *output-safe* if and only if for any $c \in \text{C}^P_{\text{out}}$, and every n and $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle$ such that

$$\text{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle,$$

if there is e in Q such that

$$\mathsf{chan}(e) = c,$$

then

$$t^{\text{sys}} \leq \mathsf{time}(e).$$

If we think of timestamps of events reaching the output channels of a program as actuator deadlines, then a system execution is output-safe just as long as no actuator deadline is ever missed.

We say that Exec^{sys} is *safe* if and only if Exec^{sys} is actor-safe and output-safe.

We say that e *immediately produces* e' in Exec^{sys} if and only if there is A, n, $\langle \langle A, \langle s, \alpha \rangle \rangle, r \rangle$, n', and $\langle A, s', \alpha' \rangle$ such that the following are true:

1. $\text{Exec}^{\text{sys}}_{2n+1} = \langle \langle A, \langle s, \alpha \rangle \rangle, r \rangle$;
2. $e \in \alpha$;
3. $\text{Exec}^{\text{sys}}_{2n'+1} = \langle A, s', \alpha' \rangle$;
4. $e' \in \alpha'$;
5. $n < n'$;
6. for every n'' such that $n < n'' < n'$, if there is $\langle A', s'', \alpha'' \rangle$ such that $\text{Exec}^{\text{sys}}_{2n''+1} = \langle A', s'', \alpha'' \rangle$, then $A' \neq A$.

We say that e *causally produces* e' in Exec^{sys} if and only if there is $e_1, \ldots, e_N$ such that $e_1 = e$, $e_N = e'$, and for every i such that $1 \leq i < N$, e_i immediately produces e_{i+1} in Exec^{sys}.

Notice that if e causally produces e', then

$$\mathsf{time}(e') > \mathsf{time}(e) + (\mathsf{delay}\, P)(\mathsf{chan}(e), \mathsf{chan}(e')).$$

Theorem 3.4.5. *If* Exec^{sys} *is safe, then* $\mathsf{prog}\, \text{Exec}^{\text{sys}}$ *is safe.*

Proof. We examine two consecutive start transitions of an actor A in Exec^{sys}, and show that the program time of the corresponding input actions strictly increases.

There are n_1, n_2, $\langle \langle A, \langle s_1, \alpha_1 \rangle \rangle, r_1 \rangle$, $\langle \langle A, \langle s_2, \alpha_2 \rangle \rangle, r_2 \rangle$, $\langle Q_1, \iota_1, \varepsilon_1, \rho_1, \pi_1, t^{\text{sys}}_1 \rangle$, $\langle Q_2, \iota_2, \varepsilon_2, \rho_2, \pi_2, t^{\text{sys}}_2 \rangle$ such that the following are true:

1. $n_1 < n_2$;

2. $\text{Exec}^{\text{sys}}_{2n_1} = \langle Q_1, \iota_1, \varepsilon_1, \rho_1, \pi_1, t_1^{\text{sys}} \rangle$;
3. $\text{Exec}^{\text{sys}}_{2n_2} = \langle Q_2, \iota_2, \varepsilon_2, \rho_2, \pi_2, t_2^{\text{sys}} \rangle$;
4. $\text{Exec}^{\text{sys}}_{2n_1+1} = \langle \langle A, \langle s_1, \alpha_1 \rangle \rangle, r_1 \rangle$;
5. $\text{Exec}^{\text{sys}}_{2n_2+1} = \langle \langle A, \langle s_2, \alpha_2 \rangle \rangle, r_2 \rangle$;
6. for every n such that $n_1 < n < n_2$, if there is $\langle \langle A', \langle s', \alpha' \rangle \rangle, r' \rangle$ such that $\text{Exec}^{\text{sys}}_{2n+1} = \langle \langle A, \langle s', \alpha' \rangle \rangle, r' \rangle$, then $A' \neq A$.

$\text{Exec}^{\text{sys}}_{2n_1+1}$ and $\text{Exec}^{\text{sys}}_{2n_2+1}$ are the two consecutive start transitions of A in Exec^{sys}.
We will show that $\mathsf{time}(\alpha_1) < \mathsf{time}(\alpha_2)$.
For every event e in α_2, either e is in Q_1 or it is not.
If it is not in Q_1, since e is in Q_2, we will trace its source, i.e. the finish transition of an actor, an event that causally produces it, or an input transition, in the states and transitions that preceed $\text{Exec}^{\text{sys}}_{2n_2}$.
Note that for every $e \in \alpha_2$, $\mathsf{time}(e) = \mathsf{time}(\alpha_2)$, so it is sufficient to show that $\mathsf{time}(e) > \mathsf{time}(\alpha_1)$.
For every $e \in \alpha_2$ one of the following is true:

1. $e \in Q_1$ and by condition 2 of the definition of safe to process 3.4 for A for start transition $\text{Exec}^{\text{sys}}_{2n_1+1}$,
$$\mathsf{time}(e) > \mathsf{time}(\alpha_1) - (\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^A_{\text{in}}) = \mathsf{time}(\alpha_1);$$
2. $e \notin Q_1$ and one of the following is true:
   a) there exists $e' \in Q_1$ that causally produces e, so $\mathsf{time}(e') \leq \mathsf{time}(e)$, and by condition 2 of the definition of safe to process 3.4 for A for start transition $\text{Exec}^{\text{sys}}_{2n_1+1}$,
   $$\mathsf{time}(e') > \mathsf{time}(\alpha_1) - (\mathsf{delay}\, P)(\mathsf{chan}(e'), \mathrm{C}^A_{\text{in}});$$
   b) there exists A', e', and $\langle s', \alpha' \rangle$ such that the following are true:
      i. $\varepsilon_1(A') = \langle s', \alpha' \rangle$;
      ii. $e' \in \alpha'$, e' causally produces e, and $\mathsf{time}(e') \leq \mathsf{time}(e)$;
      iii. by condition 3 of the definition of safe to process 3.4 for A for start transition $\text{Exec}^{\text{sys}}_{2n_1+1}$,
      $$\mathsf{time}(\alpha') > \mathsf{time}(\alpha_1) - (\mathsf{delay}\, P)(\mathrm{C}^{A'}_{\text{in}}, \mathrm{C}^A_{\text{in}});$$
   c) there exists n', and $e' \in \mathsf{label}_{\text{in}} \langle P, R \rangle$ such that the following are true:
      i. $\text{Exec}^{\text{sys}}_{2n'+1} = e'$;
      ii. e' causally produces e;
      iii. $n' > n_1$ implying that $\mathsf{time}(e') > t_1^{\text{sys}}$ by the input-first property of executions;

iv. by condition 1 of the definition of safe to process 3.4 for A for start transition $\text{Exec}^{\text{sys}}_{2n_1+1}$,

$$t_1^{\text{sys}} \geq \mathsf{time}(\alpha_1) - (\mathsf{delay}\ P)(\text{C}^P_{\text{in}}, \text{C}^A_{\text{in}}).$$

□

We say that Exec^{sys} is *fair* if and only if $\mathsf{prog}\,\text{Exec}^{\text{sys}}$ is actor-fair.
We say that Exec^{sys} is *correct* if and only if Exec^{sys} is safe and fair.

Theorem 3.4.6. *If* Exec^{sys} *is correct, then* $\mathsf{prog}\,\text{Exec}^{\text{sys}}$ *is correct.*

Proof. By definition if Exec^{sys} is correct then Exec^{sys} is safe and fair, or $\mathsf{prog}\,\text{Exec}^{\text{sys}}$ is safe and $\mathsf{prog}\,\text{Exec}^{\text{sys}}$ is actor-fair.

Therefore it remains to be shown that $\mathsf{prog}\,\text{Exec}^{\text{sys}}$ is output-fair.

Let n, $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$, and e be such that

$$\text{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle,$$

$$e \in Q,$$

and

$$\mathsf{chan}(e) \in \text{C}^P_{\text{out}}.$$

We will show that there is n' such that $n \leq n'$ and

$$\text{Exec}^{\text{sys}}_{2n'+1} = e.$$

Since Exec^{sys} is output-safe, $t^{\text{sys}} \leq \mathsf{time}(e)$.

By definition of an execution, there is n'', $\langle Q'', \iota'', \varepsilon'', \rho'', \pi'', t^{\text{sys}\prime\prime}\rangle$, and $\epsilon > 0$ such that $\mathsf{time}(e) + \epsilon \leq t^{\text{sys}\prime\prime}$.

$e \notin Q''$ because $t^{\text{sys}\prime\prime} > \mathsf{time}(e)$ and Exec^{sys} is output-safe.

$t^{\text{sys}\prime\prime} > t^{\text{sys}}$ therefore $n'' > n$.

Since e is in Q, e is not in Q'', and $\mathsf{chan}(e) \in \text{C}^P_{\text{out}}$, there is n' such that $n \leq n' < n''$ and $\text{Exec}^{\text{sys}}_{2n'+1} = e$. □

3.5 Summary

At the beginning of the chapter we defined programs and actors. Actors are stateful components with input and output channels that consume input actions from their input channels, or sets of events with the same timestamp, and produce output actions, or sets of events with at most one event per output channel. Programs are then sets of compatible actors, i.e., actors whose input and output channels do not overlap.

We then defined program executions by introducing a transition system. A state of that transition system, consists of the set of events circulating in the program, Q, the set of actors that have not read any inputs and are idle, represented by ι, and the set of actors that have

read inputs and have started processing them, represented by ε. An actor moves from $\mathsf{dom}\,\iota$ to $\mathsf{dom}\,\varepsilon$ by performing a start transition: it reads or consumes a set of events in Q that are in its input channels and that form an input action, i.e., have the same timestamp. It moves from $\mathsf{dom}\,\varepsilon$ back to $\mathsf{dom}\,\iota$ by performing a finish transition that produces a set of output events in the actor's output channels. Input transitions create new events at the program's input channels and output transitions remove events from the program's output transitions.

We first constrained program executions so that they agree with the intuition of timestamps representing a notion of time. We thus defined safe program executions to be ones that have actors process their inputs in timestamp order, and that produce events to the environment in timestamp order. We further ruled out problematic executions that forever ignore events or actors by defining fair program executions.

In Theorem 3.3.4, we showed that program executions are deterministic: independent of transition interleaving, if executions are correct, i.e., if they process events in timestamp order and they are fair, then, given the same inputs, every actor will be invoked with the same sequence of inputs and the outputs to the environment will be the same.

Last, we showed in Theorem 3.3.6, that for well-defined programs, i.e., programs that contain no cycles of zero logical-time delay, a correct execution always exists for program inputs that are non-Zeno.

Next, we naturally extended programs to systems. Systems model the execution of a program on a uniprocessor platform. The state of a system extends the program state with three elements: t^{sys}, π, and ρ. Variable t^{sys} captures the real time or system time. As mentioned in previous chapters, input and output transitions bind program and system time. Time advances on time transitions.

Systems also associate a computation time with each actor that reads inputs in a start transition. An actor then is not ready to finish, or to produce its outputs, until it has been executed on the processor for a total time equal to its computation time. Variable π tracks the actor that is executing on the processor.

Because of the splitting of the processing of events in programs in start and finish transitions, the assignment of computation time to actors comes naturally in system executions. The remaining computation time for each actor that is ready for processing is tracked with the ρ function of the system state: $\rho(A)$ is initialized to A's computation time at the start transition of A and whenever a time transition of delay δ is performed, $\rho(A)$ is reduced by δ if π is equal to A.

As was the case with program executions, we would like to constraint system executions to ones that satisfy the discrete-event semantics of the underlying program. However because of the interplay with system time, we cannot simply constraint system executions to ones that satisfy timestamp order, for if we do that, we will also allow for executions that happen to be safe, by "guessing" what the future inputs will be. Thus, we introduce the notion of safe-to-process analysis, and define safe system executions to be ones that require an actor to be safe to process every time it reads inputs and performs a start transition.

We define the program projection of a system execution to be the residual program execution when all system specific parts of a system execution are removed. In Theorem 3.4.5

we show that if a system execution is safe, then its program projection is safe as well, or that the safe-to-process analysis associated with the safety or system executions is sufficient to guarantee the timestamp order processing of events. In Theorem 3.4.6 we show that if a system execution is correct then its program projection is correct as well.

Chapter 4

PTIDES Schedulability

In this chapter we address the schedulability problem for a uniprocessor system. We formally define the problem, prove that the earliest-deadline-first scheduling policy is optimal, and show that the schedulability problem can be reduced to a finite-state reachability problem. Finally, we describe how to carry out this reduction using timed automata.

The boundedness of the state space will be based on two observations. First, we will show that if the number of events in any state of a PTIDES execution exceeds a specific bound, then it can be shown that that PTIDES execution will lead to a deadline violation. The second observation is that the absolute value of real-time or of a timestamp of an event are not individually both necessary for the execution of a PTIDES program. In fact, it is only the difference between the two that is needed for the execution of a PTIDES program. Furthermore, we show that this difference can be upper and lower bounded.

4.1 Definition

By Theorem 3.3.6, every well defined program will be able to execute correctly when presented with any non-Zeno input signal. For a system, because of the fact that each actor is associated with computation time, this is clearly not the case. The problem is to decide whether a given system will be able to execute correctly when presented with any non-Zeno input signal from some given class of such signals.

First we need to address a technical issue. Our definition of a system $\langle P, R\rangle$ includes a function R which returns for every actor a set of possible computation times. When an actor A reads a set of input events, in order to model the different possible computation times for A, one is chosen nondeterministically from the set $R(A)$ and is associated with the processing of those specific input events. Since the ability to execute a system correctly depends on those choices we need to be able to quantify system executions over all possible such choices.

Recall from the previous chapter that the operator $\mathsf{trace}^{A}_{\text{start}}$ extracts the sequence of start transitions of actor A from a system execution Exec^{sys}. In an execution, "start" transitions

represent the reading of input events that are safe to process by an actor. The label of a start transition $\langle\langle A, \langle s, \alpha\rangle\rangle, r\rangle$ contains the actor A, its state s, the input action α, which is a set of events on the inputs of A that have the same timestamp, and r which is the computation time that actor A will have to execute for in the processor before the processing of the events is complete. In a system, a start transition label is a pair of a program start transition, the part $\langle A, \langle s, \alpha\rangle\rangle$, which is common in program and system executions, paired with the computation time choice r, which appears only in system executions. The elements of the sequence $\mathsf{trace}^A_{\text{start}}\,\text{Exec}^{\text{sys}}$ will be start transition labels or pairs of the form $\langle\langle A, \langle s, \alpha\rangle\rangle, r\rangle$.

For every $A \in P$, we write $\mathsf{req}^A(\text{Exec}^{\text{sys}})$ for:

$$\mathsf{map}(\mathsf{second}, \mathsf{trace}_{\text{start}}\,\text{Exec}^{\text{sys}}),$$

where second is a function that returns the second element of a pair.

Recall that the map function, defined in the previous chapter, applies the function given as its first argument to every element of the sequence given as its second argument and returns the result. The application of second extracts exactly the computation time choice off of each start transition label and $\mathsf{req}^A(\text{Exec}^{\text{sys}})$ is the sequence of computation times that actor A spent to complete each input invocation in execution Exec^{sys}.

We write $\mathsf{OR}(\langle P, R\rangle)$ for a set such that for every $O \in \mathsf{OR}(\langle P, R\rangle)$, O is a function from P such that for every $A \in P$, $O(A) \in \mathscr{S}_\infty\, R(A)$[1].

In other words, $\mathsf{OR}(\langle P, R\rangle)$ is a set of *oracles*, where an oracle is a function that for every actor returns an infinite sequence of computation times valid for that actor in the system $\langle P, R\rangle$.

Proposition 4.1.1. *For every execution* Exec^{sys} *of a system* $\langle P, R\rangle$*, there exists* $O \in \mathsf{OR}(\langle P, R\rangle)$ *such that for every* $A \in P$, $\mathsf{req}^A(\text{Exec}^{\text{sys}}) \sqsubseteq$ [2]$O(A)$.

As we mentioned earlier, the question of whether a system is schedulable should be associated with an input model. A system will not and should not be expected to work properly if its inputs are completely unconstrained. We will thus now introduce the notion of a combination of a system and a class of input signals. This is of course not rare. Schedulability analyses are always associated with specific input models, such as sporadic or periodic, as was discussed in the introduction.

Definition 4.1.2. A *model* is an ordered pair $\langle\langle P, R\rangle, I\rangle$ such that the following are true:

1. $\langle P, R\rangle$ is a well defined system;
2. $I \subseteq \text{Sig}(\text{C}^P_{\text{in}})$ is a set of input signals for program P.

[1] We write $\mathscr{S}_N\, A$ for the set of sequences of size N whose elements are in A.

[2] For two sequences A and B, we write $A \sqsubseteq B$ if A is a prefix of B.

Assume a model $\langle\langle P, R\rangle, I\rangle$.

We discuss now the question of when a model should be deemed schedulable. Intuitively, in a schedulable model, the system is expected to be able to execute correctly, i.e., to not miss any deadlines, for every possible input to the system allowed by I. Furthermore, for a given input, the system should be able to execute correctly for all potential computation time choices allowed by specification R.

We say that $\langle\langle P, R\rangle, I\rangle$ is *schedulable* if and only if for any $s \in I$, for every $O \in \mathsf{OR}(\langle P, R\rangle)$, there is a correct execution Exec^{sys} of $\langle P, R\rangle$ such that $\mathsf{in}\,\text{Exec}^{\text{sys}} = s$ and for every $A \in P$, $\mathsf{req}^A(\text{Exec}^{\text{sys}}) \sqsubseteq O(A)$.

In other words, for every input signal s allowed by the model, i.e., $s \in I$, and for all allowed computation time choices O for the actors of the system, there exists an execution Exec^{sys} that reads all the events of the input, $\mathsf{in}\,\text{Exec}^{\text{sys}} = s$, the computation times of every actor match the ones chosen, $\mathsf{req}^A(\text{Exec}^{\text{sys}}) \sqsubseteq O(A)$, and it is correct: it processes all events, i.e., it is fair, it processes events only when they are safe to process, i.e., it is actor-safe, and it does not miss any deadlines, i.e., it is output safe.

To make this definition more clear, note that if each actor is associated with a unique computation time, i.e., for every $A \in P$, there is an r such that $R(A) = \{r\}$, then the second universal quantification $O \in \mathsf{OR}(\langle P, R\rangle)$ and the last requirement $\mathsf{req}^A(\text{Exec}^{\text{sys}}) \sqsubseteq O(A)$ can simply be removed: the corresponding model is schedulable if and only if for every $s \in I$, there is a correct execution Exec^{sys} of $\langle P, R\rangle$ such that $\mathsf{in}\,\text{Exec}^{\text{sys}} = s$.

A convenient tool in approaching schedulability problems is the identification of a particular scheduling strategy that is optimal, in the sense that if a system is schedulable, then it is also schedulable under that particular strategy. For uniprocessor platforms, the earliest-deadline-first (EDF) strategy has been proven optimal over a variety of different schedulability problems. To make it applicable here, we first need to formalize a notion of deadline for the events circulating inside the program of a system.

We write $\mathsf{deadline}\,P$ for a function from $\bigcup\{\mathsf{InputActions}(\mathrm{C}_{\text{in}}^A) \mid A \in P\}$ to $\mathbb{T}$, where $\mathsf{InputActions}(\mathrm{C}_{\text{in}}^A)$ is the set of input actions of sort C_{in}^A, or a set of events on the input channels of A that share the same timestamp, such that for every $\alpha \in \bigcup\{\mathsf{InputActions}(\mathrm{C}_{\text{in}}^A) \mid A \in P\}$,

$$(\mathsf{deadline}\,P)(\alpha) = \mathsf{time}(\alpha) + (\mathsf{delay}\,P)(\mathsf{chan}(\alpha), \mathrm{C}_{\text{out}}^P).$$

Recall from the previous chapter that $\mathsf{time}(\alpha)$ is the shared timestamp of the events in input action α, and that $(\mathsf{delay}\,P)(\mathsf{chan}(\alpha), \mathrm{C}_{\text{out}}^P)$ is the minimum delay between the channels of the events in α and the output channels of the program.

The definition of deadline matches the interpretation of the timestamp of an event at an output port as the time at which the actuation should be performed. For the actuation command to be valid, it should be issued at system time earlier than when specified by the timestamp. Processing an input action α by an actor A will result in events at the output channels of A, whose processing will result in other events, and eventually, because of the constraint that actors are constant-delay and output-homogeneous, α will result in events in all outputs ports c for which $(\mathsf{delay}\,P)(\mathsf{chan}(\alpha), c) \neq \infty$. Among those, the one with

the smallest timestamp determines the urgency of the input action, or, in other words, its deadline.

Furthermore, note that the $\mathsf{deadline}\, P$ function is defined on input actions of an actor, since those represent the schedulable units in a system execution. In a system execution, the actors that are eligible for scheduling are those that had safe-to-process events in their input channels and have read those inputs, or have executed a start transition. The set of those actors are exactly the ones for which a mapping exists in the function ε of the system state. The set of safe-to-process events that was read as part of the start transition constitutes the input action on which $\mathsf{deadline}\, P$ will be called.

Informally, we will also refer to the deadline of a single event. In that case, we simply assume that the event is lifted to the input action that consists solely of that event. Since all the events that constitute an input action have the same timestamp, their deadline and the deadline of the input action match.

Leading to the definition of earliest-deadline-first executions, we will first define eager executions. Roughly an execution is eager when there is no unnecessary system time delay, or when system transitions are performed as soon as possible. System time elapses in a system execution using time transitions. Time transitions are labeled with the amount of time that passes between the start and the end state. Therefore, in the following, if $\text{Exec}^{\text{sys}}_{2n+1} \in \mathbb{T}$, then $\text{Exec}^{\text{sys}}_{2n+1}$ is a time transition that corresponds to system time elapsing by an amount equal to $\text{Exec}^{\text{sys}}_{2n+1}$.

We say that Exec^{sys} is *eager* if and only if for every $A \in P$, the following are true:

1. for any n and $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle$ such that

$$\text{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle,$$

and any d such that A is safe to process in $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} + d \rangle$, if $\text{Exec}^{\text{sys}}_{2n+1} \in \mathbb{T}$, then

$$\text{Exec}^{\text{sys}}_{2n+1} \leq d;$$

2. for any n, $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle$, such that

$$\text{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle,$$

$$\mathsf{dom}\, \varepsilon \neq \emptyset,$$

and

$$\pi = \texttt{NULL},$$

if $\text{Exec}^{\text{sys}}_{2n+1} \in \mathbb{T}$, then

$$\text{Exec}^{\text{sys}}_{2n+1} = 0;$$

3. for any n and $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle$ such that

$$\text{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle,$$

$A \in \mathsf{dom}\,\varepsilon$, and

$$\rho(A) = 0,$$

if $\text{Exec}^{\text{sys}}_{2n+1} \in \mathbb{T}$, then

$$\text{Exec}^{\text{sys}}_{2n+1} = 0.$$

The first constraint states that an actor will read its inputs as soon as they are safe to process. The second states that the processor π will be left idle, or `NULL`, only if no inputs are available for processing. Formally, no inputs are available when there is no mapping between actors and input actions in ε, since the latter tracks the safe-to-process events with which actors have been invoked. The last constraint states that when the processing of an actor is finished, the resulting outputs will be produced with no delay. The ρ function tracks the remaining computation time of every actor that is ready for processing. Note that the domains of ρ and ε match.

Focusing on eager executions allows us to rule out executions that behave lazily in a way that could only hurt the schedulability of a system.

We now define EDF executions as eager executions. We say that Exec^{sys} is *earliest-deadline-first* if and only if Exec^{sys} is eager and for any n, $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$, and $\langle s, \alpha\rangle$ such that

$$\text{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle,$$

and

$$\pi \notin \{A \mid \text{there is } A \text{ such that } \varepsilon(A) = \langle s, \alpha\rangle, \text{ and for every } A' \text{ and } \langle s', \alpha'\rangle \text{ such that } \varepsilon(A') = \langle s', \alpha'\rangle,\ (\mathsf{deadline}\,P)(\alpha) \leq (\mathsf{deadline}\,P)(\alpha')\},$$

if $\text{Exec}^{\text{sys}}_{2n+1} \in \mathbb{T}$, then

$$\text{Exec}^{\text{sys}}_{2n+1} = 0.$$

The constraint above states that no time is allowed to elapse if the processor is not assigned to the actor that is invoked with events that have a minimal deadline among other safe-to-process events in the system.

In other words, a system execution is earliest-deadline-first just as long as it is eager with respect to start and finish transitions, and at each time instant, allocates the processor to the actor processing the events with the smallest deadline.

Proposition 4.1.3. *If* Exec^{sys} *is earliest-deadline-first, then* Exec^{sys} *is fair.*

Proof. We first show that if there is n, $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$, A, and $\langle s, \alpha\rangle$ such that $\text{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$, $\varepsilon(A) = \langle s, \alpha\rangle$, then there is n' and $\langle Q', \iota', \varepsilon', \rho', \pi', t^{\text{sys}\prime}\rangle$ such that $n < n'$, $\text{Exec}^{\text{sys}}_{2n'} = \langle Q', \iota', \varepsilon', \rho', \pi', t^{\text{sys}\prime}\rangle$, and $A \in \mathsf{dom}\,\iota'$.

By the definition of a system execution there is n'' and $\langle Q'', \iota'', \varepsilon'', \rho'', \pi'', t^{\text{sys}\prime\prime}\rangle$ such that $n'' > n$, $\text{Exec}^{\text{sys}}_{2n''} = \langle Q'', \iota'', \varepsilon'', \rho'', \pi'', t^{\text{sys}\prime\prime}\rangle$ and $t^{\text{sys}\prime\prime} > \mathsf{time}(\varepsilon(A))$.

If $\varepsilon''(A) \neq \langle s, \alpha\rangle$ then n' exists and is between n and n''.

If $\varepsilon''(A) = \langle s, \alpha\rangle$ then the following are true:

- all new events that arrive through input transitions will have a greater deadline than α,
- if there is event $e \in Q''$ with $\mathsf{deadline}(\{e\}) < \mathsf{deadline}(\alpha)$ then there is A' and $\langle s', \alpha' \rangle$ such that $\varepsilon''(A') = \langle s', \alpha' \rangle$ and $\mathsf{deadline}(\alpha') \leq \mathsf{deadline}(\{e\})$,

Using the same argument as in the proof of 3.3.6, within a finite number of steps, the set of events and input actions with deadline smaller than $\mathsf{deadline}(\alpha)$ will be exhausted and thus A will eventually occupy the processor for $\rho(A)$ time units and the system will execute a finish transition for A.

We now show that if there is n, $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle$, A, and e such that $\text{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle$, $A \in \mathsf{dom}\, \iota$, $e \in Q$, and $\mathsf{chan}(e) \in \text{C}^A_{\text{in}}$, then there is n', $\langle Q', \iota', \varepsilon', \rho', \pi', t^{\text{sys}\prime} \rangle$, s', and α' such that $n < n'$, $\text{Exec}^{\text{sys}}_{2n'} = \langle Q', \iota', \varepsilon', \rho', \pi', t^{\text{sys}\prime} \rangle$, $A \in \mathsf{dom}\, \varepsilon'$, $\varepsilon'(A) = \langle s', \alpha' \rangle$ and $e \in \alpha'$.

We use the same reasoning as the previous case: by the definition of a system execution there is n'' and $\langle Q'', \iota'', \varepsilon'', \rho'', \pi'', t^{\text{sys}\prime\prime} \rangle$ such that $n'' > n$, $\text{Exec}^{\text{sys}}_{2n''} = \langle Q'', \iota'', \varepsilon'', \rho'', \pi'', t^{\text{sys}\prime\prime} \rangle$ and $t^{\text{sys}\prime\prime} > \mathsf{time}(\varepsilon(A))$.

If $e \notin Q''$ then n' exists and is between n and n''.

If $e \in Q''$ then the following are true:

- all new events that arrive through input transitions will have a greater deadline than e,
- for any event $e' \in Q \setminus \{e\}$ such that $\mathsf{time}(e') \leq \mathsf{time}(e) - (\mathsf{delay}\, P)(\mathsf{chan}(e), \text{C}^A_{\text{in}})$, $(\mathsf{deadline}\, P)(\{e'\}) \leq (\mathsf{deadline}\, P)(\{e\})$;
- for any A' and $\langle s', \alpha' \rangle$ such that $\varepsilon(A') = \langle s', \alpha' \rangle$, and $\mathsf{time}(\alpha') \leq \mathsf{time}(e) - (\mathsf{delay}\, P)(\text{C}^{A'}_{\text{in}}, \text{C}^A_{\text{in}})$, $(\mathsf{deadline}\, P)(\alpha') \leq (\mathsf{deadline}\, P)(\{e\})$;

As before, eventually, there will be no events or input actions that would make actor A not be safe to process and A will execute a start transition with an input action that includes e. □

The following is immediate from Proposition 4.1.3:

Theorem 4.1.4. *If* Exec^{sys} *is earliest-deadline-first, then* Exec^{sys} *is correct if and only if* Exec^{sys} *is safe.*

The following lemma states that for a given input and actor computation times, if there is a correct system execution, then there will also be a correct eager execution.

In other words, in deciding the schedulability of a system, we can focus on eager system executions.

Lemma 4.1.5. *For any $s \in I$ and every $O \in \mathsf{OR}(\langle P, R\rangle)$, if there is a correct execution Exec^{sys} of $\langle P, R\rangle$ such that $\mathsf{in}\,\text{Exec}^{\text{sys}} = s$, and for every $A \in P$, $\mathsf{req}^A(\text{Exec}^{\text{sys}}) \sqsubseteq O(A)$, then there is a correct eager execution $\text{Exec}^{\text{sys}}_{\text{eager}}$ of $\langle P, R\rangle$ such that $\mathsf{in}\,\text{Exec}^{\text{sys}}_{\text{eager}} = s$, and for every $A \in P$, $\mathsf{req}^A(\text{Exec}^{\text{sys}}_{\text{eager}}) \sqsubseteq O(A)$.*

Proof. Assume there is s, $O \in \mathsf{OR}(\langle P, R\rangle)$, and Exec^{sys} such that Exec^{sys} is a correct execution of $\langle P, R\rangle$, $\mathsf{in}\,\text{Exec}^{\text{sys}} = s$, and for every $A \in P$, $\mathsf{req}^A(\text{Exec}^{\text{sys}}) \sqsubseteq O(A)$.

We will describe how to transform Exec^{sys} into an execution $\text{Exec}^{\text{sys}\prime}$ such that $\text{Exec}^{\text{sys}\prime}$ is eager, $\text{Exec}^{\text{sys}\prime}$ is correct, $\mathsf{in}\,\text{Exec}^{\text{sys}\prime} = s$, and for every $A \in P$, $\mathsf{req}^A(\text{Exec}^{\text{sys}\prime}) = \mathsf{req}^A(\text{Exec}^{\text{sys}})$.

Let n be the smallest index such that $\text{Exec}^{\text{sys}}_{2n+1} = d_t$ where $d_t \in \mathbb{T}$ and $d_t > 0$.

We write s for state $\text{Exec}^{\text{sys}}_{2n}$.

We write $\text{Exec}^{\text{sys}}(s\ldots)$ for $\langle \text{Exec}^{\text{sys}}_{2n+1}, \text{Exec}^{\text{sys}}_{2n+2}, \ldots\rangle$.

Let $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$ be such that $\text{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$.

First, if there is an actor $A \in \mathsf{dom}\,\rho$ such that $\rho(A) = 0$, then because Exec^{sys} is fair, there is $n' > n$ such that $\text{Exec}^{\text{sys}}_{2n'+1}$ is the corresponding finish transition that removes A from $\mathsf{dom}\,\rho$. In order for the new execution to satisfy the eagerness constraints $\text{Exec}^{\text{sys}}_{2n'+1}$ has to be moved at the beginning of $\text{Exec}^{\text{sys}}(s\ldots)$.

We write s' for the state of the new execution that follows all finish transitions for actors A with $\rho(A) = 0$.

Next, for any actor A that is safe to process in s', we move the corresponding start transition from $\text{Exec}^{\text{sys}}(s\ldots)$ after state s'. Let s'' be the state that follows the start transitions. Let $\langle Q'', \iota'', \varepsilon'', \rho'', \pi'', t^{\text{sys}\prime\prime}\rangle$ be such that $s'' = \langle Q'', \iota'', \varepsilon'', \rho'', \pi'', t^{\text{sys}\prime\prime}\rangle$.

Next, let $d_{\min}$ be the smallest delay in $\mathbb{T}$ such that there is an actor A that is safe to process in $\langle Q'', \iota'', \varepsilon'', \rho'', \pi'', t^{\text{sys}\prime\prime} + d_{\min}\rangle$.

If $d_{\min} < d_t$ then we split the time transition that follows s'' to two time transitions with delays $d_{\min}$ and $d_t - d_{\min}$. Let t' be the time transition that follows s'' and $d_{t'}$ its delay.

If $\mathsf{dom}\,\varepsilon''$ is not empty and $\pi'' = \texttt{NULL}$ then we choose a time transition in $\text{Exec}^{\text{sys}}(s''\ldots)$ during which an actor A from $\mathsf{dom}\,\varepsilon''$ is executing. Let d_A be the delay of that time transition.

If d_A is smaller than $d_{t'}$ then we replace the t' time transition with t_A followed by a time transition with duration $d_A - d_{t'}$ during which $\pi = \texttt{NULL}$.

If d_A is larger than $d_{t'}$ then we split t_A into two transitions: one with duration $d_A - d_{t'}$ executing A and one with duration $d_{t'}$ executing `NULL`, and context-switch to A before time transition t'.

At this point the new execution including time transition t' is eager therefore we can restart the process outlined above in the target state of t'. □

The following theorem makes the same statement as before for earliest-deadline-first executions. It states the optimality of the EDF scheduling policy for our systems.

Theorem 4.1.6. *For any $s \in I$ and every $O \in \mathsf{OR}(\langle P, R\rangle)$, if there is a correct execution Exec^{sys} of $\langle P, R\rangle$ such that $\mathsf{in}\,\text{Exec}^{\text{sys}} = s$, and for every $A \in P$, $\mathsf{req}^A(\text{Exec}^{\text{sys}}) \sqsubseteq O(A)$, then there is a correct earliest-deadline-first execution $\text{Exec}^{\text{sys}}_{\text{EDF}}$ of $\langle P, R\rangle$ such that $\mathsf{in}\,\text{Exec}^{\text{sys}}_{\text{EDF}} = s$, and for every $A \in P$, $\mathsf{req}^A(\text{Exec}^{\text{sys}}_{\text{EDF}}) \sqsubseteq O(A)$.*

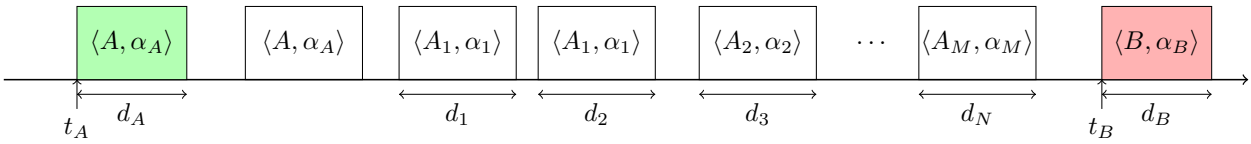


Figure 4.1: Execution that violates EDF properties.

Proof. Given a correct execution Exec^{sys} of $\langle P, R\rangle$ as described in the theorem, from lemma 4.1.5 we can construct an eager execution $\text{Exec}^{\text{sys}}_{\text{eager}}$ such that $\mathsf{in}\,\text{Exec}^{\text{sys}}_{\text{eager}} = s$, and for every $A \in P$, $\mathsf{req}^A(\text{Exec}^{\text{sys}}_{\text{eager}}) \sqsubseteq O(A)$.

We will show how we can convert $\text{Exec}^{\text{sys}}_{\text{eager}}$ into an EDF execution $\text{Exec}^{\text{sys}}_{\text{EDF}}$. The conversion consists of the following steps: identify the earliest point at which $\text{Exec}^{\text{sys}}_{\text{eager}}$ violates the EDF property; fix the violation; produce an intermediate correct execution which is EDF up to and including the initial violation point; convert the intermediate correct execution into an eager execution; repeat the process on the rest of the execution.

Assume that figure 4.1 depicts a violation of the EDF property in $\text{Exec}^{\text{sys}}_{\text{eager}}$. The horizontal axis represents system time and each rectangle labeled $\langle A, \alpha\rangle$ represents the uninterrupted execution of input action α by actor A on the processor.

For the execution to violate EDF we assume that $(\mathsf{deadline}\,P)(\alpha_A) > (\mathsf{deadline}\,P)(\alpha_B)$ and B is safe to process α_B when A starts processing α_A. Furthermore, assume that α_B has the smallest deadline among all input actions that are safe to process at time t_A and that t_B is the first time after t_A that α_B starts processing.

While the execution should process α_B at time t_A, it instead processes α_A. In order to produce an EDF execution, α_B has to be processed at time t_A. We will place as much as possible of the d_B processing time of α_B at time t_A and shift the processing of α_A to the future as appropriate.

The input actions $\alpha_1, \ldots, \alpha_M$ correspond to input actions that are "caused" by the processing of α_A. In other words, $\alpha_1, \ldots, \alpha_M$ are the input actions that contain events that causally depend on the events of α_A. In the rearrangement of $\text{Exec}^{\text{sys}}_{\text{eager}}$ into an EDF execution, that causal ordering has to be respected. In effect, this requires the processing of α_A to pushed into the processing of $\alpha_1, \ldots, \alpha_M$ and finally into the region where α_B was previously executing.

Shifting the processing of α_A will change the times at which the processing of α_A and $\alpha_1, \ldots, \alpha_M$ terminate. Therefore, the start transitions of $\alpha_1, \ldots, \alpha_M$ and the finish transitions of α_A and $\alpha_1, \ldots, \alpha_M$ will have to be shifted as well.

The intermediate shifted execution is correct. Notice that the shift does not change the program time of any of the actions. If α_i was safe to process at time t_i then α_i is guaranteed to be safe to process at any time greater than t_i. Furthermore, remember that $(\mathsf{deadline}\,P)(\alpha_A) > (\mathsf{deadline}\,P)(\alpha_B)$ and note that for every i such that $1 \le i \le M$, $(\mathsf{deadline}\,P)(\alpha_A) \le (\mathsf{deadline}\,P)(\alpha_i)$. In the intermediate execution, the processing of α_A and all α_i's terminate either at the same time as in the original execution or before B terminates.

The intermediate execution might not eagerly execute start transitions. We can tranform it into an eager one using the process described in lemma 4.1.5. □

4.2 Decidability

In the previous section we showed that in order to decide whether a model is schedulable we can narrow our search in the EDF executions of the system. Furthermore, since the properties of actor-safe and EDF system executions are local properties, we could construct a transition system whose traces are prefixes of actor-safe EDF system executions. The question of whether a non-output-safe, actor-safe, and EDF execution exists, is equivalent to whether a state in which an event misses its deadline is reachable. Is the reachability problem of a deadline miss state decidable? At first the answer seems negative since the state space of a system is infinite. However, since we are only interested in the schedulability of the system we can abstract away a big part of the state space. First, we abstract all states of a system that contain events that have missed their deadline under a new state called `error`.

Note that we have constrained the actors in a program to be output homogeneous and constant delay. Effectively this means that the timing properties of the executions of a system, and thus its schedulability, do not depend on the actor states or on the event values, and thus, those can also be abstracted away.

Next, intuitively, it should be the case that in correct executions, or executions that do not miss deadlines, the number of events in any state of the program could not grow unboundedly. Events are associated with a computation time requirement and a deadline, and thus the accumulation of too many computation requirements should conclusively lead the system to a deadline miss. One complication that arises in our programs is the fact that the deadline of an event in a channel does not only depend on that channel but also on the path that the event has followed to reach that channel. Specifically, the deadline of an event is a function of its timestamp that in principle could grow unboundedly, e.g., if the event circles around a program loop. However, what is really of interest in order to bound the number of events is the notion of relative deadline, which does not solely depend on the timestamp but rather on the difference between the timestamp and the current system time. That difference can be shown to be bounded in all correct executions for all events in every channel of a program. The lower bound naturally follows from the definition of deadline and output-safety. The upper bound, which claims that the timestamp of an event in a channel cannot grow too much relatively to system time, is a consequence of safety and the requirement that actors start processing events when they are safe to process.

Theorem 4.2.1. *For every correct execution* Exec^{sys} *of* $\langle P, R\rangle$*, every* n *and* $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$ *such that*

$$\text{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle,$$

and any $e \in Q$,

$$-(\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\mathrm{out}}) \leq \mathsf{time}(e) - t^{\mathrm{sys}} \leq (\mathsf{delay}\, P)(\mathrm{C}^P_{\mathrm{in}}, \mathsf{chan}(e)).$$

Proof. Assume that there exists correct execution $\mathrm{Exec}^{\mathrm{sys}}$ of $\langle P, R\rangle$, n, $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\mathrm{sys}}\rangle$, and e such that

$$\mathrm{Exec}^{\mathrm{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\mathrm{sys}}\rangle,$$
$$e \in Q,$$

and

$$\mathsf{time}(e) - t^{\mathrm{sys}} > (\mathsf{delay}\, P)(\mathrm{C}^P_{\mathrm{in}}, \mathsf{chan}(e)).$$

If $\mathsf{chan}(e) \in \mathrm{C}^P_{\mathrm{in}}$ then $(\mathsf{delay}\, P)(\mathrm{C}^P_{\mathrm{in}}, \mathsf{chan}(e)) = 0$, $t^{\mathrm{sys}} = \mathsf{time}(e)$ at the time of the input transition that produces e and $t^{\mathrm{sys}} \geq \mathsf{time}(e)$ thereafter.

Therefore, it cannot be the case that $\mathsf{chan}(e) \in \mathrm{C}^P_{\mathrm{in}}$, or there is actor A such that $\mathsf{chan}(e) \in \mathrm{C}^A_{\mathrm{out}}$.

Furthermore, there is n', $\langle Q', \iota', \varepsilon', \rho', \pi', t^{\mathrm{sys}\prime}\rangle$, s, α, and r such that

$$n' < n,$$

$$\mathrm{Exec}^{\mathrm{sys}}_{2n'} = \langle Q', \iota', \varepsilon', \rho', \pi', t^{\mathrm{sys}\prime}\rangle,$$
$$\mathrm{Exec}^{\mathrm{sys}}_{2n'+1} = \langle\langle A, \langle s, \alpha\rangle\rangle, r\rangle,$$

and

$$\mathsf{time}(e) = \mathsf{time}(\alpha) + (\mathsf{delay}\, A)(\mathsf{chan}(e)).$$

Since $\mathrm{Exec}^{\mathrm{sys}}$ is a correct execution, A has to be safe to process in $\mathrm{Exec}^{\mathrm{sys}}_{2n'}$. However,

$$\begin{aligned}
&\mathsf{time}(\alpha) - (\mathsf{delay}\, P)(\mathrm{C}^P_{\mathrm{in}}, \mathrm{C}^A_{\mathrm{in}}) = \\
&(\mathsf{time}(e) - (\mathsf{delay}\, A)(\mathsf{chan}(e))) - ((\mathsf{delay}\, P)(\mathrm{C}^P_{\mathrm{in}}, \mathsf{chan}(e)) - (\mathsf{delay}\, A)(\mathsf{chan}(e))) = \\
&\mathsf{time}(e) - (\mathsf{delay}\, P)(\mathrm{C}^P_{\mathrm{in}}, \mathsf{chan}(e)) > t^{\mathrm{sys}} \geq t^{\mathrm{sys}\prime}
\end{aligned}$$

The last two inequalities are by assumption and since $n' < n$, respectively. Hence, our initial assumption lead to the conclusion that there exists a start transition of actor A from a state in which A is not safe to process, or that $\mathrm{Exec}^{\mathrm{sys}}$ is not a correct execution, which is a contradiction.

Assume there exists correct execution $\mathrm{Exec}^{\mathrm{sys}}$ of $\langle P, R\rangle$, n, $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\mathrm{sys}}\rangle$, and e such that

$$\mathrm{Exec}^{\mathrm{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\mathrm{sys}}\rangle,$$
$$e \in Q,$$

and

$$\mathsf{time}(e) - t^{\mathrm{sys}} < -(\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\mathrm{out}}).$$

If $\mathsf{chan}(e) \in \mathrm{C}^P_{\mathrm{out}}$, the inequality becomes

$$\mathsf{time}(e) > t^{\mathrm{sys}}$$

which would imply that $\mathrm{Exec}^{\mathrm{sys}}$ is not output-safe. Therefore, there is A such that $\mathsf{chan}(e) \in \mathrm{C}^A_{\mathrm{in}}$. Furthermore, because we have constrained actors to be output homogeneous, and by definition of $(\mathsf{delay}\, P)$ there exists a path from $\mathsf{chan}(e)$ to $\mathrm{C}^P_{\mathrm{out}}$ with delay equal to $(\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\mathrm{out}})$, it is easy to see that after e and its descendants are processed, an event with timestamp $\mathsf{time}(e) + (\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\mathrm{out}})$ will appear at one of the programs output channels. Hence, there is n', $\langle Q', \iota', \varepsilon', \rho', \pi', t^{\mathrm{sys}\prime}\rangle$, and e' such that

$$n' > n,$$

$$\mathrm{Exec}^{\mathrm{sys}}_{2n'} = \langle Q', \iota', \varepsilon', \rho', \pi', t^{\mathrm{sys}\prime}\rangle,$$
$$e' \in Q',$$
$$\mathsf{chan}(e') \in \mathrm{C}^P_{\mathrm{out}},$$

and

$$\mathsf{time}(e') = \mathsf{time}(e) + (\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\mathrm{out}}).$$

Furthermore, since $n' > n$,

$$t^{\mathrm{sys}\prime} \geq t^{\mathrm{sys}} > \mathsf{time}(e) + (\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\mathrm{out}}) = \mathsf{time}(e')$$

or

$$t^{\mathrm{sys}\prime} > \mathsf{time}(e')$$

which implies that $\mathrm{Exec}^{\mathrm{sys}}$ is not output safe, therefore we again reached a contradiction. □

Theorem 4.2.2. *For every correct execution* $\mathrm{Exec}^{\mathrm{sys}}$ *of* $\langle P, R\rangle$*, every* n *and* $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\mathrm{sys}}\rangle$ *such that*

$$\mathrm{Exec}^{\mathrm{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\mathrm{sys}}\rangle,$$

and every $A \in P$*, the following are true:*

1. *for every* $c \in \mathrm{C}^A_{\mathrm{in}}$,

$$|\{e \mid e \in Q \text{ and } \mathsf{chan}(e) = c\}| \leq \frac{(\mathsf{delay}\, P)(\mathrm{C}^P_{\mathrm{in}}, c) + (\mathsf{delay}\, P)(c, \mathrm{C}^P_{\mathrm{out}})}{\inf R(A)};$$

2. *for every* $c \in \mathrm{C}^A_{\mathrm{out}}$,

$$|\{e \mid e \in Q \text{ and } \mathsf{chan}(e) = c\}| \leq \frac{(\mathsf{delay}\, P)(\mathrm{C}^P_{\mathrm{in}}, c) + (\mathsf{delay}\, P)(c, \mathrm{C}^P_{\mathrm{out}})}{\inf R(A)} + 1.$$

Proof. Assume A, c, n, $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle$, e, and N such that

$$A \in P,$$

$$c \in \mathrm{C}^A_{\text{in}},$$

$$\mathrm{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle,$$

$$e \in Q,$$

$$\mathsf{chan}(e) = c,$$

$$\mathsf{time}(e) = \max \{\mathsf{time}(e') \mid e' \in Q \text{ and } \mathsf{chan}(e') = c\},$$

and

$$N = |\{e \mid e \in Q \text{ and } \mathsf{chan}(e) = c\}|.$$

The deadline of e is $\mathsf{time}(e) + (\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\text{out}})$.

Note that system time cannot grow larger than $\mathsf{time}(e) + (\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\text{out}})$ before e reaches an actuator channel, otherwise e would miss its deadline and $\mathrm{Exec}^{\text{sys}}$ would not be output-safe.

Let $t^{\text{sys}\prime}$ be the system time that A finishes processing e. Since $\mathrm{Exec}^{\text{sys}}$ is output-safe, it has to at least be the case that $t^{\text{sys}\prime} \leq \mathsf{time}(e) + (\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\text{out}})$

Because e has the largest timestamp between the events in channel c, t^{sys} and $t^{\text{sys}\prime}$ are separated by at least N executions of actor A, or

$$t^{\text{sys}\prime} - t^{\text{sys}} \geq N \cdot \inf R(A).$$

Hence, since e cannot miss its deadline in $\mathrm{Exec}^{\text{sys}}$, it is necessary that

$$N \cdot \inf R(A) + t^{\text{sys}} \leq \mathsf{time}(e) + (\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\text{out}})$$

$$N \leq \frac{\mathsf{time}(e) - t^{\text{sys}} + (\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\text{out}})}{\inf R(A)}$$

Using the bound from 4.2.1 for the difference between $\mathsf{time}(e) - t^{\text{sys}}$,

$$N \leq \frac{(\mathsf{delay}\, P)(\mathrm{C}^P_{\text{in}}, \mathsf{chan}(e)) + (\mathsf{delay}\, P)(\mathsf{chan}(e), \mathrm{C}^P_{\text{out}})}{\inf R(A)}$$

Assume A, c, n, $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle$, e, and N such that

$$A \in P,$$

$$c \in \mathrm{C}^A_{\text{out}},$$

$$\mathrm{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}} \rangle,$$

$$e \in Q,$$

$$\mathsf{chan}(e) = c,$$
$$\mathsf{time}(e) = \min\{\mathsf{time}(e') \mid e' \in Q \text{ and } \mathsf{chan}(e') = c\},$$

and

$$N = |\{e \mid e \in Q \text{ and } \mathsf{chan}(e) = c\}|.$$

Let $t^{\text{sys}\prime}$ be the time of the finish transition of A that produced e.
Because e does not to miss its deadline in Exec^{sys}, it has to be:

$$t^{\text{sys}} \leq \mathsf{time}(e) + (\mathsf{delay}\, P)(c, \mathrm{C}^P_{\text{out}})$$

By choosing e to have the smallest timestamp between events in c, it can be inferred that since $t^{\text{sys}\prime}$, A has been executed $N-1$ times, therefore

$$t^{\text{sys}} - t^{\text{sys}\prime} \geq N \cdot \inf R(A).$$

Combining the two inequalities:

$$\begin{aligned}(N-1)\cdot \inf R(A) + t^{\text{sys}\prime} &\leq \mathsf{time}(e) + (\mathsf{delay}\, P)(c, \mathrm{C}^P_{\text{out}}) \\ N &\leq \frac{\mathsf{time}(e) - t^{\text{sys}\prime} + (\mathsf{delay}\, P)(c, \mathrm{C}^P_{\text{out}})}{\inf R(A)} + 1\end{aligned}$$

Again using 4.2.1:

$$N \leq \frac{(\mathsf{delay}\, P)(\mathrm{C}^P_{\text{in}}, c) + (\mathsf{delay}\, P)(c, \mathrm{C}^P_{\text{out}})}{\inf R(A)} + 1.$$

□

Theorem 4.2.2 allows to further abstract under the `error` state all those system states that have channels with more events than the specified bound.

We now show that for schedulability of a system $\langle P, R\rangle$ it is sufficient to abstract the ranges of the computation requirement function R with a single worst-case point.

We write $\mathsf{worst}\, R$ for a function from P to $\mathscr{P}_{\geq 1}\, \mathbb{T}$ such that for every $A \in P$,

$$(\mathsf{worst}\, R)(A) = \{\sup R(A)\}.$$

Theorem 4.2.3. *$\langle\langle P, R\rangle, I\rangle$ is schedulable if and only if $\langle\langle P, \mathsf{worst}\, R\rangle, I\rangle$ is schedulable.*

Proof. Since $\mathsf{OR}(\langle P, \mathsf{worst}\, R\rangle) \subseteq \mathsf{OR}(\langle P, R\rangle)$, it is easy to that if $\langle\langle P, R\rangle, I\rangle$ is schedulable then $\langle\langle P, \mathsf{worst}\, R\rangle, I\rangle$ is schedulable.

We next prove that if $\langle\langle P, \mathsf{worst}\, R\rangle, I\rangle$ is schedulable then $\langle\langle P, R\rangle, I\rangle$ is schedulable.

Note that since for every $A \in P$, $|(\mathsf{worst}\, R)(A)| = 1$ it is also the case that $|\mathsf{OR}(\langle P, \mathsf{worst}\, R\rangle)| = 1$. Specifically the unique $O_{\mathsf{worst}} \in \mathsf{OR}(\langle P, \mathsf{worst}\, R\rangle)$ is such that for every $A \in P$, for every $i \in \mathbb{N}$, $O_{\mathsf{worst}}(A)(i) = \sup R(A)$.

We assume that $\langle\langle P, \mathsf{worst}\, R\rangle, I\rangle$ is schedulable and show that $\langle\langle P, R\rangle, I\rangle$ is schedulable as well.

Let $s \in I$. Since $\langle\langle P, \mathsf{worst}\, R\rangle, I\rangle$ is schedulable, there is a correct execution Exec^{sys} of $\langle P, \mathsf{worst}\, R\rangle$ such that $\mathsf{in}\,\text{Exec}^{\text{sys}} = s$ and for every $A \in P$, $\mathsf{req}^A(\text{Exec}^{\text{sys}}) \sqsubseteq O_{\mathsf{worst}}(A)$.

Let $O \in \mathsf{OR}(\langle P, R\rangle)$. Using Exec^{sys}, we will construct a correct execution $\text{Exec}^{\text{sys}\prime}$ of $\langle P, R\rangle$ such that $\mathsf{in}\,\text{Exec}^{\text{sys}\prime} = s$ and for every $A \in P$, $\mathsf{req}^A(\text{Exec}^{\text{sys}\prime}) \sqsubseteq O(A)$.

We will now describe how we can replace the computation time requirement of a start transition in Exec^{sys} from r to some $r' \leq r$. Using that construction, we can produce the required $\text{Exec}^{\text{sys}\prime}$ described above.

Let i, A, s, and α be such that $\text{Exec}^{\text{sys}}_{2i+1}$ is a start transition and $\text{Exec}^{\text{sys}}_{2i+1} = \langle\langle A, \langle s, \alpha\rangle\rangle, r\rangle$.

Further, let j be the smallest $j > i$ such that there is $d \in \mathbb{T}$, $\langle Q_1, \iota_1, \varepsilon_1, \rho_1, \pi_1, t_1^{\text{sys}}\rangle$, $\langle Q_2, \iota_2, \varepsilon_2, \rho_2, \pi_2, t_2^{\text{sys}}\rangle$, such that $\text{Exec}^{\text{sys}}_{2j+1} = d$, $\text{Exec}^{\text{sys}}_{2j} = \langle Q_1, \iota_1, \varepsilon_1, \rho_1, \pi_1, t_1^{\text{sys}}\rangle$, $\text{Exec}^{\text{sys}}_{2j+2} = \langle Q_2, \iota_2, \varepsilon_2, \rho_2, \pi_2, t_2^{\text{sys}}\rangle$, $\pi_1 = A$, $\rho_2(A) \leq r - r'$.

In other words $\text{Exec}^{\text{sys}}_{2j+1}$ is the first time transition for which the remaining execution time of $\rho(A)$ falls below $r - r'$.

Therefore we can replace the start transition $\text{Exec}^{\text{sys}}_{2i+1}$ with $\langle\langle A, \langle s, \alpha\rangle\rangle, r'\rangle$, subtract $r - r'$ from the value of $\rho(A)$ from all states between $\text{Exec}^{\text{sys}}_{2i+2}$ and $\text{Exec}^{\text{sys}}_{2j}$; set the value of $\rho(A)$ to 0 in all states that follow $\text{Exec}^{\text{sys}}_{2j}$ up to the corresponding finish transition in Exec^{sys}, replace the time transition $\text{Exec}^{\text{sys}}_{2j+1}$ with time transition of delay $\rho_1(A) - (r = r')$, add a context switch of `NULL` and a time transition of delay $r - r'$. □

The following is immediate:

Corollary 4.2.4. *For every correct execution* Exec^{sys} *of* $\langle P, \mathsf{worst}\, R\rangle$*, every* n *and* $\langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle$ *such that*

$$\text{Exec}^{\text{sys}}_{2n} = \langle Q, \iota, \varepsilon, \rho, \pi, t^{\text{sys}}\rangle,$$

and every $A \in P$*, the following are true:*

1. *for every* $c \in \text{C}^A_{\text{in}}$*,*

$$|\{e \mid e \in Q \text{ and } \mathsf{chan}(e) = c\}| \leq \frac{(\mathsf{delay}\, P)(\text{C}^P_{\text{in}}, c) + (\mathsf{delay}\, P)(c, \text{C}^P_{\text{out}})}{\sup R(A)};$$

2. *for every* $c \in \text{C}^A_{\text{out}}$*,*

$$|\{e \mid e \in Q \text{ and } \mathsf{chan}(e) = c\}| \leq \frac{(\mathsf{delay}\, P)(\text{C}^P_{\text{in}}, c) + (\mathsf{delay}\, P)(c, \text{C}^P_{\text{out}})}{\sup R(A)} + 1.$$

Lastly, note that both system time and event logical times can grow unboundedly. However, since all operations of the EDF transition system, i.e., checking if an actor is safe to process and comparing deadlines of events, only involve differences between logical time and system time, we can replace both system time and event timestamps with a value that tracks

for each event the difference of the two. Moreover, because of Theorem 4.2.1, that value will be bounded.

This last observation together with the reductions shown before, effectively shows that if we assume a time domain $\mathbb{T}$ with some finite resolution, or, in other words, if we discretize time with some fixed quantum, then the state space of the EDF transition system is finite. If the class of input signals is also such that it can be described with a finite state system, then the product of the two systems will also have finite state and hence reachability of the `error` state will be decidable. One such a class is, for example, sporadic sources, which require finite state since after the minimum interarrival time has elapsed the distance from the last event does not have to be tracked.

In the next section we will further show that the schedulability problem remains decidable even in the case that the time domain is dense, or equal to the set of real numbers.

4.3 Reduction to reachability of timed-automata

Our goal in this section will be to show that the EDF transition system described in the previous section can be implemented using finite discrete state and real-valued clocks that can only be reset and linearly compared.

Starting from the last point of the previous section, namely the sufficiency of system time and timestamp difference, we observe that it is possible to track the relative time that an event is in the program by associating a *timer* and a *delay* with each event. At input transitions, the timer and the delay are set to zero. At time transitions, the timers of all events are incremented by the elapsed time. At start transitions, the timer and delay of one of the events of an input transition are chosen to represent the input action. At finish transitions, the delay of each event in the output action is set equal to the sum of the delay of the input action and the delay of the corresponding output channel of the actor, and the timer of each event in the output action is set equal to the timer of the input action. It is easy to see that the difference between the delay and the timer is always equal to the difference between the logical time of the event and system time. Since the latter was shown to be bounded, we can impose a bound on the value of both timers and delays. We set a limit for the value of a timer, and exactly when the timer crosses the limit, we reset it and subtract the value of the limit from the corresponding delay so that their difference does not change. In that way both timers and delays remain bounded.

So far, we have argued that the part of the EDF transition system that handles events has a bounded discrete state and uses timers that are linearly compared and reset. Therefore, it can be implemented with a timed automaton. What remains is to see whether the part of the system that deals with actor execution can also be implemented using finite state and clocks.

In the EDF transition system, actor execution is tracked using the ρ function. At the start transition of an actor A, $\rho(A)$ is set equal to a value in $R(A)$, which from Theorem 4.2.3 can be fixed to $\sup R(A)$, and the ρ value of the actor that is executing decreases as time

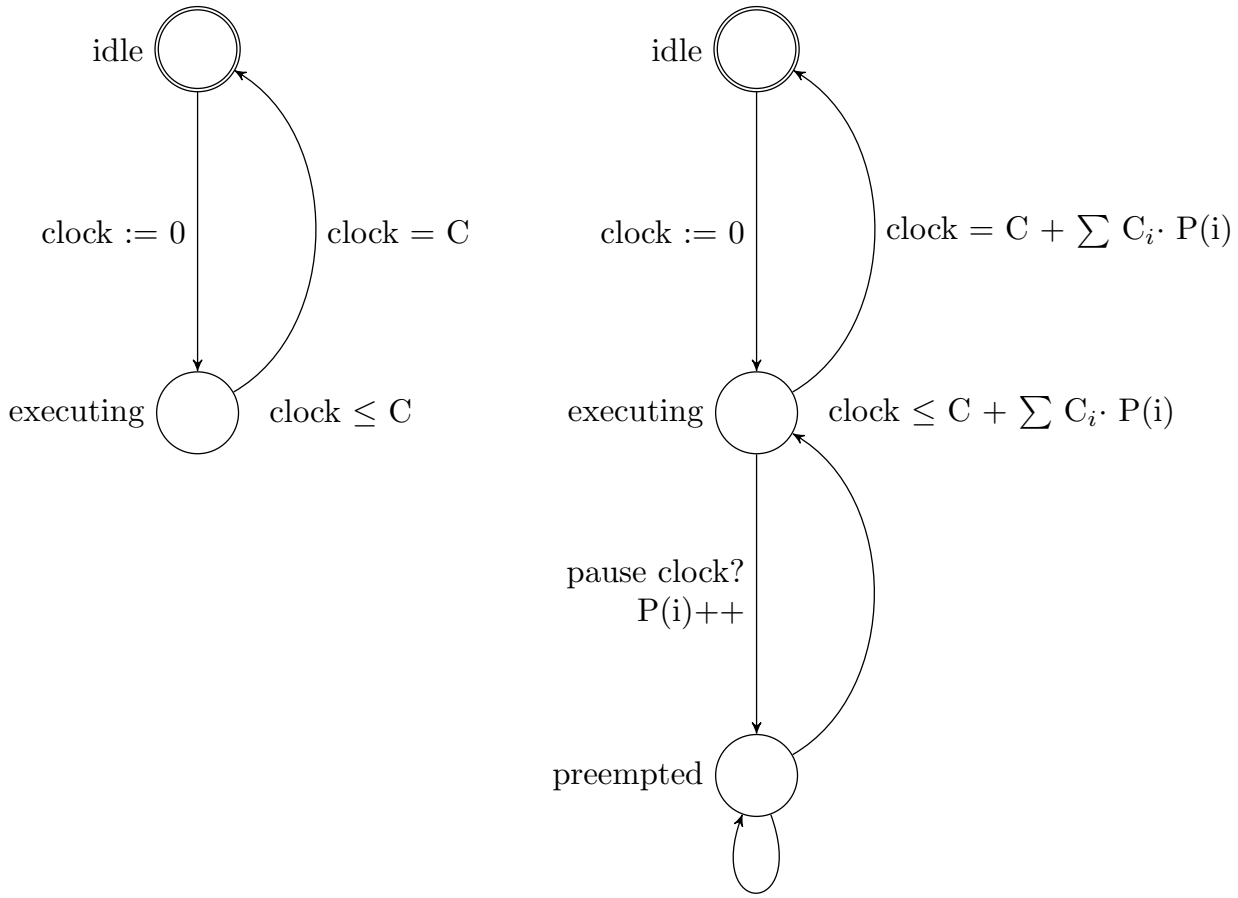


Figure 4.2: Modeling actor preemption with Timed Automata.

elapses at time transitions. An actor completes its execution when its ρ value is equal to zero. Note that this is equivalent to setting $\rho(A)$ to zero initially and increase it until it reaches $\sup R(A)$. To show that this functionality can be implemented with clocks that do not freeze when an actor is not executing, we augment the state with a value that tracks the total preemption time of an actor. When an actor A is first assigned the processor, $\rho(A)$ is set to zero, and at time transitions all ρ values are increased by the elapsed time. When an actor A is preempted by another actor B, $\sup R(B)$ is added to the preemption time of A. An actor A finishes executing when its ρ value is equal to the sum of $\sup R(A)$ and its preemption time. The scheme above is correct since, in EDF, when B preempts A, B will finish executing before A executes again. Lastly, note that an actor is added to ρ's domain when the actor is first allocated the processor and not at the start transition. This is because any delay that follows the allocation time can be accounted for precisely, whereas the interval between start time and allocation time cannot.

The mechanism is also explained in Figure 4.2. The left side of the figure shows how a timed automaton would track the execution of an actor if no preemption was allowed. Its

operation is pretty intuitive: on the transition from the idle to the executing location a clock is reset, and when the clock becomes equal to the execution time of the actor the automaton moves back to the idle location. On the right side one can see that when the automaton moves from the executing to the preempted location, ideally we would want the clock that tracks the execution time to *pause*. Since that is not possible in timed automata and because when an actor is preempted by another actor, the latter is guaranteed to terminate before the former gets to run again, we just record the preemption event in the automaton. That information is then used to decide when the actor actually finishes execution and needs to move from the executing state to the idle state.

With actor execution, all parts of the system have been shown to require finite discrete state and continuous variables that behave like clocks. Hence, the EDF transition system can be implemented as a timed automaton. In order to formally describe the equivalent timed automaton, we use timed automata with deadlines and priorities, as introduced by Bornot et al. [8], because we found them to significantly simplify modeling. Specifically, timed automata with deadlines allow us to specify the urgency of transitions. Given that EDF executions are eager and the fact that the alternative of timed safety automata would require us to encode such urgency constraints using complex invariant conditions, timed automata with deadlines proved to be very useful.

We write $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle$ for the resulting timed automaton with deadlines and priorities that simulates safe EDF executions for system $\langle P, R\rangle$, where b is the chosen limit of the event timers. Of course, in a timed automaton implementation of the system, the inputs also have to be described using a timed automaton. An *input model* of sort C is a timelock-free TADP (see [7] and formally defined below) such that the label set of the automaton is a subset of $L \subseteq C \cup \{\tau\}$, and in any run of the automaton, for each time instant, and every $c \in C$, there can only be one transition with label c.

We postulate a nonempty class $\mathbb{X}$ of *clock symbols*.

Assume a nonempty subset X of $\mathbb{X}$.

Definition 4.3.1. An X-valuation is a function from X to $\mathbb{T}$.

We write $\mathrm{V}(X)$ for the set of all X-valuations.

Definition 4.3.2. An X-constraint is a member of the smallest set of formulas Γ such that the following are true:

1. for every $x \in X$ and $r \in \mathbb{Q}$, the following are true:
   a) $x \leq r \in \Gamma$;
   b) $r \leq x \in \Gamma$;
2. for every $x_1, x_2 \in X$ and $r \in \mathbb{Q}$, the following are true:
   a) $x_1 - x_2 \leq r \in \Gamma$;
   b) $r \leq x_1 - x_2 \in \Gamma$;

3. for every $\gamma \in \Gamma$, $\neg\gamma \in \Gamma$;

4. for every $\gamma_1, \gamma_2 \in \Gamma$, $\gamma_1 \wedge \gamma_2 \in \Gamma$.

We write $\Gamma(X)$ for the set of all X-constraints.

For every X-valuation v and every X-constraint γ, we say that v *satisfies* γ, and write $\models \gamma(v)$, if and only if one of the following is true:

1. there is $x \in X$ and $r \in \mathbb{Q}$ such that one of the following is true:
   a) $\gamma = x \leq r$ and $v(x) \leq r$;
   b) $\gamma = x \geq r$ and $r \leq v(x)$;
2. there is $x_1, x_2 \in X$ and $r \in \mathbb{Q}$ such that one of the following is true:
   a) $\gamma = x_1 - x_2 \leq r$ and $v(x_1) - v(x_2) \leq r$;
   b) $\gamma = r \geq x_1 - x_2$ and $r \leq v(x_1) - v(x_2)$;
3. there is γ' such that $\gamma = \neg\gamma'$, and v does not satisfy γ';
4. there is γ_1 and γ_2 such that $\gamma = \gamma_1 \wedge \gamma_2$, and v satisfies γ_1 and γ_2.

Definition 4.3.3. A *timed automaton with deadlines and priorities* (*TADP*) is an ordered sextuple $\langle S, s_{\text{init}}, L, X, T, \preceq \rangle$ such that the following are true:

1. S is a finite set of discrete states;
2. $s_{\text{init}} \in S$ is the initial set;
3. L is a finite set of actions;
4. X is a subset of $\mathbb{X}$;
5. $T \subseteq S \times \Gamma(X) \times \Gamma(X) \times L \times \mathscr{P} X \times S$ is a set of transitions such that for every $\langle s_1, \gamma, \delta, l, U s_2 \rangle \in T$, γ is the guard of the transition, δ its deadline, U is the set of clocks to be reset, and for every $v \in \mathrm{V}(X)$, if $\models \delta(v)$, then $\models \gamma(v)$;
6. $\preceq$ is a priority order on L.

Assume a TADP $\langle S, s_{\text{init}}, L, X, T, \preceq \rangle$.

We write $\longrightarrow_{\langle S, s_{\text{init}}, L, X, T, \preceq \rangle}$ for a ternary relation between $S \times \mathrm{V}(X)$, L, and $S \times \mathrm{V}(X)$ such that for every $\langle s_1, v_1 \rangle \in S \times \mathrm{V}(X)$, $l \in L$, and $\langle s_2, v_2 \rangle \in S \times \mathrm{V}(X)$,

$$\longrightarrow_{\langle S, s_{\text{init}}, L, X, T, \preceq \rangle} (\langle s_1, v_1 \rangle, l, \langle s_2, v_2 \rangle)$$

if and only if there is γ, δ, and U such that the following are true:

1. $\langle s_1, \gamma, \delta, l, U, s_2 \rangle \in T$ and $\models \gamma(v_1)$, and for every γ', δ', l', U', and s_2' such that $\langle s_1, \gamma', \delta', l', U', s_2' \rangle \in T$ and $\models \gamma'(v_1)$, $l' \preceq l$;

2. for every $x \in X$,
$$v_2(x) = \begin{cases} 0 & \text{if } x \in U; \\ v_1(x) & \text{otherwise.} \end{cases}$$

We write $\langle s_1, v_1 \rangle \xrightarrow{l}_{\langle S, s_{\text{init}}, L, X, T, \preceq \rangle} \langle s_2, v_2 \rangle$ if and only if $\longrightarrow_{\langle S, s_{\text{init}}, L, X, T, \preceq \rangle} (\langle s_1, v_1 \rangle, l, \langle s_2, v_2 \rangle)$.

We write $\dashrightarrow_{\langle S, s_{\text{init}}, L, X, T, \preceq \rangle}$ for a ternary relation between $S \times \mathrm{V}(X)$, $\mathbb{T}$, and $S \times \mathrm{V}(X)$ such that for every $\langle s_1, v_1 \rangle \in S \times \mathrm{V}(X)$, $d \in \mathbb{T}$, and $\langle s_2, v_2 \rangle \in S \times \mathrm{V}(X)$,

$$\dashrightarrow_{\langle S, s_{\text{init}}, L, X, T, \preceq \rangle} (\langle s_1, v_1 \rangle, d, \langle s_2, v_2 \rangle)$$

if and only if the following are true:

1. $s_1 = s_2$, and for every γ, δ, l, U, and s_2' such that $\langle s_1, \gamma, \delta, l, U, s_2' \rangle \in T$, and every $d' < d$, $\not\models \delta(v_1')$, where v_1' is an X-valuation such that for every $x \in X$,
$$v_1'(x) = v_1(x) + d'.$$

2. for every $x \in X$,
$$v_2(x) = v_1(x) + d.$$

We write $\langle s_1, v_1 \rangle \xdashrightarrow{d}_{\langle S, s_{\text{init}}, L, X, T, \preceq \rangle} \langle s_2, v_2 \rangle$ if and only if $\dashrightarrow_{\langle S, s_{\text{init}}, L, X, T, \preceq \rangle} (\langle s_1, v_1 \rangle, d, \langle s_2, v_2 \rangle)$.

Definition 4.3.4. A *run* of $\langle S, s_{\text{init}}, L, X, T, \preceq \rangle$ is an infinite sequence R such that the following are true:

1. there is $v_{\text{init}} \in \mathrm{V}(X)$ such that for every $x \in X$, $v_{\text{init}}(x) = 0$, and $R(0) = \langle s_{\text{init}}, v_{\text{init}} \rangle$;

2. for every $n \in \mathbb{N}$, one of the following is true:

   a) $R(2 \cdot n) \xrightarrow{R(2 \cdot n+1)}_{\langle S, s_{\text{init}}, L, X, T, \preceq \rangle} R(2 \cdot n + 2)$;

   b) $R(2 \cdot n) \xdashrightarrow{R(2 \cdot n+1)}_{\langle S, s_{\text{init}}, L, X, T, \preceq \rangle} R(2 \cdot n + 2)$;

We say that s is *reachable* in $\langle S, s_{\text{init}}, L, X, T, \preceq \rangle$ if and only if there is a run R of $\langle S, s_{\text{init}}, L, X, T, \preceq \rangle$, n, and $v \in V(X)$ such that $R(n) = \langle s, v \rangle$.

Assume a run R of $\langle S, s_{\text{init}}, L, X, T, \preceq \rangle$.

We write $\mathsf{lapse}\, R$ for a function from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{T}$ such that for every $n_1, n_2 \in \mathbb{N}$,

$$(\mathsf{lapse}\, R)(n_1, n_2) = \sum \{R(2 \cdot n + 1) \mid n_1 \leq n < n_2 \text{ and } R(2 \cdot n + 1) \in \mathbb{T}\}.$$

We say that R is *divergent* if and only if for every $t \in \mathbb{T}$, there is $n \in \mathbb{N}$ such that

$$t \leq (\mathsf{lapse}\, R)(0, n).$$

We say that $\langle S, s_{\text{init}}, L, X, T, \preceq \rangle$ is *timelock-free* if and only if for every run R of $\langle S, s_{\text{init}}, L, X, T, \preceq \rangle$, and every $n \in \mathbb{N}$, there is a divergent run R' of $\langle S, s_{\text{init}}, L, X, T, \preceq \rangle$ such that for any $n' < n$,

$$R'(2 \cdot n') = R(2 \cdot n')$$

and

$$R'(2 \cdot n' + 1) = R(2 \cdot n' + 1).$$

Previously we described how, in a timed automaton, it is possible to simulate the difference between real-time and timestamps for each event a timer clock and an accumulated delay value. In summary, when a new event arrives in the system, it is associated with a new clock that is reset and an accumulated delay value set to zero. For new events that are generated when an actor finishes executing, the same clock as the actor input is used and the new accumulated delay values are adjusted according to the actor delay values. Since the accumulated delay and clock difference is bounded, and the clock grows continuously, we can keep both bounded by resetting the clock exactly when it crosses a specific limit and adjusting at the same time the accumulated value so that their difference stays the same.

We express the clock limit value as a multiple of the greatest common divisor of the actor delays, thereby restricting the delay accumulator values to a finite domain, as described next.

Assume $b \in \mathbb{N}$.

We write $\mathrm{R}_{\mathrm{AD}}(b)$ for a function from $\mathsf{chan}(P)$ to $\mathscr{P}_{\text{fin}}\, \mathbb{Q}$ such that for every $c \in \mathsf{chan}(P)$,

$$\mathrm{R}_{\mathrm{AD}}(b)(c) = \{i \cdot g \mid i \in \mathbb{Z} \text{ and } -(\mathsf{delay}\, P)(c, \mathrm{C}^P_{\mathrm{out}}) \leq i \cdot g \leq (\mathsf{delay}\, P)(\mathrm{C}^P_{\mathrm{in}}, c) + b \cdot g\},$$

where $g = \mathrm{GCD}(\{(\mathsf{delay}\, A)(c) \mid A \in P \text{ and } c \in \mathrm{C}^A_{\mathrm{out}}\})$.

We write $\mathrm{b}_{\mathrm{queue}}$ for a function from $\mathsf{chan}(P)$ to $\mathbb{N}$ such that for every $c \in \mathsf{chan}(P)$ and every $A \in P$,

$$\mathrm{b}_{\mathrm{queue}}(c) = \begin{cases} \left\lfloor \frac{(\mathsf{delay}\, P)(\mathrm{C}^P_{\mathrm{in}}, c) + (\mathsf{delay}\, P)(c, \mathrm{C}^P_{\mathrm{out}})}{(\sup R)(A)} \right\rfloor & \text{if } c \in \mathrm{C}^A_{\mathrm{in}}; \\ \left\lfloor \frac{(\mathsf{delay}\, P)(\mathrm{C}^P_{\mathrm{in}}, c) + (\mathsf{delay}\, P)(c, \mathrm{C}^P_{\mathrm{out}})}{(\sup R)(A)} \right\rfloor + 1 & \text{if } c \in \mathrm{C}^A_{\mathrm{out}} \cap \mathrm{C}^P_{\mathrm{out}}. \end{cases}$$

In order to track the execution of actors, we associate a clock with each actor. The clock is reset every time an actor starts executing in the processor. The execution is complete when the clock value turns equal to the actor's execution time, if the actor is not preempted. We maintain in the state a map from actors to preemption delays to manage actor preemption. When an actor is preempted by another actor, the execution time of the latter is added to the map value for the former. When the preempted actor resumes we compare its clock to the sum of its execution time and the map value.

Furthermore, since relative deadlines were shown to be bounded, if an actor is preempted too many times the events processing will lose their deadline. Observe that when an actor

A is preempted k times by an actor B, we know that at least $(k-1)\cdot(\sup R(B))$ time units have passed since A started executing. Therefore a combination of other actor execution times is a possible preemption delay for an actor as long as it does not imply that more time than the actor's maximum relative deadline has gone by.

We write $\mathrm{R_{PD}}$ for a function from P to $\mathscr{P}_{\mathrm{fin}}\,\mathbb{Q}$ such that for every $A \in P$ and every $r \in \mathbb{Q}$, $r \in \mathrm{R_{PD}}(A)$ if and only if there is a subset P' of $P \setminus \{A\}$, and a function f from P' to $\mathbb{N}$ such that

$$\sum \{f(A')\cdot(\sup R)(A') \mid A' \in P'\} \leq (\mathsf{delay}\ P)(\mathrm{C}^P_{\mathrm{in}}, \mathrm{C}^A_{\mathrm{in}}) + (\mathsf{delay}\ P)(\mathrm{C}^A_{\mathrm{in}}, \mathrm{C}^P_{\mathrm{out}})$$

and

$$r = \sum \{(f(A')+1)\cdot(\sup R)(A') \mid A' \in P'\}.$$

For every $n \in \mathbb{N}$, we fix a distinct clock symbol $\mathtt{x}_n$, and for every actor A, a distinct clock symbol $\mathtt{x}_A$.

We write $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle$ for a TADP $\langle S, s_{\mathrm{init}}, L, X, T, \preceq\rangle$ such that the following are true:

1. S is the set of all s such that one of the following is true:
   (a) s is a ordered quadruple $\langle Q, \varepsilon, \rho, \pi\rangle$ such that the following are true:
      i. Q is a function from $\mathsf{chan}(P)$ such that for every $c \in \mathsf{chan}(P)$, $Q(c) \in \mathscr{S}_{\leq \mathrm{b_{queue}}(c)}(X \times \mathrm{R_{AD}}(b)(c))$;
      ii. there is a subset P_{exec} of P such that ε is a function from P_{exec} such that for any $A \in P_{\mathrm{exec}}$,
      $$\varepsilon(A) \in X \times \bigcup \{\mathrm{R_{AD}}(b)(c) \mid c \in \mathrm{C}^A_{\mathrm{in}}\};$$
      iii. there is a subset P_{run} of $\mathsf{dom}\,\varepsilon$ such that ρ is a function from P_{run} such that for any $A \in P_{\mathrm{run}}$,
      $$\rho(A) \in \mathrm{R_{PD}}(A);$$
      iv. $\pi \in \{\mathtt{NULL}\} \cup P$;
   (b) $s = \mathtt{error}$;
2. s_{init} is an ordered quadruple $\langle Q_{\mathrm{init}}, \varepsilon_{\mathrm{init}}, \rho_{\mathrm{init}}, \pi_{\mathrm{init}}\rangle$ such that the following are true:
   (a) Q_{init} is a function from $\mathsf{chan}(P)$ such that for every $c \in \mathsf{chan}(P)$,
   $$Q_{\mathrm{init}}(c) = \langle\ \rangle;$$
   (b) $\varepsilon_{\mathrm{init}}$ is the empty function;
   (c) ρ_{init} is the empty function;
   (d) $\pi_{\mathrm{init}} = \mathtt{NULL}$;

3. $L = (\mathrm{C}_{\mathrm{in}}^P) \cup \{\texttt{start-}A, \texttt{finish-}A \mid A \in P\} \cup P \cup (\mathrm{C}_{\mathrm{out}}^P) \cup \{\texttt{bound}\} \cup \{\texttt{miss}\}$;
4. $X = \{\mathtt{x}_n \mid n \in \mathbb{N} \text{ and } n < \sum \{\mathrm{b}_{\mathrm{queue}}(c) \mid c \in \mathsf{chan}(P)\}\} \cup \{\mathtt{x}_A \mid A \in P\}$;
5. T is a subset of $S \times \Gamma(X) \times \Gamma(X) \times L \times \mathscr{P} X \times S$ such that one of the following is true:
   (a) for every $\langle Q_1, \varepsilon_1, \rho_1, \pi_1\rangle \in S$, every $\gamma \in \Gamma(X)$, every $\delta \in \Gamma(X)$, every $l \in L$, every $U \in \mathscr{P} X$, and every $\langle Q_2, \varepsilon_2, \rho_2, \pi_2\rangle \in S$,

$$\langle\langle Q_1, \varepsilon_1, \rho_1, \pi_1\rangle, \gamma, \delta, l, U, \langle Q_2, \varepsilon_2, \rho_2, \pi_2\rangle\rangle \in T$$

   if and only if one of the following is true:
      i. there is $c \in \mathrm{C}_{\mathrm{in}}^P$ and i such that the following are true:
         A. $|Q_1(c)| < \mathrm{b}_{\mathrm{queue}}(c)$;
         B. $i = \min\{j \mid$ for every $c' \in \mathsf{chan}(P)$, every δ, and any $n < |Q_1(c')|$, $Q_1(c')(n) \neq \langle \mathtt{x}_j, \delta\rangle$, and for any $A \in \mathsf{dom}\,\varepsilon_1$ and every δ, $\varepsilon_1(A) \neq \langle \mathtt{x}_j, \delta\rangle\}$;
         C. $\gamma = \texttt{true}$;
         D. $\delta = \texttt{false}$;
         E. $l = c$;
         F. $U = \{\mathtt{x}_i\}$;
         G. for every $c' \in \mathsf{chan}(P)$,

$$Q_2(c') = \begin{cases} Q_1(c') \cdot \langle\langle \mathtt{x}_i, 0\rangle\rangle & \text{if } c' = c; \\ Q_1(c') & \text{otherwise;} \end{cases}$$

         H. $\varepsilon_2 = \varepsilon_1$;
         I. $\rho_2 = \rho_1$;
         J. $\pi_2 = \pi_1$;

      This transition corresponds to an input transition that creates a new event in the system. $\mathtt{x}_i$ is the clock that gets associated with the event. By condition B, $\mathtt{x}_i$ has not been associated with any other event. The sequence of events that corresponds to input channel c, the channel of the new event, gets extended with the pair $\langle \mathtt{x}_i, 0\rangle$ so the accumulated delay of the new event is set to 0.
      ii. there is A, $I \subseteq \mathrm{C}_{\mathrm{in}}^A$, $c \in I$, and $\langle x, d\rangle$ such that the following are true:
         A. for every $c' \in I$, $|Q_1(c')| > 0$, and $\mathsf{head}\, Q_1(c) = \langle x, d\rangle$;
         B. $A \notin \mathsf{dom}\,\varepsilon_1$;
         C. $\gamma = \gamma_{\text{action}} \wedge \gamma_{\text{safe-to-process}}$, where the following are true:
            (1) $\gamma_{\text{action}} = \bigwedge\{d' - x' = d - x \mid$ there is $c' \in I$ such that $\mathsf{head}\, Q_1(c') = \langle x', d'\rangle$;
            (2) $\gamma_{\text{safe-to-process}} = \gamma_1 \wedge \gamma_2 \wedge \gamma_3$, where the following are true:

a. $\gamma_1 = x \geq d - (\mathsf{delay}\, P)(\mathrm{C}_{\mathrm{in}}^P, \mathrm{C}_{\mathrm{in}}^A)$;
b. $\gamma_2 = \bigwedge\{d' - x' + (\mathsf{delay}\, P)(c', \mathrm{C}_{\mathrm{in}}^A) > d - x \mid c' \notin I \text{ and } \mathsf{head}\, Q(c') = \langle x', d' \rangle\}$;
c. $\gamma_3 = \bigwedge \{d' - x' + (\mathsf{delay}\, P)(\mathrm{C}_{\mathrm{in}}^{A'}, \mathrm{C}_{\mathrm{in}}^A) > d - x \mid \varepsilon_1(A') = \langle x', d' \rangle\}$;

D. $\delta = \gamma$;
E. $l = \texttt{start-}A$;
F. $U = \emptyset$;
G. for every $c' \in \mathsf{chan}(P)$,

$$Q_2(c') = \begin{cases} \mathsf{tail}\, Q_1(c') & \text{if } c' \in I; \\ Q_1(c') & \text{otherwise;} \end{cases}$$

H. $\varepsilon_2 = \varepsilon_1 \cup \{\langle A, \langle x, d \rangle\rangle\}$;
I. $\rho_2 = \rho_1$;
J. $\pi_2 = \pi_1$;

This transition corresponds to a start transition by actor A. In a start transition, the actor reads a set of events from its input channels that form an input action α, i.e., they share the same timestamp. The set I is the set of channels with the events that form that input action. Condition γ_{action} guarantees that those events have the same timestamp. The three conjuncts of condition $\gamma_{\text{safe-to-process}}$ correspond to the three conditions of the definition of safe to process 3.4. The transition chooses one of the events of the input action, whose clock and accumulated delay are $\langle x, d \rangle$, and adds them with actor A to ε.

iii. there is $\langle x, d \rangle$ such that the following are true:
A. for any $c \in (\mathrm{C}_{\mathrm{out}}^{\pi_1} \setminus \mathrm{C}_{\mathrm{out}}^P)$, $|Q_1(c)| < \mathrm{b}_{\mathrm{queue}}(c)$;
B. $\varepsilon_1(\pi_1) = \langle x, d \rangle$;
C. $\gamma = \mathtt{x}_{\pi_1} = (\sup R)(\pi_1) + \rho_1(\pi_1)$;
D. $\delta = \gamma$;
E. $l = \texttt{finish-}\pi_1$;
F. $U = \emptyset$;
G. for every $c \in \mathsf{chan}(P)$,

$$Q_2(c) = \begin{cases} Q_1(c) \cdot \langle \langle x, d + (\mathsf{delay}\, \pi_1)(c) \rangle \rangle & \text{if } c \in \mathrm{C}_{\mathrm{out}}^{\pi_1}; \\ Q_1(c) & \text{otherwise;} \end{cases}$$

H. $\varepsilon_2 = \varepsilon_1 \setminus \{\langle A, \varepsilon_1(A) \rangle\}$;
I. $\rho_2 = \rho_1 \setminus \{\langle A, \rho_1(A) \rangle\}$;
J. $\pi_2 = \texttt{NULL}$;

This transition corresponds to a finish transition of the currently executing actor π_1. Since $\delta = \gamma$, it is an eager transition, and it is enabled when the actor clock $\mathtt{x}_{\pi_1}$ is equal to the sum of worst-case computation time of the actor and the time that it has been preempted $\rho_1(\pi_1)$. New events are generated on the output channels of the actor, associated with the clock of the input action that the actor was processing, and with an accumulated delay adjusted according to the delay of the actor for the corresponding output channel. The actor is removed from ε and ρ, and the processor is set to `NULL`.

iv. there is $c \in \mathrm{C}^P_{\text{out}}$ and $\langle x, d\rangle$ such that the following are true:
   A. $\mathsf{head}\, Q_1(c) = \langle x, d\rangle$;
   B. $\gamma = x = d$;
   C. $\delta = \gamma$;
   D. $l = c$;
   E. $U = \emptyset$;
   F. for every $c' \in \mathsf{chan}(P)$,

$$Q_2(c') = \begin{cases} \mathsf{tail}\, Q_1(c') & \text{if } c' = c; \\ Q_1(c') & \text{otherwise;} \end{cases}$$

   G. $\varepsilon_2 = \varepsilon_1$;
   H. $\rho_2 = \rho_1$;
   I. $\pi_2 = \pi_1$;

   This transition corresponds to an output transition. This is also an eager transition, since $\delta = \gamma$. It is enabled when the value of the clock x of an event in an output channel c, becomes equal to the accumulated delay associated with the event. If the system time was t^{sys} at that point in a system execution, then $t^{\text{sys}} - x + d$ would be equal to the timestamp of the event, and thus, when x is equal to d, t^{sys} would be equal to the timestamp of the event. Finally, the event is removed from the output queue $Q(c)$.

v. there is A and $\langle x, d\rangle$ such that the following are true:
   A. $\varepsilon_1(A) = \langle x, d\rangle$;
   B. $A \notin \mathsf{dom}\, \rho_1$;
   C. $\pi_1 = \mathtt{NULL}$;
   D. $\gamma = \bigwedge\{(\mathsf{deadline}\, P)(A, \langle x, d\rangle) \leq (\mathsf{deadline}\, P)(A', \langle x', d'\rangle) \mid \varepsilon_1(A') = \langle x', d'\rangle\}$,

      where

$$(\mathsf{deadline}\, P)(A, \langle x, d\rangle) = d - x + (\mathsf{delay}\, P)(\mathrm{C}^A_{\text{in}}, \mathrm{C}^P_{\text{out}});$$

   E. $\delta = \gamma$;
   F. $l = A$;

G. $U = \{\mathtt{x}_A\}$;
H. $Q_2 = Q_1$;
I. $\varepsilon_2 = \varepsilon_1$;
J. ρ_2 is a function from $\mathsf{dom}\,\rho_1 \cup \{A\}$ such that for every $A' \in \mathsf{dom}\,\rho_1 \cup \{A\}$,

$$\rho_2(A') = \begin{cases} 0 & \text{if } A' = A; \\ \rho_1(A') + (\sup R)(A) & \text{otherwise;} \end{cases}$$

K. $\pi_2 = A$;

This transition corresponds to a scheduler transition that puts actor A on the processor when no other actor is currently executing. Actor A executes its inputs for the first time and thus A is not in $\mathsf{dom}\,\rho$. The guard of the transition guarantees that the deadline of the input action that A is processing is the smallest among all other safe-to-process input actions. The clock of the actor $\mathtt{x}_A$ is reset, and the computation time of the actor is added to the preemption times of all other actors that have started executing, i.e., are in $\mathsf{dom}\,\rho$.

vi. there is A and $\langle x, d\rangle$ such that the following are true:
A. $\varepsilon_1(A) = \langle x, d\rangle$;
B. $A \notin \mathsf{dom}\,\rho_1$;
C. $\pi_1 \neq \mathtt{NULL}$;
D. $\gamma = (\mathsf{deadline}\,P)(A, \langle x, d\rangle) < (\mathsf{deadline}\,P)(\pi_1, \varepsilon_1(\pi_1)) \wedge$
$\bigwedge\{(\mathsf{deadline}\,P)(A, \langle x, d\rangle) \leq (\mathsf{deadline}\,P)(A', \langle x', d'\rangle) \mid \varepsilon_1(A') = \langle x', d'\rangle\}$
where
$$(\mathsf{deadline}\,P)(A, \langle x, d\rangle) = d - x + (\mathsf{delay}\,P)(\mathrm{C}^A_{\mathrm{in}}, \mathrm{C}^P_{\mathrm{out}});$$
E. $\delta = \gamma$;
F. $l = A$;
G. $U = \{\mathtt{x}_A\}$;
H. $Q_2 = Q_1$;
I. $\varepsilon_2 = \varepsilon_1$;
J. ρ_2 is a function from $\mathsf{dom}\,\rho_1 \cup \{A\}$ such that for every $A' \in \mathsf{dom}\,\rho_1 \cup \{A\}$,

$$\rho_2(A') = \begin{cases} 0 & \text{if } A' = A; \\ \rho_1(A') + (\sup R)(A) & \text{otherwise;} \end{cases}$$

K. $\pi_2 = A$;

This transition corresponds to a scheduler transition that puts actor A on the processor and preempts the currently executing actor. Actor A executes its inputs for the first time and thus A is not in $\mathsf{dom}\,\rho$. The guard of the transition

guarantees that the deadline of the input action that A is processing is the smallest among all other safe-to-process input actions and specifically strictly smaller than the deadline of the input action currently being processed. The strict inequality guarantees that input actions with equal deadlines will not perpetually preempt each other. The clock of the actor $\mathtt{x}_A$ is reset, and the computation time of the actor is added to the preemption times of all other actors that have started executing, i.e., are in $\mathsf{dom}\,\rho$.

vii. there is A and $\langle x, d\rangle$ such that the following are true:
   A. $\varepsilon_1(A) = \langle x, d\rangle$;
   B. $A \in \mathsf{dom}\,\rho_1$;
   C. $\pi_1 = \mathtt{NULL}$;
   D. $\gamma = \bigwedge\{(\mathsf{deadline}\,P)(A, \langle x, d\rangle) \leq (\mathsf{deadline}\,P)(A', \langle x', d'\rangle) \mid \varepsilon_1(A') = \langle x', d'\rangle\}$,
   where
   $$(\mathsf{deadline}\,P)(A, \langle x, d\rangle) = d - x + (\mathsf{delay}\,P)(\mathrm{C}^A_{\mathrm{in}}, \mathrm{C}^P_{\mathrm{out}});$$
   E. $\delta = \gamma$;
   F. $l = A$;
   G. $U = \emptyset$;
   H. $Q_2 = Q_1$;
   I. $\varepsilon_2 = \varepsilon_1$;
   J. $\rho_2 = \rho_1$;
   K. $\pi_2 = A$;

   This transition corresponds to a scheduler transition that resumes the execution of actor A. The guard of the transition guarantees that the deadline of the input action that A is processing is the smallest among all other safe-to-process input actions. The clock of the actor $\mathtt{x}_A$ is not reset and the preemption times of other actors are not updated since the actor had earlier started executing and was preempted.

viii. there is x and $g = \mathrm{GCD}(\{(\mathsf{delay}\,A)(c) \mid A \in P \text{ and } c \in \mathrm{C}^A_{\mathrm{out}}\})$ such that the following is true:
   A. $\gamma = x = b \cdot g$;
   B. $\delta = \gamma$;
   C. $l = \mathtt{bound}$;
   D. $U = \{x\}$;
   E. for every $c \in \mathsf{chan}(P)$, any $n < |Q_1(c)|$, and every d,
   $$Q_2(c)(n) = \begin{cases} \langle x, d - b \cdot g\rangle & \text{if } Q_1(c)(n) = \langle x, d\rangle; \\ Q_1(c)(n) & \text{otherwise;} \end{cases}$$

F. for any $A \in \mathsf{dom}\,\varepsilon_1$ and every d,

$$\varepsilon_2(A) = \begin{cases} \langle x, d - b \cdot g \rangle & \text{if } \varepsilon_1(A) = \langle x, d \rangle; \\ \varepsilon_1(A) & \text{otherwise;} \end{cases}$$

G. $\rho_2 = \rho_1$;

H. $\pi_2 = \pi_1$;

This transition is used to keep the space of possible accumulated delays of events in the automaton finite, despite the possibility of loops in the program. Exactly when any event clock x crosses a limit value, equal to $b \cdot g$, x is reset. For the semantics of the execution not to be affected, the same limit value is subtracted from the accumulated delay d of any event that is associated with that clock, so that the difference $x - d$ stays the same.

(b) for every $\langle Q, \varepsilon, \rho, \pi \rangle \in S$, every $\gamma \in \Gamma(X)$, every $\delta \in \Gamma(X)$, every $l \in L$, and every $U \in \mathscr{P}\, X$,

$$\langle \langle Q, \varepsilon, \rho, \pi \rangle, \gamma, \delta, l, U, \texttt{error} \rangle \in T$$

if and only if one of the following is true:

i. there is $c \in \mathrm{C}_{\mathrm{in}}^P$ such that the following are true:

A. $|Q(c)| = \mathrm{b}_{\mathrm{queue}}(c)$;

B. $\gamma = \texttt{true}$;

C. $\delta = \gamma$;

D. $l = c$;

E. $U = \emptyset$;

ii. there is $c \in (\mathrm{C}_{\mathrm{out}}^{\pi} \setminus \mathrm{C}_{\mathrm{out}}^P)$ such that the following are true:

A. $|Q(c)| = \mathrm{b}_{\mathrm{queue}}(c)$;

B. $\gamma = \mathtt{x}_\pi = (\sup R)(\pi) + \rho(\pi)$;

C. $\delta = \gamma$;

D. $l = \texttt{finish-}\pi$;

E. $U = \emptyset$;

iii. there is c and $\langle x, d \rangle$ such that the following are true:

A. there is n such that $Q(c)(n) = \langle x, d \rangle$;

B. $\gamma = x \geq d + (\mathsf{delay}\, P)(c, \mathrm{C}_{\mathrm{out}}^P)$;

C. $\delta = \gamma$;

D. $l = \texttt{miss}$;

E. $U = \emptyset$;

iv. there is A and $\langle x, d \rangle$ such that the following are true:

A. $\varepsilon(A) = \langle x, d \rangle$;

B. $\gamma = x \geq d + (\mathsf{delay}\ P)(\mathrm{C}^A_{\mathrm{in}}, \mathrm{C}^P_{\mathrm{out}})$;
C. $\delta = \gamma$;
D. $l = \texttt{miss}$;
E. $U = \emptyset$;

The set of these transitions are enabled when the execution has reached or can demonstrably reach a deadline violation. Transitions i and ii correspond to the case where an event is added to a queue that has reached its maximum size, and transitions iii and iv to the case where the deadline of an event, either in a queue or while being processed by an actor, is reached.

6. $\preceq$ is the least order on L such that the following are true:

   (a) for every $c \in \mathrm{C}^P_{\mathrm{in}}$, every $A_1, A_2, A_3 \in P$, $\texttt{start-}A_1, \texttt{finish-}A_2, A_3, \texttt{bound} \preceq c$;
   (b) for every $c \in \mathrm{C}^P_{\mathrm{out}}$, $c \preceq \texttt{start-}A_1, \texttt{finish-}A_2, A_3, \texttt{bound}$;
   (c) for every $c \in \mathrm{C}^P_{\mathrm{out}}$, $\texttt{miss} \preceq c$.

   The $\preceq$ order defines the priority of the transitions of the timed automaton. At each time instant, input transitions are executed first, then start, finish, scheduler, and clock bound transitions, then output transitions, and last deadline miss transitions.

Definition 4.3.5. An *input model* of sort C is a timelock-free TADP $\langle S, s_{\mathrm{init}}, L, X, T, \preceq \rangle$ such that the following are true:

1. $L \subseteq C \cup \{\tau\}$;
2. for every $c \in C$, every run R of $\langle S, s_{\mathrm{init}}, L, X, T, \preceq \rangle$, and every n_1 and n_2 such that $R(n_1) = c$ and $R(n_2) = c$,
$$0 < (\mathsf{lapse}\ R)(n_1, n_2).$$

Assume an input model IM of sort C.

We write $\mathsf{sig}^{\mathrm{prog}}\ IM$ for a subset of $\mathrm{Sig}(C)$ such that for every $s \in \mathrm{Sig}(C)$, $s \in \mathsf{sig}^{\mathrm{prog}}\ IM$ if and only if there is a run R of IM such that

$$\{\langle c, t\rangle \mid \text{there is } v \text{ such that } \langle c, t, v\rangle \in s\} = \{\langle R(2 \cdot n + 1), (\mathsf{lapse}\ R)(0, n)\rangle \mid n \in \omega \text{ and } R(2 \cdot n + 1) \in C\}.$$

Figure 4.3 shows a prefix of a run of the TADP that corresponds to the system of Figure 2.6,

Remark about priority of input transitions: The correctness of $\mathsf{TADP}\ \langle\langle P, R\rangle, b\rangle \mid\mid IM$ depends on correctly implementing the input priority property described earlier in Definition 3.4.3(4). Specifically, in order to guarantee that start transitions are restricted to actors that are safe to process, it is necessary that, for every time instant, any input transition happens before other transitions of the timed automaton. We focus on a specific channel

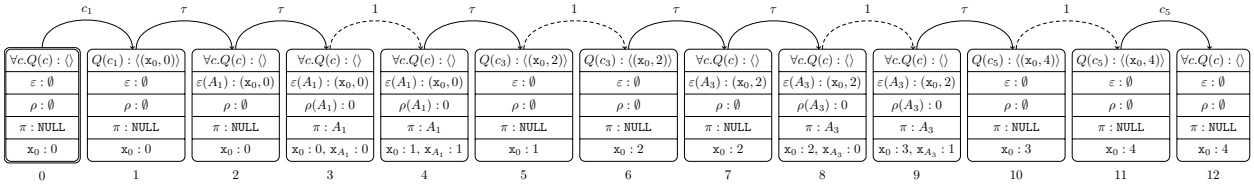


Figure 4.3: A prefix of a run of the TADP of the system of Figure 2.6 corresponding to the system execution of Figure 3.2.

$c \in \mathrm{C}_{\mathrm{in}}^{P}$, and distinguish between two cases: the input model of c can be described with a deterministic timed automaton or not. In the former case, the transitions of IM with label c will be eager, and a higher priority, $\tau \preceq c$, is sufficient to guarantee the input priority property. In the latter case, the IM will include a transition with label c that is delayable or lazy, i.e. it is not necessarily taken as soon as its guard becomes true. Assigning priority $\tau \preceq c$ will block every transition τ as soon as and for as long as that guard is true. Therefore, the implementation of the input priority property that uses $\preceq$ for inputs that correspond to non-deterministic transitions of the IM, is incorrect. Notice that a per time instant instead of global priority is required. That notion of priority can be implemented with the help of an extra clock x_p. The clock is reset in every transition with label τ, and the guard $x_p > 0$ is conjucted to the guards of all IM transitions with label c. This combination guarantees that for each time instant, a τ transition cannot be followed by an input transition c.

We say that $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle \mid\mid IM$ is *safe* if and only if for every s, $\langle \texttt{error}, s\rangle$ is not reachable in $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle \mid\mid IM$.

Theorem 4.3.6. *The following are equivalent:*

1. *for every* $s \in \mathsf{sig}^{\mathrm{prog}}\, IM$, *there is a safe earliest-deadline-first execution* $\mathrm{Exec}_{\mathrm{EDF}}^{\mathrm{sys}}$ *of* $\langle P, \mathsf{worst}\, R\rangle$ *such that* $\mathsf{in}\, \mathrm{Exec}_{\mathrm{EDF}}^{\mathrm{sys}} = s$;
2. *for every* $b > 0$, $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle \mid\mid IM$ *is safe.*

Proof. First, note that statement 1 is not equivalent to the following:

For every $s \in \mathsf{sig}^{\mathrm{prog}}\, IM$, for every earliest-deadline-first execution $\mathrm{Exec}_{\mathrm{EDF}}^{\mathrm{sys}}$ of $\langle P, \mathsf{worst}\, R\rangle$ such that $\mathsf{in}\, \mathrm{Exec}_{\mathrm{EDF}}^{\mathrm{sys}} = s$, $\mathrm{Exec}_{\mathrm{EDF}}^{\mathrm{sys}}$ is safe.

The two statements are not equivalent since there is nothing in the definition of earliest-deadline-first execution that forces it to produce outputs when system time is equal to timestamps of events in the output channels of the program.

However, the following is true: if there is a non-safe earliest-deadline-first system execution on an input signal s such that there is an output transition for every event that arrives at an output channel at a system time less than or equal to the timestamp of the event, then there is no safe earliest-deadline-first system execution on that input signal.

We assume that the scheduler transitions of a $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle$, i.e. execute v., preempt vi., and resume vii. transitions, or transitions that change the value of π to some actor

$A \neq \texttt{NULL}$ have smaller priority than the other transitions. This allows us to assume that at every time instant in a run of a $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle \;||\; IM$ there is at most one scheduler transition, and further that for every two actors $A, A' \in \mathsf{dom}\,\rho$, $\mathtt{x}_A \neq \mathtt{x}_{A'}$ since no two such clocks can be reset at the same time instant (scheduler transitions are the only transitions that reset actor clocks). This transition priority assumption is not necessary for the theorem to be true; it is only made in order to simplify the proof.

If $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle = \langle S, s_{\text{init}}, L, X, T, \preceq\rangle$, we define a relation $\mathcal{R} \subseteq \mathsf{states}(\langle\rangle, P\rangle R \times (S \times V(X))$ between states of the system $\langle P, R\rangle$ and states of the timed automaton $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle$ such that for every $\langle Q_e, \iota_e, \varepsilon_e, \rho_e, \pi_e, t_e^{\text{sys}}\rangle \in \mathsf{states}(\langle\rangle, P\rangle R$ and every $\langle\langle Q_t, \varepsilon_t, \rho_t, \pi_t\rangle, v_t\rangle \in S \times V(X)$, $\langle\langle Q_e, \iota_e, \varepsilon_e, \rho_e, \pi_e, t_e^{\text{sys}}\rangle, \langle\langle Q_t, \varepsilon_t, \rho_t, \pi_t\rangle, v_t\rangle\rangle \in \mathcal{R}$ if and only if the following true:

1. for every $c \in \mathsf{chan}(P)$, $|\{e \in Q_e \mid \mathsf{chan}(e) = c\}| = |Q_t(c)|$ and for every i and $\langle x, d\rangle$ such that $Q_t(c)(i) = \langle x, d\rangle$, there is $e \in Q_e$ such that $\mathsf{chan}(e) = c$ and $t_e^{\text{sys}} - \mathsf{time}(e) = v_t(x) + d$;

2. $\mathsf{dom}\,\varepsilon_e = \mathsf{dom}\,\varepsilon_t$ and for every $A \in \mathsf{dom}\,\varepsilon_e$ there is s, α, and $\langle x, d\rangle$ such that $\varepsilon_e(A) = \langle s, \alpha\rangle$, $\varepsilon_t(A) = \langle x, d\rangle$, and $\mathsf{time}(\alpha) = t_e^{\text{sys}} + d - v_t(x)$;

3. $\mathsf{dom}\,\rho_e = \mathsf{dom}\,\varepsilon_t$ and for every $A \in \mathsf{dom}\,\varepsilon_t$ one of the following is true:

   a) $A \notin \mathsf{dom}\,\rho_t$ and $\rho_e(A) = (\mathsf{worst}\,R)(A)$;

   b) $A \in \mathsf{dom}\,\rho_t$ and one of the following is true:

      i. for every $A' \in \mathsf{dom}\,\rho_t$, $v(\mathtt{x}_{A'}) > v(\mathtt{x}_A)$ and $\rho_e(A) = (\mathsf{worst}\,R)(A) - v_t(\mathtt{x}_A) + \rho_t(A)$

      ii. the set $S = \{v(\mathtt{x}_C) \mid C \in \mathsf{dom}\,\rho_t \text{ and } v(\mathtt{x}_C) < v(\mathtt{x}_A)\}$ is nonempty, and there is $B \in \mathsf{dom}\,\rho_t$ such that $v(\mathtt{x}_B) = \max S$, and $\rho_e(A) = (\mathsf{worst}\,R)(A) - v_t(\mathtt{x}_A) + \rho_t(A) - ((\mathsf{worst}\,R)(B) - v_t(\mathtt{x}_B) + \rho_t(B))$;

4. $\pi_e = \pi_t$.

The intuition behind constraint 3 is the following:

At any point in a run of the $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle$, if $A, B \in \mathsf{dom}\,\rho_t$ and $v_t(\mathtt{x}_A) > v_t(\mathtt{x}_B)$, then B started executing on the processor after A did, since actor clocks are only reset when an actor is first executed and never again during its execution. The relation $v_t(\mathtt{x}_A) > v_t(\mathtt{x}_B)$ therefore implies the following: (i) B has a smaller deadline than A, and (ii) B will be fully executed before A gets to run again.

From the observation above, we conclude that out of all the actors $A' \in \mathsf{dom}\,\rho_t$ with $v_t(\mathtt{x}_A) > v_t(\mathtt{x}_{A'})$, the one with the largest clock value is the one that immediately preempted actor A. Let this be actor C. If $\rho_t(A) = 0$ when C preempts A then the amount of time that A has executed for is given by $v_t(\mathtt{x}_A) - v_t(\mathtt{x}_C)$. If $\rho_t(A) \neq 0$, then A was preempted by other actors that completed their execution before C preempted A, and thus the amount of

time that A has executed for has to be adjusted to be $v_t(\mathtt{x}_A) - v_t(\mathtt{x}_C) - (\rho_t(A) - \rho_t(C) - (\mathsf{worst}\, R)(C))$.

We now continue with the proof of the theorem and show that 1 implies 2 and vice versa.

$1 \Rightarrow 2$ We prove the contrapositive $\neg 2 \Rightarrow \neg 1$:

Assuming that there is b such that $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle \mid\mid IM$ is not safe, we show that there exists $s \in \mathsf{sig}^{\text{prog}}\, IM$ and earliest-deadline-first execution $\text{Exec}^{\text{sys}}_{\text{EDF}}$ of $\langle P, \mathsf{worst}\, R\rangle$ such that $\mathsf{in}\,\text{Exec}^{\text{sys}}_{\text{EDF}} = s$ and $\text{Exec}^{\text{sys}}_{\text{EDF}}$ is not safe. Together with the observation we made about statement 1 in the beginning of the proof this is sufficient to show that statement 1 is not true.

Since $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle \mid\mid IM$ is not safe, there exists s such that $\langle \mathtt{error}, s\rangle$ is reachable in $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle \mid\mid IM$, or there exists a run R of $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle \mid\mid IM$ and n such that $R(n) = \langle \mathtt{error}, s\rangle$.

Let s be a signal such that:

$$\{\langle c, t\rangle \mid \text{there is } v \text{ such that } \langle c, t, v\rangle \in s\} = \{\langle R(2\cdot n+1), (\mathsf{lapse}\, R)(0, n)\rangle \mid n \in \omega \text{ and } R(2\cdot n+1) \in C\}.$$

Then $s \in \mathsf{sig}^{\text{prog}}\, IM$.

From run R, we construct an earliest-deadline-first execution $\text{Exec}^{\text{sys}}_{\text{EDF}}$ of $\langle P, \mathsf{worst}\, R\rangle$ such that $\mathsf{in\ prog}\,\text{Exec}^{\text{sys}}_{\text{EDF}} = s$ and $\text{Exec}^{\text{sys}}_{\text{EDF}}$ is not safe.

There is s_{init}, s^{IM}_{init}, and v_{init} such that $R(0) = \langle\langle s_{\text{init}}, s^{IM}_{\text{init}}\rangle, v_{\text{init}}\rangle$

Note that $\langle \mathsf{state}_{\text{init}}(\langle P, R\rangle), \langle s_{\text{init}}, v_{\text{init}}\rangle\rangle \in \mathcal{R}$.

For each transition of $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle$ in run R we add the corresponding transition of the system $\langle P, \mathsf{worst}\, R\rangle$, until we hit the `error` state in R. From that point on, the resulting system execution is guaranteed not to be safe in any way that it gets extended.

We describe a correspondence between transitions of $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle$ in a run R and system execution transitions such that the following is true: for every $n \leq \lfloor |R|/2 \rfloor$, there is s, s^{IM}, and v such that $R(2\cdot n) = \langle\langle s, s^{IM}\rangle, v\rangle$, and for every s^{sys} such that $s^{\text{sys}} \in \mathsf{states}(\langle P, R\rangle)$ and $\langle\langle s, v\rangle, s^{\text{sys}}\rangle \in \mathcal{R}$, the system execution transition that corresponds to $R(2\cdot n+1)$ is enabled in s^{sys} and if...

We can show that if there is $s^{\text{sys}} \in \mathsf{states}(\langle P, \mathsf{worst}\, R\rangle)$, n, s_1^{TADP}, s_1^{IM}, and v_1 such that $R(2\cdot n) = \langle\langle s_1^{\mathsf{TADP}}, s_1^{IM}\rangle, v_1\rangle$, $\langle s^{\text{sys}}, \langle s^{\mathsf{TADP}}, v\rangle\rangle \in \mathcal{R}$, and $R(2\cdot n+2) = \langle\langle s_2^{\mathsf{TADP}}, s_2^{IM}\rangle, v_2\rangle$ then the following are true:

- if $s_2^{\mathsf{TADP}} \neq \mathtt{error}$ and there is $c \in \text{C}^P_{\text{in}}$ such that $R(2\cdot n+1) = c$, then there is $s^{\text{sys}\prime}$ such that $s^{\text{sys}} \xrightarrow{c} s^{\text{sys}\prime}$ and $\langle s^{\text{sys}\prime}, \langle s_2^{\mathsf{TADP}}, v_2\rangle\rangle \in \mathcal{R}$;

- if there is $c \in \mathrm{C}_{\text{out}}^{P}$ such that $R(2 \cdot n + 1) = c$ then there is $s^{\text{sys}\prime}$ such that $s^{\text{sys}} \xrightarrow{c} s^{\text{sys}\prime}$ and $\langle s^{\text{sys}\prime}, \langle s_2^{\mathsf{TADP}}, v_2 \rangle \rangle \in \mathcal{R}$;
- if there is $A \in P$ such that $R(2 \cdot n + 1) = \texttt{start-}A$, then there is $s \in \mathsf{states}(A)$, $\alpha \in \mathsf{InputActions}(\mathrm{C}_{\text{in}}^{A})$, and $s^{\text{sys}\prime}$ such that $s^{\text{sys}} \xrightarrow{\langle A, \langle s, \alpha \rangle \rangle} s^{\text{sys}\prime}$ and $\langle s^{\text{sys}\prime}, \langle s_2^{\mathsf{TADP}}, v_2 \rangle \rangle \in \mathcal{R}$;
- if $s_2^{\mathsf{TADP}} \neq \texttt{error}$ and there is $A \in P$ such that $R(2 \cdot n + 1) = \texttt{finish-}A$ then there is $s \in \mathsf{states}(A)$, $\alpha \in \mathsf{OutputActions}(\mathrm{C}_{\text{out}}^{A})$, and $s^{\text{sys}\prime}$ such that $s^{\text{sys}} \xrightarrow{\langle A, \alpha, s \rangle} s^{\text{sys}\prime}$ and $\langle s^{\text{sys}\prime}, \langle s_2^{\mathsf{TADP}}, v_2 \rangle \rangle \in \mathcal{R}$;
- if there is $A \in P$ such that $R(2 \cdot n + 1) = A$ then there is $s^{\text{sys}\prime}$ such that $s^{\text{sys}} \xrightarrow{A} s^{\text{sys}\prime}$ and $\langle s^{\text{sys}\prime}, \langle s_2^{\mathsf{TADP}}, v_2 \rangle \rangle \in \mathcal{R}$;
- if $R(2 \cdot n + 1) = \texttt{bound}$ then $\langle s^{\text{sys}}, \langle s_2^{\mathsf{TADP}}, v_2 \rangle \rangle \in \mathcal{R}$;
- if there is $d \in \mathbb{T}$ such that $R(2 \cdot n + 1) = d$ then there is $s^{\text{sys}\prime}$ such that $s^{\text{sys}} \overset{d}{\dashrightarrow} s^{\text{sys}\prime}$ and $\langle s^{\text{sys}\prime}, \langle s_2^{\mathsf{TADP}}, v_2 \rangle \rangle \in \mathcal{R}$.

Let n be the smallest n such that there is s^{IM} and v such that $R(2 \cdot n) = \langle \langle \texttt{error}, s^{IM} \rangle, v \rangle$ and let s^{TADP} be such that $R(2 \cdot n - 2) = \langle \langle s^{\mathsf{TADP}}, s^{IM} \rangle, v \rangle$ (note that error transitions do not change the state of the input model TADP).

Using the previous observation we can indeed construct a prefix of a system execution Exec^{sys} of length n such that $\langle \text{Exec}^{\text{sys}}(n-1), s^{\mathsf{TADP}} \rangle \in \mathcal{R}$, which implies that Exec^{sys} is not safe.

Furthermore, because start and finish transitions of the TADP are urgent, and the action with the smallest deadline is chosen to execute at each point it can be shown that the constructed prefix satisfies the earliest-deadline-first conditions.

$2 \Rightarrow 1$ Again we prove the contrapositive $\neg 1 \Rightarrow \neg 2$.

Assume $s \in \mathsf{sig}^{\text{prog}}\, IM$ such that for every earliest-deadline-first execution $\text{Exec}_{\text{EDF}}^{\text{sys}}$ of $\langle P, \mathsf{worst}\, R \rangle$ such that $\mathsf{in\ prog}\, \text{Exec}_{\text{EDF}}^{\text{sys}} = s$, $\text{Exec}_{\text{EDF}}^{\text{sys}}$ is not safe.

We will show that for every $b > 0$ $\mathsf{TADP}\, \langle \langle P, R \rangle, b \rangle \mid\mid IM$ is not safe.

Let Exec^{sys} be such an earliest-deadline-first execution such that $\mathsf{in\ prog}\, \text{Exec}^{\text{sys}} = s$, Exec^{sys} is actor-safe but Exec^{sys} is not safe. Note that we can always produce an actor-safe earliest-deadline-first execution. Furthermore we assume that at every time instant the transitions in Exec^{sys} follow the order input, finish, start, and then context-switch.

By assumption and the definition of safety, Exec^{sys} cannot be output-safe. We transform Exec^{sys} so that all output transitions that can be made in time are indeed made in time. There exist output transitions for which that is not possible, since otherwise there would exist a safe earliest-deadline-first execution and that goes against the assumption. Let $\text{Exec}_{\text{EDF}}^{\text{sys}}$ be the resulting execution.

We construct a correspondence between system execution transitions and TADP transitions, in a symmetrical way to the way we did it in the first part of the proof. Thus

in the same way we can show how to construct a run R of $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle$ for which there is n such that $R(n) = \texttt{error}$.

Let t_{miss} be the system time when the first deadline miss occurs in $\text{Exec}^{\text{sys}}_{\text{EDF}}$. We set b to be equal to t_{miss}. This guarantees that no clock bound transitions are necessary in R.

For any given choice of b we can transform R to a run of $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle$ by splitting time transitions using clock bound transitions whenever an event clock grows beyond the specified bound at the old target state.

Let R^{TADP} be the resulting run.

Let R^{IM} be a run of IM such that

$$\{\langle c, t\rangle \mid \text{there is } v \text{ such that } \langle c, t, v\rangle \in s\}$$
$$= \{\langle R^{IM}(2 \cdot n + 1), (\mathsf{lapse}\, R^{IM})(0, n)\rangle \mid n \in \omega \text{ and } R^{IM}(2 \cdot n + 1) \in C\}.$$

R^{IM} exists because $s \in \mathsf{sig}^{\text{prog}}\, IM$.

Given R^{TADP} and R^{IM} we construct a run R of $\mathsf{TADP}\,\langle\langle P, R\rangle, b\rangle \mid\mid IM$ by interleaving the two runs appropriately.

□

Theorem 4.3.6, along with Theorem 4.1.4 and 4.1.6, establishes the decidability of the schedulability problem for our systems.

Chapter 5

Verification of Discrete-Event Models

In this chapter, we take a step back, and examine the verification of discrete-event programs without assigning them real-time semantics. The general goal will be to find types of constraints other than real-time that will allow us to bound the state-space of a discrete-event program. One main difference in the transition systems that we will see in this chapter is that, since the processing of events will not require any physical time, the "start" and "finish" transitions of previous chapters will be generally combined in one transition.

First, we introduce a basic, *deterministic DE* (DDE) model, where actors are simple constant (and known) delays. An actor in DDE delays every input event by a constant delay Δ, which means that if the input event has timestamp τ then the actor produces a corresponding output event with timestamp $\tau + \Delta$. A constant delay actor cannot be represented by an equivalent timed automaton, as the latter would need an unbounded number of clocks, one for every input event that may arrive within an interval Δ.

Nevertheless, we can show that the strong deterministic properties of DDE allow its state space to be reduced to a finite *lasso*. The latter can be used for exhaustive model-checking of both *signal* and *state* queries. An example of a signal query is "*is there an execution where an event with timestamp* > 10 *occurs in channel* c." An example of a state query is "*is there a reachable state where channels* c_1 *and* c_2 *contain two events with timestamps* τ_1, τ_2*, such that* $|\tau_1 - \tau_2| \leq 2$." The lasso can also be used to show that every DDE model can be transformed to an equivalent timed automaton (TA) model.

We also introduce two extensions to the basic DDE model: *non-deterministic DE* (NDE) and *DE with timed automata* (DETA). In NDE, actors are non-deterministic delays, specified by an interval, say, $[l, u]$, so that an input event is delayed by some arbitrary $\delta \in [l, u]$. In DETA, actors are either constant delays, or timed automata. A timed automaton M can be viewed as an actor which reacts to input events arriving on a given channel c by taking a discrete transition labeled with input c (we require that M be *receptive*, that is, always be able to accept any input). M can spontaneously choose to generate an output event on a given channel c' by taking a discrete transition labeled with c'.

Finally, we discuss expressiveness of the above models. We show that DDE $\subset$ NDE and DDE $\subset$ TA $\subset$ DETA, where all inclusions are strict. We also show that NDE $\not\subseteq$ TA, and

conjecture that TA $\not\subseteq$ NDE and NDE $\not\subseteq$ DETA.

5.1 Deterministic timed discrete-event models

We abstract away event values, so events are only timestamped tokens. Formally, an event is represented by a timestamp $\tau \in \mathbb{R}_{\geq 0}$, where $\mathbb{R}_{\geq 0}$ is the set of non-negative reals. The set of naturals is denoted $\mathbb{N} = \{0, 1, 2, ...\}$.

Syntax

A DDE model is a finite labeled directed graph $G = (A, C, D)$ such that

- A is the set of nodes of G. Each node is called a *DE actor* or *actor* in short.
- $C \subseteq A \times A$ is the set of edges of G. Each edge $c \in C$ is called a *channel*.
- $D : A \to \mathbb{N}$ is a (total) function mapping each actor $a \in A$ to a non-negative integer number called the *delay* of a. Note that $D(a)$ may be 0.

Let $c = (a, b) \in C$. Then c is an *output* channel of a and an *input* channel of b. We use $C^{in}(a)$ and $C^{out}(a)$ to denote the sets of input and output channels of an actor a, respectively. Let $C(a) = C^{in}(a) \cup C^{out}(a)$. By definition, G is a *closed* model, in the sense that all input channels are connected. In fact, every channel has a unique writer and a unique reader. An actor without input (respectively, output) channels is called a *source* (respectively, *sink*).

An example of a DDE model is given Figure 5.1. The model has three actors, a_1, a_2, a_3, with delays $1, 1, 0$, respectively, and four channels (the four arrows).

A *channel state* for a DE model G is a total function $r : C \to 2^{\mathbb{R}_{\geq 0}}$ which maps every channel $c \in C$ to a finite set of events initially pending on c. In Figure 5.1, the bullets annotating channels c_1, c_2 specify an *initial channel state*. In this case there are two initial events, both with timestamp 3.

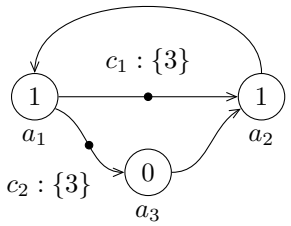


Figure 5.1: A DDE model.

Partial order: Our model allows cyclic graphs and zero-delay actors. However, we require that every cycle visits at least one actor a such that $D(a) > 0$. This condition effectively allows to "break" zero-delay loops, and to define a partial order $\prec$ on the set of actors A, so that $a \prec a'$ iff there exists a path from a to a' such that for any actor a'' in the path (including a but excluding a') we have $D(a'') = 0$. The order $\prec$ is essential for ensuring that actors are fired in timestamp order (Lemma 5.1.1) which in turn yields important deterministic properties of the DDE model. For the example of Figure 5.1, the order $\prec$ is $a_3 \prec a_2$.

Operational Semantics

To a given DDE model G and initial channel state r_0, we will associate a *timed transition system* $TTS(G, r_0) = (S, s_0, \rightarrow)$, where S is its set of states, $s_0 = (r_0, 0)$ is its (unique) initial state, and $\rightarrow \subseteq S \times A \times S$ is its transition relation, defined below. A state $s \in S$ is a pair (r, t), where r is a channel state and $t \in \mathbb{R}_{\geq 0}$ is a *global timestamp*.

The initial global timestamp is 0.

Given a channel state $r : C \rightarrow 2^{\mathbb{R}_{\geq 0}}$, let $\tau_{\min}(r) = \min \bigcup_{c \in C} r(c)$. That is, $\tau_{\min}(r)$ is the minimum timestamp among all currently pending events in r. Given actor $a \in A$ and r, we denote by $\tau_{\min}(a, r)$ the minimum timestamp among all currently pending events in the input channel(s) of a at r. That is, $\tau_{\min}(a, r) = \min \bigcup_{c \in C^{in}(a)} r(c)$. Note that $\tau_{\min}(a, r) \geq \tau_{\min}(r)$ for any a, r. By convention we set $\min \emptyset = \infty$. This implies that for an empty channel state r, we have $\tau_{\min}(r) = \infty$. Also, if $C^{in}(a) = \emptyset$, that is, if a has no input channels, then $\tau_{\min}(a, r) = \infty$ for all r.

We say that an actor $a \in A$ is *enabled* at state $s = (r, t)$, denoted *enabled*(a, s), if $\tau_{\min}(a, r) = \tau_{\min}(r) = t$. That is, a is enabled at s if there is at least one event pending in one of the inputs of a which has timestamp τ no greater than the smallest timestamp in r and τ agrees with the global time t. We say that a is *strongly enabled* at s if *enabled*(a, s) and there is no actor $b \neq a$ such that *enabled*(b, s) and $b \prec a$. That is, a is strongly enabled at s if it is enabled and there is no actor b which is also enabled at s and which comes before a according to $\prec$.

We next define the operation of *firing* an actor, and the effect that this has on the state. Intuitively, firing an actor a at state $s = (r, t)$ consists in removing the event $\tau_{\min}(a, r)$ from all input channels of a that contain this event, and adding the event $\tau_{\min}(a, r) + D(a)$ to each output channel of a. Formally, we define an auxiliary function $f(a, r, d)$ which, given actor $a \in A$, channel state r, and delay $d \in \mathbb{R}_{\geq 0}$, returns a channel state r' defined as follows:

$$r'(c) = \begin{cases} r(c) - \{\tau_{\min}(a, r)\} & \text{if } c \in C^{in}(a) \\ r(c) \cup \{\tau_{\min}(a, r) + d\} & \text{if } c \in C^{out}(a) \\ r(c) & \text{otherwise.} \end{cases} \tag{5.1}$$

We are now ready to define the transition relation $\rightarrow$ of $TTS(G, r_0)$. $\rightarrow$ has two types of transitions: *discrete transitions* of the form $s \xrightarrow{a} s'$, such that a is strongly-enabled in state $s = (r, t)$ and $s' = (r', t)$ with $r' = f(a, r, D(a))$, and *timed transitions* of the form $s \xrightarrow{\delta} s'$,

where $s, s' \in S$, $a \in A$, and $\delta \in \mathbb{R}_{\geq 0}$. Timed transitions are enabled at a state $s = (r, t)$ when no discrete transition is enabled at s and when r is not empty. In that case, it must be $t < \tau_{\min}(r)$ and $\tau_{\min}(r) \neq \infty$. Then, a timed transition $s \xrightarrow{\delta} (r, t')$ occurs, with $\delta = \tau_{\min}(r) - t$ and $t' = t + \delta = \tau_{\min}(r)$.

Remarks: (1) Global time is not affected by a discrete transition and channel state is not affected by a timed transition. (2) In $TTS(G, r_0)$ there cannot be two timed transitions in a row. (3) According to the rule $\tau_{\min}(a, r) = \infty$, source actors are never enabled, and therefore never fire. We use source actors simply to allow for initial events at the inputs of some actors.

A state $s = (r, t)$ is a *deadlock*, that is, has no outgoing transitions, iff r is empty, that is, for every $c \in C$, $r(c) = \emptyset$.

An *execution* of $TTS(G, r_0)$ is a sequence of states $\rho = s_0, s_1, ...$ such that there is a (discrete or timed) transition from every s_i to s_{i+1}. We require ρ to be *maximal*, that is, either infinite or ending in a deadlock state. Note that if the DDE model contains a loop that is reachable from some initial event, there will not be a deadlock state.

Lemma 5.1.1. *Let $\rho = s_0, s_1, ...$ be an execution of $TTS(G, r_0)$ and let a be an actor of G. For any transitions $s_i \xrightarrow{a} s_{i+1}$ and $s_j \xrightarrow{a} s_{j+1}$ in ρ such that $s_i = (r_i, t_i)$, $s_j = (r_j, t_j)$, and $i < j$, we have $t_i < t_j$.*

Lemma 5.1.1 states that every actor is fired in timestamp order, and in particular, that it cannot be fired more than once before time elapses.

$TTS(G, r_0)$ has several deterministic properties. First, by definition, if $s \xrightarrow{a} s'$ then there is no $s'' \neq s'$ such that $s \xrightarrow{a} s''$. Second, $TTS(G, r_0)$ has the so-called "diamond property":

Lemma 5.1.2. *If $s \xrightarrow{a} s_1$ and $s \xrightarrow{b} s_2$, then there is a unique s' such that $s_1 \xrightarrow{b} s'$ and $s_2 \xrightarrow{a} s'$.*

Let $\rho = s_0, s_1, \ldots$ be an execution of $TTS(G, r_0)$ where $G = (A, C, D)$ and $s_i = (r_i, t_i)$. The *signal* of a channel $c \in C$ under execution ρ, denoted σ_c^ρ, is defined to be the set of all events occurring in c along the entire execution: $\sigma_c^\rho = \bigcup_{i \in \mathbb{N}} r_i(c)$.

Lemma 5.1.3 (Kahn property [18])**.** *For any $c \in C$ and any two executions ρ_1 and ρ_2, $\sigma_c^{\rho_1} = \sigma_c^{\rho_2}$.*

Because of the Kahn property, we can write σ_c for the unique signal of a channel c. This can be viewed as the denotational semantics of DDE models.

5.2 Boundedness of DDE

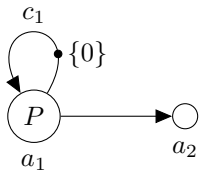


Figure 5.2: Periodic clock.

In this section we study boundedness of the state-space of DDE models. Let us begin with an illustrative example. Figure 5.2 shows a DDE model $G = (A, C, D)$, with $A = \{a_1, a_2\}$, $C = \{c_1 : (a_1, a_1), c_2 : (a_1, a_2)\}$, $D(a_1) = P$, and $D(a_2) = 0$. This model captures a periodic source with period P, generating events at times $P, 2P, \cdots$. The model includes actor a_1 which delays its input by P and a sink actor a_2. If the delay of an actor is non-zero, it is drawn inside the actor. Zero delays are not drawn. The model has two channels, c_1, c_2. Channel c_1, a self-loop of a_1, contains an initial event with timestamp 0. This is the only initial event in the system (empty initial event sets on channels are not drawn). The initial event models the seed of the periodic source. Actor a_1 adds a delay of P to the event's timestamp, outputs the event to c_2, which represents the source's output, and starts anew a cycle where the initial event is replaced with an event with timestamp P.

The initial channel state is $r_0 = \{(c_1, \{0\}), (c_2, \{\})\}$. A prefix of a path in the transition system $TTS(G, r_0)$ is the following:

$$
\begin{array}{llll}
s_0 : (c_1 : \{0\}, c_2 : \{\}, t = 0) & \xrightarrow{a_1} & s_1 : (c_1 : \{P\}, c_2 : \{P\}, t = 0) & \xrightarrow{P} \\
s_2 : (c_1 : \{P\}, c_2 : \{P\}, t = P) & \xrightarrow{a_2} & s_3 : (c_1 : \{P\}, c_2 : \{\}, t = P) & \xrightarrow{a_1} \\
s_4 : (c_1 : \{2P\}, c_2 : \{2P\}, t = P) & \xrightarrow{P} & s_5 : (c_1 : \{2P\}, c_2 : \{2P\}, t = 2P) & \xrightarrow{a_1} \\
s_6 : (c_1 : \{3P\}, c_2 : \{2P, 3P\}, t = 2P) & \xrightarrow{a_2} & s_7 : (c_1 : \{3P\}, c_2, \{3P\}, t = 2P) & \xrightarrow{P} \cdots
\end{array}
$$

Note that in state s_2 there are two events with timestamps equal to $\tau_{\min} = P$ but they are both strongly enabled since it is neither the case that $a_1 \prec a_2$ nor $a_2 \prec a_1$. Furthermore, it is easy to see that the signal in c_1 is $\sigma_{c_1} = \{i \cdot P \mid i \in \mathbb{N}\}$ and the signal in c_2 is $\sigma_{c_2} = \{i \cdot P \mid i \in \mathbb{N}_{>0}\}$.

As it can be seen from the above example, $TTS(G, r_0)$ is generally infinite-state. There are two potential sources of infinity of state-space in DE models. First, the timestamps may grow unbounded, as is the case with the above example. Second, it is unclear whether the set of events on each channel remain bounded. This is true in the above example, but is it generally true? In Section 5.2 we show that this is true for all DDE models. Then in Section 5.2 we show how timestamps can also be bounded.

Bounding the number of events in the channels

Let us begin by providing some intuition about why the number of events in an execution of $TTS(G, r_0)$ remains bounded.

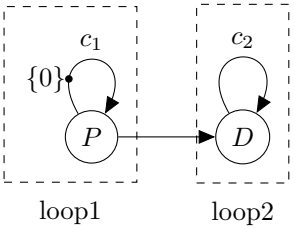


Figure 5.3: Loop example.

Consider the example in Figure 5.3. The set of events produced by "loop1" in channel c_1 is $\{i \cdot P \mid i \in \mathbb{N}\}$. Each new event with timestamp t that enters "loop2" from "loop1", will result in an infinite set of events $\{t + j \cdot D \mid j \in \mathbb{N}_{>0}\}$ in channel c_2. Therefore the set of all events in channel c_2 will be $\{i \cdot P + j \cdot D \mid i, j \in \mathbb{N}_{>0}\}$.

Because $P, D \in \mathbb{N}$, the timestamp of any event that appears in c_2 can be written as $k \cdot \mathsf{gcd}(P, D)$ for some k. In fact, there exists n, such that for all $k > n$, there exist positive i and j such that $k \cdot \mathsf{gcd}(P, D) = i \cdot P + j \cdot D$. So eventually all multiples of $\mathsf{gcd}(P, D)$ appear as timestamps of events in c_2.

Note that in every reachable state $s = (r, t)$ of $TTS(G, r_0)$, for $G = (A, C, D)$, an upper bound on the timestamp of any event is $\tau_{\min}(r) + \max\{D(a) \mid a \in A\}$, and a lower bound is $\tau_{\min}(r)$. Hence, because event timestamps in c_2 are separated by at least $\mathsf{gcd}(D, P)$, the number of events in c_2 satisfies

$$|r(c_2)| \leq \left\lceil \frac{\max\{D(a) \mid a \in A\}}{\mathsf{gcd}(D, P)} \right\rceil \text{ in any state } s = (r, t).$$

In general, let $G = (A, C, D)$ be a DE model and let r_0 be an initial channel state for G. Let $TTS(G, r_0) = (S, s_0, \to)$. Consider a state $s = (r, t) \in S$. Recall that r is a function $r : C \to 2^{\mathbb{R}_{\geq 0}}$. The *size* of r, denoted $|r|$, is defined to be

$$|r| := \sum_{c \in C} |r(c)|$$

Lemma 5.2.1. *For any two states $s_1 = (r_1, t_1)$ and $s_2 = (r_2, t_2)$ of $TTS(G, r_0)$, if $s_1 \to s_2$ then $\tau_{\min}(r_1) \leq \tau_{\min}(r_2)$.*

Proof. If there is $\delta \in \mathbb{R}_{\geq 0}$ such that $s_1 \xrightarrow{\delta} s_2$, then $r_1 = r_2$ so the statement holds.

If there is a such that $s_1 \xrightarrow{a} s_2$, then, by definition, for a channel c, either $r_2(c) = r_1(c)$, or $r_2(c) \subset r_1(c)$, or $r_2(c) = r_1(c) \cup \{\tau_{\min}(r_1) + D(a)\}$.

In the first two cases, $\min r_2(c) \geq \min r_1(c) \geq \tau_{\min}(r_1)$. In the third case, since $\min r_1(c) \geq \tau_{\min}(r_1)$ and $\tau_{\min}(r_1) + D(a) \geq \tau_{\min}(r_1)$, it will also be true that $\min r_2(c) \geq \tau_{\min}(r_1)$.

Therefore, in all cases, $\min r_2(c) \geq \tau_{\min}(r_2)$. Note also that $\tau_{\min}(r_2) = \min \bigcup_{c \in C} r_2(c) = \min_{c \in C}(\min r_2(c))$. Therefore $\tau_{\min}(r_2) \geq \min_{c \in C} \tau_{\min}(r_1) = \tau_{\min}(r_1)$. □ □

Lemma 5.2.2. *For every reachable state $s = (r, t)$ of $TTS(G, r_0)$ where $G = (A, C, D)$, if an event with timestamp τ is not an initial event then $\tau \leq \tau_{\min}(r) + \max_{a \in A} D(a)$.*

Proof. We write $D_{\max}$ for $\max_{a \in A} D(a)$.

Let $c \in C$ be the channel of the event. Since the event is not an initial event, i.e., $\tau \notin r_0(c)$, there is t', r_1, r_2, and a such that $(r_1, t') \xrightarrow{a} (r_2, t') \rightarrow^* (r, t)$, or, τ was produced by a discrete transition that precedes the current state s.

Therefore, $\tau = \tau_{\min}(r_1) + D(a) \leq \tau_{\min}(r_1) + D_{\max}$, and because of Lemma 5.2.1, $\tau \leq \tau_{\min}(r) + D_{\max}$. □ □

Theorem 5.2.3 (Boundedness of channels)**.** *There exists $K \in \mathbb{N}$ such that for every reachable state $s = (r, t)$ of $TTS(G, r_0)$, $|r| \leq K$.*

Proof. Let τ be the timestamp of an event in r. If the event is an initial event then $\tau \leq \max_{c \in C} r_0(c)$. Otherwise, by Lemma 5.2.2, $\tau \leq \tau_{\min}(r) + \max_{a \in A} D(a)$.

Therefore, an upper bound of the timestamp of any event is:

$$U = \max\left(\max_{c \in C} r_0(c), \tau_{\min}(r) + \max_{a \in A} D(a)\right)$$

and a lower bound is obviously:

$$L = \tau_{\min}(r).$$

Since by Lemma 5.2.1, $\tau_{\min}(r)$ is monotonically increasing, and because $\tau_{\min}(r) \geq 0$:

$$\begin{aligned} U - L &= \max\left(\max_{c \in C} r_0(c), \tau_{\min}(r) + \max_{a \in A} D(a)\right) - \tau_{\min}(r) \\ &= \max\left(\max_{c \in C} r_0(c) - \tau_{\min}(r), \tau_{\min}(r) + \max_{a \in A} D(a) - \tau_{\min}(r)\right) \\ &= \max\left(\max_{c \in C} r_0(c) - \tau_{\min}(r), \max_{a \in A} D(a)\right) \\ &\leq \max\left(\max_{c \in C} r_0(c), \max_{a \in A} D(a)\right) \end{aligned}$$

Let $R_{max} = \max\left(\max_{c \in C} r_0(c), \max_{a \in A} D(a)\right)$.

Let $fr(\tau)$ be the fractional part of a timestamp.

In any channel state r, the fractional part of any timestamp will be equal to the fractional part of timestamp of an initial event, since for every actor a, $D(a) \in \mathbb{N}$.

Therefore in any channel state r, an upper bound on the number of different timestamps, or the number of events in the state, is:

$$K = R_{max} \cdot \left|\{fr(\tau) \mid \tau \text{ is the timestamp of an initial event}\}\right|.$$

□

Bounding timestamps

In $TTS(G, r_0)$ timestamps of events can still grow unbounded. Moreover, there is the additional global timestamp which grows unbounded too. Nevertheless, it is easy to see how to transform $TTS(G, r_0)$ in order to obtain an equivalent *bounded timed transition system*, which we will denote $BTS(G, r_0)$. To define $BTS(G, r_0)$, we introduce some notation. Let $s = (r, t)$ be a state of $TTS(G, r_0)$. Let $\delta \in \mathbb{R}_{\geq 0}$ be such that $\delta \leq \tau_{\min}(r)$. Then we denote by $r - \delta$ the new channel state r' obtained from r by decrementing all timestamps in r by δ.

We are now ready to define $BTS(G, r_0)$. Its states are channel states, that is, the global timestamp is dropped. On the other hand, $BTS(G, r_0)$ has both discrete and timed transitions, like $TTS(G, r_0)$. A timed transition in $BTS(G, r_0)$ has the form $r \xrightarrow{\delta}_b r'$ where $\delta = \tau_{\min}(r)$ and $r' = r - \tau_{\min}(r)$. A discrete transition in $BTS(G, s_0)$ has the form $r \xrightarrow{a}_b r'$ with $r' = f(a, r, D(a))$, such that $\tau_{\min}(a, r) = \tau_{\min}(r) = 0$. That is, in $BTS(G, r_0)$, we keep track of time elapsing by appropriately decrementing the timestamps of pending events.

In the following, we write $\tau_{0,1}, \ldots, \tau_{0,N}$ for the set of N initial events in $\bigcup_{c \in C} r_0(c)$.

Lemma 5.2.4. *For every state r of $BTS(G, r_0)$, there exists a sequence σ^r such that $|\sigma^r| = N$, $\sigma^r_i \in \{-1, 1\}$, and for every event e in r, the following is true about its timestamp τ_e:*

$$\tau_e = n + \sum_{i=1}^{N} c_i \cdot \tau_{0,i} \text{ where } n \in \mathbb{Z} \text{ and } c_i \in \{0, \sigma^r_i\}.$$

Proof. We will prove the statement using induction on the transition relation of $BTS(G, r_0)$.

We first show that there is a sequence σ that satisfies the statement at the initial state r_0. The sequence σ is such that for every i, $\sigma_i = 1$. Now we need to show that for that particular σ, for every event in r_0, there is a sequence c^e and an $n \in \mathbb{Z}$ such that $c^e_i \in \{0, \sigma_i\}$ and if τ_e is the timestamp of the event then

$$\tau_e = n + \sum_{i=1}^{N} c^e_i \cdot \tau_{0,i}$$

In r_0 every event is an initial event, so its timestamp will be one of $\tau_{0,1}, \ldots, \tau_{0,N}$. Assume it is initial event e_j with timestamp $\tau_{0,j}$.

We set $c_i^{e_j}$ to be equal to 0 if $i \neq j$ and σ_i or 1 if $i = j$. We also set n to be 0.

Indeed for those choices, $n + \sum_{i=1}^{N} c_i^{e_j} \cdot \tau_{0,i} = 0 + c_j^{e_j} \cdot \tau_{0,j} = \tau_{0,j}$

Now assuming that the statement is true at a state r, we will show that it will be true for any state r' such that $r \to r'$.

By assumption, there exists sequence σ^r for r, such that for every event e with timestamp τ_e in r, there exists $n^{r,e} \in \mathbb{Z}$ and a sequence $c^{r,e}$ such that $\tau_e = n^{r,e} + \sum_{i=1}^{N} c_i^{r,e} \cdot \tau_{0,i}$.

If $r \xrightarrow{a} r'$ for some $a \in A$, then $\sigma^{r'} = \sigma^r$ and for every event e with timestamp τ_e in r' one of the following is true about e:

1. e is also in r, $c^{r',e} = c^{r,e}$, and $n^{r',e} = n^{r,e}$, or

2. e is not in r, there is event e' in r such that $\tau_e = \tau_{e'} + D(a)$, $c^{r',e} = c^{r,e'}$, and $n^{r',e} = n^{r,e'} + D(a)$.

It is easy to see that for $c^{r',e}$ and $n^{r',e}$ as defined above, the property is satisfied.

If $r \xrightarrow{\delta} r'$ where $\delta = \tau_{\min}(r)$, let $e_{\min}$ be the event in r with the minimum timestamp $\tau_{\min}$. By assumption, there are $n^{r,e_{\min}}$ and $c^{r,e_{\min}}$ such that $\tau_{\min} = n^{r,e_{\min}} + \sum_{i=1}^{N} c_i^{r,e_{\min}} \tau_{0,i}$.

We claim that the following $\sigma^{r'}$ satisfies the property for state r':

$$\sigma_i^{r'} = \begin{cases} -c_i^{r,e_{\min}} & \text{if } c_i^{r,e_{\min}} \neq 0 \\ \sigma_i^r & \text{otherwise} \end{cases}$$

Indeed, let e be any event in r with timestamp $\tau_e = n^{r,e} + \sum_{i=1}^{N} c_i^{r,e} \tau_{0,i}$. The timestamp of e in r' will be:

$$\tau'_e = n^{r,e} - n^{r,e_{\min}} + \sum_{i=1}^{N} (c_i^{r,e} - c_i^{r,e_{\min}}) \cdot \tau_{0,i}.$$

We set $n^{r',e}$ to be $n^{r,e} - n^{r,e_{\min}}$ and

$$c_i^{r',e} = \begin{cases} 0 & \text{if } c_i^{r,e} = c_i^{r,e_{\min}} \\ \sigma_i^{r'} & \text{otherwise} \end{cases}$$

We have to show that:

$$\tau'_e = n^{r',e} + \sum_{i=1}^{N} c_i^{r',e} \cdot \tau_{0,i}.$$

or that $c_i^{r',e} = c_i^{r,e} - c_i^{r,e_{\min}}$.

By definition of $c^{r',e}$ this is equivalent to $\sigma_i^{r'} = c_i^{r,e} - c_i^{r,e_{\min}}$ if $c_i^{r,e} \neq c_i^{r,e_{\min}}$.

If $c_i^{r,e_{\min}} \neq 0$, then $c_i^{r,e}$ has to be 0 for $c_i^{r,e} \neq c_i^{r,e_{\min}}$ to be true, so $c_i^{r,e} - c_i^{r,e_{\min}} = -c_i^{r,e_{\min}} = \sigma_i^{r'}$ where the last equation is true by definition of $\sigma^{r'}$.

If $c_i^{r,e_{\min}} = 0$, then $c_i^{r,e}$ has to be σ_i^r for $c_i^{r,e} \neq c_i^{r,e_{\min}}$ to be true, so $c_i^{r,e} - c_i^{r,e_{\min}} = \sigma_i^r = \sigma_i^{r'}$ where the last equation is true by definition of $\sigma^{r'}$. □

Theorem 5.2.5. *The set of reachable states of $BTS(G, r_0)$ is finite.*

Proof. At any state r of $BTS(G, r_0)$, if an event in r is not initial, then its timestamp is smaller than $\max_{a \in A} D(a)$.

Therefore, for every timestamp τ the following is true:

$$\tau \leq \max\left(\max_{c \in C} r_0(c), \max_{a \in A} D(a)\right)$$

It is also obviously true that $\tau \geq 0$.

By Lemma 5.2.4, and the fact that every timestamp in any state of $BTS(G, r_0)$ has a lower and upper bound, we conclude that there can only be a finite number of events across all states of $BTS(G, r_0)$. □

Is it easy to show that a bisimulation exists between TTS and BTS. In particular, let $s = (r, t)$ be a reachable state of $TTS(G, r_0)$. It can be easily shown, by induction on the transition relation of $TTS(G, r_0)$, that s satisfies $t \leq \tau_{\min}(r)$. We define the relation R between states of $TTS(G, r_0)$ and states of $BTS(G, r_0)$, so that R contains all pairs $((r, t), r - t)$. It can be checked that R is a bisimulation relation.

5.3 Extended discrete-event models

In this section we introduce extensions to the DDE model.

Non-deterministic DE

The *non-deterministic DE model* (NDE) extends DDE by allowing actors with variable delays, specifically intervals.

The syntax of an NDE model is almost the same as that of a DDE model. It is a labeled graph $G = (A, C, D)$, with A and C being as in a DDE model, and D associating an interval instead of a fixed value to each actor. Intervals must be nonempty, and can be of the form $[l, u]$, (l, u), (l, ∞), and so on, for $l, u \in \mathbb{N}$. When the interval is $[l, l]$ we simply write $D(a) = l$. We allow loops, but require that every loop visits at least one actor a such that $l(a) \geq 1$, where $l(a)$ is the lower bound of $D(a)$. The partial order $\prec$ is also defined in NDE, so that $a \prec a'$ iff there exists a path from a to a' such that for any actor a'' in the path (including a but excluding a') we have $l(a'') = 0$.

The semantics of NDE is defined as a timed transition system, as with DDE. Given an NDE graph G, and an initial channel state r_0, $TTS(G, r_0)$ is defined to be the tuple $(S, s_0, \to)$ where S and s_0 are as in DDE, and the transition relation $\to$ contains both discrete and timed transitions. A discrete transition of $TTS(G, r_0)$ is of the form $(r, t) \xrightarrow{a} (r', t)$ where a is strongly enabled in (r, t) and $r' = f(a, r, d)$, for some $d \in D(a)$. The definition of strongly enabled for NDE is the same as in DDE and uses the partial order $\prec$ as defined above. The timed transitions of $TTS(G, r_0)$ are defined in the same way as in DDE.

We point out that the above semantics allows to "reorder" events, in the sense that an event produced in a channel could have timestamp smaller than the events already in the channel.

However, execution of actors is still guaranteed to happen in timestamp order. Also, since we are not currently using multisets, if an event is added to a channel which already has an event with the same timestamp then the two events are merged into one.

DE with timed automata

The *DE with timed automata model* (DETA) extends DDE by allowing actors to be modeled as timed automata. This extension allows actors in a discrete-event program to model environment behavior as well as have more elaborate internal behavior than DDE and NDE.

Like DDE and NDE, a DETA model is represented by a labeled graph. In the case of DETA, a label is either a fixed delay or a timed automaton (TA). Formally, a DETA model is a graph $G = (A, C, L)$, with A and C being as in a DDE model, and L being a labeling function which maps every actor $a \in A$ to either a delay $d \in \mathbb{N}$, or a TA $M = (Q, q_0, X, I, E)$, where:

- Q is the set of *locations* of M, and $q_0 \in Q$ is its *initial location*.
- X is the set of *clocks* of M. Both Q and X are finite sets.
- I is the *invariant function* which maps every $q \in Q$ to a simple convex constraint of the form $\bigwedge_i x_i \leq k_i$, where $x_i \in X$ are clocks and $k_i \in \mathbb{N}$ are constants.
- E is the set of *transitions* of M. A transition is a tuple $h = (q, c, q', \phi, X')$ where:
  - $q, q' \in Q$: q is the source and q' the destination location of h.
  - $c \in C(a)$, i.e., c is either an input or an output channel of actor a.
  - ϕ is a simple constraint on clocks, called the *guard* of h.
  - $X' \subseteq X$ is a subset of clocks to be *reset* by h.

We require that every TA M in a DETA model be *receptive*, that is, able to accept any input event at any state. Formally, for every location q of M, and for every input channel c of a, the union of all guards of all outgoing transitions from q labeled with c must cover the whole space of clock valuations, that is, must be equivalent to the guard *true*.

We allow loops in DETA models, but we assume conservatively that the delay introduced by TA actors could be zero. Therefore we require that every loop visits at least one constant delay actor with delay ≥ 1. The partial order $\prec$ is defined for DETA in the same way as for DDE, by treating TA actors like zero-delay actors.

Before defining the semantics of DETA models we briefly recall the semantics of TA. A state of a TA M is a pair (q, v) where $q \in Q$ is a location and v is a *clock valuation*, that is, a function $v : X \to \mathbb{R}_{\geq 0}$ mapping every clock of M to a non-negative real value. We will use the term *TA state* for a pair (q, v), to avoid confusion with states of DE models, which we sometimes for clarity call *channel states*. The initial TA state of M is defined to be $(q_0, \vec{0})$, where $\vec{0}$ is the valuation assigning 0 to all clocks. M defines two types of transitions on this state-space: discrete and timed transitions. A discrete transition is possible from TA state (q, v) to TA state (q', v'), denoted $(q, v) \xrightarrow{c}_M (q', v')$, if M has a transition $h = (q, c, q', \phi, X')$ such that: (1) v satisfies the guard ϕ, denoted $v \models \phi$; (2) $v' = v[X' := 0]$, which means that $v'(x) = 0$ if $x \in X'$ and $v'(x) = v(x)$ otherwise; and (3) v' satisfies the invariant of the destination location q', denoted $v' \models I(q')$. A timed transition of delay $\delta \in \mathbb{R}_{\geq 0}$ is possible from TA state (q, v) to TA state (q, v'), denoted $(q, v) \xrightarrow{\delta}_M (q, v')$, if: (1) $v' = v + \delta$, which means that $v'(x) = v(x) + \delta$ for all $x \in X$; and (2) $v' \models I(q)$. The latter condition, together with our assumption on the form of invariants, ensures that the progress of time

from v to v' does not violate any urgency constraints at location q. Note that, since $I(q)$ is downwards-closed, $v + \delta \models I(q)$ implies that for any $\delta' \leq \delta$, we also have $v + \delta' \models I(q)$.

We are now ready to define the semantics of DETA models. Consider a DETA model $G = (A, C, L)$ and an initial channel state r_0. Let A_{TA} be the subset of A such that $a \in A_{TA}$ iff $L(a)$ is a TA. For $a \in A_{TA}$, we denote the TA $L(a)$ by M_a. Then, G and r_0 define the timed transition system $TTS(G, r_0)$. A state of $TTS(G, r_0)$ is a triple (r, w, t) where r is a channel state, w is a total function mapping actors in A_{TA} to TA states, and $t \in \mathbb{R}_{\geq 0}$ is a global timestamp. For given $a \in A_{TA}$, $w(a)$ represents the TA state which M_a is currently at.

Like the other timed transition systems defined earlier, $TTS(G, r_0)$ has two types of transitions: discrete and timed. A discrete transition has the form $(r, w, t) \xrightarrow{a} (r', w', t)$, for $a \in A$, and is possible if:

- either $a \notin A_{TA}$, that is, $L(a) \in \mathbb{N}$, in which case $r' = f(a, r, L(a))$ and $w' = w$;
- or $a \in A_{TA}$, in which case
  1. either a has an input channel c such that $t \in r(c)$, in which case:
     a) r' is obtained from r by removing the event with timestamp t from c, that is, $r'(c) = r(c) - \{t\}$ and $r'(c') = r(c')$ for all $c' \neq c$.
     b) w' is obtained from w by having M_a take the discrete transition $w(a) \xrightarrow{c}_{M_a} w'(a)$ in reaction to the event in c, and having all other TA retain their state, that is, $w'(a') = w(a)$ for all $a' \in A_{TA}, a' \neq a$.
  2. or a has an output channel c such that $w(a) \xrightarrow{c}_{M_a} w'(a)$ and for all $a' \in A_{TA}$ s.t. $a' \neq a$, we have $w'(a') = w(a)$. In this case, $r'(c) = r(c) \cup \{t\}$ and $r'(c') = r(c')$ for all $c' \neq c$.

The case $a \notin A_{TA}$ corresponds to the case where a standard DDE actor fires, that is, an actor introducing a deterministic delay. The case $a \in A_{TA}$ corresponds to the case where a TA actor fires, that is, makes a discrete transition. In this case, the following subcases are possible:

- Either a consumes an event from an input channel and reacts to it (Case 1). Note that since M_a is assumed to be receptive, the transition $w(a) \xrightarrow{c}_{M_a} w'(a)$ is guaranteed to exist. Also note that it is by definition impossible for a TA actor to consume multiple events from multiple input channels in a single transition. This is true even when all these events may have the same timestamp. On the other hand, in that case the TA actor will consume all these events in a series of discrete *simultaneous* transitions, that is, without time passing in-between these transitions.
- Or a "spontaneously" produces an event to an output channel (Case 2).

A timed transition in $TTS(G, r_0)$ has the form $(r, w, t) \xrightarrow{\delta} (r, w', t + \delta)$, for $\delta \in \mathbb{R}_{\geq 0}$, and is possible if:

1. $t + \delta \leq \tau_{\min}(r)$; and

2. for all $a \in A_{TA}$, M_a has a timed transition by δ, that is, $w(a) \xrightarrow{\delta}_{M_a} w'(a)$ is a valid transition.

That is, a timed transition by δ is possible if it is possible for every TA in the system to let time elapse by δ, and also if this does not violate the urgency of any pending event in the system. Note that it is possible in DETA to have several timed transitions in a row.

An example of a DETA model is provided in the left part of Figure 5.4. There are four actors in this model, one of which, a_2, is a TA actor. The automaton for a_2 has two locations, q_0, q_1, and a single clock x. The invariant at q_1 is $x \leq 1$, whereas the invariant at q_0 is *true* and therefore not shown. The guard in the transition from q_0 to q_1 is also *true*, and not shown either. The label $x := 0$ means that x is reset on the corresponding transition (absence of such a label means that the clock is not reset). In the transitions of the automaton, we use the label $c?$ instead of c when c is an input channel, to emphasize the fact that the actor consumes an event from c. Similarly, we use $c!$ when c is an output channel, to emphasize the fact that the actor produces an event in c. A sample execution of this DETA model is provided in the right part of Figure 5.4.

Executions and signals in NDE and DETA: The notions of executions and signals can be easily extended from DDE to NDE and DETA models. Because of non-determinism in both NDE and DETA, Lemmas 5.1.2 and 5.1.3 do not hold in neither NDE nor DETA. This means in particular that the signal σ_c^ρ of a given channel c in these models generally depends on the execution ρ. For NDE and DETA models, we define σ_c to be the union of σ_c^ρ over all executions ρ.

Unboundedness of NDE and DETA: Boundedness does not hold for neither NDE nor DETA models. We can show that if we feed a variable delay with a periodic stream of events we can construct a sporadic stream which, in turn, if fed into a periodic loop causes Theorem 5.2.3 to fail. Figure 5.5 illustrates the idea. In this model, the variable

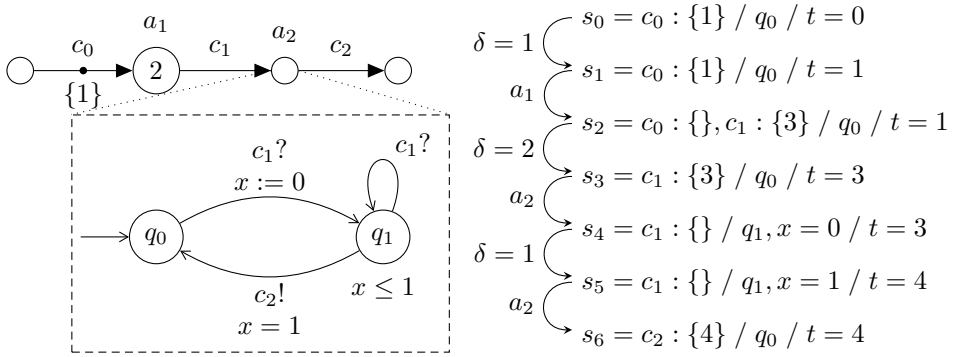


Figure 5.4: A DETA model (left) and a sample execution (right).

delay is implemented as a TA, resulting in a DETA model. The variable delay can also be implemented as an NDE actor a with $D(a) = [l, u]$, resulting in an NDE model.

More precisely, assume that the input stream of a has period P, as shown in Figure 5.5, and that $P > u$. Then the TA shown in the figure correctly implements a variable delay and every event coming out of the loop with period P will be given a variable delay $[l, u]$. Let a add delay $l + \frac{u-l}{i}$ to its i^{th} input event. This will result in an output stream of events $\{l + u - l, P + l + \frac{u-l}{2}, 2 \cdot P + l + \frac{u-l}{4}, \ldots\}$.

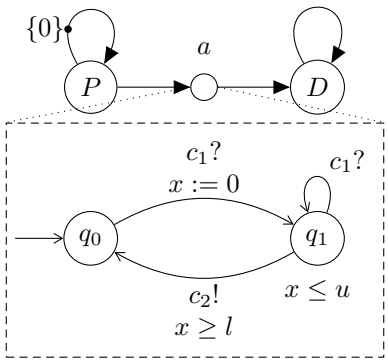


Figure 5.5: Unbounded DETA model.

In general, if a signal of the form $\{i \cdot P + \frac{x}{2^i} \mid i \in \mathbb{N}\}$ is fed into a loop with delay D, then the signal in the loop will contain events $\{i \cdot D + j \cdot P + \frac{x}{2^j} \mid i, j \in \mathbb{N}\}$. This set of events has the property that there is no bound $K \in \mathbb{N}$ such that for every n the number of events in window $[n, n + 1]$ is less than K. Intuitively the reason is that for any K, an n can be found such that the equation $i \cdot D + j \cdot P = n$ has more than K solutions, and since, for large enough j, $x/2^j < 1$, the window $[n, n + 1]$ will contain more than K events.

5.4 Verification

We begin by defining the types of verification queries that we are interested in.

Let G be a DE model (i.e., a DDE, NDE, or DETA model) with set of channels C and let r_0 be an initial channel state. Let ρ be an execution of $TTS(G, r_0)$. Recall that σ_c^ρ, for channel $c \in C$, denotes the set of all events (timestamps) that occur in c along execution ρ in $TTS(G, r_0)$. A *signal query* is a query of the form "does σ_c^ρ satisfy some property ϕ?", where ϕ is a property written in (some subclass of) first-order logic. For instance, the property "an event occurs in c" can be written as $\phi := \exists \tau : \tau \geq 0$. The property "two events occur in c at two distinct times in the interval $[1, 2]$" can be written as $\exists \tau_1, \tau_2 : \tau_1 \neq \tau_2 \wedge 1 \leq \tau_1, \tau_2, \leq 2$. The property "two events occur in c separated by at most 1 time unit" can be written as $\exists \tau_1, \tau_2 : |\tau_1 - \tau_2| \leq 1$.

We are also interested in queries which involve states of $TTS(G, r_0)$. A *state query* asks whether there exists a reachable state $s = (r, t)$ such that r satisfies some property ψ. Again,

we can imagine various types of properties ψ. For example, given constant $k \in \mathbb{N}$, ψ could be the expression $|r| > k$, which states that there are more than k events pending in the system, or the expression $|r(c)| > k$, for given $c \in C$, which states that there are more than k events pending on channel c. ψ could also be an expression such as those mentioned for signal queries above, stating, for example, that r contains an event with a certain timestamp or timestamp bounds, two events with a certain time difference, etc.

Channel signals are denotational semantics of DE models and signal queries allow to express natural properties on these. State queries are also important, as they refer to system snapshots as well as to implementation properties such as buffer space requirements. In the rest of this section we discuss how signal and state queries can be automatically checked on DDE models.

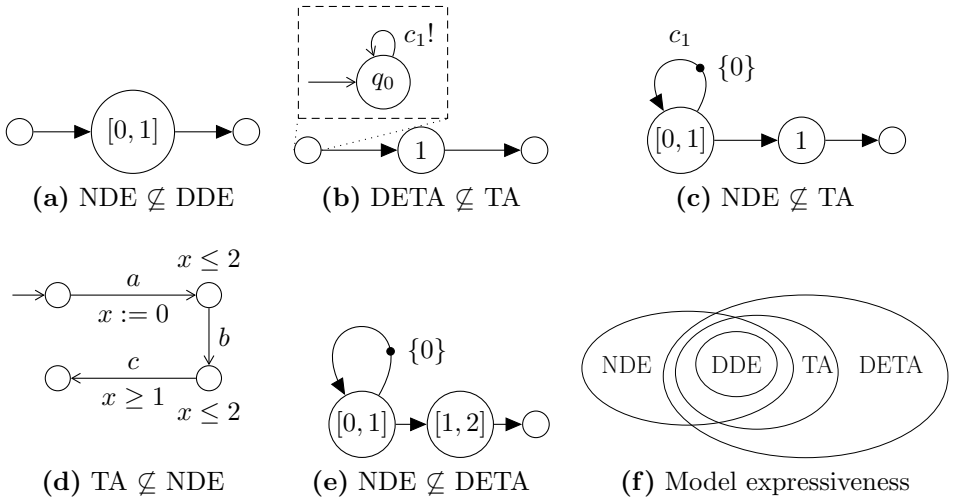
First, consider signal queries. They can be checked with the help of a *lasso* derived from BTS. This lasso is a finite and deterministic transition system (deterministic in the sense that every state has at most one successor), derived from BTS by merging all enabled discrete transitions from a given state s into a single *supertransition* where all corresponding actors fire. The diamond property (which by the bisimulation property also holds on BTS) ensures that this transformation is valid. We can analyze this lasso and compute, for every channel c, an affine expression that describes σ_c. Then we can reduce the problem of checking whether σ_c satisfies a signal query ϕ to an SMT (satisfiability modulo theory) problem. For example, the affine expression for channel c_2 for the example of Figure 5.3 is $i \cdot P + j \cdot D$, with $i, j \in \mathbb{N}_{>0}$. Checking whether there exist two events τ_1, τ_2 in σ_{c_2} such that $\tau_1 - \tau_2 = 5$, can then be reduced to checking satisfiability of the expression $\tau_1 = i_1 \cdot P + j_1 \cdot D \wedge \tau_2 = i_2 \cdot P + j_2 \cdot D \wedge \tau_1 - \tau_2 = 5$.

Second, for state queries, we can again use the lasso and the bisimulation of BTS and TTS to compute an affine expression characterizing the set of timestamps on a *per state* basis. We can then reduce the problem of whether there exists a reachable state satisfying a property ψ to a series of SMT problems, one for every affine expression computed for every state in the lasso.

5.5 Expressiveness

In this section we discuss how the various DE models introduced above are related to each other, as well as to timed automata, in terms of expressiveness. We write $\mathcal{A} \subseteq \mathcal{B}$ if for every model G in formalism $\mathcal{A}$ there exists a model G' in formalism $\mathcal{B}$ such that G and G' are equivalent in terms of denotational semantics, i.e., channel signals. More precisely, G and G' are equivalent if they refer to the same set of channels C, and for every $c \in C$, $\sigma_c^G = \sigma_c^{G'}$, where $\sigma_c^G, \sigma_c^{G'}$ are the signals of c in G, G', respectively.

To be able to compare the DE models with timed automata, we view TA as a subclass of DETA models. Concretely, suppose M is a TA whose transitions are labeled with $c_1, c_2, ..., c_n$. Then M can be seen as a DETA model with $n+1$ actors, $a, a_1, ..., a_n$, where a is a source actor labeled by M, and $a_1, ..., a_n$ are sink actors connected to a. In this interpretation, every label c_i of M is seen as a distinct output channel of a.



(a) NDE $\not\subseteq$ DDE **(b)** DETA $\not\subseteq$ TA **(c)** NDE $\not\subseteq$ TA

(d) TA $\not\subseteq$ NDE **(e)** NDE $\not\subseteq$ DETA **(f)** Model expressiveness

Figure 5.6: Models used in expressiveness discussion.

The expressiveness results, summarized in Figure 5.6(f), are as follows:

DDE $\subsetneq$ NDE: DDE $\subseteq$ NDE because the fixed delay d can be expressed as the interval $[d, d]$. NDE $\not\subseteq$ DDE because NDE allows non-deterministic behavior but DDE does not. Indeed, the NDE model of Figure 5.6(a) produces a single event on channel c at time $t \in [0, 1]$, but this is impossible to express in DDE.

DDE $\subsetneq$ TA: TA $\not\subseteq$ DDE because TA allows non-deterministic behavior but DDE does not. The example of Figure 5.6(a) can be easily constructed with TA. To see why DDE $\subseteq$ TA, consider the lasso defining the signals of a DDE model, discussed in Section 5.4. The affine expressions describing the channel signals can be directly transformed into parallel compositions of simple TA with periodic self-loops. For example, $(2 + 3 \cdot i) \cup (3 + 7 \cdot j)$ can be trivially transformed into the parallel composition of two TA.

DDE $\subsetneq$ DETA: DDE $\subseteq$ DETA because every DDE model is by definition a DETA model (one that has no TA actors). DETA $\not\subseteq$ DDE again because of non-determinism.

TA $\subsetneq$ DETA: As defined above, TA is by definition a subclass of DETA. To see that DETA $\not\subseteq$ TA, consider the DETA model shown in Figure 5.6(b). In this model, every event produced by the TA on channel c_1 is delayed by the constant delay actor by exactly 1 time unit. Since there is no bound on the number of events that can be produced on c_1 in a time interval of size 1, an equivalent TA model would require an unbounded number of clocks.

NDE $\not\subseteq$ TA: To see this, consider the example of Figure 5.6(c). Similarly to the model of Figure 5.6(b), in this model the number of events that can be produced in an interval of size 1 on channel c_2 is unbounded, and for every such event a TA implementation would require a separate clock to produce the corresponding event on channel c_3.

We also conjecture that the TA of Figure 5.6(d) cannot be implemented in NDE. This TA produces three events, a, b, c, in that order, with the constraint that the distance between a and c is in the interval $[1, 2]$. This behavior requires both non-deterministic delays and some form of synchronization to ensure that b occurs before c, which does not appear to be implementable in NDE.

We finally conjecture that the NDE model of Figure 5.6(e) cannot be implemented in DETA. The loop in the model can produce unbounded bursts of events in a finite window. A variable delay needs to be introduced for every such event. In DETA, variable delays can only be implemented using timed automata, which only have a finite number of clocks.

Chapter 6

Conclusions and Future Work

We first provided a formalization of the PTIDES programming framework. We proved that for a rich subset of PTIDES programs, the earliest-deadline-first scheduling policy is optimal with regards to feasibility. We then showed that the schedulability problem can be reduced to a finite-state reachability problem, and described the reduction using timed automata. The finite-state reduction entailed the following steps: perform a schedulability preserving abstraction of the states of a PTIDES program, and then show that the space of the states that are visited by executions that do not miss deadlines is bounded.

Intuitively the above was based on two observations. First, if the number of events in a PTIDES state exceeds a limit, then any execution from that state can be shown to lead to deadline violation. Since every event is associated with an amount of execution demand from the processor, overly increasing the number of events will result in an amount of execution demand that will be impossible to be met by the processor. The second observation is that the absolute value of real-time or of a timestamp of an event are not both necessary for the execution of a PTIDES program. Indeed, the difference between real-time and the timestamp of an event is sufficient to correctly execute the program. That difference is bounded from above in all schedulable states; that is, if real-time grows too much relative to the timestamp of an event, then the event is bound to miss its deadline, and it is bounded from below in all executions because actors are guaranteed to process events in timestamp order.

We used the same intuition of using relative instead of absolute timestamps to show the boundedness of the state space of a discrete-event program. We further explored this discrete-event system by enriching actor specifications and compared its expressiveness to timed automata.

There are some questions that still remain open about the schedulability of PTIDES programs. First and foremost, as described in section 3.2, we constrained our actors to be output-homogeneous and constant delay. The combination of the two constrains allowed us to statically determine event deadlines and was necessary in proving that the earliest-deadline-first algorithm is optimal. They do not appear to be necessary to either bound the number of events during the execution of a PTIDES execution, or to bound the difference between real-time and timestamp of events. However, as we hint in section 6.1, lifting the

non output-homogeneous constraint allows for events that do not have a statically defined deadline and makes it difficult to come up with an optimal scheduling policy, at least when the input model is unconstrained.

The second question relates to the complexity of the schedulability problem. The reduction to reachability in timed automata provides a decision procedure for schedulability; however, further investigation could reveal more practical and efficient schedulability tests in the case of constrained input models, e.g. sporadic and periodic. Section 6.2 describes the initial steps for a reduction of the schedulability of PTIDES programs with no loops, constant delays, and sporadic inputs to generalized multiframe tasks, a sporadic task system model developed by Baruah et al. [2]. The techniques described in chapter 5 should allow us to deal with periodic loops as well.

6.1 Scheduling with register actors

We begin by explaining the semantics of the register actor. The register actor models a memory cell: a memory location that can be overwritten and read. We will refer to the contents of that memory location as the register's *value*. The register actor has two input channels, one called "input" and the other called "trigger", and one output channel. The register's value is set by events in the input channel. When an event in the input channel is processed, the register's value is overwritten and set equal to the value of that event. Processing an event from the trigger channel corresponds to reading the value of the register: when an event in the trigger channel is processed, an event in the output channel is produced whose value is equal to the register's value.

Assume an event e with timestamp τ is at the input channel of a register. Because the processing of e will not result in an event at the output channel that will eventually reach an actuator, e is not by itself associated with a deadline. However, in the presence of an event e' with timestamp $\tau' > \tau$ at the trigger channel, because e has to be processed before e' is, e inherits the deadline of e'.

Since events that are targeted to the input channel of a register are not a priori associated with a deadline, the question that arises is how should their processing be scheduled. We call events that are targeted towards the input channel of a register, and have not inherited a deadline from a trigger channel event, *soft events*. We call all other events *hard events*. We know that if only hard events are present in a network, the earliest-deadline-first scheduling policy is optimal.

One naive scheduling policy that takes into account soft events is the following: use earliest-deadline-first among hard events and schedule soft events only when no hard events are available.

We show an example below, in Figure 6.1, where the naive scheduling policy results in a deadline miss, even though a schedule that does not miss the deadline exists.

At time 0, requests arrive at sensors S_1 and S_3. Two events e_1 and e_3 are generated with timestamp 0. The deadline of e_3 is 5, since the delay on the path from sensor S_3 to actuator

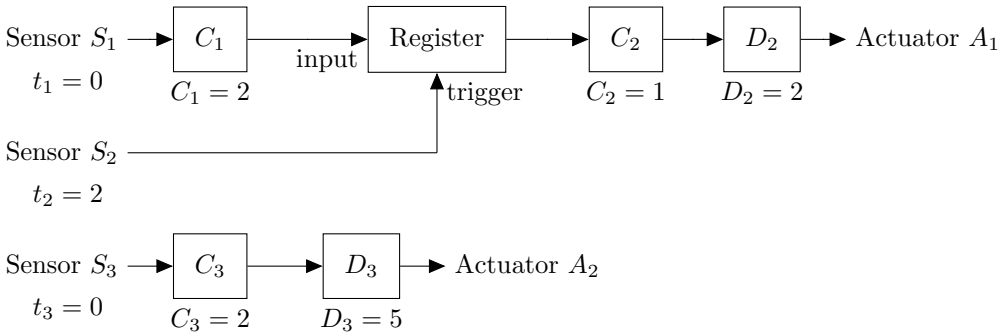


Figure 6.1: Register example.

A_2 is 5. Note that e_1 does not have a deadline because no events have arrived at the trigger channel of the register actor. As shown in the scheduling diagram below, in Figure 6.2, the naive scheduling policy schedules actor C_2 for execution from time 0 to 2. At time 2, a request arrives at sensor S_2 and an event e_2 with timestamp 2 is created. Its deadline is determined by the path S_2, register, C_2, D_2, A_1 and is equal to 4. Because the timestamp of e_2 is bigger than the timestamp of e_1, the latter inherits the deadline of e_2 and becomes a hard event. In order to guarantee that the register actor processes events in timestamp order, e_1 has to be processed by the register actor before e_2 and hence C_1 executes next from time 2 to 4. Assuming that the register reacts to both input and trigger instantaneously, C_2 starts processing at time 4 and therefore the deadline is missed.

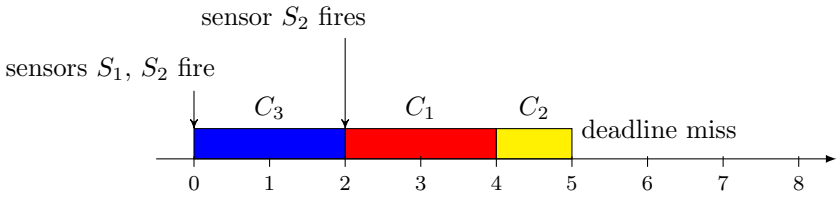


Figure 6.2: Naive policy schedule.

The schedule that does not miss the deadline is shown in Figure 6.3 below. Actor C_1 processes event e_1 from 0 to 2. At time 2, both e_2 and e_3 are available but because the deadline of e_2 is smaller (4), actor C_2 executes from 2 to 3, and then C_3 executes from 3 to 5. The difference from the naive scheduling policy is that at time 0, even though a hard event was available, the scheduler chooses to process a soft event. This schedule can be justified with several rationales.

At time 0, the deadline of the hard event e_3 is 5 and its computation demand is 2, therefore its execution can be delayed at least until time 3, making the choice to process the soft event e_1 not problematic.

At time 0, we consider all possible events that could arrive in the future and decide whether we should process e_1 or e_3. One of the possibilities will be that e_2 arrives at 2 and

hence the decision to process e_1.

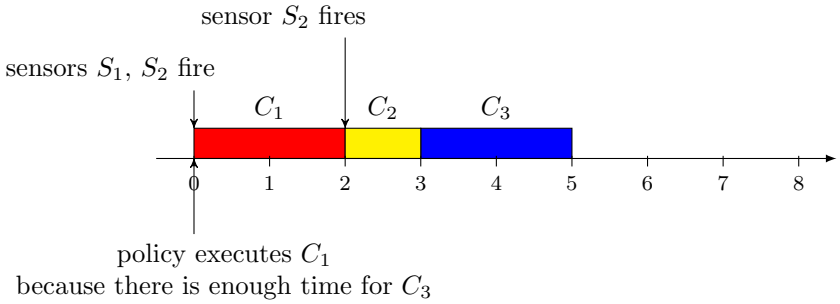


Figure 6.3: Clairvoyant policy schedule.

The problems that arise from the discussion above are two:

- Does there exist a scheduling policy that optimally schedules PTIDES programs in the presence of both soft and hard events?
- Does the answer to the previous question change if we add assumptions about the input model (e.g. periodic, sporadic, etc.)?
- Is the schedulability decision under that scheduling policy decidable?

6.2 Relation to Multiframe Tasks

Generalized multiframe tasks is a model for describing sporadic task systems. It was introduced by Baruah et al. [2] together with a pseudo-polynomial algorithm for deciding the feasibility of systems described using it.

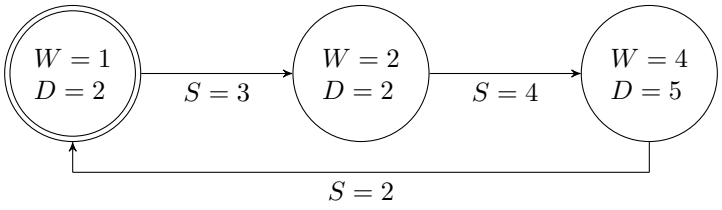


Figure 6.4: Generalized multiframe task example.

An example of generalized multiframe task is given in Figure 6.4. Each node in the graph represents a *frame* which is associated with an execution time W and a relative deadline D. Every node has one outgoing edge which is labeled with a separation time S. Informally, an execution of a multiframe task is a walk on the graph such that a job is released every time a node or frame is visited. If a frame is visited at time t then a job is released which requires

computation time equal to W and has an absolute deadline equal to $t + D$. If two frames F_1 and F_2 are separated by an edge with separation time S, and F_1 is visited at time t, then the execution transitions from F_1 to F_2 no earlier than time $t + S$.

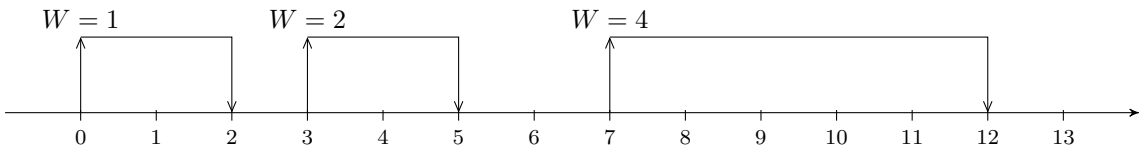


Figure 6.5: Example frame arrival sequence for task in Figure 6.4

Figure 6.5 shows an example execution of the multiframe task shown in Figure 6.4. The execution starts at time 0, when the task is in its initial frame. A job with computation requirement $W = 1$ is released and its has absolute deadline equal to 2. After 3 time units, the task moves to the second frame and releases the next job. This time gap could have any value greater than or equal to 3.

When a PTIDES program consists solely of actors with a single input and a single output channel, it can be broken into parallel actor chains. As we saw in Chapter 2, when the sensor inputs are sporadic, the feasibility problem of such PTIDES programs can be reduced to the feasibility of a set of sporadic tasks. The reduction fails when the program contains actors that are connected to multiple input channels, multiple output channels, or both. Figure 6.6(a) shows an example of a PTIDES program where actor C_1 has two outputs. If sensor S1 takes a measurement at time t, an event e is generated at the input channel of D_1 with timestamp t. If $D_2 < D_3$, the absolute deadline of e is $t + D_1 + D_2$. The processing of e by actor C_1 will result in two events, both with timestamp $t + D_1$, one at the input of D_2 with deadline $t + D_1 + D_2$ and the other at the input of D_3 with deadline $t + D_1 + D_3$. From the perspective of the scheduler, the processing of e requires the processing of three "jobs", one with execution demand C_1 and deadline $t + D_1 + D_2$, one with execution demand C_2 and deadline $t + D_1 + D_2$, and one with execution demand C_3 and deadline $t + D_1 + D_3$. From the above description we can see that the multiframe task at Figure 6.6(b) produces the same demand for the processor as the program at Figure 6.6(a).

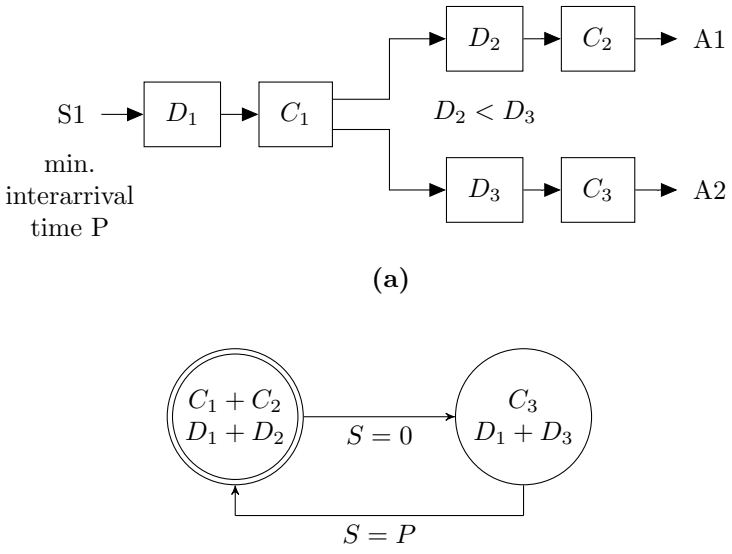


Figure 6.6: PTIDES program with split paths and equivalent multiframe task.

If a PTIDES program contains an actor with multiple inputs, e.g., the PTIDES program in Figure 6.7(a), the equivalent multiframe task system includes multiple multiframe tasks and non-zero separation times. In the case of Figure 6.7(a), assuming $D_1 > D_2$, the equivalent multiframe task system, shown in Figure 6.7(b), has two multiframe tasks. The one on the left simulates the execution of events that arrive at sensor S1, and the one on the right simulates the execution of events that arrive at S2. The separation time between the first and second frame of the left multitask is equal to $D_1 - D_2$. If an event arrives at S1 at time t, a job is released at t with computation demand C_1 and absolute deadline $t + D_1 + D_3$. Then at time $t + D_1 - D_2$ another job is released with computation demand C_3 and the same absolute deadline. We call that job J_3. The previous statement is not completely accurate because J_3 will be released at time $t + D_1 - D_2$ only if the execution of C_1 has been completed by the time. This might not be possible if the execution demand of C_1 is larger than $D_1 - D_2$ or if its execution is preempted by some job with smaller deadline. However, in all cases, the reason why J_3 might not be released at $t + D_1 - D_2$, is some other job with absolute deadline smaller than or equal to the deadline of J_3. Therefore, assuming an earliest-deadline-first execution policy, if we were to release J_3 at $t + D_1 - D_2$ we would get an execution that is equivalent to the canonical one from the perspective of schedulability.

The second reason for concern about the multiframe task system in Figure 6.7(b), is on how accurately it simulates the case where two events arrive at S1 and S2 with time difference $D_1 - D_2$, for example, the case where an event arrives at S1 at time t and an event arrives at S2 at time $t + D_1 - D_2$. The PTIDES program of Figure 6.7(a), will process both events with timestamp $t + D_1$ when they arrive at the inputs of D_3, whereas the multiframe task system will generate and process two jobs, one corresponding to each event. Generally,

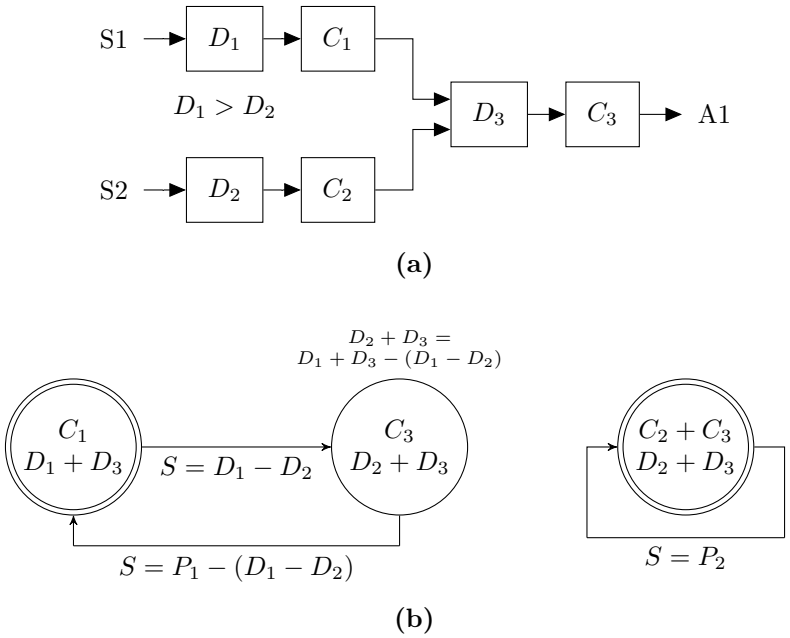


Figure 6.7: PTIDES program with merging paths and equivalent multiframe task.

the equivalent multiframe task system cannot accurately handle simultaneous events at the inputs of an actor. However, on the one hand, the schedulability of the resulting multiframe task system trivially gives us a sufficient condition for the schedulability of the PTIDES program. On the other hand, less trivially but still intuitively, if the model of the inputs of S1 and S2 is sporadic, and there exists an execution of the PTIDES program that both misses its deadline and processes simultaneous events, then we can perturb the input pattern by delaying the arrival of the input by a "small" amount and still get a schedule that misses a deadline and does not process simultaneous events. With that intuition, the schedulability of the equivalent multiframe task system is also a necessary condition for the schedulability of the initial PTIDES program.

Finally, if we want to strictly stick to the generalized multiframe task model, as described in [2], then the period P_1 is also required to be greater than $D_1 - D_2$. However, it seems plausible to extend the analysis presented in that paper to cover the case where a multiframe task has a period or a minimum interarrival time that is independent of the separation times between the different frames, as shown in Figure 6.8.

Lastly, the analysis we presented here, talks about PTIDES programs that do not have any loops. However, as discussed in chapter 5, if a sporadic input source feeds into a loop, then there exists a sporadic input signal that results in a Zeno-signal in some channel of the loop. Therefore, loops in a PTIDES program will probably make sense with a more constrained input model than just a sporadic one. In that case it should be possible to analyze such PTIDES programs using the techniques presented in the same chapter and reduce them to a model whose execution pattern in conjunction with a set of multiframe

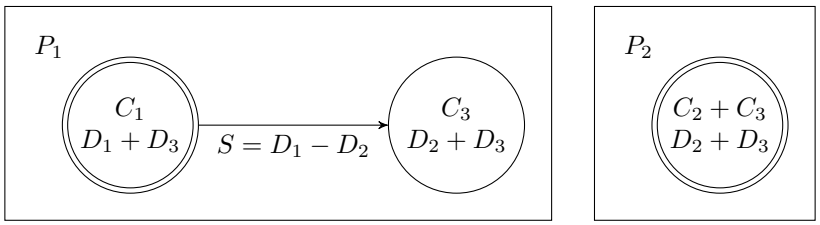


Figure 6.8: PTIDES program with merging paths and equivalent multiframe task.

tasks is analyzable.

6.3 DE Verification

The verification problems for NDE and DETA remain open. We are currently exploring ideas to constrain the model to regain, or statically check for, boundedness, which would enable transformation of bounded DETA models to timed automata. We are also currently working on extracting affine expressions directly from DDE models (without the use of lassos) and then extending this technique to NDE, which would allow verification of signal queries on NDE despite unboundedness. TA are another possible symbolic representation of signals, natural in DETA models. It is easy to see how to transform TA signal representations by fork, join, constant- and variable-delay actors, but not how to compute fixpoints which seems needed for general cyclic networks.

Another direction for future work is investigating model-checking of general temporal logics against DE models, or coming up with new logics especially designed for DE models.

Another direction is to enrich expressiveness of DDE and NDE models, for instance, by adding control-expressive actors such as synchronizers, which from the comparison between NDE and TA appear important. Adding values to events is another possibility for extending DE models in general, including DETA models.

Bibliography

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[2] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Systems*, 17:5–22, 1999.

[3] Sanjoy K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24:93–128, 2003. 10.1023/A:1021711220939.

[4] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190, 1990.

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

[6] Gregory Bollella and James Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[7] Sébastien Bornot, Gregor Gossler, and Joseph Sifakis. On the construction of live timed systems. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 109–126. Springer Berlin / Heidelberg, 2000.

[8] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In Willem-Paul Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 103–129. Springer Berlin Heidelberg, 1998.

[9] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[10] Patricia Derler, Eidson John, Goose Stuart, Edward A. Lee, and Michael Zimmer. Deterministic execution of ptides programs. Technical Report UCB/EECS-2013-65, EECS Department, University of California, Berkeley, May 2013.

[11] Patricia Derler, Edward A. Lee, and Slobodan Matic. Simulation and implementation of the ptides programming model. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications (DS-RT '08)*, pages 330–333, October 2008.

[12] ML Dertouzos. Control robotics: the procedural control of physical processes," information processing 74, 1974.

[13] John Eidson, Edward A. Lee, Slobodan Matic, Sanjit A. Seshia, and Jia Zou. Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE (special issue on CPS)*, 100(1):45 – 59, January 2012.

[14] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149 – 1172, 2007.

[15] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 125–149. Springer Berlin / Heidelberg, 2002.

[16] Arkadeb Ghosal, ThomasA. Henzinger, ChristophM. Kirsch, and MarcoA.A. Sanvido. Event-driven programming with logical execution times. In Rajeev Alur and GeorgeJ. Pappas, editors, *Hybrid Systems: Computation and Control*, volume 2993 of *Lecture Notes in Computer Science*, pages 357–371. Springer Berlin Heidelberg, 2004.

[17] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84 – 99, jan 2003.

[18] G. Kahn. The semantics of a simple language for parallel processing. *Information Processing*, 74:471–475, 1974.

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

[20] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 114–123, New York, NY, USA, 2007. ACM.

[21] EdwardA. Lee and Haiyang Zheng. Operational semantics of hybrid systems. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 25–53. Springer Berlin Heidelberg, 2005.

[22] Insup Lee, Susan B Davidson, and Victor Fay-Wolfe. Motivating time as a first class entity. Technical report, 1987.

[23] Allen Leung, Krishna V Palem, and Amir Pnueli. Timec: A time constraint language for ilp processor compilation. *Constraints*, 7(2):75–115, 2002.

[24] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237 – 250, 1982.

[25] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[26] Eleftherios Matsikoudis, Christos Stergiou, and Edward A. Lee. On the schedulability of real-time discrete-event systems. In *13th International Conference on Embedded Software (EMSOFT)*, September 2013. Montreal, Canada.

[27] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Cambridge, MA, USA, 1983.

[28] AK Mok. Annotating ada for real-time program synthesis. *Proc. Computer Assurance*, 1987.

[29] C. Norström, A. Wall, and Wang Yi. Timed automata as task models for event-driven systems. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 182–189, 1999.

[30] Christos Stergiou, Stavros Tripakis, Eleftherios Matsikoudis, and Edward A. Lee. On the verification of timed discrete-event models. In *11th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, August 2013. Buenos Aires, Argentina.

[31] M. Stigge, P. Ekberg, Nan Guan, and Wang Yi. The digraph real-time task model. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 71 –80, april 2011.

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

[33] Yang Zhao, Jie Liu, and E.A. Lee. A programming model for time-synchronized distributed real-time systems. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07. 13th IEEE*, pages 259–268, 2007.

[34] Jia Zou. *From Ptides to PtidyOS, Designing Distributed Real-Time Embedded Systems*. PhD thesis, EECS Department, University of California, Berkeley, May 2011.

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