Time at Your Service

Schedulability Analysis of Real-Time and Distributed Services

Mohammad Mahdi Jaghoori

Time at Your Service

Schedulability Analysis of Real-Time and Distributed Services

Proefschrift

ter verkrijging van

de graad van doctor aan de Universiteit Leiden

op gezag van de Rector Magnificus prof. dr. Paul van der Heijden

volgens besluit van het College voor Promoties

te verdedigen op maandag 20 december 2010

klokke 13.45 uur

door

Mohammad Mahdi Jaghoori

geboren te Mashhad, Iran
in 1982

The work reported in this thesis has been carried out at the Center for Mathematics and Computer Science (CWI) in Amsterdam and Leiden Institute of Advanced Computer Science at Leiden University, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). This research was supported by the European IST-33826 STREP project CREDO on Modeling and Analysis of Evolutionary Structures for Distributed Services.

The cover illustration depicts a real-time solution to the dining philosophers problem. This solution is referred to as "A Late Philosopher" in Section 5.3.
Cover design by Samira Sedghi.
ISBN: 978-90-6464-445-0

Acknowledgments

Thank you all. This, at the highest level of abstraction, shows my thanks and gratitude to all who made it possible for me to accomplish my PhD. I will try to refine it a bit here, but am sure this refinement will miss some traces. My apologies in advance. The English acknowledgment is complementary to the Farsi one on the other side of the book.

I would like to thank my PhD supervisor Frank de Boer. Throughout these years, he was a unique source of inspirations for me. Since my first visit to CWI for the PhD interview, he was more of a friend to me than a supervisor. Next to Frank, I spent hours and hours pushing the frontiers of science with Tom Chothia, my co-promoter. As it is obvious from my publications record, I would not be writing this thesis if it were not for them.

I spent most of my days at CWI with these people, in addition to Frank and Tom, alphabetically listed: Lăcrămioara Aştefănoaei, Marcello Bonsangue, Dave Clarke, David Costa, Susanne van Dam, Fernando Filho, Bo Gao, Stijn de Gouw, Immo Grabe, Helle Hansen, Michiel Helvensteijn, Stephanie Kemper, Natallia Kokash, Christian Krause, Clemens Kupke, Rodrigo Laiola, Delphine Longuet, Huiye Ma, Ziyan Maraikar, Jacopo Mauro, Young-Joo Moon, José Proença, Jan Rutten, Alexandra Silva, Sun Meng, Yanjing Wang, Joost Winter. Some of them have left already, some of them have joined only recently. We sometimes had scientific discussions, but I am to thank them mainly because they made my life easy, so far away from my family and my country, and basically made my stay in Amsterdam possible, by being good friends before anything else.

What I liked most during these years was travelling. I was involved in a European project called Credo. We travelled a lot, mainly for project meetings, all around the world during which I learned scientific collaboration, presentation, discussion, and sightseeing. During all these meetings, I also got to know many other nice people related to the Credo project and, in one way or another, they all played a role on my way towards this thesis. As the smallest sign of appreciation, I will name them, alphabetically: Bernhard Aichernig, Christel Baier, Tobias Blechmann, Andreas Griesmayer, Einar Broch Johnsen, Joachim Klein, Sascha Klüppelholz, Marcel Kyas, Wolfgang Leister, Willem-Paul de Roever, Rudolf Schlatte, Andries Stam,

Martin Steffen, Simon Tschirner, Xuedong Liang, and Wang Yi.

The fourth year of my PhD was funded by Leiden University. I should thank everyone in Leiden who helped me in different ways. Joost Kok supported me, especially in the last few weeks leading to my defense. I also thank him for nominating my thesis for the prestigious C. J. Kok prize and his efforts to schedule my defense in 2010.

I wish all those who helped and supported me scientifically, and more importantly non-scientifically, the best and success and prosperity in life and afterwards.

Mahdi Jaghouri
November 2010

Contents

Chapter 1

Introduction

These days computers are not just meant for large business corporations or just for controlling the launch of shuttles and rockets. Since a few years ago, they have been sitting on the desks of almost every house, and they are all connected via the Internet. Today software is also along the same trend changing from a desktop product to an Internet service. Many people check the weather on the Internet instead of looking out the window. You can plan your travels, from flight and hotel reservations to the sights you want to see, right from your desk at home.

On the other hand, nowadays, computers are controlling many of the devices we use every day. A normal daily schedule is like this. Your clock wakes you up in the morning; then you have coffee from a coffee machine, take your breakfast from the refrigerator, and after breakfast have the dishes washed by the dishwasher; you take the bus to work, make a few phone calls during the day, have your lunch warmed up in the microwave, and in the afternoon you struggle with the air-conditioning to adjust your room temperature; at the end of the day you print some papers to read in the evening after your favorite show on TV has finished. You are most probably interacting with a computer when you use a device like the ones just mentioned.

Furthermore, computer processors have stopped getting faster due to physical restrictions. Higher speeds can only be achieved by increasing the number of processors and the level of parallelism. This implies that future computer systems, including real-time and embedded software, need to be distributed.

Working in real time, all these distributed software and services need to be timely and respond to the requests in time. An important aspect of real time software is how it schedules its tasks to be executed. In this thesis, we introduce a software paradigm based on object orientation in which real-time objects are enabled to specify their own scheduling strategy. The nature of this paradigm allows for distributed deployment of these objects.

Concurrent Objects with Schedulers The term concurrency is used in computer science to refer to a situation that multiple process are running at the same time. Different formalizations are proposed to model concurrent systems. Process calculi (like CCS [37] and CSP [21]), Petri-nets [40], automata models (like IO automata [33, 32]) and Actors [20, 1] are examples of these concurrency models. Different concurrency models are usually compared regarding their concurrency units, possible ways of communication between them (e.g., shared memory or message passing, synchronous or asynchronous), their syntax and their semantics, etc.

The concurrency model we work with assigns a processor to each object: objects are thus concurrent and are the units of concurrency. Concurrent objects are an extension of the actor model [20, 1] with sophisticated means for synchronization. We use an operational interpretation of the actor model in which objects have sequential pieces of code specifying their behavior. A concurrent object may be used to model a software service residing on an independent computer. As in the actor model, concurrent objects communicate only by sending asynchronous messages and have queues for receiving them. This fits very well the common approach in deploying distributed systems where different nodes are completely asynchronous and communicate only by message passing.

In a real-time setting, a deadline is assigned to each message specifying the time before which the intended job should be accomplished. An object has a method for each message it can handle. Receiving a message schedules the corresponding method, i.e., a new process is created to execute the method which is added to the queue. The scheduling policy of the object determines in which order the processes in the queue are executed. We allow each concurrent object to define its own scheduling policy (rather than, for instance, assuming "First Come First Served (FCFS)" by default) with the condition that inserting a new message in the queue cannot preempt the currently running method.

Actors and concurrent objects are suitable for modular modeling, because of the asynchronous nature of communication and the encapsulation of computation (i.e., having no shared variables). This means that objects can be developed independently, and later put in the context of bigger systems. Nevertheless, for an object to produce a specific service, messages should be sent to it in a correct sequence.

The high-level view of the object behavior is given in its behavioral interface. A behavioral interface given in a timed automaton, specifies how the object expects input messages and how it would reply accordingly. Schedulability analysis for a real time system consists of checking whether all tasks can be finished within their deadlines. We propose a modular method for schedulability analysis. For individual object analysis, the behavioral interface is used a driver, because it models the acceptable input behavior of the object. The real usage of the object would need to be compatible with this expected usage.

One of the advantages of this technique is the possibility of analyzing real-time systems with their application-specific scheduling strategies specified at a high-level. This is in contrast to traditional approaches where scheduling is left to the underlying operating system. Actors with scheduling policies can be used for modeling, analyzing and synthesizing systems with context-specific scheduling strategies.

1.1 The *Credo* Context

This thesis has been carried out in the context of the *Credo* project. Therefore, this work can be applied in the context of the *Credo* methodology [17]. Current software development methodologies follow a component-based approach in modeling distributed systems. The *Credo* methodology aims at their shortcoming concerning an integrated formalism to model highly reconfigurable distributed systems at different phases of design, i.e., systems that can be reconfigured in terms of a change to the network structure or an update to the components. Moreover, the high complexity of such systems requires tool-supported analysis techniques.

The core of the *Credo* tool suite consists of two different *executable* modeling languages: Reo [4] is an executable *dataflow* language for high-level description of the dynamic reconfigurable *network* of connections between the components; Creol [27] is an *object-oriented* modeling language, used to provide an abstract but executable model of the implementation of the individual components. Figure 1.1 illustrates the relation between these modeling languages and their relation to existing programming languages. At shown here, schedulability analysis is performed at the level of object-oriented modeling of components.

This thesis focuses on the part of the *Credo* tool suite that offers an automated technique for *schedulability analysis* of individual objects [25, 24]. We use the *timed automata* of UPPAAL to model objects and their behavioral interfaces. Given a specification of a scheduling policy (e.g., shortest deadline first) for an object, we use UPPAAL to analyze the object with respect to its behavioral interface in order to ensure that tasks are accomplished within their specified deadlines (see Figure 1.2).

1.2 Thesis Overview and Contributions

We bridge the gap between object orientation and automata theory for schedulability analysis. Object orientation is a widely accepted paradigm for modeling and programming. On the other hand, there is a rich theory based on timed automata for model checking and schedulability analysis of systems.

Creol is a concurrent object modeling language for specifying abstract behaviors of distributed systems and protocols. We extend Creol with the possibility of specifying real-time information in **Chapter 2**. Our extension of Creol is in a descriptive approach: real-time execution information is added as delays between standard Creol statements and deadlines are added to (asynchronous) method calls as schedulability requirements [12, 15]. Additionally, each concurrent object can specify its context-specific scheduling policy.

For schedulability analysis, we take advantage of recent advances in automata theory that allow analysis of non-uniformly recurring tasks. At the first step, we show in **Chapter 3** how to use timed automata to model actors [14, 23, 24] as the pure asynchronous setting of concurrent objects. We develop a modular technique for schedulability analysis of the actor models: **Chapter 4** shows how the problem

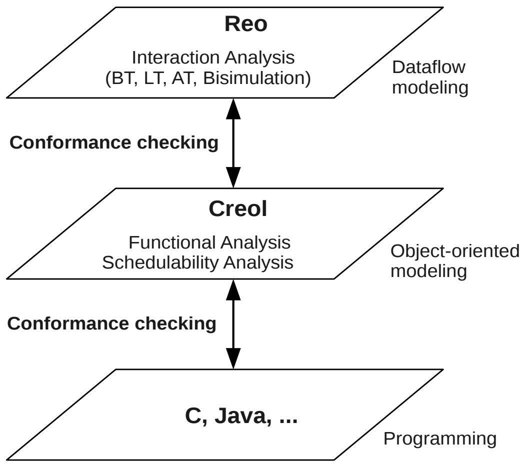


Figure 1.1: Overview of modeling levels and analysis in *Credo*

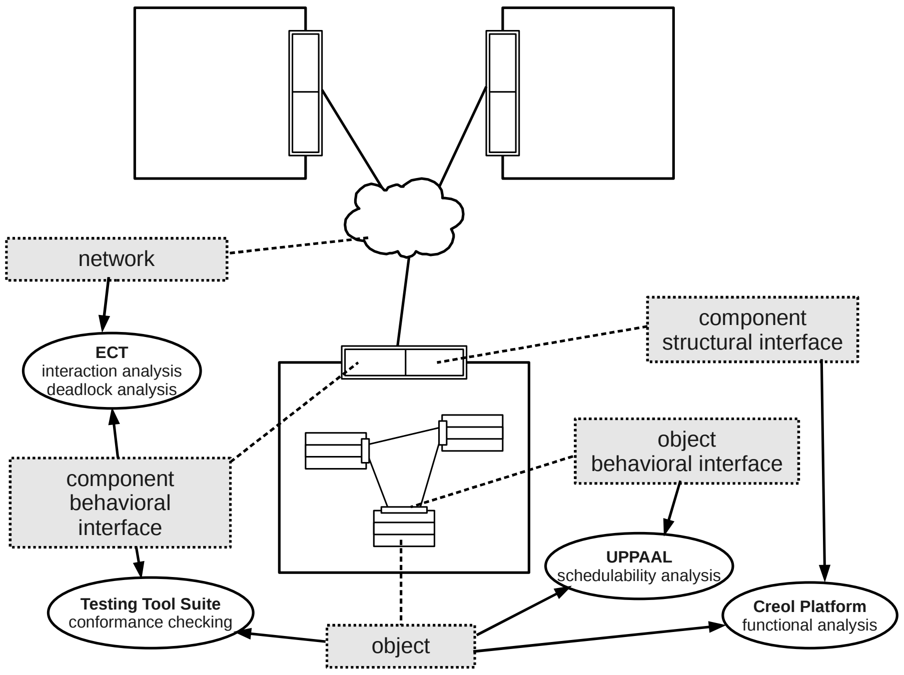


Figure 1.2: End user perspective of the *Credo* Tools

of schedulability can be formulated as the reachability problem of timed automata; in this chapter, we discuss how to analyze a system of actors together or each actor in isolation [24]. When a system is too big to be analyzed, one can argue about its schedulability based on the analysis of individual actors and their compatibility, as explained in **Chapter 5** [25]. We show how these techniques can be automated with the help of an existing model checker, namely UPPAAL.

Finally, we explain in **Chapter 6** how we can apply our theory to the analysis of real-time Creol models. To this end, we develop a timed-automata semantics for Creol based on the actor-based automata framework [22] of Chapter 3. Alternatively, one can develop abstract automata models and show that a concrete Creol code conforms to the verified automata models [15].

A list of publications that are cited above and are related to the theories developed in this thesis are collected below:

[12] Frank de Boer, Tom Chothia, and Mohammad Mahdi Jaghoori. Modular schedulability analysis of concurrent objects in Creol. In *Proc. Fundamentals of Software Engineering (FSEN'09)*, volume 5961, pages 212–227, 2009.

[14] Frank S. de Boer, Immo Grabe, Mohammad Mahdi Jaghoori, Andries Stam, and Wang Yi. Modeling and analysis of thread-pools in an industrial communication platform. In *Proc. 11th International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *LNCS*, pages 367–386. Springer, 2009

[15] Frank S. de Boer, Mohammad Mahdi Jaghoori, and Einar B. Johnsen. Dating concurrent objects: Real-time modeling and schedulability analysis. In *Proc. 21st International Conference on Concurrency Theory (CONCUR'10)*, volume 6269 of *LNCS*, pages 1–18, 2010

[22] Mohammad Mahdi Jaghoori and Tom Chothia. Timed automata semantics for analyzing Creol. In *Proc. Foundations of Coordination Languages and Software Architectures (FOCLASA'10)*, EPTCS 30, pages 108–122, 2010

[23] Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia, and Marjan Sirjani. Task scheduling in Rebeca. In *Proc. Nordic Workshop on Programming Theory (NWPT'07)*, 2007. extended abstract

[24] Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia, and Marjan Sirjani. Schedulability of asynchronous real-time concurrent objects. *J. Logic and Alg. Prog.*, 78(5):402 – 416, 2009

[25] Mohammad Mahdi Jaghoori, Delphine Longuet, Frank S. de Boer, and Tom Chothia. Schedulability and compatibility of real time asynchronous objects. In *Proc. RTSS'08*, pages 70–79. IEEE CS, 2008

1.3 Case Studies

Modeling and analysis is described with the help of numerous running examples. Each chapter will have as running example the model of a Mutual Exclusion handler, called a `MutEx`. We will use other examples and case studies to further illustrate the techniques. We will end each chapter with an industrial case study, thread-pools in the ASK system explained below.

MutEx A `MutEx` object provides mutual exclusive access to a given resource that is shared between two objects. A `MutEx` may communicate with the two objects on its left and right which try to enter a critical section. In this example, each user object sends its request to the `MutEx` and gets a permit from the `MutEx` when it is safe to use the resource. In a real-time model, we want to put a time bound on the waiting time for each user after it sends a request until it gets a permit.

Coordinator A coordinator provides multi-way barrier synchronization; we will model and analyze a three-way coodinator. In contrast to `MutEx`, which provides mutual exlusion, a coordinator requires all parties to be ready before they can continue.

Peer-to-Peer Peer-to-peer systems are now a commonly used way of sharing data. Contrasted to a client-server architecture, these systems are called peer-to-peer because all nodes can act both as a server and a client; in other words, they are all peers. In such systems, there are a number of nodes that share some data each. Each data item has a key, for instance, for a song the key is the name of the song. Each peer node can search for new data using their keys, e.g., by providing the name of the song it looks for. If another node has a song with the given name, then it can provide the data, namely, the song itself, to the requester node.

We model and analyze a file-sharing system with hybrid peer-to-peer architecture (like in Napster), where a central server (called the *broker*) keeps track of the keys for the data in every node. Each node, upon creation, registers its data with the broker. Later it queries the broker for some new data, and the broker connects it to the node which has the data.

ASK System: An Industrial Case Study [14]

We use an object-oriented paradigm to model different thread pools in the context of the ASK system, an industrial communication platform. Thread pools are often used as a pattern to increase the throughput and responsiveness of software systems. Implementations of thread pools may differ considerably from each other, which urges the need to analyze these differences in a formal manner. We will apply automata theory to analyze their schedulability

ASK is an industrial software system for connecting people to each other. The system uses intelligent matching functionality in order to find effective connections between requesters and responders in a community. ASK has been developed by Almende [2], a Dutch research company focusing on the application of self-organization techniques in human organizations and agent-oriented software systems. ASK provides mechanisms for matching users requiring information or services with potential suppliers. Based on information about earlier established contacts and feedback of users, the system learns to bring people into contact with each other in the most effective way. Typical applications for ASK are workforce planning, customer service, knowledge sharing, social care and emergency response. Customers of ASK include the European mail distribution company TNT Post, the cooperative financial services provider Rabobank and the world's largest pharmaceutical company Pfizer. The amount of people using a single ASK configuration varies from several hundreds to several thousands.

The software of ASK can be technically divided into three parts: the *web front-end*, the *database* and the *contact engine*. The *web front-end* acts as a configuration dashboard, via which typical domain data like users, groups, phone numbers, mail addresses, interactive voice response menus, services and scheduled jobs can be created, edited and deleted. This data is stored in a *database*, one for each configuration of ASK. The feedback of users and the knowledge derived from earlier established contacts are also stored in this database. Finally, the *contact engine* consists of a number of components, which handle inbound and outbound communication with the system and provide the intelligent matching and scheduling functionality.

At the heart of each component in the ASK system is a thread-pool. Depending on the sysetm load, there will be many tasks to be processed in each component. The overall performance of the system depends on the scheduling of the tasks by the thread-pools.

Chapter 2

Concurrent Objects in Real-Time Creol

Concurrent objects are an evolution of the actor model with more sophisticated synchronization patterns and type systems. As in actors, concurrent objects have dedicated processors and communicate only by asynchronous message passing. Due to their intrinsic asynchrony, actor-based and concurrent object languages can be used for modeling classical concurrent and distributed applications, as well as, modern web services.

A concurrent object has an unbounded queue for storing its incoming messages. At the initial state, a number of objects are created statically, and an initialization message is implicitly put in their queues. The system runs as objects send messages to each other and the methods corresponding to the received messages are executed. The order of executing the messages in the queue is called the *scheduling strategy*. Actors usually process their incoming messages in an FCFS manner, i.e., first come first served; some concurrent object languages leave the scheduling strategy unspecified, i.e., they consider a nondeterministic execution of the messages. In real-time settings, the correct behavior of a system depends on the correct choice of the scheduling strategy; this requires the ability to specify different scheduling strategies (besides FCFS) for individual actors.

We first present Creol, a language for specifying concurrent object models, as introduced by Johnsen et al. [27]. Creol is strongly typed and allows for type safe dynamic behavior, e.g., dynamic class upgrades [28], dynamic service recovery [8], etc. In this thesis, we work on a complementary feature of Creol. We extend Creol with real-time information and explicit specification of scheduling policies. The semantics of Real-Time Creol will be given using rewrite logic as an extension of the original semantics of Creol in [27].

2.1 Background: Modeling in Creol

The basic concept of the Creol modeling language [27] is to provide a formal object-oriented solution for modeling distributed systems. Like the standard actor model, concurrent objects in Creol have dedicated processors. Since there is no shared data and there is a single thread of execution in each object, objects act as *monitors* for usage of their locally encapsulated data. Allocation of an object's processor to the processes in Creol is based on cooperative scheduling, i.e., once a process is executing a method in an object, it is not preempted unless the method voluntarily releases the processor.

Concurrent objects communicate by asynchronous method calls. However, the caller can choose to wait for an answer, thus simulating synchronous method calls. Creol objects are typed by interfaces, nevertheless, classes can implement as many interfaces as necessary. The notion of co-interfaces can be used to restrict who can call the methods provided in interfaces. As a result, the callee can communicate with the caller in a type-safe manner.

Creol is backed by its formal operational semantics and its strong typing allows for dynamic class upgrades [45]. This thesis is not concerned about the type system and dynamic features of Creol. In this chapter, we focus on adding scheduling policies to Creol and real-time information with the purpose of schedulability analysis.

In Creol, object references are typed only by interfaces, allowing for a strongly typed language with support for features such as multiple inheritance and type-safe dynamic class upgrades [45]. We augment an interface with a notion of high-level behavioral specification given in a timed automaton; this is given by the modeler as the first step in modeling an object. A class can implement multiple interfaces. In this case, the timed automata for behavioral interfaces should be interleaved; intuitively, allowing more behavior than each automaton separately. The justification is that a class with multiple interfaces can participate in multiple protocols independently. Creol supports multiple inheritance for both interfaces and classes. Interfaces form a subtype hierarchy, which is distinct from inheritance at the level of classes used merely for code reuse [28]. In a subtype hierarchy, the timed automata for the inherited behavioral interfaces should synchronize on similar actions; intuitively, allowing less behavior in the subtype. The justification is that a subtype should be a refinement of its supertypes [39].

2.1.1 Defining a Class in Creol

A simplified syntax for Creol, used in the subsequent chapters, is given in Figure 2.1. Classes and interfaces can have parameters. Class instances can communicate by objects given as class parameters, called the *known objects*. We can thus define the static topology of the system. The class behavior is defined in its methods, where a method is a sequence of statements separated by semicolon. For expressions, we assume the normal arithmetics and thus do not give the details.

The start-up message in Creol is called run, which defines the active behavior of the objects. Each object in Creol, shows reactive behavior upon receiving messages. It means that upon receipt of a message, a new process is created inside the object for executing the method corresponding to that message. The execution of a method cannot be interrupted arbitrarily. However, the currently executing process may, at its own discretion, release the processor (at some predefined processor release point using

b	:	Boolean
e	:	Expression
g	:	Guard
m	:	Method
n	:	Identifier
s	:	Statement
t	:	Label
x	:	Object

$$
\begin{array}{rcl}
in & ::= & \textbf{interface}\ n([n:n]^{*}_{,})\ [\textbf{inherits}\ [n[(e^{+}_{,})]^{?}]^{+}_{,}]^{?} \\
 & & \textbf{begin}\ [[\textbf{with}\ n]^{?}\ \textbf{op}\ n]^{+}\ \textbf{end} \\
cl & ::= & \textbf{class}\ n([n:n]^{*}_{,})\ \textbf{implements}\ [n[(e^{+}_{,})]^{?}]^{+}_{,} \\
 & & \textbf{begin}\ [\textbf{var}\ Vdcl]^{*}\ [mtd]^{+}\ \textbf{end} \\
Vdcl & ::= & n^{+}_{,} : [int \mid bool] \\
mtd & ::= & [\textbf{with}\ n]^{?}\ \textbf{op}\ n\ ==\ S \\
S & ::= & s \mid s; S \\
s & ::= & n := e \mid !x.m() \mid t!x.m() \mid [n :=]^{?} t.get \mid \textbf{release} \mid \textbf{await}\ g \\
 & & \mid \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od} \mid \textbf{if}\ b\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{skip} \\
g & ::= & b \mid t? \mid g \wedge g \mid g \vee g
\end{array}
$$

Figure 2.1: BNF grammar for Creol (adapted from [27]) where * and $^{+}$ show repetition for at least 0 and 1 times, respectively; $^{?}$ denotes an optional element; and, a subscript $_{,}$ implies a comma-separated list.

the **await** or **release** keywords) allowing other enabled processes to start execution. When a process is executing, it is not interrupted until it finishes or reaches a release point. Release points can be conditional, written as **await** g. If g is satisfied, the process keeps the control; otherwise, it releases the processor. The rest of the process is disabled while the guard g is not satisfied. When the processor is free, an enabled process is nondeterministically selected and started, i.e., Creol does not specify any scheduling policy. The **release** statement unconditionally releases the processor and the continuation of the process is immediately enabled, i.e., it can immediately start to compete with other enabled processes for execution.

The guard in a conditional release point can also test whether an asynchronous call has been accomplished. This is handled implicitly by the semantics with a completion notification. Examples of a release point are '**await** !taken' and '**await** l?' in Figure 2.4. Notice that waiting for the completion of a call without releasing the processor blocks the caller object, disallowing other processes in the object to use the processor.

To make a system run, one should provide the initialization script. In the Creol convention, one can have a class for the initialization. This is similar to the class containing the main method in Java.

If a method invocation p is associated with a label t, written as t!p(), the sender can wait for a reply using the blocking statement n := t.**get** or in a nonblocking way by including t? in a conditional release point, e.g., as in **await** t?. A reply is sent back automatically when the called method finishes. Before the reply is available, executing **await** t? releases the processor whereas a blocking `get` statement does not. While the processor is not released, the other processes in the object do not get a chance for execution; if `get` is related to a self call and its reply is not yet available, it forces synchronous execution of the called method. Standard usages of asynchronous method calls include the statement sequence $x := o!m(\overline{e}, d)$; $v := x.$`get` which encodes a *blocking call*, abbreviated $v := o.m(\overline{e}, d)$ (often referred to as a synchronous call), and the statement sequence $x := o!m(\overline{e}, d)$; `await` $x?$; $v := x.$`get` which encodes a non-blocking, *preemptible call*, abbreviated `await` $v := o.m(\overline{e}, d)$.

```
class MutEx (left: Entity, right: Entity) begin
 var taken : bool
 op initial ==
    taken := false
 op reqL ==
    await !taken;
    taken := true;
    l ! left.grant();
    await l?;
    taken := false
 op reqR ==
    await !taken;
    taken := true;
    r ! right.grant();
    await r?;
    taken := false
end
```

Figure 2.2: A Creol class for mutual exclusion.

In standard Creol, different invocations of a method call are associated with different values of the label. For instance executing the statement `t!p()` twice results in two instances of the label t. This allows for distinguishing the return values of the calls. Dynamic labels give rise to an infinite state space for non-terminating reactive systems. This is fine as long as we use simulation or testing, but for verification purposes we need to abstract from this. One way of such abstraction is explained later in Chapter 6.

Figure 2.2 shows the Creol code for a mutual exclusion handler object. An instance of `MutEx` should be provided with two instances of a class Entity representing the two objects (on its left and right) trying to get hold of the `MutEx` object. To do so, they may call reqL or reqR, respectively. The request is suspended if the object is already taken; otherwise, it is granted. The `MutEx` waits until the requester entity finishes its operation (in its grant method).

2.1.2 Creol Semantics in Rewrite Logic

Creol has a semantics defined in Rewriting logic [35] which can be used directly as a language interpreter in Maude [9]. The semantics of standard Creol is explained in detail in [27] and can be used for the analysis of Creol programs. In this section we focus on the semantics of some core Creol statements and we extend it with Real-Time specifications.

The state space of a Creol program is given by terms of the sort `Configuration` which is a set of objects, messages, and futures. The empty configuration is denoted

$$\texttt{rl [}\mathit{skip}\texttt{] : } \langle o,\ a,\ \texttt{\{}l \mid \texttt{skip;}s\texttt{\}} \circ q \rangle \longrightarrow \langle o,\ a,\ \texttt{\{}l \mid s\texttt{\}} \circ q \rangle \ .$$

$$\begin{array}{l}\texttt{rl [}\mathit{assign}\texttt{] : } \langle o,\ a,\ \texttt{\{}l \mid x\texttt{:=}e\texttt{;}s\texttt{\}} \circ q \rangle \\ \longrightarrow \texttt{if } x \in \texttt{dom}(l) \texttt{ then } \langle o,\ a,\ \texttt{\{}l\texttt{[}x \mapsto [\![e]\!]^{none}_{aol}\texttt{]} \mid s\texttt{\}} \circ q \rangle \\ \qquad\qquad\qquad \texttt{else } \langle o,\ a\texttt{[}x \mapsto [\![e]\!]^{none}_{aol}\texttt{]},\ \texttt{\{}l \mid s\texttt{\}} \circ q \rangle \texttt{ fi } .\end{array}$$

$$\texttt{rl [}\mathit{release}\texttt{] : } \langle o,\ a,\ \texttt{\{}l \mid \texttt{release;}s\texttt{\}} \circ q \rangle \longrightarrow \langle o,\ a,\ \texttt{firstEn(schedule(\{}l \mid s\texttt{\},}q\texttt{))} \rangle \ .$$

$$\begin{array}{l}\texttt{crl [}\mathit{await1}\texttt{] : \{}\langle o,\ a,\ \texttt{\{}l \mid \texttt{await } e\texttt{;}s\texttt{\}} \circ q \rangle\ \texttt{c\}} \\ \longrightarrow \texttt{\{}\langle o,\ a,\ \texttt{\{}l \mid s\texttt{\}} \circ q \rangle\ \texttt{c\} if } [\![e]\!]^{c}_{aol} \ .\end{array}$$

$$\begin{array}{l}\texttt{crl [}\mathit{await2}\texttt{] : \{}\langle o,\ a,\texttt{\{}l \mid \texttt{await } e\texttt{;}s\texttt{\}} \circ q \rangle\ \texttt{c\}} \\ \longrightarrow \texttt{\{}\langle o,\ a,\ \texttt{\{}l \mid \texttt{release;await } e\texttt{;}s\texttt{\}} \circ q \rangle\ \texttt{c\} if } \neg[\![e]\!]^{c}_{aol} \ .\end{array}$$

$$\texttt{rl [}\mathit{context\text{-}switch}\texttt{] : } \langle o,\ a,\ \texttt{\{}l \mid \epsilon\texttt{\}} \circ q \rangle \longrightarrow \langle o,\ a,\ \texttt{firstEn(}q\texttt{)} \rangle \ .$$

Figure 2.3: The semantics of Creol in Maude.

`none` and white-space denotes the associative and commutative union operator on configurations. Objects are defined as tuples

$$\langle o,\ a,\ q\ \rangle$$

where o is the identifier of the object, a is a map which defines the values of the attributes of the object, and q is the task queue. Tasks are of sort `Task` and consist of a statement s and the task's local variables l. We denote by $\{l|s\} \circ\ q$ the result of appending the task $\{l|s\}$ to the head of the queue q. For a given object, the first task in the queue is the *active task* and the first statement of the active task to be executed is called the *active statement*.

Let σ and σ' be maps, x a variable name, and v a value. Then $\sigma(x)$ denotes the lookup for the value of x in σ, $\sigma[x \mapsto v]$ the update of σ such that x maps to v, $\sigma \circ \sigma'$ the composition of σ and σ', and $\mathrm{dom}(\sigma)$ the domain of σ. Given a mapping, we denote by $[\![e]\!]^c_\sigma$ the evaluation of an expression e in the state given by σ and the global configuration c (the latter is only used to evaluate the polling of futures; e.g., `await x?`).

Rewrite rules execute statements in the active task in the context of a configuration, updating the values of attributes or local variables as needed. For an active task $\texttt{\{}l \mid s\texttt{\}}$, these rules are defined inductively over the statement s. Some (representative) rules are presented in Figure 2.3. Rule *skip* shows the general set-up, where a `skip` statement is consumed by the rewrite rule. Rule *assign* updates either the local variable or the attribute x with the value of an expression e evaluated in the current state. The suspension of tasks is handled by rule *release*, which places the active task in the task queue. Rules *await1* and *await2* handle conditional release points.

The auxiliary function `schedule` in fact defines the (local) *task scheduling policy* of the object which in standard Creol is nondeterministic. In the next subsection, we will add explicit scheduling policies to objects. After release or when a method finishes the processor is idle. At this point the first enabled task in the queue is picked by the auxiliary function `firstEn`; this function moves the first enabled task to the head of the queue. Since the valuation of the enabling condition of a task may change in the course of time, it has to be checked when a new task must be started.

2.2 Real-Time in Creol[1]

We explain how to model real-time systems using concurrent objects in Creol. To this end, we develop an extension of the Creol language with support for specifying real-time information. As in the previous section, we aim at schedulability analysis and therefore we choose for a descriptive approach; namely, we add best and worst case execution information to methods and deadlines to method calls rather than clocks and time-outs that can affect the system behavior. Furthermore, scheduling strategies can be added to object definitions, which may depend on the remaining deadlines of tasks.

In real-time Creol, normal statements are executed instantaneously. We add a special statement `duration` to specify delays during execution. A `duration` statement takes two parameters specifying the best and worst case execution delays. There is a special case of the blocking `get` statement. This statement may take as much time as needed until the corresponding method call finishes; however, as soon as the callee finishes, the blocking `get` will succeed.

Every method call, including self calls, can be given a deadline; the deadline is given as an extra parameter to the method call. This deadline specifies the relative time before which the corresponding method should be scheduled and completed. Since we do not have message transmission delays, the deadline expresses the time until a reply is received. Thus, it corresponds to an *end-to-end* deadline. Self-calls may inherit the remaining deadline of the caller method by using the keyword `deadline`. As explained in Chapter 4, to be able to model check schedulability, *all* method calls must have a finite deadline.

In this chapter, we first give a complete semantics of Real-Time Creol in Real-Time Maude, which extends the original rewrite semantics of Creol. We cannot perform complete analysis on this semantics, because first of all, it is infinite state, and secondly, the analysis results given by Real-Time Maude is incomplete (due to decidability constraints). We then provide a finite-state abstraction of Creol with timed automata semantics in terms of an extension of the framework in the previous chapter; this semantics can be readily used for schedulability analysis, as well as model checking of other properties, in UPPAAL.

Figure 2.4 repeats the Creol code for a mutual exclusion handler object annotated with timing information. The duration statements specify the delays and the numbers in parantheses show the deadlines required for the method calls.

[1]The rest of this chapter is an improvement and extension of the results published in [15].

```
class MutEx (left: Entity, right: Entity) begin
  var taken : bool
  op initial ==
     duration(1,1); taken := false
  op reqL ==
     duration(1,2); await !taken;
     duration(4,4); taken := true;
     l ! left.grant(10);
     await l?;
     duration(2,3); taken := false
  op reqR ==
     duration(1,2); await !taken;
     duration(4,4); taken := true;
     r ! right.grant(10);
     await r?;
     duration(2,3); taken := false
end
```

Figure 2.4: A Creol class for mutual exclusion with timing information.

2.2.1 Real-Time Creol's Semantics

Objects in real-time Creol are given explicit scheduling policies. This is reflected in the object tuple

$\langle o,\ p,\ a,\ q\ \rangle$

such that p defines the scheduling policy. The function schedule now takes also a scheduling policy as parameter. At the end of this section, we define two example scheduling policies FCFS and EDF. Furthermore, tasks in Real-Time Creol have a special local variable *deadline* that holds the remaining deadline of the task.

The rewrite rules of the Real-Time Creol semantics are given in Figure 2.5. The first rule ensures that a `duration` statement may terminate only if its best case execution time has been reached. In order to facilitate the conformance testing discussed in Chapter 6, we define a global clock clock(t) in the configurations (where t is of sort Time) to time-stamp observable events. These observables are the invocation and return of method calls. Rule *async-call* emits a message to the callee $[\![e]\!]^{none}_{(a \circ l)}$ with method m, actual parameters $[\![\overline{e}]\!]^{none}_{(a \circ l)}$ including the deadline, a fresh future identifier n, which will be bound to the task's so-called destiny variable [13], and, finally, a time stamp t. In the (method) *activation* rule, the function task transforms such a message into a task which is placed in the task queue of the callee by means of the scheduling function schedule. The function task creates a map which assigns the values of the actual parameters to the formal parameters (which includes the deadline variable) and which assigns the future identity to the destiny variable. The statement of the created `crl [`*duration*`]:` $\langle o,\ \{l \mid \texttt{duration}(b,w);s\} \circ q\rangle \longrightarrow \langle o,\ \{l \mid s\} \circ q\rangle$ `if` $b \leq 0$.

`crl [`*async-call*`]:` $\langle o,\ a,\ \{l \mid x{:=}e!m(\bar{e});s\} \circ q\rangle$ `clock`(t)
$\longrightarrow \langle o,\ a,\ \{l[x \mapsto n] \mid s\} \circ q\rangle\ \ m(t, [\![e]\!]^{none}_{aol}, [\![\bar{e}]\!]^{none}_{aol}, n)\ n$ `if fresh`(n) .

`crl [`*activation*`]:` $\langle o,\ p,\ \{l \mid s\} \circ q\rangle\ \ m(t,o,\bar{v})$
$\longrightarrow \langle o,\ p,\ \{l \mid s\} \circ$ `schedule(task(`$m(o,\bar{v})$`),`p`,`q`)` $\rangle$.

`crl [`*return*`]:` $\langle o,\ a,\ \{l \mid \texttt{return}(e);s\} \circ q\rangle\ \ n$ `clock`(t)
$\longrightarrow \langle o,\ a,\ \{l \mid s\} \circ q\rangle$ `clock`(t) $\langle n, [\![e]\!]^{none}_{(aol)}, t\rangle$ `if` $n = l(\texttt{destiny})$.

`crl [`*get*`]:` $\langle o,\ a,\ \{l \mid x := e.\texttt{get};s\} \circ q\rangle\ \ \langle n,v,t\rangle$
$\longrightarrow \langle o,\ a,\ \{l \mid x := v;s\} \circ q\rangle\ \ \langle n,v,t\rangle$ `if` $[\![e]\!]^{none}_{aol} = n$.

`crl [`*tick*`]:` $\{C\} \longrightarrow \{\delta(C)\}$ `in time` t `if` $t <$ `mte`$(C) \wedge$ `canAdvance`(C) .

`op canAdvance: Configuration` $\rightarrow$ `Bool .`
`eq canAdvance(C1 C2) = canAdvance(C1)` $\wedge$ `canAdvance(C2) .`
`eq canAdvance(`$\langle o,\ p,\ a,\ \{l \mid \texttt{duration}(b,w);s\} \circ q\ \rangle$`) =` $w > 0$.
`eq canAdvance(`$\langle o,\ p,\ a, \{l \mid x := e.\texttt{get};s\} \circ q\ \rangle\ \ n$`) = true if` $n = [\![x]\!]^{none}_{(\bar{a}\circ\bar{l})}$.
`eq canAdvance(C) = false [owise] .`

`op mte: Configuration` $\rightarrow$ `Time .`
`eq mte(C1 C2) = min(mte(C1), mte(C2)) .`
`eq mte(`$\langle o,\ p,\ a, \{l \mid \texttt{duration}(b,w);s\} \circ q\ \rangle$`) =` w .
`eq mte(C) =` ∞ `[owise] .`

`op` δ_1`: Task Time` $\rightarrow$ `Task .`
`eq` $\delta_1(\{l \mid s\}, t) = \{l[\texttt{deadline} \mapsto l(\texttt{deadline}) - t] | s\}$.

`op` δ_2`: TaskQueue Time` $\rightarrow$ `TaskQueue .`
`eq` $\delta_2(\{l|s\} \circ q, t) = \delta_1(\{l|s\}, t) \circ \delta_2(q,t)$.
`eq` $\delta_2(\epsilon, t) = \epsilon$

`op` δ_3`: Task Time` $\rightarrow$ `Task .`
`eq` $\delta_3(\{l|\texttt{duration}(b,w);s\}, t) = \{l[\texttt{deadline} \mapsto l(\texttt{deadline}) - t] \mid \texttt{duration}(b-t, w-t);s\}$.
`eq` $\delta_3(\{l|s\}, t) = \{l[\texttt{deadline} \mapsto l(\texttt{deadline}) - t]|s\}$ `[owise] .`

`op` δ`: Configuration Time` $\rightarrow$ `Configuration .`
`eq` δ`(C1 C2, `t`) =` δ`(C1,`t`)` δ`(C2,`t`) .`
`eq` δ`(clock(`t'`),`t`) = clock(`$t'+t$`) .`
`eq` $\delta(\langle o,\ p,\ a,\ \{l|s\} \circ q\ \rangle, t) = \langle o,\ a,\ \delta_3(\{l|s\}) \circ \delta_2(q)\rangle$.
`eq` δ`(C, `t`) = C [owise] .`

Figure 2.5: The semantics of Real-Time Creol in Real-Time Maude

```
op EDF: → Policy .
op FCFS: → Policy .

op schedule: Task Policy TaskQueue → TaskQueue .
eq schedule(t, EDF, q) = scheduleEDF(t,q) .
eq schedule(t, FCFS, q) = scheduleFCFS(t,q) .

op scheduleEDF: Task TaskQueue → TaskQueue .
eq scheduleEDF({l|s}, {l′|s′} ∘ q) =
   if l(deadline) < l′(deadline) then {l|s} ∘ {l′|s′} ∘ q
                              else {l′|s′} ∘ scheduleEDF({l|s}, q) .
eq scheduleEDF({l|s}, ϵ) = {l|s} .

eq scheduleFCFS: Task TaskQueue → TaskQueue .
eq scheduleFCFS({l|s}, q) = q ∘ {l|s} .
```

Figure 2.6: FCFS and EDF scheduling policies for real time Creol

task consist of the body of the method. Rule *return* adds the return value from a method call to the future identified by the task's destiny variable and time stamps the future at this time. Rule *get* describes how the `get` operation obtains the returned value.

The global advance of time is captured by the rule *tick*. This rule applies to global configurations in which all active statements are duration statements which have not reached their worst execution time or blocking get statements. These conditions are captured by the predicate canAdvance in Figure 2.5. When the *tick* rule is applicable, time may advance by any value t below the limit determined by the auxiliary *maximum time elapse* [38] function mte, which finds the lowest remaining worst case execution time for any active task in any object in the configuration. Note that the blocking `get` operation allows time to pass arbitrarily while waiting for a return.

When time advances, the function δ, besides advancing the global clock, determines the effect of advancing time on the objects in the configuration, using the auxiliary functions δ_i, for $i = 1, 2, 3$, defined in Figure 2.5, to update the tasks. The function δ_1 decreases the deadline of a task. The function δ_2 applies δ_1 to all queued tasks; δ_2 has no effect on an empty queue ϵ. The function δ_3 additionally decreases the current best and worst case execution times of the active `duration` statements.

Specifying Scheduling Strategies

At the highest level of abstraction, no scheduling strategy is specified as in standard Creol. This amounts to nondeterministic selection of methods from a bag of suspended processes. If a property is proved to hold in this setting, it will hold for any scheduling strategy. In real-time setting, however, this can easily lead to non-schedulability.

Real-time systems usually need special-purpose designed strategies in order to be schedulable: usually the task with the shortest deadline or another one needed by this task must be scheduled and executed first.

In real-time Creol, a scheduling strategy should be defined in the function schedule. Figure 2.6 gives the definition of two scheduling policies as example: First Come First Served (FCFS) and Earliest Deadline First (EDF). Since always the first task in the queue will be executed, scheduling amounts to inserting new tasks in the right place in the queue. An FCFS scheduler enqueues the tasks in their order of arrival regardless of any priorities. An EDF scheduler uses the remaining deadline of each task as its priority; the task that has a smaller remaining deadline has a higher priority. Recall that every task (i.e., method invocation) has a local variable called deadline which is updated dynamically to reflect the remaining deadline of the task at any time. This is an example of dynamic priority scheduling.

Other strategies can be defined in a similar way. One can implement a fixed priority scheduler following the same approach as in EDF, but using a fixed priority table for task types. Fixed priorities may be assigned to task invocations rather than task types. Resource-aware Creol models may need extra functions to handle issues like priority inversion and inheritance. These problems are not addressed in this thesis but flexible ways to specify rich scheduling policies at a high level could be an interesting trend of future research.

2.3 Case Studies

2.3.1 Coordinator

The coordinator class, in Figure 2.7 is taken from [27] and is concretized for three-way synchronization (while abstracting data away). The Coordinator class implements three interfaces, each with one method, e.g., **with** Any **op** m1 that defines method m1 which can be called from objects of *any* type. The method init in an object is immediately executed upon object creation. The method run specifies active behavior of the object; it will have to compete with other tasks in the queue after init has finished.

Execution of the method run begins with a delay of 2 to 3 time units; this delay represents some computation that is abstracted away. The first await statement in this method would initially block because the variables $s1$, $s2$ and $s3$ are initially false. This line actually waits until the three parties of synchronization are ready. These variables are set in methods $m1$, $m2$ and $m3$, respectively. We look at $m1$; the other two methods are similar.

The first time $m1$ is called it passes the first await statement because of the initial values of the variables *sync* and $s1$. After 1 time unit, it sets $s1$ to true to announce the readiness of this party. It will then wait until synchronization takes place. It must reset $s1$ to allow the next round of synchronization, because the *run* method waits until all parties have finished their first round.

The main synchronized task is modeled in the method *body* which is called synchronously from the *run* method; this is modeled using the blocking `get` statement

```
interface C1 begin with Any op m1 end
interface C2 begin with Any op m2 end
interface C3 begin with Any op m3 end
class Coordinator implements C1, C2, C3 begin
   var s1,s2,s3,sync : bool
   op init ==
      s1 := false;
      s2 := false;
      s3 := false;
      sync := true;
      duration(2, 4)
   op body ==
      duration(2, 5)
   op run ==
      duration(1, 2);
      await (s1 /\ s2 /\ s3);
      duration(1, 1);
      b!body(10);       // force sync call with deadline 10
      b.get;
      sync := false;
      duration(1, 2);
      await (~s1 /\ ~s2 /\ ~s3);
      duration(1, 1);
      sync := true;
      !run(50) // deadline = 50
   with Any op m1 ==
      await (sync /\ ~s1);
      duration(1, 1);
      s1 := true;
      await ~sync;
      duration(1, 1);
      s1 := false
   with Any op m2 ==  ...like m1...
   with Any op m3 ==  ...like m1...
end
```

Figure 2.7: A real-time 3-way coordinator in Creol.

on the local label *b*. This task is again modeled simply as a delay which represents some computation that is abstracted away.

In this example, every communicated message is considered to be visible and therefore a delay has been specified between every two method calls. Furthermore, there is a delay between the processor release points. For schedulability analysis of this model, we will shows in Chapter 6 how to generate timed automata automatically from this Creol code. These automata models can then be used for schedulability analysis using UPPAAL as explained in the following chapters.

2.3.2 Thread-Pools

Figure 2.8 shows a Creol model of a thread pool. The model defines a `Thread` class and the `ResourcePool` class. The task list is modeled implicitly in terms of the message queue of an instance of the `ResourcePool` class. The variable `size` represents the number of available threads, i.e., instances of the `Thread` class. The variable `pool` is used to hold a reference to those threads that are currently not executing a task. This class uses a predefined data type Set which is available in Creol. We make use of the functions add, remove and choose on variables of type Set that provide the basic operations on a set. The choose function nondeterministically chooses an element from the set without removing it.

Tasks are modeled in terms of the method start inside the `Thread` class. For our analysis the functional differences between tasks is irrelevant, so the method is specified in terms of its duration only. A task ends by a call to the method `finish` of the `ResourcePool` object which adds the executing thread to the pool of available threads. Therefore, this thread can be selected again to execute another task.

Tasks are generated (by the environment) with (asynchronous) calls of the `invoke` method of the `ResourcePool` object. In case there are no available threads, the execution of the `invoke` method suspends by the execution of the `await` statement which releases control (so that a call of the `finish` method can be executed). When multiple tasks are pending and a thread becomes available, the scheduling strategy of the `ResourcePool` object determines which task should be executed next when the current task has been completed.

In this model, not all method calls are relevant for schedulability analysis. The only part that takes some time is the execution of the task itself. For schedulability analysis, we will consider the whole model as one entity to be analyzed. In Chapter 3, we will provide automata models for this case study that can be used for schedulability analysis. We will show in Chapter 6 a method to check the conformance of this Creol code to these automata models.

```
class Thread(myPool: ResourcePool) implements IThread
begin
  op run ==
    !myPool.finish()
  with ResourcePool op start ==
    skip;
    duration(5,6);
    !myPool.finish()
end

class ResourcePool(size: Int) implements IResourcePool
begin
  var pool: Set[Thread];
  op init ==
    var thread:Thread;
    pool := {};
    while (size>0) do
      thread := new Thread(this);
      size := size-1
    end
  with Any op invoke ==
    var thread Thread;
    await ¬isempty(pool);
    thread := choose(pool);
    pool := remove(pool,thread);
    !thread.start(deadline)
  with Thread op finish ==
    pool := add(pool,caller)
end
```

Figure 2.8: The thread pool

Chapter 3

Automata-Based Actor Framework

Actor model was originally introduced by Hewitt [20] as an agent-based language, and later developed by Agha [1] into a concurrent object-based model. Actors are units of distribution and concurrency. Furthermore, actors have encapsulated states and behavior; they have local variables, but no shared variables. They communicate via asynchronous (non-blocking) message passing, and arrival of messages is guaranteed.

We use the actor model as the pure asynchronous setting of concurrent objects. In this Chapter, we give an automata-based actor framework for modeling real-time systems with an operational interpretation of the actor model, as in Rebeca [43]. A system is composed of objects; templates of objects are defined in classes. When no ambiguity arises we may use the term actor for a class or an object. A class defines a method for each message it can handle. A method is a sequential code which may send messages. There is at least a dedicated method in each class, which is responsible for initialization and start-up; thus modeling its active behavior. A class can have known actors that serve as place holders for the objects that can communicate with instances of that class.

The goal of our framework is checking schedulability as will be explained later in Chapter 4; as shown in Chapter 4, we can put a finite bound on the queue lengths and obtain schedulability results that hold for any queue length. As schedulability requirement, all method calls are given a deadline; if not, the called method inherits the deadline of its caller method. Since we deal with models and not programs, the methods (or individual statements) should carry specifications of their (best- and worst-case) execution times. For this purpose, we make use of guards and invariants in timed automata. Since existing timed automata analysis tools cannot create automata instances dynamically, we do not support dynamic actor creation. Last but bot least, actors can specify their own scheduling strategies using timed automata; a good schedule brings asynchronous actors to concurrence.

In this chapter, we describe how to systematically use timed automata and UPPAAL for modeling systems of actors. First, we give an introduction to timed automata in general. Then, we demonstrate our actor framework with the help of a running example: a *mutual exclusion handler* called `MutEx`. Section 3.2.1 provides the formal actor model. We will conclude this chapter by applying and extending the framework to modeling of a thread-pool and a peer-to-peer system broker.

3.1 Background: Timed Automata

In this section, we define timed automata syntax and semantics and very briefly explain how timed automata can be used in general for modeling real-time systems. Since we will use UPPAAL [31] for the analysis, these definitions are based on the syntax and semantics used in UPPAAL. UPPAAL is a model-checker based on the theory of timed automata [3, 7] and its modeling language offers additional features such as bounded integer variables and urgency.

In short, timed automata are automata enriched with clocks. Timed automata provide a dense-time model where a clock variable evaluates to a real number. If there are multiple clocks in a model, they all progress at the same rate. Updating a clock value is done by resetting it to zero. This action happens instantaneously without stopping the clock. The current value of a clock can be checked using conjunctions over conditions of the form $c \sim n$ or $c - c' \sim n$, called clock constraints, where c and c' are clocks, n is an integer and $\sim \in \{<, \leq, =, \geq, >\}$.

DEFINITION 3.1.1 (TIMED AUTOMATA). *Suppose $\mathcal{B}(C)$ is the set of all clock constraints on the set of clocks C. A timed automaton over actions Σ and clocks C is a tuple $\langle L, l_0, \longrightarrow, I \rangle$ representing*

- *a finite set of locations L, including an initial location l_0;*
- *the set of edges $\longrightarrow \subseteq L \times \mathcal{B}(C) \times \Sigma \times 2^C \times L$; and,*
- *a function $I : L \mapsto \mathcal{B}(C)$ assigning an invariant to each location.*

□

An edge may be written as $l \xrightarrow[g\ ,\ r]{a} l'$. It means that action '$a$' may change the location l to l' by resetting the clocks in r, if clock constraints in g (as well as the invariant of l') hold. In addition, a location can be marked *urgent* which is equivalent to resetting a fresh clock x in all of its incoming edges and adding an invariant $x \leq 0$ to the location; intuitively, it means that the automaton cannot spend any time in that location [31].

A timed automaton is called *deterministic* if and only if for each $a \in \Sigma$, if there are two edges from l labeled by the same action $l \xrightarrow[g\ ,\ r]{a} l'$ and $l \xrightarrow[g'\ ,\ r']{a} l''$ then the guards g and g' are exclusive (i.e. $g \wedge g'$ is unsatisfiable).

DEFINITION 3.1.2 (TIMED AUTOMATA SEMANTICS). *A timed automaton defines an infinite labeled transition system whose states are pairs (l, u) where $l \in L$ and $u : C \to \mathbb{R}_+$ is a* clock assignment. *We denote by $\mathbf{0}$ the assignment mapping every clock in C to 0. The initial state is $(l_0, \mathbf{0})$. There are two types of transitions:*

- *delay transitions* $(l,u) \overset{d}{\to} (l,u')$ *where* $d \in \mathbb{R}_+$*, if* u' *is obtained by delaying every clock for* d *time units, written as* $u' = u + d$*, and for each* $0 \leq d' \leq d$*,* $u + d'$ *satisfies the invariant of location* l*; and,*
- *action transitions* $(l,u) \overset{a}{\to} (l',u')$ *where* $a \in \Sigma$*, if there exists* $l \xrightarrow[g\ ,\ r]{a} l'$ *such that* u *satisfies the guard* g*,* u' *is obtained by resetting all clocks in* r *and leaving the others unchanged and* u' *satisfies the invariant of* l'*.*

□

In the following, we assume that the set of actions Σ is partitioned into two disjoints sets: a set Σ_I of *input actions* $a?$ and a set Σ_O of *output actions* $a!$. A *non-observable internal actions* τ is also assumed. Let $\Sigma_\tau = \Sigma \cup \{\tau\}$. A timed automaton with inputs and outputs is a timed automaton over Σ_τ.

Timed traces A *timed sequence* $\sigma \in (\Sigma_\tau \cup \mathbb{R}_+)^*$ is a sequence of timed actions in the form of $\sigma = t_1 a_1 t_2 a_2 \ldots a_n t_{n+1}$ such that for all i, $1 \leq i \leq n$, $t_i \leq t_{i+1}$. Given a timed sequence σ, $\pi_{obs}(\sigma)$ denotes the projection of σ on Σ, intuitively deleting $t_i\tau$ occurrences. The sequence $\pi_{obs}(\sigma)$ is called the *observable timed sequence* associated to σ.

A *run* of a timed automaton A from initial state $(l_0, \mathbf{0})$ over a timed sequence $\sigma = t_1 a_1 t_2 a_2 \ldots a_n t_{n+1}$ is a sequence of transitions:

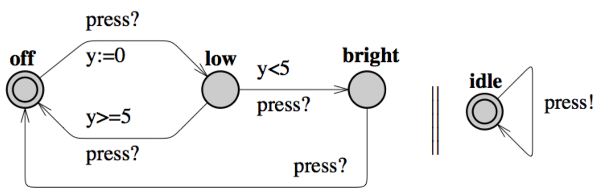
$$(l_0, \mathbf{0}) \overset{d_1}{\to} (l_0, u_0') \overset{a_1}{\to} (l_1, u_1) \overset{d_2}{\to} \cdots \overset{a_n}{\to} (l_n, u_n) \overset{d_{n+1}}{\to} (l_n, u_n')$$

where $d_1 = t_1$ and for all i, $1 < i \leq n+1$, $t_i = t_{i-1} + d_i$. The set *Traces*(A) of timed traces of A is the set of timed sequences σ for which there exists a run of A over σ. The set *Traces*$_{obs}(A)$ of observable timed traces of A is the set $\{\pi_{obs}(\sigma) \mid \sigma \in \mathit{Traces}(A)\}$.

3.1.1 Networks of Timed Automata

A system may be described as a collection of timed automata with inputs and outputs A_i $(1 \leq i \leq n)$ communicating with each other. The behavior of the system is then defined as the parallel composition of those automata $A_1 \parallel \cdots \parallel A_n$. Semantically, the system can delay if all automata can delay and can perform an action if one of the automata can perform an internal action or if two automata can synchronize on complementary actions (inputs and outputs are complementary).

Definition 3.1.3 (Semantics of a Network of Timed Automata). *Let the set of* $A_i = \langle L_i, l_i^0, \longrightarrow_i, I_i \rangle$ *form a network of* n *timed automata with the clocks* C *and actions* Σ_τ*. Let* $\bar{l_0} = (l_1^0, \ldots, l_n^0)$ *be the initial location vector. The semantics is defined as a transition system whose states are* $(l_1, \ldots, l_n, u)$ *where* $l_i \in L_i$ *and* $u : C \to \mathbb{R}_+$*. The initial state is* $s_0 = (\bar{l_0}, \mathbf{0})$*. The transition relation is defined by three types of transitions:*



(a) Lamp (b) User

Figure 3.1: The simple lamp example [6]

- *delay transitions* $(\bar{l}, u) \xrightarrow{d} (\bar{l}, u')$ *where* $d \in \mathbb{R}_+$, *if* u' *is obtained by delaying every clock for* d *time units and for each* $0 \leq d' \leq d$, u' *satisfies the invariant of every location in* $\bar{l}$*;*

- *local transitions* $(\bar{l}, u) \rightarrow (\bar{l}[l'_i/l_i], u')$ *if there exists* $l_i \xrightarrow[g\ ,\ r]{\tau} l'_i$ *such that* u *satisfies the guard* g, u' *is obtained by resetting all clocks in* r *and leaving the others unchanged and* u' *satisfies the invariants of* $\bar{l}[l'_i/l_i]$*; and,*

- *synchronization transitions* $(\bar{l}, u) \rightarrow (\bar{l}[l'_j/l_j, l'_i/l_i], u')$ *if there exists* $l_i \xrightarrow[g_i\ ,\ r_i]{a?} l'_i$ *and* $l_j \xrightarrow[g_j\ ,\ r_j]{a!} l'_j$ *such that* u *satisfies the guard* $g_i \wedge g_j$, u' *is obtained by resetting all clocks in* $r_i \cup r_j$ *and leaving the others unchanged and* u' *satisfies the invariants of* $\bar{l}[l'_j/l_j, l'_i/l_i]$.

□

Marking a location urgent in an automaton indicates that the automaton cannot spend any time in that location. In a network of timed automata, the enabled transitions from an urgent location may be interleaved with the enabled transitions from other automata (while time is frozen). Like urgent locations, *committed* locations freeze time; furthermore, if any process is in a committed location, the next step must involve an edge from one of the committed locations.

Example 3.1.4. [6] Figure 3.1(a) shows a timed automaton modeling a simple lamp. The lamp has three locations: `off`, `low`, and `bright`. If the user presses a button, i.e., synchronizes with `press?`, then the lamp is turned on. If the user presses the button again, the lamp is turned off. However, if the user is fast and rapidly presses the button twice, the lamp is turned on and becomes bright. The user model is shown in Figure 3.1(b). The user can press the button randomly at any time or even not press the button at all. The clock y of the lamp is used to detect if the user was fast ($y < 5$) or slow ($y \geq 5$).

3.1.2 Timed Automata in Uppaal

As accepted in Uppaal, we allow defining variables of type boolean and bounded integers for each automaton. Variables can appear in guards and updates. Furthermore, clocks can be reset to any integer in Uppaal. The semantics of timed automata changes such that each state will include the current values of the variables as well, i.e. (l, u, v) with v a variable assignment. An action transition $(l, u, v) \xrightarrow{a} (l', u', v')$ additionally requires v and v' to be considered in the corresponding guard and update. Theoretically [18], timed automata with variables are hybrid automata in which variables are like clocks that never progress in time.

In a network of timed automata, variables can be defined locally for one automaton, globally (shared between all automata), or as parameters to the automata. In Uppaal, timed automata templates are automata defined with a set of parameters which can be of any type, e.g., integer or channel. In a system composition, these templates are instantiated with concrete values for their parameters to form a network of timed automata.

Definition 3.1.5 (Uppaal Model). *An* Uppaal *model consists of: (1) a set of timed automata templates (TAT); (2) global declarations; and, (3) system declarations.* □

An automata template in an Uppaal model consists of a name, a set of arguments, local declarations and a timed automaton definition (as above); formally, $TAT = (tName, Args, local, Auto)$. Global and local declarations contain the definition of clocks and variables. The network of timed automata to be analyzed is defined in the system declarations by instantiating the timed automata templates.

3.2 Modeling Actors in Timed Automata[1]

The first step in modeling an actor is to specify in a timed automaton its *behavioral interface*; when it is clear from the context we may call it simply an interface. The modeler specifies the correct way of using an actor in its interface. On one hand, an interface characterizes the expected pattern of incoming messages; thus, it can be seen as the highest-level abstraction of all environments in which the actor instances can be used. On the other hand, an interface includes the object outputs; thus, it models the input/output behavior of the object while abstracting from the queue, local variables and method implementations.

The actor definition itself consists of its methods and local variables. For every actor, we model each method with a timed automata template; actor variables are shared between these automata. All automata templates for methods and behavioral interfaces are parameterized in the identity of the actor itself, and the identifiers of the actors communicating with it (namely, its known actors). For instance, a `MutEx` may

[1]This section is an improvement and extension of the results published in [24].

communicate with the two objects on its left and right which try to enter a critical section. In Section 3.3, we will show how to model these automata in UPPAAL.

Figure 3.2 shows the specification of the behavioral interface automaton for `MutEx`. It is parameterized in `self`, `Left` and `Right`; `self` is used to hold the actor identity, while `Left` and `Right` represent its known actors. An action `Left.reqL(req_d)?` means a message `reqL` is received from the actor representing the `Left` known actor, and a deadline `req_d` is assigned to it. The interface of `MutEx` includes all possibilities of receiving two requests, however, it disallows granting both requests at the same time. Furthermore, if there is a pending request, it puts a time bound of MAX_REL on the arrival of a release; otherwise, the deadline of the pending request cannot be guaranteed.

Figure 3.3 depicts the automata for the methods of the `MutEx`, called method automata. We abstract from computation, which is represented only by a time delay. However, some variables (e.g., 'taken' in `MutEx`) are needed for correct behavior of the actor. A `MutEx` is initially not taken (the 'taken' variable is set to false in `initial` method). The object on the left (specified as `Left`) may ask for the `MutEx` by sending reqL. In response, the `MutEx` sends a permitL back, if the `MutEx` is not already taken. Similarly, the message reqR may be sent by `Right`. If the `MutEx` is taken, the request is put back in queue by a self call. A release message can be sent by either object on the left or right.

The interface should include the expected timing information for the arrival of messages. Specifically, a time interval between messages is necessary to have a schedulable actor. The given `MutEx` interface keeps track of the time since every request, and expects that the corresponding release arrives no sooner than MIN_REL time units. The interface assumes that the first request is always granted (because `MutEx` is not taken initially). If the next request arrives before a release, it will be waiting for the `MutEx` to be released. Whenever a request is pending, a release must arrive before MAX_REL time units, otherwise the pending request will miss the deadline. By iterating the schedulability analysis (explained in Chapter 4) this interval can be refined so that to indicate the minimum requirement.

The timing constraints in method automata show the amount of computation needed for each step. The invariants on the locations of method automata together with the guards on edges, require that the send operations succeed; otherwise, time cannot progress. This is necessary in order to model the asynchronous nature of the message passing. In other words, outputs are urgent.

Send actions in methods and receive actions in interface automata should be assigned a deadline. Self calls inherit the remaining deadline of the parent task (called delegation), unless an explicit deadline is assigned (called invocation). Delegation may be used when a task is split over two or more methods.

Another common scenario for delegation happens when a task (say handling a 'request' in a `MutEx`) cannot be accomplished immediately (say because the `MutEx` is already 'taken'). Therefore, it needs to make a self call, which must terminate within the original deadline. In other words, the original deadline of a request requires a bound on the time until the request is granted (i.e., a permit is sent back). Variables

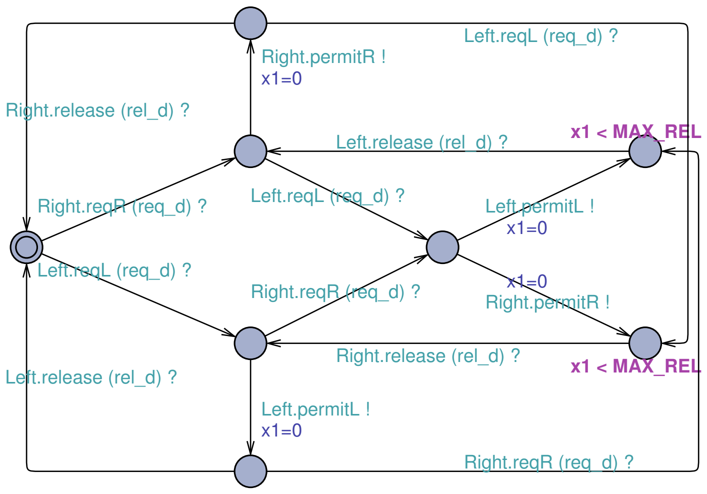


Figure 3.2: A possible interface specification for `MutEx`

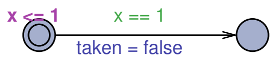
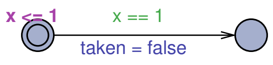
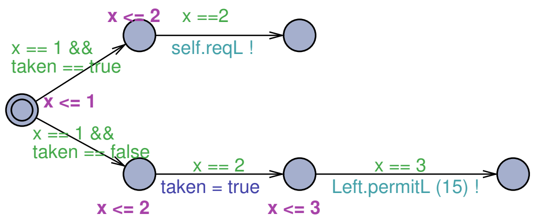
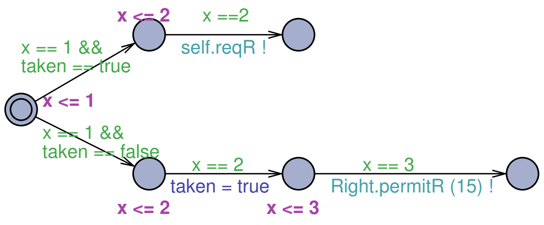
initial	
release	
reqL	 x <= 2 x ==2 self.reqL ! x == 1 && taken == true x <= 1 x == 1 && taken == false x == 2 taken = true x == 3 Left.permitL (15) ! x <= 2 x <= 3
reqR	 x <= 2 x ==2 self.reqR ! x == 1 && taken == true x <= 1 x == 1 && taken == false x == 2 taken = true x == 3 Right.permitR (15) ! x <= 2 x <= 3

Figure 3.3: Specification of a mutual exclusion handler (`MutEx`)

('taken' in this example) are used to keep the current state of the actor and thus to bound such delegation loops. In modeling, we cannot abstract from such variables, because it would let the loops in delegation to continue infinitely. This results in nonschedulability, because the (inherited) deadline becomes smaller every time.

3.2.1 The Formal Timed Actor Model

In this section, we present our formal model of actors. Each actor implements an interface that has a set of method names, which represent the methods the actor provides. Each of the methods in the actor is represented by a timed automaton. An actor also needs a queue for storing incoming messages and a scheduling strategy for determining the order of executing messages. Upon receiving a message, the corresponding method, representing the task to be executed, is inserted into the process queue. Once the currently executing automaton reaches a final state, the next task in the queue is started. Furthermore, we show in Chapter 4 that we may put a finite bound on the queue and still derive schedulability results that hold for any queue length.

We assume a finite global set of method names $\mathcal{M}$ (corresponding to the messages that can be sent and received).

Definition 3.2.1 (Behavioral Interface). *A behavioral interface B with a set of method names $M_B \subseteq \mathcal{M}$ is a deterministic timed automaton over alphabet Σ^B such that:*

- *Σ^B is partitioned into two sets*
  - *output actions: $\Sigma^B_O = \{m! | m \in \mathcal{M} \wedge m \notin M_B\}$*
  - *input actions: $\Sigma^B_I = \{m(d)? | m \in M_B \wedge d \in \mathbb{N}\}$*
- *the edges labeled with output actions have* true *as their guard*

$\square$

The behavioral interface can be viewed as a high level specification of an object. It specifies the (acceptable) observable behavior of the object. An input action $m(d)?$ represents a message m sent to the object by the environment with the deadline d. A correct implementation of the object should be able to finish an incoming call $m(d)?$ before d time units. Output actions are the methods called by this object and should be handled by (other objects in) the environment. The object may send out a message only if allowed in the interface.

A behavioral interface B can be implemented by providing implementation for the methods in M_B. Every method is represented by a timed automaton and may in turn send messages.

Definition 3.2.2 (Class). *A class R implementing the behavioral interface B is a set $\{(m_1, A_1), \ldots, (m_n, A_n)\}$ where:*

- *$M_R = \{m_1, \ldots, m_n\} \subseteq \mathcal{M}$ is a set of method names such that $M_B \subseteq M_R$; and,*
- *for all i, $1 \leq i \leq n$, A_i is a timed automaton representing method m_i with the alphabet $\Sigma_i = \{m! | m \in M_R\} \cup \{m(d)! \mid m \in \mathcal{M} \land d \in \mathbb{N}\}$*

□

Method automata only send messages while computations are abstracted into time delays. Receiving messages (and buffering them) is handled by a scheduler defined next. Sending a message $m \in M_R$ is called a self call. Self calls may or may not be assigned an explicit deadline. The self calls that are not statically assigned a deadline are called delegation. Delegation implies that the internal task (triggered by the self call) is in fact the continuation of the parent task; therefore, the delegated task inherits the (remaining) deadline of the task that triggers it. A scheduler needs to handle the inheritance of the deadlines.

The condition $M_B \subseteq M_R$ requires that a class should at least provide method implementation for handling all messages it may receive according to the interface it implements. It can add other methods which then can only be called as self calls. Checking whether it will produce correct output behavior, and if it can finish all methods in designated deadlines is explained in Chapter 4.

Scheduler An actor needs a queue for storing incoming messages before they are scheduled to be executed. In this section, we give the formal definition of a queue and scheduling strategies. Our scheduler model is inspired by and extends the ideas of task automata [16]. Tasks, being specified using timed automata, may perform self calls, which need to be handled by the scheduler, whereas in task automata tasks are just modeled with their execution time. We also need to support inheriting deadlines for delegation. We assume that the first task in the queue is the currently running task.

For each task, a queue needs to store the method name and its deadline. Furthermore, it needs a clock to keep track of the time since the task is triggered. This enables us to check if a deadline is missed. In theory a queue can be unbounded. We show in Chapter 4 that we may put a finite bound on the queue and still derive schedulability results that hold for any queue length.

Definition 3.2.3 (Unbounded Queue). *An unbounded queue q is a list of tasks together with an infinite set of clocks C_q. Each task is written as $m(d, c)$ where m is a method name, $d \in \mathbb{N}$ is its deadline and $c \in C_q$ keeps track of how long the task has been in the queue. The remaining deadline of m is $d - c$.* □

To have the complete behavior of an object defined, a scheduling strategy (e.g., Earliest Deadline First) should be defined in terms of a scheduler function. A scheduler is used to insert tasks into the queue; it also assigns an unused queue clock to the new task to keep track of the time remaining until its deadline. Typically to determine where in the queue to insert the new task, a scheduler could dynamically examine the remaining time of each task in the queue. However, since the scheduler is to be defined statically, dynamic clock values are not available. Therefore, the scheduler function returns the set of all possibilities for putting the new task in the queue depending on different possibilities of clock values.

Definition 3.2.4 (Scheduler Function). *Given a queue q with clocks C_q and a method name m with deadline d, a scheduler function '$sched(q, m(d))$' returns a set of triples $\{(G, c, q')\}$, where*

- *G is a guard on clocks in C_q (possibly based on d);*
- *$c \in C_q$ is a clock not used (i.e., not assigned to any tasks) in q; and,*
- *q' is a queue with the clocks C_q and represents the queue q after inserting $m(d, c)$ in a particular position as implied by the guard G.*

□

We define an overloading of the scheduler function as $sched(q, m(d, c))$ that inserts a task into the queue using a given clock c. By reusing the deadline and the clock already assigned to a task in the queue, we can model inheriting the deadline. In the case of delegation, the clock assigned to the currently running task is reused.

A scheduler function is *preemptive* if it can place the new task in the first position. Recall that the first task in the queue is the task that is currently running. In this actor framework, we only consider non-preemptive schedulers. Section 4.2.2 briefly discusses the decidability of preemptive scheduling.

Example 3.2.5. Consider a queue $q = [m_1(d_1, c_1), \ldots, m_k(d_k, c_k)]$ with the set of clocks C_q. A 'first come first served' scheduler would always put the new job at the back of the queue. This scheduler would not impose any constraints:

$$sched\_FCFS(q, m(d)) = \{(true, c, [m_1(d_1, c_1), \ldots, m_k(d_k, c_k), m(d, c)])\}$$

where $c \in C_q$ is not assigned to any task in q.

A non-preemptive 'earliest deadline first' scheduler would insert tasks into the queue based on the remaining deadlines of the existing queue members:

$$\begin{aligned}
&sched\_EDF(q, m(d)) = \\
&\quad \{(d < d_2 - c_2, c, [m_1(d_1, c_1), m(d, c), m_2(d_2, c_2), \ldots, m_k(d_k, c_k)]), \\
&\quad (d_2 - c_2 \leq d < d_3 - c_3, c, [m_1(d_1, c_1), m_2(d_2, c_2), m(d, c), \ldots, m_k(d_k, c_k)]), \\
&\quad \ldots \\
&\quad (d_{k-1} - c_{k-1} \leq d < d_k - c_k, c, [m_1(d_1, c_1), m_2(d_2, c_2), \ldots, m(d, c), m_k(d_k, c_k)]), \\
&\quad (d_k - c_k \leq d, c, [m_1(d_1, c_1), m_2(d_2, c_2), \ldots, m_k(d_k, c_k), m(d, c)])\}
\end{aligned}$$

where $c \in C_q$ is not assigned to any task in q. The scheduler function cannot reorder the queue, so here we assume as an invariant that the tasks already in the queue are in the right order.

In our Uppaal implementation, we will use a timed automaton to act as both the queue and the scheduler function (cf. Section 3.3).

3.2.2 Timed Actor Model Semantics

An object is an instance of a class together with a specific scheduler and a queue. In a system, a number of objects run concurrently, each maintaining a queue of the tasks it has to perform. Without loss of generality, we assume that the sets of method names for different objects are disjoint.

Definition 3.2.6 (System). *Consider a set of objects $O_1, ..., O_n$ as instances of the classes $R_1, ..., R_n$, respectively. Each object O_i has its own queue and scheduler function $sched_i$. Assume that each class R_i implements the behavioral interface B_i. This set of objects is a system if for every class R_j, we have $Act^O_j \subseteq \big(M_{R_j} \cup \bigcup_{1 \le i \le n} M_{B_i}\big)$ where Act^O_j is the set of calls made by methods of R_j.* □

The semantics of a system is defined by a timed automaton, called the system automaton. In this automaton, we do not assume any upper bound on the queue size and therefore the system automaton may become infinite state. In order to make it finite state, however, we will show in Chapter 4 that we can put a finite bound on the queue length and obtain schedulability results that hold for any queue length.

Definition 3.2.7 (System Automaton). *The system automaton for a given system, as defined in Definition 3.2.6, with method names $\mathcal{M}_S = \bigcup_{1 \le i \le n} M_{R_i}$ is a timed automaton $S = (L_S, l_S, \longrightarrow_S, I_S)$ over the alphabet $\Sigma_S = \mathcal{M}_S$ and the clocks C_S:*

- *The set of clocks C_S is the union of all sets of clocks for the method automata plus the queue clocks of each object.*
- *The locations of the system are the product of each of the locations of objects together with a queue, i.e., $\{\ (l_1, q_1), (l_2, q_2), \dots, (l_n, q_n)\ \}$.*
- *The initial location l_S is:*

  $$\{\ (start(A_1), [m_1(d_1, c_1)]), \dots, (start(A_n), [m_n(d_n, c_n)])\ \}$$

  where m_i is the 'initial' method of the class R_i, A_i its timed automaton, d_i is the deadline for this initial method and c_i is one of the object's queue clocks.
- *The edges $\longrightarrow_S$ are defined with the rules in Figure 3.4. We write $\xrightarrow[g\ ;\ r]{m}_S$ for an edge of S with action m, guard g and update r.*
- *The invariant of a location is defined as the conjunction of the location invariants of all currently executing object locations.*
- *A location $\{(l, q), \dots\}$ is marked urgent if l is final and q has more than one element.*

□

$$\{(l,q),\ldots\} \xrightarrow[g\wedge G\ ;\ X, c_i=0]{m}_S \{(l',q'),\ldots\} \qquad [\mathit{invocation}]$$

$$\mathit{if}\ (l \xrightarrow[g\ ;\ X]{m(d)!} l)\ \&\ (G,c_i,q') \in \mathit{sched}(q,m(d))\ \&\ m \in M_l$$

$$\{(l,[m_1(d,c),\ldots]),\ldots\} \xrightarrow[g\wedge G\ ;\ X]{m}_S \{(l',q'),\ldots\} \qquad [\mathit{delegation}]$$

$$\mathit{if}\ (l \xrightarrow[g\ ;\ X]{m!} l')\ \&\ (G,c,q') \in \mathit{sched}([m_1(d,c),\ldots],m(d,c))\ \&\ m \in M_l$$

$$\{(l_1,q_1),(l_2,q_2)\ldots\} \xrightarrow[g\wedge G\ ;\ X, c_i=0]{m(d)}_S \{(l_1',q_1),(l_2,q_2'),\ldots\} \qquad [\mathit{remote\ invocation}]$$

$$\mathit{if}\ (l_1 \xrightarrow[g\ ;\ X]{m(d)!} l_1')\ \&\ (G,c_i,q_2') \in \mathit{sched}(q_2,m(d))\ \&\ m \in M_{l_2}$$

$$\{(l,[m_1(d',c'),m_2(d,c)\ldots]),\ldots\} \xrightarrow[C_l=0]{}_S \{(l',[m_2(d,c)\ldots])\ldots\} \qquad [\mathit{context\ switch}]$$

$$\mathit{if}\ l\ \mathit{is\ final}\ \ \&\ \ C_l = \mathit{local\_clocks}(m_2)\ \ \&\ \ l' = \mathit{start}(m_2)$$

$$\{(l,q),\ldots\} \xrightarrow[g\ ;\ X]{}_S \{(l',q),\ldots\}\ \ \mathit{if}\ \ (l \xrightarrow[g\ ;\ X]{} l') \qquad [\mathit{internal}]$$

Figure 3.4: Reductions for a System

In Figure 3.4, the following helper function and notations are used. Function $start(m)$ returns the initial location of the automaton for method m. Locations in method automata with no outgoing transitions are called *final*. Furthermore, M_l is the set of methods provided by the object that is in location l.

The first three rules in this figure take care of message passing including the enqueuing of the message by the receiver. The first two rules handle self calls and therefore the queue of the same object is used. In case of delegation, the clock of the current task is reused and thus the deadline is inherited. A remote method invocation results in an action with an observable deadline value. This deadline value is comparable (i.e., should be greater than or equal) to its corresponding deadline in the behavioral interface of the receiving object. Chapter 5 gives the details of compatibility check with respect to behavioral interfaces.

Whenever the execution of the current task finishes, the context switch rule makes sure the next method in the queue is executed, if there is any. When the last task in the queue of an object finishes, no context-switch takes place until a new task is added to the queue. If in a location $\{(l,q),\ldots\}$, l is final and q has more than one element, the location is marked urgent. This forces context switch to happen as soon as it is possible.

Finally, if an object can do an invisible action, it also appears in the system automaton as an invisible action.

3.3 Using UPPAAL for Modeling

In this section, we explain how to use UPPAAL [31] to model and simulate systems of actors. We model interfaces, methods implementations and schedulers (which in turn include a queue) with timed automata. The actor behavior can be obtained by making a network of these timed automata in UPPAAL. The details are explained with the help of the `MutEx` example. The automata for the interafce and methods of `MutEx` are given in Figures 3.5 and 3.6, respectively. These automata are comparable to the ones in Figures 3.2 and 3.3.

Communication We use two channels invoke and delegate for sending messages. The channel invoke has three dimensions (parameters), the message name, the sender and the receiver, e.g., invoke[**release**][self][Left]! replaces Left.**release** in the interface of MutEx. Notice that in interfaces, incoming messages are modeled as outputs and outgoing message are modeled as inputs; this is necessary for handling deadlines explained next. In method automata, by setting both sender and receiver to self, one can invoke a self call (when a deadline is to be given). The delegate channel is used for delegation. The self call made using the delegate channel inherits the deadline of the currently running task (it is taken care of by the scheduler automaton). Since a delegation is used only for self calls, no sender is specified (it has only two parameters). Figure 3.6 shows the interface and the method reqL of `MutEx` modeled in UPPAAL (cf. Figure 3.2).

Deadlines We take advantage of the fact that when two edges synchronize, UPPAAL performs the updates on the emitter (with ! sign) before the receiver (with ? sign). Hence we can use a global variable **deadline**. The emitter sets the deadline value into this variable which is read by the receiver. The receiver, however, cannot use this deadline value in its guard, as guards are evaluated before updates. To improve performance, we will define the **deadline** variable as *meta*. Meta veriables in UPPAAL are not stored in the state-space and are mainly used for passing a value when two transitions are synchronizing on a channel.

3.3.1 Modeling the Scheduler

A scheduler function (Definition 3.2.4) can be implemented as a scheduler automaton. This automaton also contains a queue as in Definition 4.2.3. Figure 3.7 shows the general structure of a scheduler automaton. This general picture does not specify any specific scheduling strategy. Unlike the Definition 3.2.4, the scheduler automata applies the scheduling strategy at dispatch time instead of insertion time, but the resulting behavior is the same. The reason is to enable using deadlines in the strategy. As explained in the previous subsection, the deadline value cannot be used (in the guard) on the same transition where a message is received.

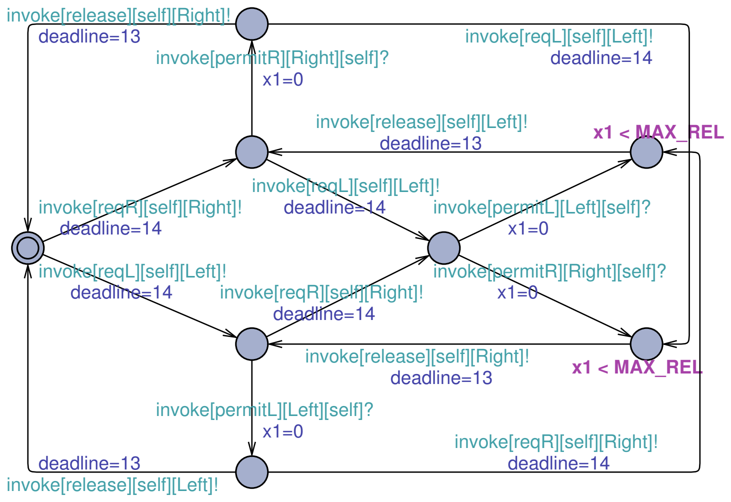

invoke[release][self][Right]!
deadline=13
invoke[permitR][Right][self]?
x1=0
invoke[reqL][self][Left]!
deadline=14
invoke[release][self][Left]!
deadline=13
x1 < MAX_REL
invoke[reqR][self][Right]!
deadline=14
invoke[reqL][self][Left]!
deadline=14
invoke[permitL][Left][self]?
x1=0
invoke[reqL][self][Left]!
deadline=14
invoke[reqR][self][Right]!
deadline=14
invoke[permitR][Right][self]?
x1=0
invoke[release][self][Right]!
deadline=13
x1 < MAX_REL
invoke[permitL][Left][self]?
x1=0
invoke[reqR][self][Right]!
deadline=14
deadline=13
invoke[release][self][Left]!

Figure 3.5: The behavioral interface for the `MutEx` object.

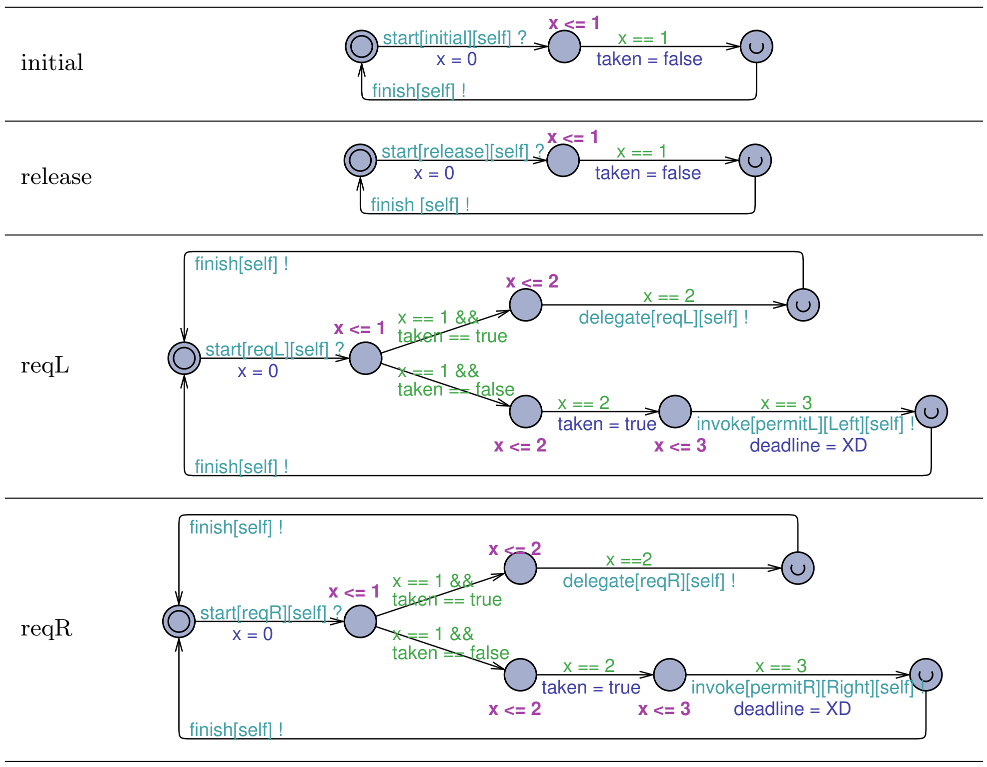


Figure 3.6: Modeling method and interface automata in UPPAAL

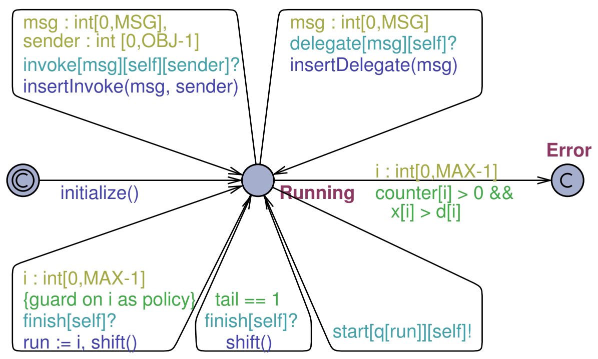


Figure 3.7: A general scheduler automaton

```
void initialize(){
    q[0] = op_init;
    d[0] = INIT_DEADLINE;
    ca[0] = 0;
    counter[0] = 0;    // clock x[0] is assigned
    tail = 1;
}
```

Figure 3.8: Initializing an object

Queue The triple $m(d, x)$ for each task in the queue is modeled using the arrays q, d and x, respectively. We assume a maximum length of MAX for the queue. We will show in Chapter 4 that we can find such a maximum for queues of schedulable objects. The array ca shows the clock assigned to each message (task), such that 'd[ca[i]] − x[ca[i]]' represents the remaining deadline of q[i] at any time. counter[i] holds the number of tasks using clock x[i]. A clock is free if its counter is zero. When delegation is used, the counter becomes greater than one.

Initializaton The initialization of a queue takes place in the initialize funciton. This transition is taken before any method in any actor is started, because its start location is committed. The function shown in Figure 3.8 puts the initilization method op_init in the queue and assign the first free clock to it.

Input-enabledness A scheduler for a class R should allow receiving any message in M_R at any time. In Figure 3.7, there is an edge (top-left in the picture) that allows receiving a message on the invoke channel (from any sender). To allow any message and sender, 'select' expressions are used. The expression msg : int [0, MSG] nondeterministically selects a value between 0 and MSG for msg. This is equivalent to

```
void insertDelegate(int msg){
   q[tail] = msg;
   ca[tail] = ca[run];
   counter[ca[tail]] ++;
   tail ++;
}

void insertInvoke(int m, int snd){
   int c, i;
   c = MAX;
   for (i = 0; c == MAX; i++) {
      if (counter[i][s] == 0) {
         c = i;
      }
   }
   q[tail] = m;
   ca[tail] = c;
   x[c] = 0;
   d[c] = deadline;
   counter[c] = 1;
   tail++;
}
```

Figure 3.9: Inserting a message into the queue

adding a transition for each value of msg. Similarly, any sender ($\text{sender} : \text{int}[0, \text{OBJ}-1]$) can be selected. This message is put at the tail of the queue ($\text{q[tail]} = \text{msg}$), and a free clock ($\text{counter[c]} == 0$) is assigned to it ($\text{ca[tail]} = \text{c}$), and the deadline value is recorded ($\text{d[c]} = \textbf{deadline}$); this is handled in the function `insertInvoke` shown in Figure 3.9. The synchronization between this transition and the method automata corresponds to the *invocation* rules in Figure 3.4.

A similar transition accepts messages on the delegate channel (top-right in the picture). In this case, the clock already assigned to the currently running task (parent task) is assigned to the internal task ($\text{ca[tail]} = \text{ca[run]}$); this is handled in the function `insertDelegate` shown in Figure 3.9. In a delegated task, no sender is specified (it is always self). The variable run shows the index of the currently running task in the queue (which is not necessarily the first task). This handles the rule *delegation*.

Scheduling Strategy When a message is added to an empty queue, it will be immediately executed. When a method is finished (synchronizing on finish channel), it is taken out of the queue (by shift() given in Figure 3.10). If it is not the last in the queue, the next method to be executed should be chosen based on a specific

```
void shift(int a) {
    counter[ca[a]] --;
    if (counter[ca[a]] == 0) d[ca[a]] = MD;
    tail--;
    while (a < tail && a < MAX-1) {
        q[a] = q[a+1];
        ca[a] = ca[a+1];
        a++;
    }
    q[a] = 0;
    ca[a] = 0;
}
```

Figure 3.10: Shifting the queue during context switch

scheduling strategy (by assigning the right value to run). For a *concrete* scheduler, the guard and update of run should be well defined. If run is always assigned 0 during context switch, the automaton serves as a First Come First Served (FCFS) scheduler. An Earliest Deadline First (EDF) scheduler can be encoded using a guard like:

```
i < tail && i != run &&
forall (m : int[0,MAX-1])
  (m == run) || (x[ca[i]] - x[ca[m]] >= d[ca[i]] - d[ca[m]])
```

and i will show the task with the smallest remaining deadline. Notice that $x[a] - x[m] \geq d[a] - d[m]$ is equivalent to $d[m] - x[m] \geq d[a] - x[a]$. The rest ensures that an empty queue cell (i < tail) or the currently finished method (run) is not selected.

A fixed priority scheduler can be implemented in a much similar way to an EDF scheduler explained above. As pointed out in Chapter 2, we do not deal with complex issues like priority inversion and inheritance in this thesis, which can form an interesting trend for future research.

If the currently running method is the last in the queue, nothing needs to be selected (i.e., if tail == 1 we only need to shift). The second step in context-switch is to start the method selected by run. Having defined start as an urgent channel, the next method is immediately scheduled (if queue is not empty).

Error The scheduler automaton moves to the Error state if a deadline is missed (x[i] > d[i]). The guard counter[i] > 0 checks whether the corresponding clock is currently in use, i.e., assigned to a message in the queue. Furthermore, to make sure no queue overflow occurs, the property to check should include $tail <= MAX$.

3.4 Thread Pools Extension[2]

In this section, we model a thread-pool using timed automata in Uppaal as an exntension of the actor framework presented in this chapter. We model a thread-pool as a scheduler automaton taking tasks from a queue and dispatching them among concurrent threads. This model can be seen as an extension of our actor framework to a situation in which objects share the message/task queue.

We separate the task queue in two parts: an *execution* part and a *buffer*. The execution part includes the tasks that are being executed. This part needs one slot for each thread and is therefore as big as the number of threads; we assume a fixed number of threads given a priori. Before beginning their execution, tasks are queued based on a given scheduling strategies, e.g., EDF, FPS, etc., in the rest of the queue (i.e., the buffer part).

In the rest of this section, we show two approaches in modeling concurrent threads sharing a task queue. At a higher level of abstraction, we can assume that the threads run in parallel as if each has its own processing unit. We can alternatively model a time-sharing scheduling policy where the 'executing' threads share the processor; therefore, each task runs a period of time before it is interrupted by the scheduler to run the next one. In both cases, when a task reaches the execution part, it will not be put back to the buffer part. We call this *weak non-preemption*, i.e., in the special case of one thread, it behaves like a non-preemptive scheduler. The scheduler (responsible for dispatching methods) and the queue (responsible for receiving messages) can be modeled in the same automaton or separately.

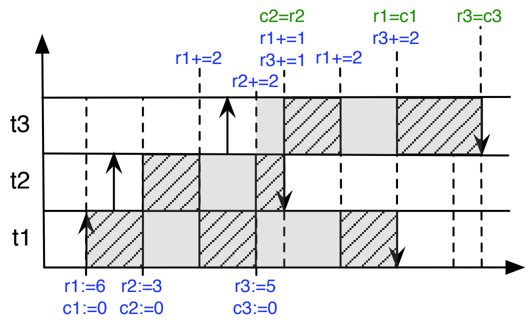
3.4.1 Time-Sharing

In this model, execution threads share one CPU. Therefore, the tasks in the execution part of the queue are interleaved. We call a thread active if a task is assigned to it. At its turn, each active thread gets a fixed time slot (called a *quantum*) for execution. If the assigned task does not finish within this quantum, the thread is preempted and the control is given to the next active thread. Recall that we use weak non-preemption, i.e., once a task is in the execution part it cannot be put back into the buffer part.

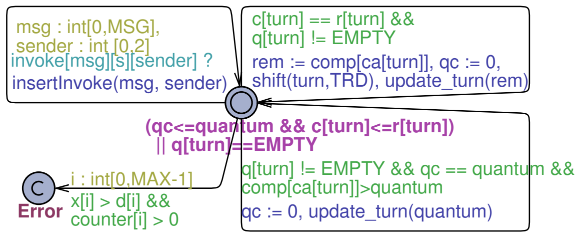
In this model, each task is modeled only as a computation time. This abstraction is necessary to enable the modeling of preemption of tasks at any arbitrary time (i.e., the selected quantum). Figure 3.11.(a) shows intuitively how three threads are scheduled. The up-arrows show when a task is released. A down-arrow indicates the completion of the task, after which the thread remains inactive in this scenario. The tasks assigned to `t1`, `t2` and `t3` have the computation times of 6, 3 and 5, respectively, and the preemption quantum is 2.

We associate to each thread a clock `c` and an integer variable `r` for response time, i.e., the execution plus idle time, which is updated dynamically while an active thread is idle. A task finishes when its clock reaches the expected response time value (`c=r` shown in green in Figure 3.11.(a)). When a task is assigned to a thread, only at the

[2]This section has been published in [14].



(a) Executing three tasks (quantum = 2)


msg : int[0,MSG],
sender : int [0,2]
invoke[msg][s][sender] ?
insertInvoke(msg, sender)
c[turn] == r[turn] &&
q[turn] != EMPTY
rem := comp[ca[turn]], qc := 0,
shift(turn,TRD), update_turn(rem)
(qc<=quantum && c[turn]<=r[turn])
|| q[turn]==EMPTY
C
Error
i : int[0,MAX-1]
x[i] > d[i] &&
counter[i] > 0
q[turn] != EMPTY && qc == quantum &&
comp[ca[turn]]>quantum
qc := 0, update_turn(quantum)

(b) Scheduler including a queue

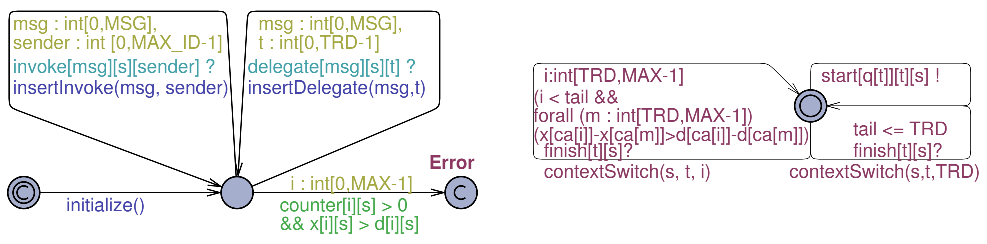
Figure 3.11: Modeling a time-sharing scheduler for a thread-pool

next quantum the thread becomes active, i.e., the thread clock is reset to zero and `r` is given the computation time of the task. In Figure 3.11.(a), the active period of the threads is shown in gray, during which the hatched pattern denotes when the thread has the CPU. At every context-switch (shown by dashed lines in Figure 3.11.(a)), the response-time variables of all idle threads are increased to reflect their recent idle time.

Figure 3.11.(b) shows the formal model of the time-sharing scheduler, which includes the message queue. The clock `qc` is used to keep track of time slots. The invariant on the initial location of the automaton ensures progress when a context-switch should occur and on the other hand it does not deadlock when the queue is empty (`q[turn]==EMPTY`). The edges on the right-side of the automaton model context-switch; in `update_turn` response-time variables are updated:

```
for (i = 0; i < TRD; i++) {
  if (q[i] != EMPTY) {
    if (i != turn) r[i] += quantum;
    else comp[ca[i]] -= quantum;  // remaining computation time
} }
```

The first check `q[i] != EMPTY` makes sure that the thread `i` is active, i.e., a task is assigned to it. The variable `turn` shows the thread that was just running. For threads



(a) A queue shared between threads (b) An EDF scheduler

Figure 3.12: Scheduler and queue separated for handling parallel threads in Uppaal

`i != turn` the response time `r[i]` is increased to cover their idle time, whereas for the thread that is just stopped we update `comp`, the remaining computation time. The value of `comp` is used when a task finishes before a quantum is reached, e.g., tasks `t2` and `t3` in Figure 3.11.(a). In this case, the response time of idle threads is increased by `comp` instead of `quantum`.

Finally the variable `turn` is updated at every context-switch, such that the next active thread is selected. If it happens that there are no more active threads, i.e., the last task just finished, `turn` will keep its old value, as modeled in the for loop below:

```
turn = (turn + 1) % TRD;
for (i=0; q[turn]==EMPTY && i<TRD-1; i++) {
  turn = (turn + 1) % TRD;
}
```

The edges on the left model insertion of tasks into the queue using the `insertInvoke` function, in which the scheduling policy can be modeled. In this model, the deadline or priority values for tasks can be modeled statically. Each queue slot is assigned a clock `x` which shows how long a task sits in that queue slot. The automaton also takes care that when a task misses its deadline (`x[i] > d[i]`), it goes to the `Error` state. The verification property should additionally check that the queue is not full.

3.4.2 Parallel Threads

In this model, every thread is assumed to have a dedicated processing unit, but they share one task queue. This model is more accurate when we can rely on the fact that the real system will run on a multi-core CPU and each thread will in fact run in parallel to the others. In this model, the queue and the scheduling strategy are modeled in separate automata. Figure 3.12.(a) shows a queue of size `MAX` which stores the tasks in the order of their arrival. This automaton is parameterized in `s` which holds the identity of the object. It accepts any message from any sender on the `invoke` channel, using the Uppaal 'select' statement on `msg` and `sender`. To check for deadlines, a clock `x` is assigned to each task in the queue, which is reset when the task is added, i.e., in `insertInvoke` function.

This model allows us to specify tasks as timed automata; therefore, tasks can create subtasks with self-calls. As a result, we don't need `c` and `r`. The `delegate` channel is dedicated to self calls that create subtasks inheriting the parent's deadline. To identify the parent, it receives the thread identity as `t`. Inheriting the deadline is modeled by reusing the clock `x` assigned to the parent task (which is in turn assigned to thread `t`). The number of tasks (and subtasks) assigned to clock `x[i]` is stored in `counter[i]`. This is handled in the `insertDelegate` function. The queue goes to `Error` state if a task misses its deadline (`x[i] > d[i]`) or the queue is full.

Figure 3.12.(b) shows how a scheduling strategy can be implemented. This automaton should be replicated for every thread, thus parameterized in `t` as well as the object identity `s`. The different instances of this automaton will be assigned each to one slot in the queue, namely `q[t]`. This example models an EDF (earliest deadline first) scheduling strategy. The remaining time to the deadline of a task at position `i` in the queue is obtained by `x[ca[i]]-d[ca[i]]`. When the thread `t` finishes its current task (`finish[t][s]`), it selects the next task from the buffer part of the queue for execution by putting it in `q[t]`; next, it is started (`start[q[t]][t][s]`).

3.5 A Peer-to-Peer Case Study[3]

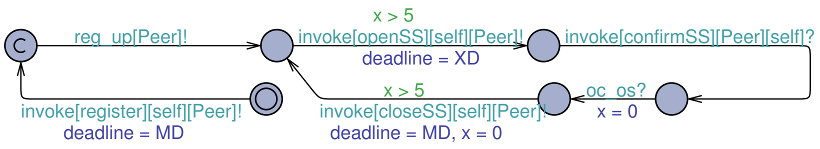
In this section, we explain how to model the real-time aspects of the peer-to-peer system using timed automata and the UPPAAL model checker [31]. We model the broker object in the peer-to-peer model, which is the most heavily loaded entity in this system. In the real-time model of an object, we add explicit schedulers to object specifications. As explained before, the model of an object consists of its behavioral interface and its methods automata.

3.5.1 Interface Inheritance

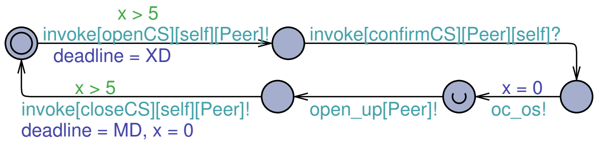
The first thing to model for an object is its behvaioral interface. The broker needs to handle server-related and client-related messages. It makes sense to define two interfaces for handling these messages separately, and then the broker interface inherits (and synchronizes) the behavior provided by these two interfaces. A server registers its data with the broker to initialize its operation. We opt for a simple scenario, i.e., each server or client handles only one request at a time. We also assume at this level of abstraction, that openCS is always successful, i.e., every data item searched for is available.

Figure 3.13 shows the behavioral interface of the broker; the Broker interface given in Figure 3.13(c) inherits the behavior specified for ClientSide and ServerSide interfaces. The Broker interface introduces some additional behavior, i.e., the possibility of doing an update to the data stored at the broker. The Broker automaton (see Figure 3.13(c)) synchronizes with the ServerSide automaton (see Figure 3.13(a)) to ensure that an update only takes place after the data is registered. Moreover, the data at the broker is

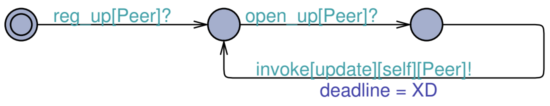
[3]This case study has been reported partly in [17].


C
req_up[Peer]!
x > 5
invoke[openSS][self][Peer]!
deadline = XD
invoke[confirmSS][Peer][self]?
x > 5
invoke[closeSS][self][Peer]!
deadline = MD, x = 0
oc_os?
x = 0
invoke[register][self][Peer]!
deadline = MD

(a) Server-Side interface handling openSS and closeSS


x > 5
invoke[openCS][self][Peer]!
deadline = XD
invoke[confirmCS][Peer][self]?
x > 5
invoke[closeCS][self][Peer]!
deadline = MD, x = 0
open_up[Peer]!
x = 0
oc_os!

(b) Client-side interface handling openCS and closeCS



(c) Broker interface introducing extra behavior (update)

Figure 3.13: The behavior of broker is the synchronization of the above automata

updated after receiving new information (on the ClientSide). This is modeled by synchronization on the channel open_up. In Figure 3.13, we use the open_up and reg_up channels to synchronize the automata for Broker with ClientSide and ServerSide, respectively. Additionally, the automata for ClientSide and ServerSide are synchronized on the oc_os channel; this abstractly models the synchronization on port communication between the components in which the broker is not directly involved. This model allows the client side of any peer to connect to the server side of any peer (abstracting from the details of matching the peers). The confirmCS and confirmSS messages model the confirmation sent back from the broker to the open session requests by the peers. These edges synchronize with the method implementations (explained next) in order to reduce the nondeterminism in the model. At the end of the next chapter, there are some general guidelines that help improve efficiency of model checking on Timed Automata models.

In general, the behavior of the sub-type has to be a refinement of the behavior of its super-type [41]. This is achieved by computing the product of the automata describing the inherited behavior (ServerSide and ClientSide) and the automaton synchronizing them (Broker).

Methods

The methods also use the invoke channel for sending messages. Figure 3.14 shows the automata implementation of two methods for handling the openCS and register events. In openCS, and similarly in every method, the keyword sender refers to the

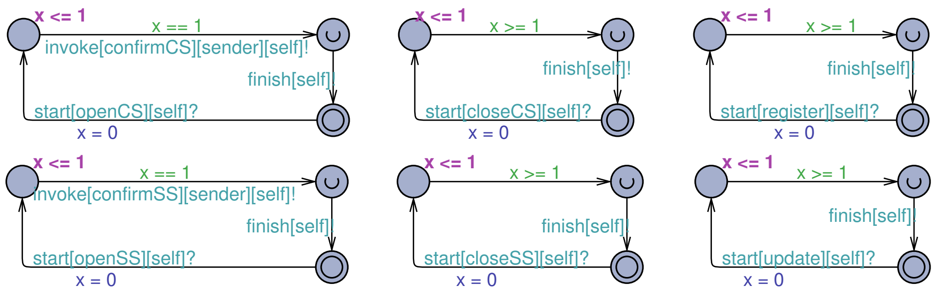


Figure 3.14: Method automata for broker

object/component that has sent the corresponding message. The scheduler should be able to start each method and be notified when the method finishes, so that it can start the next method. To this end, method automata start with a synchronization on the start channel, and finish with a transition synchronizing on the finish channel leading back to the initial location. The implementation of the methods for opening a client or a server session involve sending a *confirm* message back to the sender, while other methods are modeled merely as a time delay.

Chapter 4

Schedulability Analysis of Automata Models

A system with a single processor can execute one task at a time. When there are multiple tasks, the tasks need to be executed in turns. A schedule is one ordering of the tasks to be executed. In a hard real-time system, tasks have deadlines and must finish before their deadlines. A system with a given set of tasks is schedulable if there is an ordering for executing the tasks such that all tasks meet their deadlines.

In a nonterminating reactive system, tasks are dynamically generated. In an actor model, tasks correspond to methods which are triggered by receiving messages. The generation of tasks is not periodic; the best way to specify how tasks are generated is by means of automata. In fact the behavioral interface of an actor is the abstraction of the environments in which the actor can be used, because it specifies the allowed patterns of incoming and outgoing messages. Since a behavioral interface includes all allowed ways of calling the methods of the actor, we also call it a *driver*.

In this chapter, we define the problem of schedulability analysis for a system of actors modeled in timed automata (as described in Chapter 3). This problem is shown to be decidable by reducing it to reachability check of timed automata. For a big system, it may be practically infeasible to analyze its schedulability. Instead we show how to analyze an object in isolation with respect to its behavioral interface; its schedulability will be guaranteed if its real usage in a system is in accordance to its expected usage (see Figure 4.1). Our method makes it possible to put a finite bound on the process queue and still obtain schedulability results that hold for any queue length.

In the next chapter, we will explain in more details, how to check the compatibility of the usage of actors with their behavioral interfaces. This chapter begins with an overview of the existing automata theory for schedulability analysis, namely task automata. Then we show how can use similar ideas for schedulability analysis of actor models. The chapter is concluded with some case studies including the analysis of the thread-pool example.

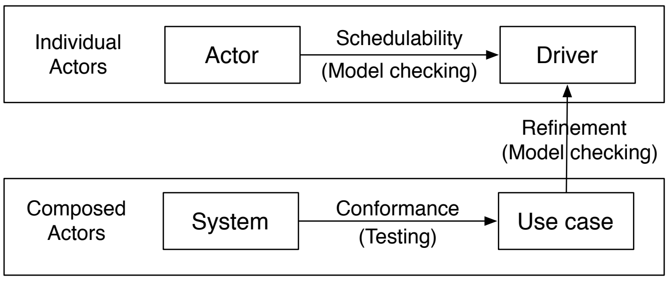


Figure 4.1: Modular approach to schedulability analysis

4.1 Background: Task Automata and Schedulability

The task automata model [16] is an extended version of timed automata [3] with asynchronous processes that are computation tasks generated (or triggered) by timed events. It can be used for specifying and analyzing real time systems with non-uniformly recurring tasks.

Definition 4.1.1 (Task Automata). *Considering a set of task types $\mathcal{P}$, a task automaton is a timed automaton extended with*

- *a partial function $M : L \hookrightarrow \mathcal{P}$ assigning task types to locations; and,*
- *a clock x_{done} which is reset whenever a task finishes.*

□

Task automata are used to specify the arrival pattern of tasks on a single processor computation unit. A task type, in this model, is an executable program represented by a triple (b, w, d), where b and w are, respectively, the best-case and worst-case execution times, and d is the deadline.

Compared with classical task models for real time systems, task automata may be used to describe tasks (1) that are generated non-deterministically according to timing constraints in timed automata, (2) that may have interval execution times representing the best case and the worst case execution times, and (3) whose completion times may influence the releases of task instances. In addition, tasks are triggered by events but cannot generate events themselves.

Intuitively, in a task automaton, a transition leading to a location in the automaton denotes an event triggering an instance of the annotated task and the guard (clock constraints) on the transition specifies the possible arrival times of the event. Semantically, an automaton may perform two types of transitions. Delay transitions correspond to the execution of a running task and idling for other tasks. Discrete

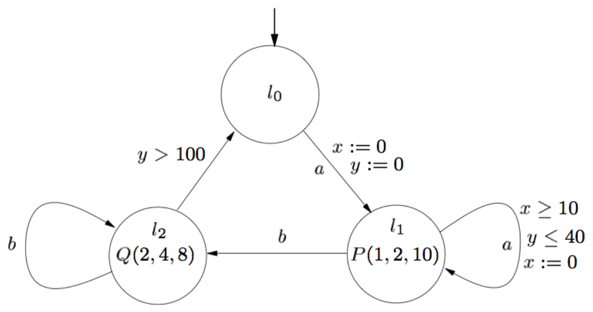


Figure 4.2: A task automaton [16]

transitions correspond to the arrival of new task instances. Whenever a task is triggered, it will be put into the queue for execution, according to a given scheduling strategy, e.g., FPS (fixed priority scheduling) or EDF (earliest deadline first).

Example 4.1.2. [16] Consider the automaton shown in Figure 4.2. It has three locations l_0, l_1, l_2, and two tasks P and Q (triggered by a and b) with interval computation times $[1, 2]$ and $[2, 4]$ (the best case and the worst case execution times), and relative deadlines 10 and 8, respectively. The automaton models a system starting in l_0 that may move to l_1 by event a at any time. This triggers the task P. In l_1, as long as the constraints $x \geq 10$ and $y \leq 40$ hold, when an event a occurs an instance of task P will be created and put into the scheduling queue. However, it cannot create more than 5 instances of P in l_1, because the constraint $y \leq 40$ will be violated after 40 time units. In fact, every instance will be computed before the next instance arrives and the scheduling queue may contain at most one task instance. Therefore, no task instance of P will miss its deadline. The system is also able to accept b, switch from l_1 to l_2, and trigger Q. Because there are no constraints labeled on the b-transition in l_2, it may accept any number of b's and create any number of Q's in zero time. However, after more than two copies of Q, the queue will be non-schedulable, i.e., a deadline may be violated. This means that the system is non-schedulable. Thus, zeno behaviors will correspond to non-schedulability, which is a natural property of the model.

Definition 4.1.3 (Task Automata Semantics). *The semantics of a task automaton $A = \langle L, l_0, \longrightarrow, I, M, x_{done}\rangle$, with respect to a given scheduling strategy* Sch*, is a labeled transition system $[\![A_{\mathrm{Sch}}]\!]$ with an initial state $(l_0, u_0, [])$ and transitions defined by the following rules:*

- $(l, u, q) \xrightarrow{a}_{\mathrm{Sch}} (l', u[r], \mathrm{Sch}(M(l'), q))$ *if* $l \xrightarrow[g\ ,\ r]{a} l'$, $u \models g$ *and* $u[r] \models I(l')$,
- $(l, u, []) \xrightarrow{t}_{\mathrm{Sch}} (l, u + t, [])$ *if* $t \in \mathbb{R}_{\geq 0}$ *and* $(u + t) \models I(l)$,

- $(l, u, P(b, w, d) :: q) \xrightarrow{t}_{\text{Sch}} (l, u + t, \text{Run}(P(b, w, d) :: q, t))$ *if* $t \in \mathbb{R}_{\geq 0}$, $t \leq w$ *and* $(u + t)$ *satisfies* $I(l)$*; and,*

- $(l, u, P(b, w, d) :: q) \xrightarrow{fin}_{\text{Sch}} (l, u[x_{done}], q)$ *if* $b \leq 0 \leq w$ *and* $u[x_{done}]$ *satisfies* $I(l)$*,*

where $P(b, w, d) :: q$ *denotes the queue with the task instance* $p(b, w, d)$ *inserted into* q *(at the first position),* $[]$ *denotes the empty queue, and* $fin \notin \Sigma$ *is a distinct action name.* □

Note that the transition rules are parameterized by Sch (scheduling strategy). Whenever it is understood from the context, we shall omit Sch from the transition relation. We have the same notion of reachability as for timed automata.

Definition 4.1.4 (Schedulability). *A task automaton is said to be schedulable if there exists a scheduling strategy such that all possible sequences of events generated by the automaton are schedulable in the sense that all associated tasks can be computed within their deadlines.* □

Theorem 4.1.5 ([16]). *The problem of schedulability for task automata with a non-preemptive scheduler is decidable.*

Fersman et al. [16] have studied the decidability of the problem of checking schedulability for task automata for different settings. The schedulability checking problem will be undecidable if the following three conditions hold: (1) the execution times of tasks are intervals, (2) the precise finishing time of a task instance may influence new task releases, and (3) a task is allowed to preempt another running task. For example, when using a non-preemptive scheduler, or when tasks have fixed computation times, schedulability for a task automaton can be reduced to a check for reachability in timed automata, and is therefore decidable.

4.2 Analyzing Actor Models[1]

In this section, we define the theory for analyzing actors modeled in the timed automata framework of Chapter 3. First, we define what schedulability means for a system of actors.

Definition 4.2.1 (Schedulable System). *A system of actors is schedulable if in its system automaton the deadlines of the tasks in the queue (including the currently executing task) never expire, i.e., there is no reachable state such that a clock of one of the tasks of the queue is greater than the deadline.* □

The definition of a system automaton given in Chapter 3 is infinite state. To be able to perform model checking, we need to make it finite. However artificially fixing

[1]This section is an improvement and extension of the results published in [24].

$$\{(l,q),\dots\} \longrightarrow_S \textit{Error} \quad \textit{if a queue length is longer than maximum} \qquad [\textit{overflow}]$$

$$\{(l,[m_1(d_1,c_1),\dots,m_k(d_k,c_k)]),\dots\} \xrightarrow[(c_1>d_1)\vee\cdots\vee(c_k>d_k)]{}_S \textit{Error} \qquad [\textit{missed deadline}]$$
$$\textit{if } l \textit{ is not final}$$

Figure 4.3: Reductions for Errors in a System

the queue may lead to false negatives. Luckily, for any given actor, we can determine an upper bound on the queue length of schedulable systems before constructing the system automaton. The lemma below gives a reasonable length for bounding the queue.

Lemma 4.2.2. *If a system automaton is schedulable then it does not put more than $\lceil d^i_{max}/b^i_{min}\rceil$ tasks into the queue of object O_i, where d^i_{max} is the longest deadline for any methods of O_i called on any transition of the method automata of all objects and b^i_{min} is the shortest termination time of any of the method automata of O_i.*

Proof. Assume that the queue of an actor reaches the length $m = \lceil d_{max}/b_{min}\rceil + 1$. We show that in this case, the actor is not schedulable. All methods are called with a deadline, and delegated deadlines are equal to, or less than, the original deadlines. Therefore, all tasks in the queue, including the last task, must have a deadline less than or equal to d_{max}.

The m tasks that at this moment exist in the queue need at least $m \times b_{min}$ to execute and terminate. If other tasks are inserted in the meantime in the queue, it may only increase this time until the original m tasks execute and terminate. Therefore, the final task terminates in time greater than $(\lceil d_{max}/b_{min}\rceil + 1) \times b_{min}$ which is strictly greater than d_{max} and so at least this task misses its deadline. □

We can calculate the best case runtime for timed automata as shown by Courcoubetis and Yannakakis [10]. The longest deadline can be found by a simple static search of all the transitions. Next, we redefine Definition 3.2.3 and put a bound on the length of the queue.

Definition 4.2.3 (Queue). *A queue q with an upper bound MAX is a list of at most MAX tasks together with a set C_q of MAX clocks. Each task is written as $m(d,c)$ where m is a method name, $d \in \mathbb{N}$ is its deadline and $c \in C_q$ keeps track of how long the task has been in the queue. The deadline of m expires when $c > d$.* □

The definition of system automaton (Definition 3.2.7, Chapter 3) is potentially infinite state and the information on each state may also be infinite as the queue could grow unboundedly. To make schedulability analysis of a system possible, we update this definition to use a bounded queue with sizes given by Lemma 4.2.2. We add to this definition a location *Error* such that a queue overflow or a missed deadline results in an error (see Figure 4.3). To deal with bounded queues, a scheduler function (cf. Definition 3.2.4) could be redefined to return the input queue unchanged when it is already full. Alternatively, we add conditions to each of the rules in Figure 3.4 to ensure that they are used only if the queue length is less than the maximum and no deadlines have been missed; thus the scheduler functions need not be redefined.

We show below that the system is schedulable if the error location is not reachable.

Theorem 4.2.4. *A system automaton is schedulable if, and only if, it cannot reach the error state when each object O_i has a queue with the length of $\lceil d^i_{max}/b^i_{min} \rceil$.*

Proof. A system automaton can reach the *error* state via the rules in Figure 4.3:

1) Lemma 4.2.2 implies that the [*queue overflow*] rule would be used if and only if the system automaton is not schedulable.

2) The guard of the [*missed deadline*] rule implies that a deadline has been missed and therefore the system automaton is not schedulable. This rule, however, excludes the cases that an actor is in its final location. In this situation, if there are other tasks in the queue, then this location is urgent and will be left in zero time and therefore no task will miss its deadline by waiting in this location. If there are no other tasks in the queue of this actor, then this actor is idle and staying in this location does not affect schedulability of this actor. In the other direction, if the system automata cannot reach the *error* state then the guard of the [*missed deadline*] rule must never hold, so no deadlines are missed. □

4.2.1 Analyzing One Actor in Isolation

Model checking a complete system for schedulability is usually not feasible due to the huge size of the state-space; this is because each actor has its own queue and actors run asynchronously in parallel. Analyzing one object per se is not possible either, because there are infinitely many ways to call its methods. To restrict how methods are called in an object, we use its behavioral interface as a *driver*; it includes all allowed ways of calling the methods of the object. The behavioral interface of an actor is the abstraction of the environments in which the actor can be used, because it specifies the allowed patterns of incoming and outgoing messages.

We showed in the previous subsection that we can put a finite bound on the queue length of actors in the context of a closed system. We reuse this lemma for the context of an open system, namely an object in the context of its behavioral interface. First we define the semantics of an object in the context of its behavioral interface.

Definition 4.2.5 (Behavior Automaton). *Suppose Q is the domain of all queues with upper bound MAX and using the clocks in C_q. Given a scheduler function sched, the behavior automaton of an actor $R = \{m_1 : A_1, \ldots, m_n : A_n\}$ with the interface D is a timed automaton $H = (L_H, l_H, \rightarrow_H, I_H)$ over the alphabet Σ_H and clocks C_H:*

- $\Sigma_H = \{m(d)! | m \notin M_R\} \cup \{m(d)? | m \in M_D\} \cup \{m | m \in M_R\}$, *where $d \in \mathbb{N}$ denotes a deadline.*

$$(l, l_d, [T_1, \ldots, T_k]) \xrightarrow[g \wedge G \ ; \ X, c_i := 0]{m(d)?}_H (l, l'_d, q') \qquad [\mathit{receive}]$$
$$\textit{if } (l_d \xrightarrow[g \ ; \ X]{m(d)?}_D l'_d) \ \ \& \ \ k \leq MAX \ \ \&$$
$$(G, c_i, q') \in \mathit{sched}([T_1, \ldots, T_k], m(d))$$

$$(l, l_d, [T_1, \ldots, T_k]) \xrightarrow[g \wedge G \ ; \ X, c_i := 0]{m}_H (l', l_d, q') \qquad [\mathit{self\ call : invocation}]$$
$$\textit{if } (l \xrightarrow[g \ ; \ X]{m(d)!}_{T_1} l') \ \ \& \ \ m \in M_R \ \ \& \ \ k \leq MAX \ \ \&$$
$$(G, c_i, q') \in \mathit{sched}([T_1, \ldots, T_k], m(d))$$

$$(l, l_d, [m_1(d, c), T_2, \ldots, T_k]) \xrightarrow[g \wedge G \ ; \ X]{m}_H (l', l_d, q') \qquad [\mathit{self\ call : delegation}]$$
$$\textit{if } (l \xrightarrow[g \ ; \ X]{m!}_{m_1} l') \ \ \& \ \ m \in M_R \ \ \& \ \ k \leq MAX \ \ \&$$
$$(G, c, q') \in \mathit{sched}([m_1(d, c), T_2, \ldots, T_k], m(d, c))$$

$$(l, l_d, [T_1, \ldots, T_k]) \xrightarrow[g \ ; \ X]{m(d)!}_H (l', l_d, [T_1, \ldots, T_k]) \qquad [\mathit{external\ call}]$$
$$\textit{if } (l \xrightarrow[g \ ; \ X]{m(d)!}_{T_1} l') \ \ \& \ \ m \notin M_R \ \ \& \ \ k \leq MAX$$

$$(l, l_d, [T_1, \ldots, T_k]) \xrightarrow[g \ ; \ X]{}_H (l', l_d, [T_1, \ldots, T_k]) \qquad [\mathit{internal\ step}]$$
$$\textit{if } (l \xrightarrow[g \ ; \ X]{}_{T_1} l') \ \ \& \ \ k \leq MAX$$

$$(l, l_d, [T_1, T_2, \ldots, T_k]) \xrightarrow[C_l := 0]{}_H (l', l_d, [T_2, \ldots, T_k]) \qquad [\mathit{context\ switch}]$$
$$\textit{if } l \in \mathit{final}(T_1) \ \ \& \ \ 2 \leq k \leq MAX \ \ \&$$
$$C_l = \mathit{local\_clocks}(T_2) \ \ \& \ \ l' = \mathit{start}(T_2)$$

$$(l, l_d, [T_1, \ldots, T_k]) \longrightarrow_H \mathit{Error} \qquad [\mathit{queue\ overflow}]$$
$$\textit{if } (k > MAX)$$

$$(l, l_d, [T_1, \ldots, m_i(d_i, c_i), \ldots]) \xrightarrow[(c_i > d_i)]{}_H \mathit{Error} \qquad [\mathit{missed\ deadline}]$$
$$\textit{if } l \notin \mathit{final}(T_1) \ \ \& \ \ 1 \leq i \leq k$$

Figure 4.4: Calculating the edges of the behavior automaton

- $C_H = C_D \cup \left(\bigcup_{i \in [1..n]} C_i\right) \cup C_q$, *where* C_i *and* C_D *are the clocks for* A_i *and* D*, respectively, and* C_q *is the set of queue clocks.*

- $L_H = \{error\} \cup \left((\bigcup_{i \in [1..n]} L_i) \times L_D \times Q\right)$*, where* L_D *and* L_i *are the sets of locations of* D *and* A_i*, respectively.*

- *The initial location* l_H *is* $(l_D, start(A_1), [m_1(d, c)])$*, where* l_D *is the initial location of* D*,* A_1 *is the automaton for the '*`initial`*' method (corresponding to* m_1*),* d *is the initial deadline and* $c \in C_q$*.*

- *The edges* $\rightarrow_H$ *are defined with the rules in Figure 4.4.*

- *The location invariants are defined as* $I(l, l_d, q) = I(l) \wedge I(l_d)$ *and* $I(error) = true$. *Furthermore, the locations* $(l, l_d, [T_1, \ldots, T_k])$ *such that* $l \in final(T_1)$ *and* $k \geq 2$ *are marked as urgent locations.*

□

In Figure 4.4, functions $start(A)$ and $final(A)$ give the initial location of A and the set of locations in A with no outgoing transitions, respectively. Each location of the behavior automaton is written as (l, l_d, q), where q is the queue, l_d is the current location of the interface and l is the current location of the method being executed, i.e., the automata corresponding to the first element of the queue. An edge of an automaton A with action a, guard g and update r is shown in this figure as $\xrightarrow[g\ ;\ r]{a}_A$. Finally, k shows the number of tasks in the queue.

When the interface allows receiving a message, or when a self call is made, the message is scheduled into the queue. In the case of delegation the clock of the currently running task is reused for the new task so that it inherits the remaining deadline. The *internal step* rule captures other transitions in a method.

When a task terminates the next task in the queue is started, unless there is no other task in the queue. In the latter case, context switch is not performed. Instead, this terminated task is exempted from the *missed deadline* rule. As soon as a new task is added to the queue, the context switch rule is enabled. Notice that the context switch rule is immediately executed when enabled, due to the urgency of the source location (cf. last bullet in Definition 4.2.5).

We build the behavior automata for an actor based on a given driver. We can then check if the actor is schedulable for the environments that match the driver. This compatibility check is explained in the next Chapter.

Definition 4.2.6 (Schedulable Object). *An object is schedulable in the context of its behavioral interface if in its behavior automaton the deadlines of the tasks in the queue (including the currently executing task) never expire, i.e., there is no reachable state such that a clock of one of the tasks of the queue is greater than the deadline.* □

Theorem 4.2.7. *A behavior automaton is schedulable if, and only if, it cannot reach the error state with a queue length of* $\lceil d_{max}/b_{min} \rceil$.

The proof of this theorem is similar to Theorem 4.2.4 concerning the schedulability of a system automaton. As a result of this theorem, we can check the schedulability of behavior automata by checking the reachability of the *error* state using the Uppaal model checker.

As explained in Chapter 3, to use Uppaal, we can make use of the automata specifying the scheduler. Such automata are finite given the queue length suggested by Lemma 4.2.2. Furthermore, in the scheduler automata, the *error* state is reachable only if a deadline is explicitly missed or if the queue is overfull; both cases imply non-schedulability. Therefore checking schedulability amounts to checking the reachability of the *error* state in the scheduler automata when it is put in a network of timed automata together with the method and driver automata. Since this reachability check is decidable, the problem of checking schedulability is also decidable.

4.2.2 Discussion on Preemptive Scheduling

We have focused our work on "non-preemptive" schedulers, i.e., schedulers that finish one task and then pick the next task to run using a given policy. Preemptive schedulers, on the other hand, will interrupt and switch between running tasks. It is exactly this switching that gives the effect of truly concurrent processes on a single CPU machine.

In the most general case, testing the schedulability of a preemptive scheduler is undecidable. Fersman et al. prove this for Task Automata [16] and their proof may be applied to our framework. In short they show that it is possible to implemmaent a 2-counter machine by encoding the counters using clocks in the interval $[0, 1]$. Their system repeatedly loops and the value of the counter is taken to be 2^{1-c} where c is the value of the clock at the end of each loop. Preemption allows the value c to be arbitrarily halved and doubled so that the counter may be incremented and decremented.

The work by Kloukinas et al. [29] uses discrete-time automata to handle preemption and proposes a methodology to cope with the state-space explosion due to it. In the dense model of time, however, the most general preemptive scheduler is undecidable because the amount of time between preemptions can be any value in the real domain.

If we are willing to restrict this interval and only allow preemption every t seconds then schedulability is decidable for every $t > 0$. This is enough to accurately model real systems; t could, for instance, be set to equal the target machine's clock speed. Schedulability is decidable in this framework because we can break up any of the method automata into a number of smaller automata each of which runs for t time units and then adds the next part of the method to the queue, so giving the scheduler the opportunity to preempt them. Unfortunately this would lead to the size of our system increasing exponentially as t decreases. We hope that a full investigation of preemption will make promising further work.

4.3 Case Studies

In this section, we explain how schedulability analysis can be used in practice. First, we analyze the `MutEx` implementation and show how the choice of scheduling policy can affect the schedulability results. It also shows the iterative method for applying schedulability analysis.

Then, we apply our technique to the industrial case study of thread pools. We first analyze the two models given in Chapter 3 with some predefined settings. Then we use an iterative approach (automated by a shell script) to analyze the model for parallel

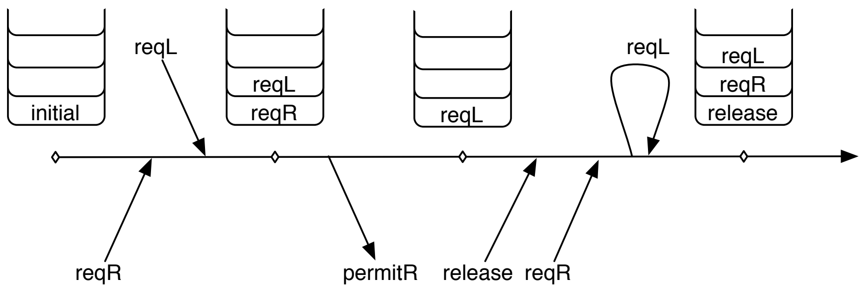


Figure 4.5: A possible trace for a fork based on FIFO scheduling policy

threads with different settings. Thus we can see the optimal number of threads given different inter-arrival and deadline values.

4.3.1 Schedulability Analysis of `MutEx`

To use Uppaal for analysis, we model behavioral interfaces, methods implementations and schedulers (which in turn include a queue) with timed automata as explained in Chapter 3. The actor behavior model (corresponding to behavior automaton) can be generated by making a network of these timed automata in Uppaal. Schedulability analysis boils down to checking the reachability of the error state in the scheduler automaton as long as the queue is not full.

With such an automated analysis process, it is easy to study the effect of different scheduling strategies on schedulability. Figure 4.5 shows a possible scenario in which 'First Come First Served (FCFS)' strategy for a `MutEx` may cause starvation, i.e., makes `MutEx` non-schedulable. This scenario is obtained by running a `MutEx` as controlled by its driver. The figure depicts the time line of a fork and its queue. The queue contents are shown only at context switch, i.e., when a method is finished and a new method is taken from queue head to start its execution (shown by a diamond on the time line). The same scenario generated by Uppaal is shown in Figure 4.6. This is part of the trace generated when checking for the reachability of the Error location.

At the end of this scenario, executing release and reqR would result in a 'permitR' to the right object for a second time, ignoring the request from the left one. This can continue infinitely. New instances of 'reqL' inherit the deadline associated to their parents, which shrinks continuously. After postponing 'reqL' for enough number of times, its deadline is missed, resulting in nonschedulability of `MutEx`. Using an Earliest Deadline First (EDF) strategy would favor old reqL to new reqR in this scenario. In addition, the EDF scheduler must give a higher priority to 'release' as opposed to 'request'.

The `MutEx` actor with the given driver needs a queue length of at least 5. Considering the driver, the reason is that 2 requests and 2 releases may be in the queue, while a delegated request can be added. Having chosen the proper scheduling strategy and

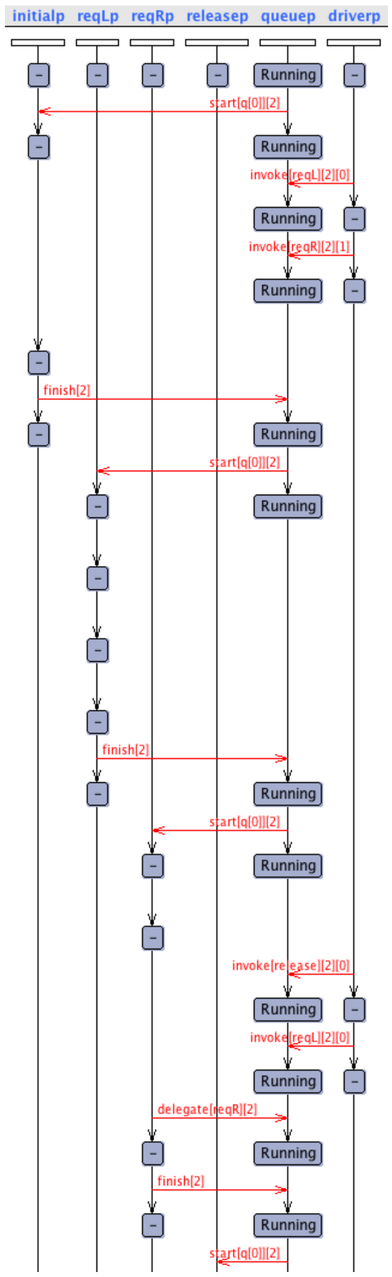


Figure 4.6: Starvation scenario for FCFS given by UPPAAL. The parameters self, Left and Right are instantiated to 2, 0 and 1, respectively. The arrows labeled invoke and delegate show enqueuing of messages. The arrows labeled start and finish determine the execution period of methods.

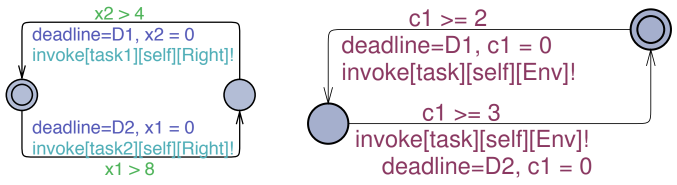


Figure 4.7: Generating task instances sequentially (left) or in parallel (right)

queue length, we can repeat the scheduling analysis (checking for reachability of Error location) to find the best values for MIN_REL and MAX_REL and the deadline values for request and release.

4.3.2 Schedulability Analysis of Thread Pools[2]

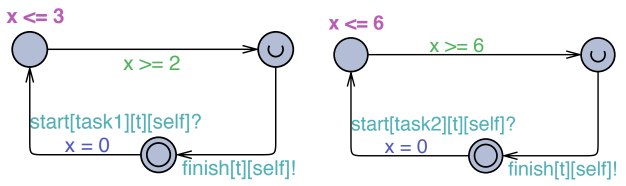
In this section, we analyze the schedulability of the timed automata models of thread pools given in the previous section. The model of a thread pool is not enough for schedulability analysis, because we need to know how fast tasks are generated and what their deadlines are. The timed automaton specifying the task generation patterns serves as the behavioral interface of the thread pool. The behavioral interface can be seen as a model of the environment and captures the work-load of the ASK system. In this section, we assume two threads.

Figure 4.7 shows two models of behavioral interfaces for our model of thread pools. In these diagrams, `Right` shows the identity of an object in the environment that sends messages task1 and task2 to the thread pool under analysis; the identity of the thread pool is given as `self`. In one model, tasks are generated independently with an inter-arrival time of at least 9 time units between every two occurrences of the instances of the same task type. In the other model, tasks are generated one after the other in a sequential manner.

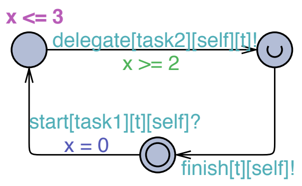
To perform schedulability analysis by model checking, we need to find a reasonable queue length to make the model finite. The execution part of the queue is as big as the number of threads, and the buffer part is at least of size one. As in single-threaded situation of objects [24], a system is schedulable only if it does not put more than $\lceil D_{max}/B_{min} \rceil$ messages in its queue, where D_{max} is the biggest deadline in the system, and B_{min} is the best-case execution time of the shortest task. As a result, schedulability is equivalent to the `Error` state not being reachable with a queue of length $\lceil D_{max}/B_{min} \rceil$. Therefore, schedulability analysis does not depend on whether an upper bound on queue length is assumed or not.

To use the time-sharing model of a thread pool, a task is modeled as a computation time. The two task types are given the computation times of 3 and 6 time units. This model is analyzed with a queue length of three where two concurrent threads are

[2]This case study has been reported partly in [14].



(a) Modeling tasks as computation times



(b) Generating a subtask

Figure 4.8: Modeling tasks for parallel threads

assumed. Given the behavioral interface with parallel task generation, the minimum deadline for which the model is schedulable is 7 and 9 for task1 and task2, respectively. For sequential task generation, the deadlines can be reduced to 5 and 6 for task1 and task2, respectively.

When the thread pool for parallel threads is applied, one can model tasks as timed automata; two simple task models are given in Figure 4.8.(a). In this model, task1 has a computation time of between 2 to 3 time units, and task2 takes 6 time units to execute. Using either of the two behavioral interfaces above, the model is schedulable with the deadlines 3 and 6 for task1 and task2, respectively. In parallel generation of tasks, parallel threads can handle the tasks faster, and therefore, smaller deadlines are needed for schedulability. It turns out that the parallelism of the threads does not affect the schedulability of tasks that are created sequentially.

More complicated models can include sub-task generation. Figure 4.8.(b) shows a model of task1 which creates an instance of task2. The schedulability analysis of this model with a queue length of three fails due to queue overflow. This implies that a fixed-sized thread pool with a too small buffer size can fail. By increasing the size of the queue to four, the model will be schedulable given a deadline of 9 to the task1 (and task2 still needs a deadline of 6). This shows that a fixed-sized thread pool can still be useful if a big enough buffer size is used.

Number of Threads

For the schedulability analysis of the ASK system, we modeled the *determinate* variant of the ASK thread pools, called dabbey. This variant has a fixed number of threads and a fixed-size task buffer. The details of the UPPAAL models corresponding to the dabbey can be found in Section 3.4.2. The dabbey is modeled in a flexible way such that by adjusting the following parameters, one can experiment with different settings of the thread pool:

- The first thing one can change is the number of threads for the modeled thread pool. By increasing the number of parallel threads, one can perform tasks faster. This means that smaller deadlines and inter-arrival times can be used.

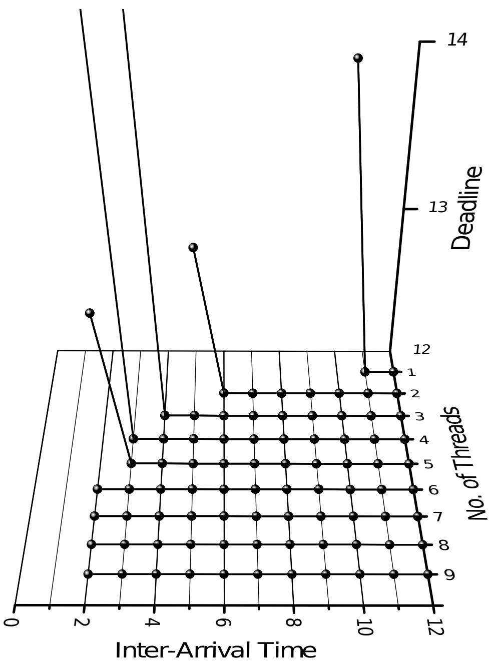


Figure 4.9: Required number of threads to deal with tasks with given deadlines and inter-arrival times

- In order to perform experiments, the deadline and inter-arrival values can be changed in order to get the right value for schedulability.
- Furthermore, one can change the number of different task types.

In order to compare the schedulability of tasks for different numbers of threads and different inter-arrival times, we created a script to automatically invoke the schedulability analysis for different parameter values using the UPPAAL command-line verifier, called `verifyta`. The output of the script can be plotted in a 3D plane. An example is given in Figure 4.9. In this example, the number of different tasks is 9, while their computation times are defined as $\{8, 9, 9, 10, 10, 10, 11, 11, 12\}$. On the Z-axis of the plot, the minimum possible deadline for the tasks to be schedulable can be read, for different numbers of available threads in the thread pool and different inter-arrival times. Several conclusions can be drawn from this diagram, like:

- For the given set of task types, thread pools with sizes 6, 7, 8 and 9 perform equally well.
- For the given set of task types, if we allow less strict deadlines for the tasks,

the minimum inter arrival times for thread pools with sizes 1, 2 or 5 can be improved. But inter-arrival times of less than 2 are not possible to handle.

- For the given set of task types, a thread pool with 2 threads (minimum inter-arrival time 5, corresponding minimum deadline 13) performs more than twice as good as a thread pool with 1 thread (minimum inter-arrival time 10, corresponding minimum deadline 14).

Such interesting results can be applied by Almende in the near future to improve the performance of the thread pools in the ASK system.

4.4 Guidelines for Efficiency

The number of peers in the system is an important factor for the schedulability. However, their number is limited by the capabilities of model checking technologies. Size of the waiting queue of the scheduler and execution times and deadlines for all methods are important for schedulability analysis, too. Since they have less effect on the feasibility of verification, they are a good starting point for the analysis, e.g., one might try to adjust the waiting queue size to check if a smaller size would make a schedulable system non-schedulable.

A minor hint: Set the search order for the verification to *Depth First* (*Options → Search Order → Depth First*). For this model, it is more likely to find counter examples in the depth of the state space, so this setting will lead to faster termination for systems, which are not schedulable.

Model Checking of the described system with two peers, seven methods, and first-come-first-served scheduling might already take several hours of CPU-time. With an additional peer, it would be infeasible in most cases. Usually abstraction makes verification process much more efficient on one hand. On the other hand it makes models harder to understand. Keep in mind that this document focuses on an illustrative presentation of the usage of the *Credo* technology, therefore efforts on abstraction of the peer-to-peer model have been kept on a basic level.

In general, there are some possibilities to gain efficiency on this level. Among these are:

- *Reduction of the number of clocks:* Unrestricted use of clocks has an exponential impact on the size of the state space. One should take care not to introduce new clocks unless they are absolutely necessary.

- *Reduction of the maximal constants used in guards:* This will reduce the exponential growth of the state space caused by the number of clocks.

- *Reduction of non-determinism:* This is for instance caused by guards, allowing a transition in a given interval (e.g. `x >= 5 and x <= 10`). The interplay between different non-deterministic guards can cause a hardly predictable partitioning of the state space. This phenomenon can be observed, e.g., if the method automata are changed to support both, worst- and best-case execution times.

- *Reduction of variables, their ranges, and their possible valuations:* Less variables mean a smaller size of a single state and less possible different valuations mean fewer distinct states.

Chapter 5

Checking Compatibility

In previous chapters, we described how to use timed automata for actor-based modeling of systems and applied model checking to schedulability analysis of a complete closed system. Checking the whole system for schedulability is subject to state space explosion, given the fact that each actor has its own processor and queue. When model checking a complete system is not feasible, we explained how to use the behavioral interface of an actor as a driver to simulate the actor's environment.

Individually schedulable actors can be used as off-the-shelf components in making bigger components and systems. One can develop in a modular manner actors that are schedulable when their actual use is compatible with their drivers. In this section, we briefly discuss one way to check for such a compatibility in the context of the 'design by contract' approach [36] based on automata [11].

In this approach, the behavioral interface of an actor is used as a driver specification which describes the most general conditions under which instances of the model are schedulable. A system developer (as opposed to the actor developer) instantiates actor models in the context of a particular configuration of connections. Intuitively, a running system of objects is compatible with their behavioral interfaces if its observable behavior is captured by the composition of the behavioral interfaces of the participating objects.

A naive check of deadlock freedom in the composition of the behavioral interfaces does not imply anything about compatibility. On one hand, behavioral interfaces are abstractions of object behaviors, therefore, any mismatch (i.e., deadlock) could be due to a spurious behavior not possible in the real system model. On the other hand, a lack of deadlock does not imply compatibility. The reason is that compatibility means that a 'send' by an object takes place at such a time that satisfies the guard in the behavioral interface of the receiver object. However, a behavioral interface has no timing constraint on its output (send) actions (cf. Definition 3.2.1) and therefore does not reflect the precise timing of the send action by the real system model.

In this chapter, we first establish a testing framework for checking refinement of timed automata. Next, we will apply this framework to testing compatibility which will be defined in terms of refinement. We will finally show that if compatibility holds for a system with individually schedulable objects, the whole system is schedulable.

5.1 Testing Refinement of Timed Automata

We assume we are given two timed automata, modeling the specification and the implementation of a real-time system. The former is a deterministic timed automaton *Spec* over the set of actions Σ_{Spec}. The implementation, also called the *model under test*, is a timed automaton MUT over the set of actions $\Sigma_{MUT} = \Sigma_{Spec} \cup \{\tau\}$. In this section, we propose a theoretical framework for testing whether MUT is a refinement of *Spec*. We define refinement as the inclusion of the observable behaviors. We give a test case generation algorithm and prove that the set of all test cases that could be generated by the algorithm is sound and complete for our notion of refinement.

Trace inclusion is a usual notion of correctness (or conformance) between a system and its specification in formal testing frameworks [19]. We use the standard notions and methods from these frameworks. However, our testing framework gives rise to some issues which are not common in standard frameworks. First, the implementation under test is a model and not a real system. We do not take advantage of our knowledge of the model, so it is still black-box testing, but the execution of test cases will be simplified. We will explain later that it comes down to model-checking for the reachability of a **Fail** location.

In a naive approach, one would take a trace from the MUT and check whether it exists in the *Spec*. In our case, since MUT is very big (it includes the method automata and object queues), we generate a test-case from *Spec*. The consequence is that the test, built from the specfication only, will not be able to fully control the system under test during its execution. This leads to a lot of non-determinism but we alleviate this by using a model checker to execute a test case.

Finally, our main goal is to find a counter-example in the case of wrong refinement. Then test cases must be as “complete” as possible to take any incorrect behavior into account, which is a novel issue in testing.

Refinement of Timed Automata[1]

In the context of timed automata, an observable behavior is either an observable action (any action except τ), the passage of time (a delay) or a lock (also called quiescence, i.e., the absence of observable actions). Actions and delays are taken into account in timed traces, which describe the possible distribution of observable actions in time. To be able to take locks into account, we need to represent them explicitly in the specification. The different kinds of locks that can occur in a timed system are (see Figure 5.1):

Deadlock No action is possible but time can go on.

Timelock Time is stopped, no action and no delay is possible (in Figure 5.1 the synchronization cannot happen due to the mismatching invariant and guard).

Zeno-timelock Infinitely many actions can occur in finite time.

[1]This chapter is an extension of the results published in [25].

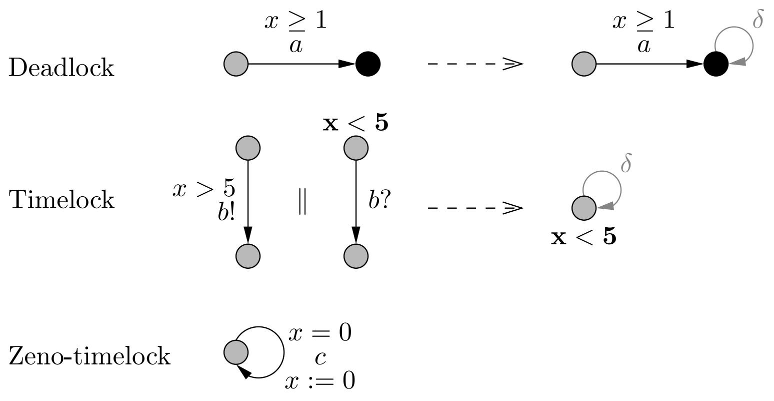


Figure 5.1: Construction of the suspense automaton

We allow deadlocks and timelocks but no Zeno-timelocks. For a correct refinement, the *MUT* may deadlock (resp. timelock) only if the specification deadlocks (resp. timelocks).[2] Locations with a deadlock or timelock are called *quiescent*. To explicitly specify quiescence in the specification, we add a loop on each blocking location labeled by a new action δ, considered to be an observable output action [44]. The automaton obtained by adding δ actions is called a *suspense automaton*.[3]

Definition 5.1.1 (Suspense automaton). *Let $A = (L, l_0, \longrightarrow, I)$ be a timed automaton over clocks C and actions Σ. The* suspense automaton *of A is the timed automaton $\Delta(A) = (L, l_0, \longrightarrow_\Delta, I)$ over C and $Act \cup \{\delta\}$ where*

$$\longrightarrow_\Delta \;=\; \longrightarrow \;\cup\; \{l \xrightarrow{\delta} l \mid l \textit{ is quiescent}\}$$

defines the transition relation. □

The traces of the suspense automaton, called suspension traces, represent all the observable behaviors of this automaton. The set of all suspension traces of a timed automaton A is the set *Traces*$(\Delta(A))$ denoted by *STraces*(A).

Definition 5.1.2 (Refinement). *MUT is said to be a* refinement *of Spec, denoted by $MUT \sqsubseteq Spec$, if and only if $STraces_{obs}(MUT) \subseteq STraces(Spec)$.* □

Since *Spec* is deterministic, checking trace inclusion becomes decidable [3, 42], but due to the size of *MUT*, it may be susceptible to state-space explosion. To avoid this, we propose a method for *testing* trace inclusion. In particular, we want to be able to exhibit a counter-example if some incompatibility is found.

[2]In the context of compatibility checking, the specification is obatined by synchronous product of the behavioral interfaces of objects, which may in turn have a deadlock or timelock.

[3]We do not call it a suspension automaton as Tretmans does [44] in order to avoid confusion with a decidable class of hybrid automata also called suspension automata [34].

5.1.1 Test Cases

We build a test case given a sequence of transitions from the specification, representing a set of traces. Such a sequence abstractly represents a desired system behavior. A test case is a deterministic timed automaton without loops whose leaves are labeled with verdicts.

Definition 5.1.3 (Test case). *Let Spec be a timed automaton over Σ_{Spec}. A* test case *for Spec is a deterministic acyclic timed automaton $TC = (L, l_0, E, I)$ over $\Sigma_{Spec} \cup \{\tau\}$. Each leaf location (i.e., locations with no outgoing transition) must be labeled with a verdict* **Pass**, **Fail** *or* **Inconc**. *Every non-leaf location l has the following properties:*

- *for all actions $a \in \Sigma_{Spec}$, there exists a transition with action a from l*
- *there exists at most one τ transition from l*
- *for all actions $a \in \Sigma_{Spec}$, for all transitions $l \xrightarrow{g_i,a,r_i} l'_i$, $1 \le i \le k-1$ and $l \xrightarrow{g_k,\tau,r_k} l'_k$ the guards are complementary: $\bigvee_{1\le i\le k} g_i$ is always true.*

We refer to a set of test cases as a test set. □

To accommodate for deadlocks and timelocks, a test case should be generated from the suspense automaton of the specification. A verdict labeling a location allows us to evaluate an execution of the test case terminating on this location. Locations labeled by **Fail** are those which are reachable with forbidden behaviors of the system (a non-specified action or an action happening outside its time constraints in the specification, for example). An execution of a test case ending on a **Pass** location means that the system fulfilled the test case requirements. When an inconclusive location **Inconc** is reached, it means that the system behaved correctly but not according to the behavior aimed by the test case. To find a counter-example to refinement, we need to search for locations marked **Fail**.

The properties of non-leaf locations ensure in particular that in each of these locations, any action is possible at any time, and only one transition can be taken given an action and a valuation of the clocks. These properties are necessary to be able to provide relevant counter-examples.

Generating a test case

The algorithm in Figure 5.3 takes as input a linear timed automaton T obtained from the specification automaton *Spec* and builds a test case *TC* as output. The input automaton T contains a sequence of transitions written as $l_{i-1} \xrightarrow{g_i,m_i,r_i} l_i$. These transitions can be either exact transitions or 'subtransitions' of $\Delta(Spec)$. We call $l_{i-1} \xrightarrow{g_i,m_i,r_i} l_i$ a subtransition, if there exists a transition $l_{i-1} \xrightarrow{g,m_i,r_i} l_i$ in $\Delta(Spec)$ such that $g \Rightarrow g_i$. Since the guards g_i may specify intervals, T represents a set of suspension traces of the specification.

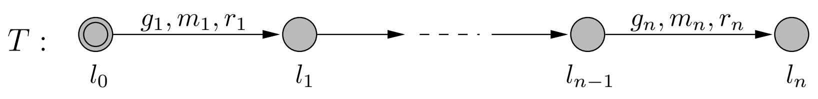


Figure 5.2: A linear timed automaton T as input to the algorithm below.

Input	A timed automaton *Spec* (may be a suspense automaton); and, A linear timed automaton T (as shown above).
Output	A timed automaton TC (the test case).

Locations:
$L(TC) = L(T) \cup \{c, f\}$

Verdicts:

Label l_n	with the verdict	**Pass**
Label c	with the verdict	**Inconc**
Label f	with the verdict	**Fail**

Transitions:

```
for every $0 \leq i < n$ repeat the following
  let $h_i$ be the invariant of $l_i$ in Spec
  add $l_i \xrightarrow{\neg h, \tau, \emptyset} f$
  add $l_i \xrightarrow{g_i \wedge h_i, m_i, r_i} l_{i+1}$
  for each action $m \in \Sigma_{Spec}$ do
    let $g_f = false$
    for each transition $l_i \xrightarrow{g,m,r} l'$ in Spec do
      let $g_f = g_f \vee g$
      if $m \neq m_i$ then
        add $l_i \xrightarrow{g \wedge h_i, m, r} c$
      else
        add $l_i \xrightarrow{g \wedge h_i \wedge \neg g_i, m, r} c$
      endif
    endfor
    add $l_i \xrightarrow{h_i \wedge \neg g_f, m, \emptyset} f$
  endfor
endfor
```

Figure 5.3: Test case generation algorithm

The sequence of transitions of T corresponds to the behavior we want to test so the last location must be labeled **Pass**. If a location has an invariant h in *Spec*, violating this invariant must make the test fail; thus, a transition labeled with τ and with guard $\neg h$ leading to **Fail** is added. Furthermore, no other transition may be taken if the invariant is violated; this is ensured by conjunction of guards of all other transitions with h.

We complete the initial linear timed automaton with allowed and forbidden behaviors. Every behavior which is not allowed in *Spec* is forbidden, so for every action, a transition labeled by this action and whose guard is the complement of all the existing guards for this action leads to a **Fail** location; this guard is computed in g_f. Any trace leading to **Fail** is an example of behavior not allowed in the specification.

Finally, every divergence from the sequence of transitions which is allowed in *Spec* leads to the **Inconc** location. Such a divergence can be either an action other than the one labeling the main transition, or the same action at a different time. In the latter case, the test is inconclusive only if the action is not taken at the right time, implied by the guard $g \wedge \neg g_i$.

Remark. The starting point for test case generation is a sequence of transitions in the specification. Given a desired reachability property φ, we can generate such a sequence of transitions automatically. We start by model-checking φ on the suspense automaton of the specification *Spec*. The diagnostic trace produced by the model-checking tool gives the sequence of moves that have to be made by this automaton and the required clock constraints needed to reach the targeted location. This method is similar to [19]. Instead of checking for a reachability property, one can also use the simulation feature of a model-checker to generate specific hand-made traces.

5.1.2 Properties of the Test Cases

Intuitively, a test case must drive the execution of the system such that actions happen in the specified order. Therefore, the execution of a test case on the system is defined as the parallel composition of the automata of the test case and the system, synchronizing on the same actions. We denote the product automaton by $TC \| MUT$. The model under test *passes* the test, denoted by MUT passes TC, if and only if the **Fail** location is not reachable in the product $TC \| MUT$. A test set $\mathcal{T}$ being a set of test cases, the model under test passes $\mathcal{T}$, denoted by MUT passes $\mathcal{T}$, if and only if for all test cases TC in $\mathcal{T}$, MUT passes TC.

Soundness The soundness requirement for a test set states that it must not reject a correct refinement. In other words, any correct refinement must pass the test set.

Definition 5.1.4 (Sound). *A test case is* sound *(or unbiased) for the refinement relation $\sqsubseteq$ if and only if*

$$MUT \sqsubseteq Spec \implies MUT \text{ passes } TC$$

A test set $\mathcal{T}$ is sound if and only if all test cases in $\mathcal{T}$ are sound. □

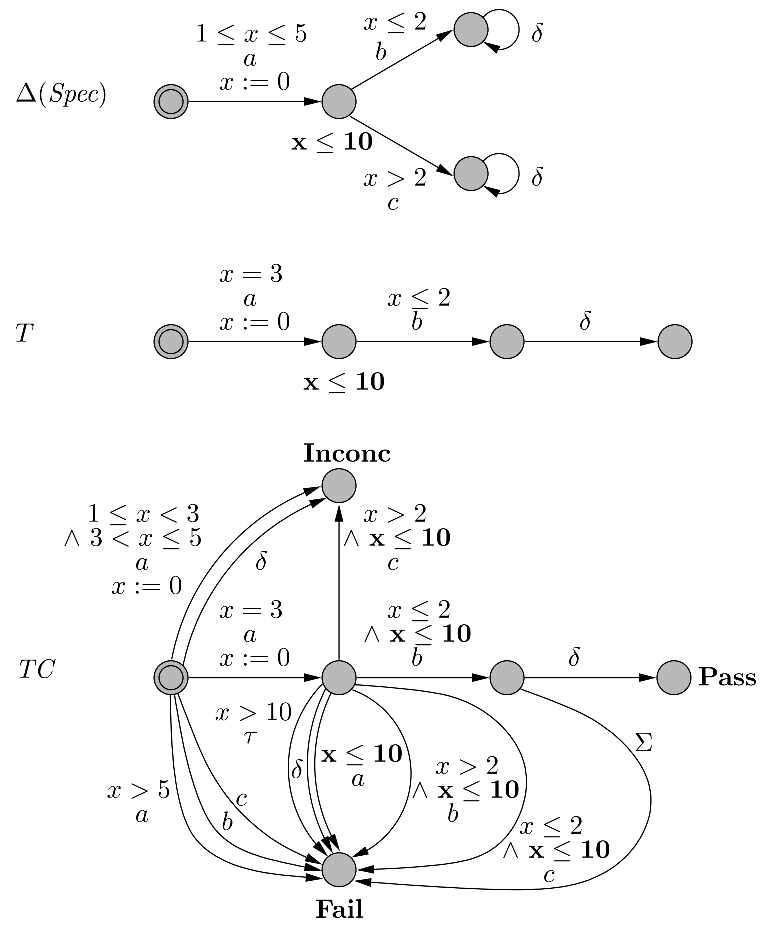


Figure 5.4: The suspense automaton of a specification *Spec*, a selected sequence of transitions T from $\Delta(\mathit{Spec})$ and the test case TC generated from T by the algorithm. The invariant is kept in bold face in the test case only to show its effect on guards. The transition labeled by Σ stands for three transitions labeled by a, b and c with guard *true*.

Theorem 5.1.5 (Soundness). *Let Spec be a deterministic timed automaton and T be a linear timed automaton built from a sequence of transitions in Spec. The test case TC generated from T and Spec by the algorithm in Figure 5.3 is* sound *for $\sqsubseteq$.*

Proof. We assume that MUT does not pass TC, i.e., there is a trace in $TC \,\|\, MUT$ that leads to the fail location $l_f = (l, f)$. This trace can be decomposed into its MUT and TC components. Every location in TC, except for c and f, can be mapped to a location in *Spec*; since *Spec* is deterministic, this mapping is unique. The diagram below, shows the decomposition of this trace and the mapping to *Spec*.

$$\begin{tikzcd}
Spec & (l_0^S, \mathbf{0}^S) \arrow[r, "{d_0,a_0}"] & \cdots \arrow[r] & (l_i^S, u_i^S) \arrow[r, "{d_i,a_i}", "/" marking] & {} \\
TC & (l_0^T, \mathbf{0}^T) \arrow[u, Rightarrow] \arrow[r, "{d_0,a_0}"] & \cdots \arrow[r] & (l_i^T, u_i^T) \arrow[u, Rightarrow] \arrow[r, "{d_i,a_i}"] & (f, u_f^T) \arrow[u, Rightarrow, "/" marking] \\
TC \,\|\, MUT & (l_0^T, l_0^M, \mathbf{0}) \arrow[u, Rightarrow] \arrow[d, Rightarrow] \arrow[r, "{d_0,a_0}"] & \cdots \arrow[r] & (l_i^T, l_i^M, u_i) \arrow[u, Rightarrow] \arrow[d, Rightarrow] \arrow[r, "{d_i,a_i}"] & (l_f^M, f, u_f) \arrow[u, Rightarrow] \arrow[d, Rightarrow] \\
MUT & (l_0^M, \mathbf{0}^M) \arrow[r, "{d_0,a_0}"] & \cdots \arrow[r] & (l_i^M, u_i^M) \arrow[r, "{d_i,a_i}"] & (l_f^M, u_f^M)
\end{tikzcd}$$

We show that the last step in the trace does not exist in *Spec*. This step is due to a transition to the fail location f of TC which may be due to one of the following cases, based on the test case generation algorithm in Figure 5.3.

- It might be a τ transition which implies that the delay d_i is not permitted by the invariant of l_i^S in *Spec*; or,
- It might be that the action a_i that happens at $u_i^T + d_i$ leads TC to f, i.e., u_i^T satisfies the guard $\neg g_f$ where g_f is the disjunction of all guards that allow a_i. As a result, action a_i is not allowed at this time in *Spec*.

The trace shown above does not exist in the suspension traces of *Spec*, while it obviously does exist in the observable suspension traces of *MUT*. Therefore, *MUT* is not a refinement of *Spec*. □

Completeness Soundness is not sufficient to ensure the relevance of test cases. A test case with no **Fail** location is sound but cannot reject any system. We also need to be sure that if the system is a wrong refinement, there exists a test case able to reject it. In other words, any system that passes the test set is a correct refinement of the specification. A complete test set rejects any wrong refinement in the system.

Definition 5.1.6 (Complete). *A test set is* complete *for the refinement relation $\sqsubseteq$ if and only if*

$$MUT \text{ passes } \mathcal{T} \implies MUT \sqsubseteq Spec$$

□

Theorem 5.1.7 (Completeness). *The set of all test cases for Spec that can be generated by the algorithm in Figure 5.3 is* complete *for* $\sqsubseteq$.

Proof. We must prove that if the system is not a refinement, there exists a test case that makes the system fail. In other words, assuming that there exists an observable suspension timed trace of *MUT* not belonging to suspension timed traces of *Spec*, we must show that there exists a test case *TC* such that the **Fail** location is reachable in the product $TC \| MUT$.

Without loss of generality, we consider $\sigma = t_1 a_1 \ldots t_k a_k \in STraces_{obs}(MUT)$. The corresponding observable run in *MUT* is the following:

$$(l'_0, \mathbf{0}) \xrightarrow{d_1} (l'_0, u'_0) \xrightarrow{a_1} (l'_1, u_1) \rightarrow \ldots \xrightarrow{d_k} (l'_{k-1}, u'_{k-1}) \xrightarrow{a_k} (l'_k, u_k)$$

If σ is not a trace of *Spec*, then two cases are possible depending on the first behavior diverging from the trace of *Spec*:

1. There exists i, $0 \leq i \leq k-1$ such that $t_1 a_1 \ldots t_i a_i \in STraces(Spec)$ and $t_1 a_1 \ldots t_i a_i t_{i+1} \notin STraces(Spec)$, i.e., a delay of $d_{i+1} = t_{i+1} - t_i$ is not possible in *Spec* at state (l_i, u_i). It means that location l_i has an invariant h that is violated by $u_i + d_{i+1}$. Let T be a sequence of $i+1$ transitions from *Spec* such that $t_1 a_1 \ldots t_i a_i \in STraces(T)$. Let *TC* be the test case generated from T by the algorithm. Since location l_i has an invariant h, there is a transition in *TC* from l_i to location f whose guard is $\neg h$ and labeled with τ. As $u_i + d_{i+1}$ does not satisfy h, location f is reachable in the product $TC \| MUT$ with the trace $t_1 a_1 \ldots t_i a_i t_{i+1} \tau$ corresponding to the run

$$((l_0, l'_0), \mathbf{0}) \xrightarrow{d_1} ((l_0, l'_0), u'_0) \xrightarrow{a_1} \ldots \xrightarrow{d_{i+1}} ((l_i, l'_i), u'_i) \xrightarrow{\tau} ((f, l'_{i+1}), u_{i+1})$$

2. There exists i, $0 \leq i \leq k-1$ such that $t_1 a_1 \ldots t_i a_i t_{i+1} \in STraces(Spec)$ and $t_1 a_1 \ldots t_i a_i t_{i+1} a_{i+1} \notin STraces(Spec)$, i.e., the action a_{i+1} is not allowed in *Spec* at time t_{i+1}. It means that there is no transition in *Spec* from location l_i labeled with a_{i+1} whose guard is satisfied by $u_i + d_{i+1}$. Let T be a sequence of $i+1$ transitions from *Spec* such that $t_1 a_1 \ldots t_i a_i t_{i+1} \in STraces(T)$. Let *TC* be the test case generated from T by the algorithm. By construction of *TC*, there is a transition from location l_i to location f labeled with a_{i+1} whose guard is the complement of all other guards of transitions from l_i labeled with a_{i+1}, let us call it g. Since $u_i + d_{i+1}$ does not satisfy any of these guards, it satisfies g. Then location f is reachable in the product $TC \| MUT$ with the trace $t_1 a_1 \ldots t_i a_i t_{i+1} a_{i+1}$ corresponding to the run

$$((l_0, l'_0), \mathbf{0}) \xrightarrow{d_1} ((l_0, l'_0), u'_0) \xrightarrow{a_1} \ldots \xrightarrow{d_{i+1}} ((l_i, l'_i), u'_i) \xrightarrow{a_{i+1}} ((f, l'_{i+1}), u_{i+1})$$

Therefore, the set of all test cases generated by the algorithm is complete for $\sqsubseteq$. $\square$

A sound and complete test set is called *exhaustive*. Exhaustivity is in general impossible to reach, since it needs an infinite test set. Thus we know that we cannot in practice find all counter examples to refinement. However, we still want to ensure a certain quality to test cases. For instance, we want to avoid useless sound test cases where all paths lead to **Pass**.

Rigidness We are interested in test cases that reject models which behave in a wrong way *along the test case*: the test case should not say **Pass** if it is possible to detect something wrong during the test case execution. We show that any test case generated by our algorithm can detect every wrong behaviors occurring along it. We can actually show that we can provide a counter-example for any incorrect refinement occurring along the sequence of transitions the test case is built from. A test case TC is *rigid* if and only if it rejects any incorrect refinement along the traces of the test case; this is formally defined below.

Definition 5.1.8 (Rigid). *Given a trace $\sigma \in STraces(Spec)$, suppose the action or delay $e \in \Sigma \cup \{\delta\} \cup \mathbb{R}_+$ is allowed after σ in MUT but not in Spec, i.e., $\sigma.e \in STraces_{obs}(MUT) \setminus STraces(Spec)$. A test case TC is rigid for the refinement relation $\sqsubseteq$ if and only if*

$$\sigma \in Traces(TC) \wedge l_0 \xrightarrow{\sigma} l_i, 1 \leq i < n \Rightarrow \sigma.e \in Traces(TC) \wedge l_0 \xrightarrow{\sigma.e} \textbf{Fail}$$

□

Intuitively, if σ ends in a non-leaf location in TC, the test case TC will observe any one-step divergence after σ. This notion is close to the notions of non-laxness in the untimed setting [26] and of strictness in the timed setting [30] but it is stronger. These notions state that if the system behaves in a non-conforming way during the execution of the test case, it must be rejected. Also in our framework, every detected divergence leads to the rejection of the system, but we can add that every divergence is actually detected. This result directly follows from the construction of the test case.

Theorem 5.1.9 (Rigidness). *Let Spec be a deterministic timed automaton and T be a linear timed automaton built from a sequence of transitions in Spec. The test case for Spec generated from T by the algorithm in Figure 5.3 is* rigid *for $\sqsubseteq$.*

Proof. We show that for every trace σ of the test case TC ending in a non-leaf location, $\sigma.e$ is a trace of TC leading to **Fail** if $\sigma.e$ is a trace of MUT and not of *Spec*.

If $e \in \Sigma \cup \{\delta\}$, let $\sigma = t_1 a_1 \ldots t_k \in Traces(TC)$ corresponding to the run

$$(l_0, \mathbf{0}) \xrightarrow{d_1} (l_0, u_0') \xrightarrow{a_1} (l_1, u_1) \rightarrow \cdots \xrightarrow{d_k} (l_{k-1}, u_{k-1}')$$

where $k - 1 \neq n$. Since $\sigma.e$ is not a trace of *Spec*, it means that e is not allowed in *Spec* at location l_{k-1} after a delay of d_k. Then, by construction of the test case TC, there is a transition from l_{k-1} to the **Fail** location labeled with action e and whose guard satisfies u'_{k-1}. Then $\sigma.e$ is a trace of TC and $l_0 \xrightarrow{\sigma.e}$ **Fail**.

If $e \in \mathbb{R}_+$, let $\sigma = t_1 a_1 \ldots a_k \in \mathit{Traces}(TC)$ corresponding to the run

$$(l_0, \mathbf{0}) \xrightarrow{d_1} (l_0, u'_0) \xrightarrow{a_1} (l_1, u_1) \rightarrow \cdots \xrightarrow{a_k} (l_k, u_k)$$

where $k \neq n$. Since $\sigma.e$ is not a trace of *Spec*, it means that a delay of e is not allowed in *Spec* at location l_k, due to an invariant h at this location in *Spec*. Then, by construction of the test case TC, there is a transition from l_k to the **Fail** location labeled with τ and whose guard $\neg h$ satsifies u_k. Then $\sigma.e$ is a trace of TC and $l_0 \xrightarrow{\sigma.e}$ **Fail**. □

5.2 Testing Compatibility

Once it has been proved that each object of a system is schedulable given a particular behavioral interface, the schedulability of the whole system is ensured if the actual use of the objects in this system is compatible with their behavioral interfaces. For each object, the behavioral interface abstractly models its observable behavior in terms of the messages it may receive and the messages it sends. In the following, B represents the parallel composition of the behavioral interfaces, and S represents the system of communicating objects (cf. Chapter 3).

First we don't consider deadlines, i.e., the action $a(d)$ in system automata (resp. $a(d)?$ in behavioral interfaces) is not distinguishable from a (resp. $a?$). Furthermore, to make the notion of compatibility coincide with that of refinement we defined in section 5.1, we assume that the synchronization between the actions $a!$ and $a?$ will appear as the simple action a. Thus B and S represent two timed automata on the same set of actions, while S also contains τ actions. Following the testing framework we defined in Section 5.1, we can test compatibility of the system with respect to the behavioral interfaces.

Definition 5.2.1 (Compatibility). *The system S is compatible with the behavioral interfaces B if and only if $S \sqsubseteq B$.* □

We defined the behavioral interfaces to be deterministic in Definition 3.2.1. This does not reduce the expressiveness, because what is important for schedulability analysis is the timed traces in the driver. This implies the decidability of the compatibility check (see for example [42] for a way of checking trace inclusion in Uppaal).

Recall from Section 5.1 that refinement only considers the observable behavior of the system. Therefore, the system behavior is restricted to the communications between different objects. Thus compatibility ensures that each object is used correctly, i.e., receives messages as specified in its behavioral interface.

Deadlines When constructing the product of behavioral interfaces, for every synchronizing pair of actions, there is one deadline (from the input action). Therefore, every timed action in the traces of the product (denoted B) can be augmented with an integer d showing this deadline.

Similarly, in the system (more precisely in method automata) every observable output action (i.e., excluding self calls) has a deadline. The input actions (which appear in the schedulers) have no specific deadlines. Therefore, in the observable traces of the system, every timed action can also be augmented with an integer d' for the deadline.

To include deadlines in the formal definition of compatibility, we need to add the condition $d \leq d'$ for every matching action where d is the deadline in B and d' is the deadline in the system. When submitting a test case, the usual parallel composition is extended to allow synchronization only on actions with compatible deadlines: an action $a(d')$ in the system will be able to synchronize with an action $a(d)$ in the behavioral interfaces if and only if $d \leq d'$.

5.2.1 Schedulability

The objective of compatibility checking is to infer the schedulability of the whole system. The objects used in a system are proved individually schedulable with respect to their behavioral interfaces. In this subsection, we formally prove that compatibility implies schedulability of the whole system provided that individual behavioral object models are schedulable. Intuitively, this means that every message in the system will be finished within the designated deadline.

Theorem 5.2.2 (System Schedulability). *A system composed of a set of objects $O_1, \ldots, O_n$ is schedulable, if the behavioral object model for every O_i is individually schedulable (cf. Definition 4.2.5) and the system is compatible with the parallel composition of the behavioral interfaces of the objects.*

Proof. To prove this we can assume that the system is compatible but not schedulable. This means that there is a trace $\delta = t_1, a_1, t_2, \ldots, a_k, t_{k+1}$ of the system automaton (cf. Definition 3.2.7) in which one of the objects, say O_j, drives the system to the Error state, i.e., either the queue of O_j is overflown or a task in its queue misses its deadline. We show that this requires the existence of a trace in B_j that drives O_j to the Error state, which contradicts the schedulability assumption.

Due to compatibility, δ_{obs} exists in the product of behavioral interfaces. This trace can be projected onto the behavioral interface of O_j alone by removing the delay-actions t_i, a_i for every a_i that is not in the action set of the behavioral interface of O_j. Call the resulting trace $\delta_{obs}|_j$. We can compute a set of traces in BOM of object O_j as $T = \{\varphi_i \ : \ \varphi_{i_{obs}} = \delta_{obs}|_j\}$. Since the object individually is schedulable, these traces do not lead to *Error* state, i.e., given this sequence of inputs and outputs, none of the tasks in the queue of O_j misses its deadline, nor a queue overflow occurs.

On the other hand, the trace δ corresponds to a run of the system:

$$(\{l_0^1, \ldots, l_0^n\}, \mathbf{0}) \xrightarrow{a_1} \cdots \xrightarrow{a_{k-1}} (\{l_{k-1}^1, \ldots, l_{k-1}^n\}, u_{k-1}) \xrightarrow{a_k} (Error, u_k)$$

where l_i^j is the location of object j after i steps. Formally, $l_i^j = (s, Q)$ where s is the location of its currently running task and Q is the current task queue. For brevity the delay transitions are not shown. We project the trace $\delta = t_1, a_1, t_2, \ldots, a_k, t_{k+1}$ onto the actions of O_j, by removing the delay-actions t_i, a_i such that $l_{i-1}^j = l_i^j$. We represent the resulting trace as $\delta|_j = u_1, b_1, u_2, \ldots, b_h, u_{h+1}$.

By considering the definition of system automaton and BOM we can show that $\delta|_j \in T$. This requires that $\delta|_j$ drives O_j to the Error state, which is in contradiction with the schedulability assumption. □

5.2.2 Executing a Test Case

We explain here why testing a model instead of a real implementation simplifies the submission of test cases to the model and why such a method is an alternative to exhaustive verification avoiding state space explosion.

Our model of a system consists of a set of communicating objects. Each object, in turn, consists of methods and a scheduler automata (the scheduler contains a queue). Therefore, a system is a network of timed automata (representing methods and schedulers of all objects) which can run in Uppaal.

All actions except communication between objects are considered internal. Submitting a test case is then the synchronized product of the system automata and the test case automaton, in which every observable action in the system must synchronize with the test case.

A test case is a deterministic automaton and intuitively represents a specific order of exchanging messages between the objects. Therefore, requiring the system to synchronize with the test case resolves part of the non-determinism in the system. However, the internal actions of different objects are non-deterministically interleaved (more precisely, those internal actions that can happen at the same time). We would then need to execute the test case several times to get a verdict. In fact, since the system under test passes the test if the **Fail** location is not reachable in the composition of the test case and the system, using a model-checker to execute the test case comes to check for the reachability of the **Fail** location. So only one check is needed. The Uppaal model checker can provide a diagnostic trace in case of failure, showing exactly how and when the system is not compatible.

Since the system behavior is controlled by the test case, it is possible to model-check the system with respect to the test case even when model-checking the whole system may not be feasible. We then avoid state space explosion by restricting verification to the behavior of the system matching the test case.

5.3 Case Studies

Individually schedulable objects can be used in making different actual systems. However, it is necessary to make sure that each object is used correctly, i.e., according to its behavioral interface. This can be tested by the compatibility check. Once compatibility is ensured, we can immediately deduce the schedulability of all objects in the system.

We analyzed the `MutEx` object in Chapter 4 and showed that it is schedulable given a customized EDF policy. In this section, we use the schedulable `MutEx` in the context of dining philosophers and bridge controller systems. The schedulability of these systems (which is deduced from compatibility) implies starvation freedom (because a schedulable `MutEx` guarantees granting requests in time).

Submitting a test case When submitting a test case, we require that any communication between two objects should synchronize with the test case, as well. Practically, this means that the sender object (in one of its methods), the receiver object (in its scheduler) and the test case should synchronize. UPPAAL does not support three-way synchronization.

Since we do not want to change the specification of the model under test, we solve the problem of three-way synchronization by splitting every action in the test case into two steps. At the first step, the sender object synchronizes with the test case, and *immediately* afterwards, the test case synchronizes with the receiver object. The urgency between these two steps is modeled by using a 'committed' location in the test case between these two steps.

Naive Philosophers

In this case, every philosopher tries to take the right fork and then the left fork. We use `MutEx` as fork. The model of a philosopher is given in appendix. We test compatibility for a system composed of four philosophers and four forks. It is well known that this naive binding results in deadlock and starvation. In fact, that is because the `MutEx` behavioral interface expects a **release** before MAX_REL, which never arrives.

We choose the deadlock property as the starting point for compatibility check. We check the product of behavioral interfaces for deadlock. Such a deadlock may occur when all philosophers have requested and been granted their left fork. We can get a diagnostic trace from UPPAAL by model checking for deadlock and use it to build a test case. As expected this test case fails (i.e., results in a counter example to compatibility). We can see that this naive binding is in fact incompatible with respect to the behavioral interface of `MutEx`.

A Right-handed Philosopher

One solution to the deadlock problem is to let one philosopher take the right fork first (while others pick up the left fork first). This right-handed philosopher has a

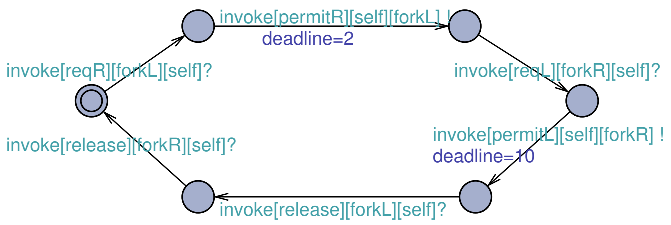

invoke[permitR][self][forkL] !
deadline=2
invoke[reqR][forkL][self]?
invoke[reqL][forkR][self]?
invoke[release][forkR][self]?
invoke[permitL][self][forkR] !
deadline=10
invoke[release][forkL][self]?

Figure 5.5: The behavioral interface for the Philosopher object.

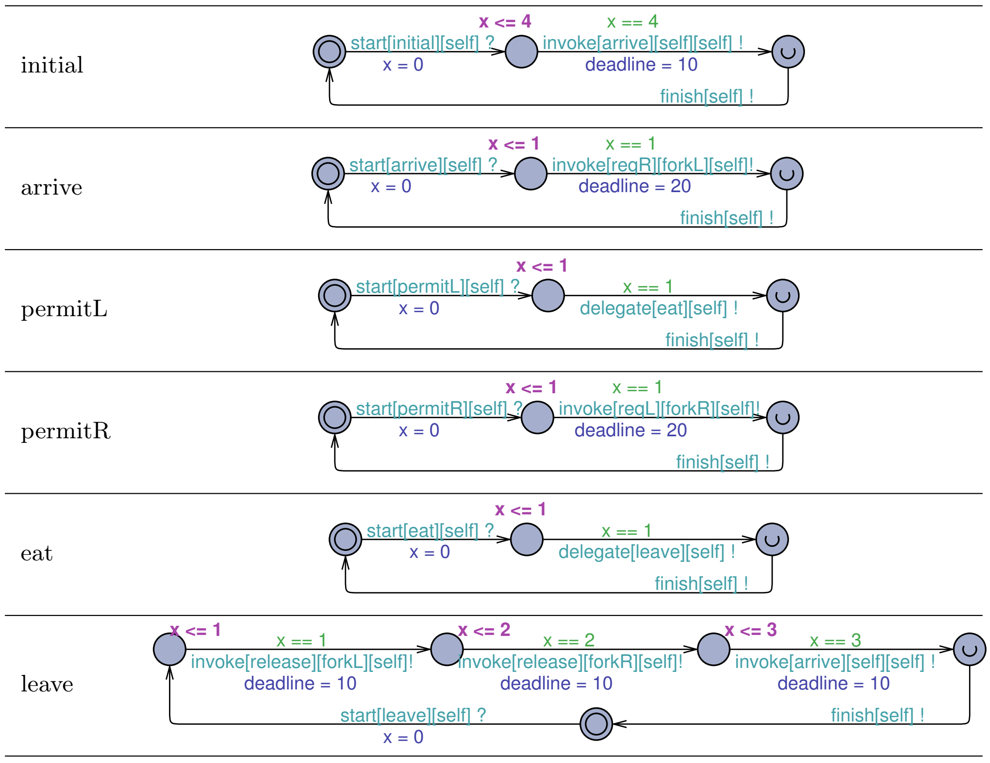


Figure 5.6: Modeling methods and interface of Philosopher

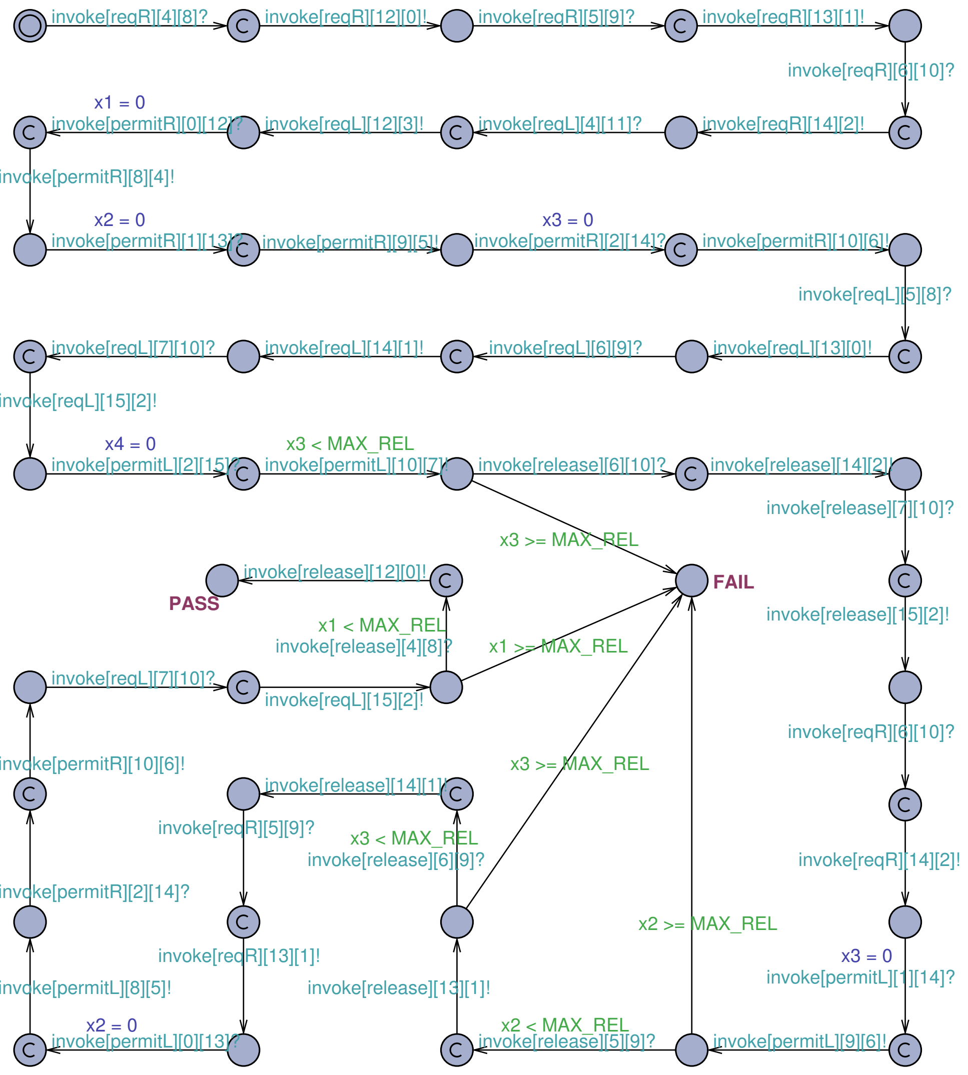


Figure 5.7: A test case for dining philosophers without testing deadlines. Every philosopher is given a chance in this test case to obtain the chopsticks and then release them. Every action is split in two steps: capturing the message sent by an object and feeding it to the queue of the receiver. Both steps happen at the same time.

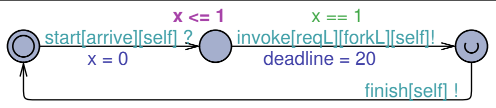
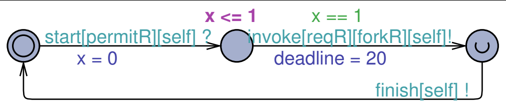
arrive	
permitR	

Figure 5.8: Modeling methods of a right-handed Philosopher

different behavioral interface as it sends reqL (to forkR) before reqR (to forkL). In this case, the product of the behavioral interfaces is deadlock free.

We select a trace in which every philosopher can perform a cycle of getting both forks until it releases them (see Figure 5.7). Interestingly, this test also fails because it takes too long for the right-handed philosopher (i.e., greater than MAX_REL) to release one of the forks (which is somehow due to the asymmetry in the model). By increasing MAX_REL in the behavioral interface of MutEx (and accordingly the deadlines such that it remains schedulable), this model of philosophers can become compatible. However, increasing the deadlines implies waiting longer before a request is granted.

A Late Philosopher

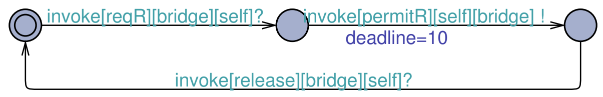
Another solution to deadlock in dining philosophers is to make one or more philosophers start later than the others. This model passes the compatibility test that the previous model failed (without increasing the deadlines). This implies the schedulability of all objects in the context of this system, which in turn implies starvation freedom.

Another result is that one can deduce an upper bound on the time between when a philosopher starts a round (requests the two forks) until he releases them. This can be done because we know that the requests are granted within the specified deadline, and the time needed for the local computation in a philosopher can also be computed.

Bridge Controller

There is a bridge which can allow at most one train to pass at a time. Therefore, the trains on the other side must wait until the bridge is free. The trains arriving on the left side of the bridge send reqL and the trains on the right send reqR. The model of a train (arriving on the left side of the bridge) is given in appendix. A train is different from philosophers as every train needs only one MutEx. We use a test case in which a train can take and release the MutEx representing the bridge.

Figure 5.11 shows the UPPAAL implementation of this test case. The committed locations have a C inside. Notice that in this figure, the inconclusive location is not



invoke[reqR][bridge][self]?
invoke[permitR][self][bridge] !
deadline=10
invoke[release][bridge][self]?

Figure 5.9: The behavioral interface for the Train object.

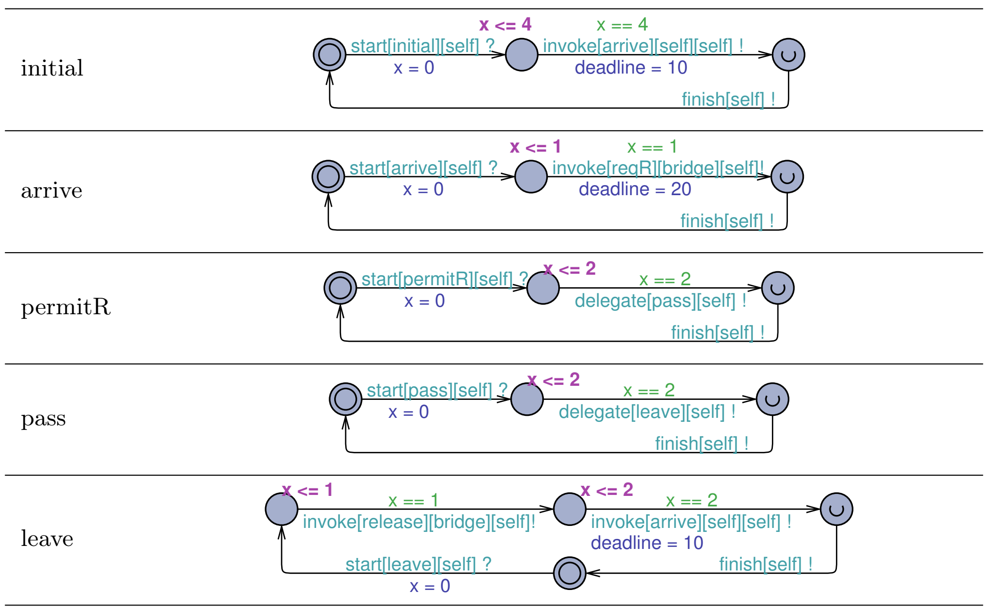


Figure 5.10: Modeling methods of a Train arriving on the right side of the bridge.

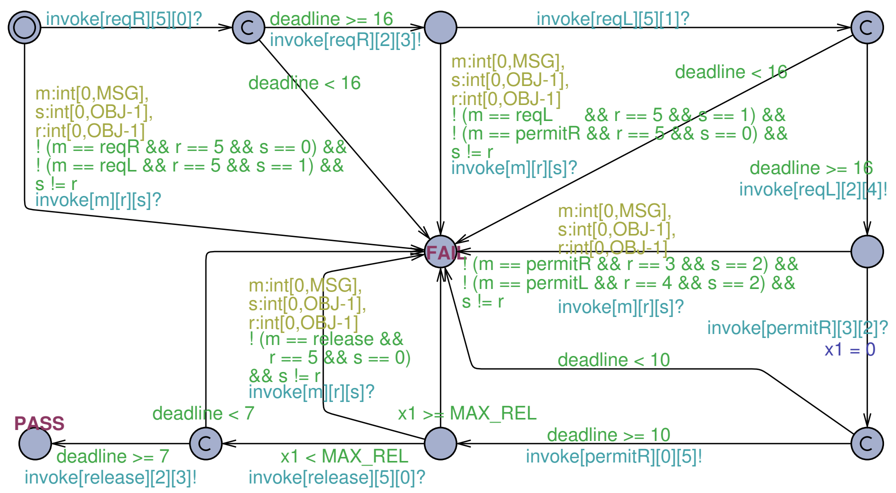


Figure 5.11: Test case for bridge controller in UPPAAL taking deadlines into account.

explicitly modeled, because we are basically interested in checking the reachability of FAIL or PASS state. If neither is reachable, then the test is inconclusive. To capture the disallowed actions at each location, we use the select feature of UPPAAL to be able to choose any action; in the guard, we exempt the allowed communication actions (by specifying the message, sender and receiver) and self calls (s != r). The correct deadline values are checked on the transitions going out of the committed locations.

The system passes this test case if the trains are not too slow (as in the Appendix). Increasing the time for passing the bridge would render the system incompatible. In that case, this test case fails by taking the transition that checks x1 >= MAX_REL and leads to FAIL (from the location at the middle bottom).

Chapter 6

Schedulability Analysis of Real-Time Creol

This chapter deals with schedulability analysis of object-oriented models specified in Real-Time Creol as introduced in Chapter 2. Although Real-Time Maude is used to specify executable semantics of Real-Time Creol, it is not suitable for model checking Creol. The rewrite semantics of Creol in maude is infinite state, and moreover, the analysis results given by Real-Time Maude is incomplete for dense time models (due to decidability constraints). We define in this section timed automata semantics for an abstraction of Real-Time Creol suitable for model checking. This semantics maps each method to one automaton. For practicality, we define this semantics in terms of a translation algorithm that generates an Uppaal model containing the automata for methods together with other necessary global declarations.

We will provide a timed automata semantics for Creol with the purpose of schedulability analysis. This semantics can also be used for analyzing other real-time properties for Creol. For practicality, this semantics is given in terms of a translation algorithm from Creol code to Uppaal input syntax, making it possible to apply Uppaal directly on the generated automata model.

This semantics is based on the automata-theoretic framework in Chapter 3 for modeling actors. However, Creol includes more sophisticated synchronization mechanisms like processor release points and asynchronous replies. To this end, the original framework needs to be extended.

This chapter finally shows an alternative approach to arguing about schedulability of Creol models. Instead of automatically generating timed automata from Creol code, one can manually create the automata models for schedulability analysis in Uppaal. To extend the analysis results to the Creol models, then we can establish conformance between the Creol and automata models.

6.1 Timed Automata Semantics For Creol[1]

In Creol a class may implement multiple interfaces. For schedulability purposes, each static interface must be associated with a behavioral interface as in Definition

[1]This section is an improvement and extension of the results published in [12, 22].

3.2.1. For outgoing messages, the behavioral interface should capture when a reply is expected (cf. Definition 3.2.1). The implementation of the class must be schedulable with respect to each of its behavioral interfaces. If a class instance can take part in protocols associated to more than one interface simultaneously, then those interfaces should be used together (i.e., interleaved) when analyzing the class for schedulability.

The semantics of a Creol class consists of the automata for its methods. When instantiating a class, it should also be associated with a scheduler automaton. Below we formally define a class in the automata semantics for Creol. In this definition, each method is considered a task and corresponds to one automaton. Furthermore, releasing the processor creates a subtask for executing the rest of the method. Subtasks do not need independent automata; they reuse the automaton for their parent method. If a subtask is created at a conditional release point, it will be disabled as long as the condition of the release point is not true. To model this, all tasks have enabling conditions.

Definition 6.1.1 (Class). *A class R implementing a set of behavioral interfaces B^* is defined as follows. Recall that each $B \in B^*$ has a set of method names M_B. R is a set $\{(m_1, E_1, A_1), \ldots, (m_n, E_n, A_n)\}$ of tasks, where*

- *$M_R = \{m_1, \ldots, m_n\} \subseteq \mathcal{M}$ is a set of task names such that $M_B \subseteq M_R$ for every $B \in B^*$. M_R includes the sub-tasks created at release points, as well as the methods; however, there is exactly one automaton for each method.*
- *for all i, $1 \leq i \leq n$, A_i is a timed automaton representing the method containing the task m_i.*
- *for all i, $1 \leq i \leq n$, E_i is the enabling condition for m_i.*

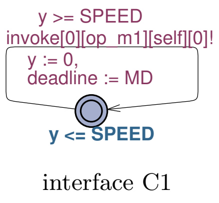
□

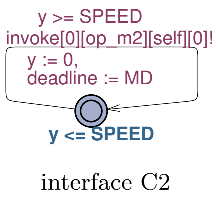
We assume that the given Creol models are correctly typed and annotated with timing information. We use the same syntax for expressions and assignments in Creol, as is used by Uppaal. This allows for a more direct translation.

We repeat the code for the coordinator example from Chapter 2 and will use it as a running example. Figure 6.2 shows the automata generated for the methods run and m1. The automata for m2 and m3 are similar. As can be seen in these automata, the method and variable names are prefixed in order to avoid name clashes with Uppaal keywords. Location labels in these automata refer to line numbers in the original Creol code in Figure 6.1 (cf. function *Loc* in Table 6.1, Section 6.1.2).

Execution of the method run starts with the transition to location 12. The automaton delays in this location as required by the duration statement at line 12. The '**await**' statement in line 13 may release the processor if its guard is not true. In the case of a release point, the rest of the method is modeled as a sub-task, e.g., the transition from location 13 to the *final* location marked `f` in Figure 6.2 generates a sub-task op_run1 if the guard associated to **await** does not hold; the processor is released by the subsequent transition with the action finish going back to the initial

```
interface C1 begin with Any op m1 end
interface C2 begin with Any op m2 end
interface C3 begin with Any op m3 end
class Coordinator implements C1, C2, C3 begin
   var s1,s2,s3,sync : bool
   op init ==
      s1 := false; s2 := false;
      s3 := false; sync := true;
      duration(2, 4)
   op body == duration(2, 5)
   op run ==
      duration(1, 2);
      await (s1 /\ s2 /\ s3);
      duration(1, 1);
      b!body(10);        // force sync call with deadline 10
      b.get;
      sync := false;
      duration(1, 2);
      await (~s1 /\ ~s2 /\ ~s3);
      duration(1, 1);
      sync := true;
      !run(50)           // deadline = 50
   with Any op m1 ==
      await (sync /\ ~s1);
      duration(1, 1);
      s1 := true;
      await ~sync;
      duration(1, 1);
      s1 := false
   with Any op m2 ==   ...like m1...
   with Any op m3 ==   ...like m1...
end
```





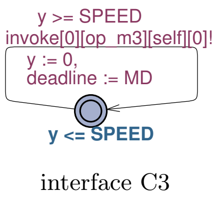
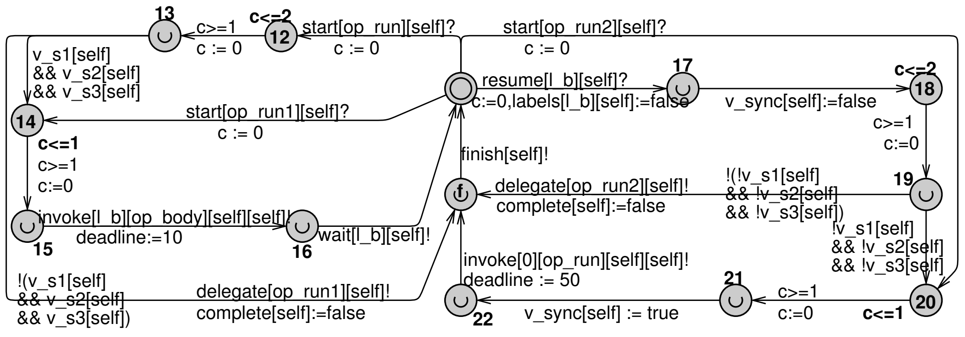
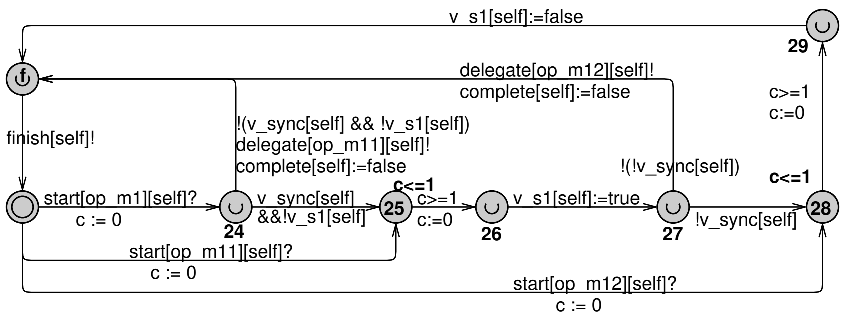


Figure 6.1: A real-time coordinator in Creol with periodic task generation in the behavioral interfaces.



(a) The automaton for method `run`



(b) The automaton for method `m1`

Task	Enabling Condition	Task	Enabling Condition
`run`	true	`m1`	true
`run1`	s1 ∧ s2 ∧ s3	`m11`	sync ∧ ∼s1
`run2`	∼s1 ∧ ∼s2 ∧ ∼s3	`m12`	∼sync

Figure 6.2: Method automata and the enabling conditions for their subtasks.

location. This sub-task inherits the remaining deadline of the original task; this is done by the scheduler when handling the delegate channel (see next subsection). The enabling condition of the sub-tasks are equivalent to the guard used in the corresponding release point (see tables in Figure 6.2). The sub-task op_run1 can be triggered with the start transition entering location 14.

The method sends a local message at line 15 and blocks afterwards on its reply; thus simulating a synchronous call. Blocking statements use the wait channel to transfer the control to the scheduler, which then can check whether the label is associated to a local call; thus it can decide whether to block the object or do a synchronous call. To allow recursive synchronous calls, the wait transition goes back to the initial state making the method reentrant, e.g., the transition from location 16 modeling the statement b.**get**. The rest of the method is executed as it resumes to location 17.

The details of the translation are explained formally in Section 6.1.2.

Abstractions

For the sake of simplicity, we abstract from parameter passing, however, it can be modeled in UPPAAL by extending the queue to hold the parameters. This is further elaborated on in Section 6.1.1.

In standard Creol, different invocations of a method call `t!p()` are associated with different instances of the label `t`; this makes it possible to distinguish the replies to the different method invocations. For instance executing the statement `t!p()` twice results in two instances of the label t. Dynamic labels give rise to an infinite state space for non-terminating reactive systems. To be able to perform model checking, we treat every label as a static tag. Therefore, different invocations of a method call with the same label are not distinguished in our framework. Alternatively, one could associate replies to message names, but this is too restrictive. By associating replies to labels, we can still distinguish the same message sent from different methods with different labels.

Another complication in translation is how to map a possibly infinite state Creol model to finite state automata. We do this by abstracting away some information. One automatic way of abstracting is as follows: variables from a finite domain can be mapped to themselves but conditions on potentially infinite variables are mapped to true. We do abstraction semi-automatically which is represented with the function *Abs* in Section 6.1.2, i.e., the user states which variables are abstracted away. In other words, we over-approximate the behavior of the Creol model. A more advanced abstraction would map potentially infinite variables to finite domains in order to narrow the over-approximation.

6.1.1 Extended Schedulers

A scheduler manages the buffering and execution of the tasks. A typical way to model *scheduler automata* is shown in Figure 6.3. A transition numbered α in this figure is referred to as t_α in the text. This model is generic except for the strategy, explained below. The keyword `self` holds the current object identifier. Declarations related to the scheduler automata including function definitions are given at the end of the chapter.

Queue The queue has the size MAX. A scheduler automaton uses multiple arrays of size MAX to implement the queue and its clocks. The message in `q[i]` is received from object `s[i]`; and, `ca[i]` points to a clock in array `x` that records how long the message has been in the queue: this message should be processed before the clock `x[ca[i]]` reaches its deadline value `d[ca[i]]`. A clock `x[i]` may be assigned to more than one task, the count of which is stored in `counter[i]`. We have shown in Chapter 4 that schedulable objects need bounded queues, i.e., with a proper MAX value, queue overflow implies nonschedulability. An `Error` location is reachable when a task in the queue explicitly misses its deadline (t_{14}). Support for parameters can be added with an array `p` such that `p[i][j]` holds the `j`th parameter of `q[i]`.

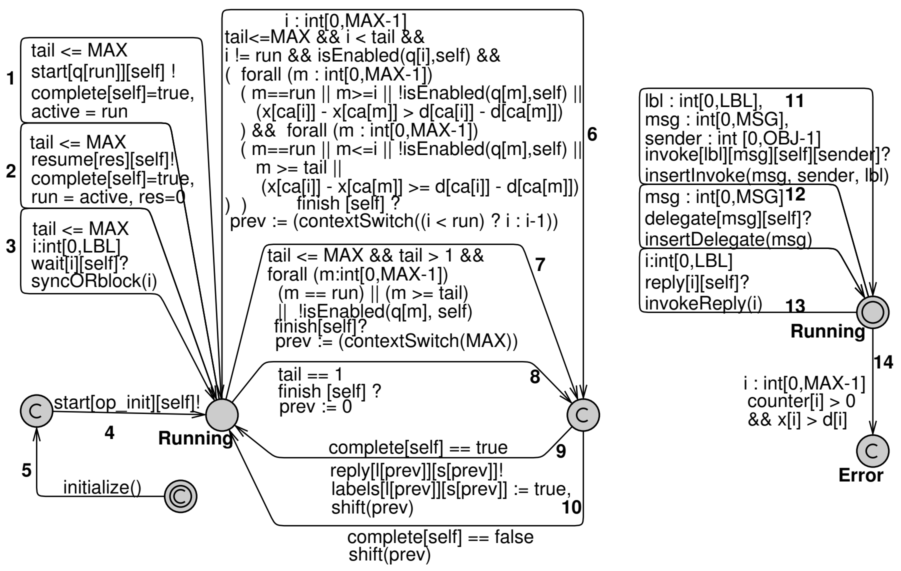


Figure 6.3: An Earliest-Deadline-First scheduler modeled in UPPAAL (details explained in the text).

Channels The start channel is for starting execution of a task (t_1 and t_4), which later gives up the processor with a signal on finish channel (t_6, t_7 and t_8). After a task has finished a reply signal is sent back to the caller (t_9); in case of release points (where complete is false) no signal is sent (t_{10}). To handle replies for self-calls, sending (t_9) and receiving (t_{13}) must be modeled in separate automata. The invoke channel is for sending/receiving messages (async call); t_{11} generates a new task with the deadline given in the variable **deadline** by putting the message name, sender and the associated label in the queue. The delegate channel generates a sub-task (t_{12}), which inherits the deadline of the parent task. The blocking statement t.**get** passes control to the scheduler using wait channel (t_3); this allows the scheduler to perform a synchronous call when needed. A blocked task is resumed using resume channel (t_2). To avoid context-switch delays, start and resume defined urgent.

Start-up An object is initialized by putting 'init' and 'run' methods in its queue (t_5), immediately followed by executing the 'init' method (t_4). During 'init', the object may receive other messages which are allowed to compete with the 'run' method for execution.

Context-switch Tasks can have enabling conditions, which may include the availability of a reply, but does not depend on clock values. Therefore, we can define in UPPAAL a boolean function isEnabled to evaluate the enabling condition for each method when needed. Whenever a task finishes, the scheduler selects another *enabled* task, based on its strategy (t_6), sets run to point to this task, and executes it (t_1). There are two special cases: (1) the last task in the queue has just finished (t_8); since run is zero and q[0] gets EMPTY after shift, the processor becomes idle until a new task arrives (t_{11}). (2) all remaining tasks in the queue are disabled (t_7); run is set to MAX to block the processor. In this case, the object may be enabled either by receiving a new task (t_{11}) or receiving a reply signal (t_{13}). The functions insertInvoke and invokeReply take care of these cases.

Labels We handle labels with a global boolean array labels, and assuming that label names are unique in each class, we can define constants for each label such that labels[t][self] uniquely identifies label t. When the condition in a release point includes t?, i.e., waiting until the reply to the call with label t is available, we replace t with labels[t][self] which is set to true by the scheduler when called method finishes (t_9). To distinguish completion of a method from processor release points, we use the variable complete. The variable complete is true by default and will be reset to false at release points (cf. rel and crel rules in Figure 6.5).

Synchronous call Executing the blocking t.**get** statement (wait channel) on a remote call blocks the object until a reply is received. However, if t is associated to a local call, the scheduler will start the called method (modeling synchronous function call); upon termination of the called method, the scheduler will resume the blocked process (resume channel). In order to allow nested synchronous invocations, each message in the queue stores a pointer to the caller method. As explained in the next section, the blocked method goes back to its initial location allowing recursive synchronous function calls. Note that this recursion is bounded with the queue length.

Scheduling strategy The selection strategy is specified as a guard on t_6. Parts of this guard ensure that we consider only non-empty queue elements (i < tail) containing an enabled task (by calling isEnabled) different from the currently running one (run). Figure 6.3 compares the remaining deadline of task i, obtained by d[ca[i]] − x[ca[i]], with other tasks in the queue; task i is selected when its deadline is strictly less than that of task m for m>=i and less than or equal for m<=i. By replacing deadlines with a priori defined task priorities, we can model fixed-priority-scheduling. By fixing run to zero, we obtain first-come-first-served strategy. Nevertheless, the model is not restricted to these strategies.

6.1.2 A Formal Encoding

We define timed automata semantics for Creol in terms of a translation algorithm. The input to this algorithm is a Creol model consisting of class and interface specifications. The output is one UPPAAL model for each class; this UPPAAL model consists of only a set of timed automata templates (*TAT*), one for each method, and the global declarations (*Dec*). The system declarations is not generated automatically, because

Function	Input	Output
$[\![.]\!] : M \to A$	A method	A timed automaton
$\underline{[\![.]\!]} : S \times L \times L \to 2^T \times 2^{L\times G} \times 2^{N\times G}$	A Creol statement, two automata locations	Part of a timed automaton including transitions and location invariants, plus a set of enabling conditions
$Loc : L \to S$	A location in a method automaton	A Creol statement in the method body
$pre : N \to N$	a variable or label name	prefixed name
$LabelReset : G \to 2^U$	A guard	UPPAAL update statements
$Abs : E \to E$	Creol expression or label	UPPAAL expression
$labels : * \to L$	overloaded	label names
$methods : * \to N$	overloaded	method names

Table 6.1: Summary of functions used in translation

it depends, among others, on the choice of scheduling strategy. We formally define the translation for a class:

$$[\![\textbf{class } C\ (a^*) \textbf{ implements } i^* \textbf{ begin } v^*\ m^* \textbf{ end}]\!] = (Dec, TAT)$$

where timed automata templates are $TAT = \{(Beh(I), C\_I, Args, Local(I)) \mid I \in i^*\} \cup \{(\underline{[\![\textbf{op } N == S]\!]}, C\_N, Args, \{\}) \mid \text{“}\textbf{op } N == S\text{”} \in m^*\}$. Functions $Beh(I)$ and $Local(I)$ return the user defined automata and their local declarations for the interface with the name I. All automata templates have as arguments: $Args = \{$`const int` arg`;` $\mid arg : Type \in a^*\} \cup \{$`const int self;`$\}$. First we define automata templates for methods and later global declarations Dec in this section.

Automata Templates Given a method declaration ‘**op** N $==$ S’, the corresponding timed automaton is computed as shown in Figure 6.5 by computing $\underline{[\![\textbf{op } N\ ==\ S]\!]}$, which uses the overloaded function $\underline{[\![.]\!]}$ to compute the automaton transitions. A cointerface given using **with** keyword is ignored. The intuition behind the translation function is given in Figure 6.4 using a notation similar to graph transformation rules. In the generated method, the locations l_0 and f refer to the unique locations representing the *initial* and *final* location of the method automaton, respectively. The final location f is urgent and is connected to l_0 to allow multiple incarnations of the method execution.

The function $\underline{[\![S]\!]}_{a,e}$ translates the statements S to a set of transitions that start from the location a and finish at e, possibly via some newly created locations. This function returns a triple (L, T, I, E) where L is a set of locations, T is a set of transitions, I is a function that assigns invariants to locations (changed by the *delay* rule), and E is a function that gives the enabling conditions for subtasks (generated by release points, namely, *rel* and *crel* rules), e.g., see tables in Figure 6.2. In Figure 6.5, we do not write I and E for rules that return empty sets for these elements; otherwise, they are written as function assignments.

Except for delay transitions, all other transitions take zero time; this is modeled by making their source location urgent. Notice that by using urgent locations, rather than committed locations, we allow for interleaving of the local transitions of parallel objects.

The partial function $Loc : L \rightarrow S$ assigns to the locations of a method automaton, their corresponding statements in the method body. This function is not defined for the initial and the urgent final locations in the generated method automata. With each automaton corresponding to exactly one method, one can use this function to trace from automata back to methods. The intuition is shown in Figure 6.4 as a label on new locations on the right hand side of *seq*, *if* and *while* rules.

The translation uses some helper functions. With function *pre*, all variable and label names are prefixed to prevent possible clash with the keywords of UPPAAL. As explained in the scheduler, there is a global boolean array *labels*. For every label t that is checked in guard g, $LabelReset(g)$ resets the corresponding element in *labels* to false. $Abs(ex)$ is the abstract of the expression or assignment ex with proper prefixing of the variables; the abstract must map the expressions to a finite domain, mapping everything to *true* is correct but not optimal. If the domains of the variable are finite we can leave them unchanged, however if they are possibly infinite then we must approximate them. The user of our translation can decide the exact approximation, as it will need to balance between state-space size and accuracy. A possible definition of these functions is given in Figure 6.6.

Next, we explain the automata expansion rules in Figure 6.5 with the help of the example in Figure 6.2.

seq The sequential composition of two statements is translated to two sequential transitions. The results of applying $\underline{[\![.]\!]}$ on these statements recursively is then put together using the special union operator: $(L_1, T_1, I_1, E_1) \uplus (L_2, T_2, I_2, E_2) = (L_1 \cup L_2, T_1 \cup T_2, I_1 \cup I_2, E_1 \cup E_2)$. This rule is applied until $\underline{[\![.]\!]}$ is applied on basic statements, explained below.

delay & skip These rules do not change any variables and thus do not affect the functional behavior of the model. A `duration` statement models delay in our specifications. The best-case and worst-case of the delay are translated, respectively, to a guard and an invariant using the clock c; see locations 12, 14 and 18 in Figure 6.2. Notice that the clock c is reset on this transition as well as *start* and *resume* transitions. By using urgent locations, this clock remains zero in other locations.

assign This transition uses the *Abs* function to properly update the values of the variables that are not abstracted away. In the coordinator example we do not abstract from any variables. Locations 17 and 21 are examples of assignment.

call & call2 A call is translated to an invoke channel. For example, location 15 shows sending the message op_body with the label l_b. If no label is used, i.e., *call* rule, the default value zero is used.

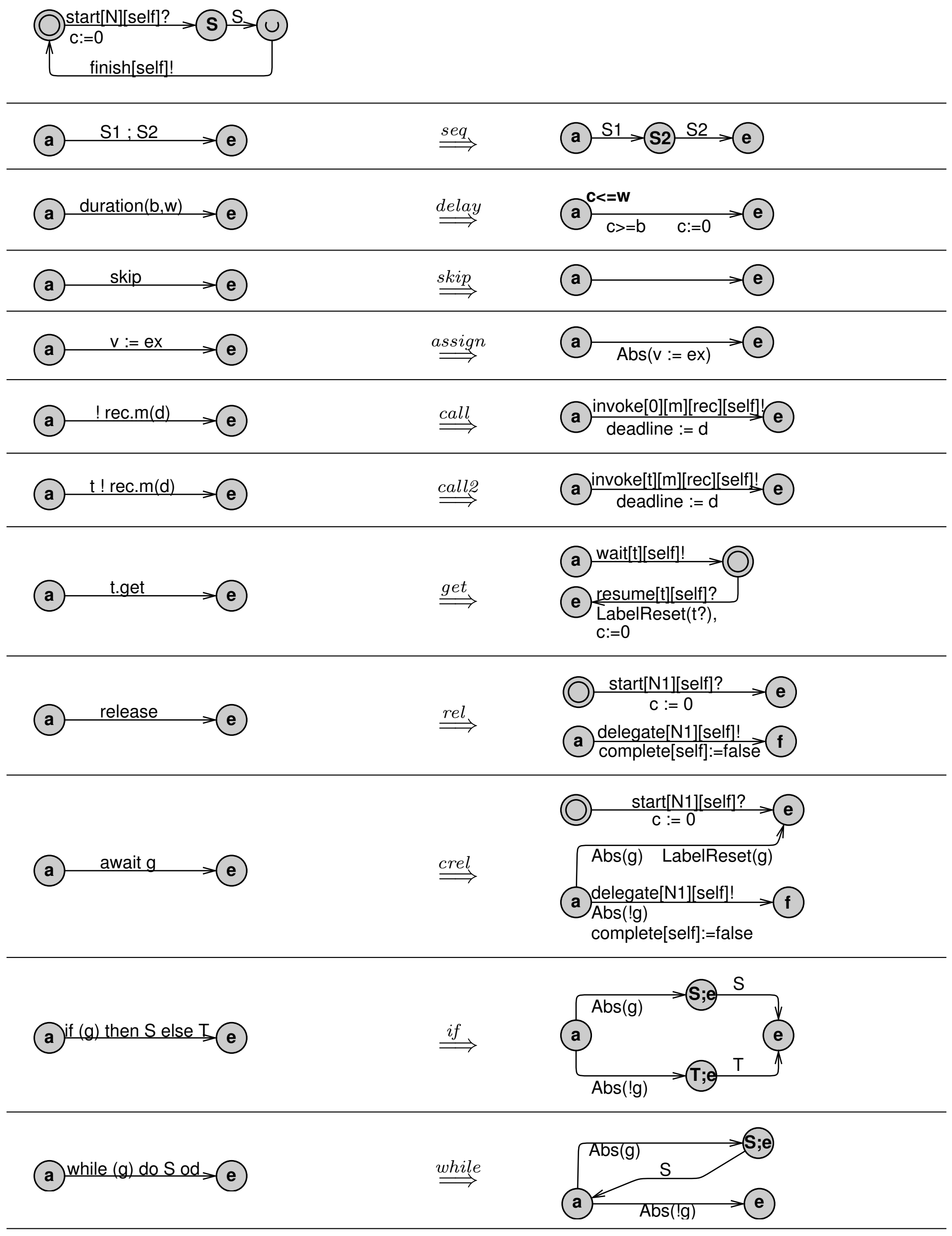


Figure 6.4: Automata expansion rules: the intuition in graph rewriting

$\llbracket \textbf{with } N' \textbf{ op } N \;==\; S \rrbracket = \llbracket \textbf{op } N \;==\; S \rrbracket = (L \cup \{l_0, a, f\}, l_0, T \cup T_1, I)$ where $\llbracket S \rrbracket_{a,f} = (L, T, I, E)$; and, $T_1 = \{l_0 \xrightarrow[c:=0]{start[N][self]?} a, f \xrightarrow{finish[self]!} l_0\}$; and,
Location f and every location $l \in L$ such that $I(l)$ is not defined is urgent.
New locations: l_0 (initial), f and a such that $Loc(a) = S$

seq	$\llbracket S_1; S_2 \rrbracket_{a,e} = \llbracket S_1 \rrbracket_{a,l} \uplus \llbracket S_2 \rrbracket_{l,e}$ New location: l such that $Loc(l) = S_2$
delay	$\llbracket duration(b, w) \rrbracket_{a,e} = \{a \xrightarrow[c \geq b \;;\; c:=0]{} e\} \wedge I(a) = (c \leq w)$
skip	$\llbracket skip \rrbracket_{a,e} = \{a \rightarrow e\}$
assign	$\llbracket v := ex \rrbracket_{a,e} = \{a \xrightarrow[Abs(v:=ex)]{} e\}$
call	$\llbracket !rec.m(d) \rrbracket_{a,e} = \{a \xrightarrow[deadline:=d]{invoke[0][pre(m)][pre(rec)][self]!} e\}$
call2	$\llbracket t!rec.m(d) \rrbracket_{a,e} = \{a \xrightarrow[deadline:=d]{invoke[pre(t)][pre(m)][pre(rec)][self]!} e\}$
get	$\llbracket t.get \rrbracket_{a,e} = \{a \xrightarrow{wait[pre(t)][self]!} l_0,\; l_0 \xrightarrow[LabelReset(t?), c:=0]{resume[pre(t)][self]?} e\}$
rel	$\llbracket release \rrbracket_{a,e} = \{a \xrightarrow[complete[self]:=false]{delegate[N1][self]!} f,\; l_0 \xrightarrow[c:=0]{start[N1][self]?} e\} \wedge E(N1) = true$ where f is the unique final location and $N1$ is a new name.
crel	$\llbracket await\ g \rrbracket_{a,e} = \{a \xrightarrow[Abs(!g) \;;\; complete[self]:=false]{delegate[N1][self]!} f,\; a \xrightarrow[Abs(g) \;;\; LabelReset(g)]{} e,$ $l_0 \xrightarrow[c:=0]{start[N1][self]?} e\} \wedge E(N1) = Abs(g)$ where $N1$ is a new name, l_0 is the initial location and f is the unique final location.
if	$\llbracket if\ (g)\ then\ S\ else\ T \rrbracket_{a,e} =$ $\llbracket S \rrbracket_{l_1,e} \uplus \llbracket T \rrbracket_{l_2,e} \uplus (\{l_1, l_2\}, \{a \xrightarrow[Abs(g)]{} l_1, a \xrightarrow[Abs(!g)]{} l_2\}, \{\}, \{\})$ New locations: l_1 and l_2 such that $Loc(l_1) = S; Loc(e)$ and $Loc(l_2) = T; Loc(e)$
while	$\llbracket while\ (g)\ do\ S\ od \rrbracket_{a,e} = \llbracket S \rrbracket_{l,a} \uplus (\{l\}, \{a \xrightarrow[Abs(g)]{} l, a \xrightarrow[Abs(!g)]{} e\}, \{\}, \{\})$ New location: l such that $Loc(l) = S; Loc(e)$

Figure 6.5: Automata expansion rules: Formal specification. The union of tuples is defined as $(L_1, T_1, I_1, E_1) \uplus (L_2, T_2, I_2, E_2) = (L_1 \cup L_2, T_1 \cup T_2, I_1 \cup I_2, E_1 \cup E_2)$.

$Abs(ex) = ex\{\texttt{labels[}pre(t)\texttt{][self]/t?}\}\{pre(x)\texttt{[self]/x}\}$ for variables x and labels t

$pre(t)$	$=$	`l_t`	for labels t
$pre(x)$	$=$	`v_x`	for variables x
$pre(m)$	$=$	`op_m`	for method names m
$pre(r)$	$=$	`o_r`	for object name r

$$
\begin{aligned}
LabelReset(g \wedge g') &= LabelReset(g) \cup LabelReset(g') \\
LabelReset(g \vee g') &= LabelReset(g) \cup LabelReset(g') \\
LabelReset(t?) &= Abs(t?)\ \texttt{= false;} \\
LabelReset(b) &= \{\}
\end{aligned}
$$

Figure 6.6: Helper functions in encoding automata templates.

get The statement t.**get** stops the current method and triggers the scheduler (on wait channel). If t is associated to a local call, the scheduler immediately starts the called method (i.e., sync call). This can be a recursive call because the *get* rule moves the method to the initial location l_0 (transition from 16 to l_0 in Figure 6.2). If t refers to a remote call, the object is blocked (no method is executed while the object is blocked). Notice that l_0 is not urgent and thus execution of a blocking **get** can take any amount of time; and resume is an urgent channel, therefore, the caller is *resumed* as soon as the callee has finished and a reply signal is available (transition from l_0 to 17 in Figure 6.2).

rel This rule releases the processor by moving the control to the final location$f$; therefore, the current task will finish. Instead, it generates a subtask N1 with the enabling condition *true*, i.e., the subtask is immediately enabled. The transition start[N1][self]? defines the subtask N1 such that it will execute the rest of the original task

crel With the **await** g statement, if g holds, the processor is not released and the method continues (e.g., transition 25 to 26 in Figure 6.2). If g does not hold, it releases the processor as in the *rel* rule. For example in Figure 6.2, control moves from location 13 to the final location f, and the generated subtask is modeled with the transition from l_0 to 14.

if & while These rules create a branch based on the valuation of the abstract of guard g. This abstraction could turn this branch into a nondeterministic choice. The *if* rule makes sure that both branches are joined afterwards, whereas the *while* rules closes it as a loop.

Global Declarations In Figure 6.7, we define some helper functions we need for computing the global declarations *Dec* for a given Creol class with the signature '**class** $C(a^*)$ **implements** i^* **begin** v^* m^* **end**'. We use $E(m)$ as a short hand to return the element E returned by $\underline{[\![m]\!]}$ for $m \in m^*$.

$$
\begin{array}{lcl}
\mathit{starts} & = & \{E(m) \mid m \in m^*\} \\
\mathit{tasks} & = & \{N \mid \textbf{op}\ N == S \in m^*\} \ \cup\ \{n \mid (n, en) \in E(m), m \in m^*\}
\end{array}
$$

$$
\mathit{labels} \ = \ \bigcup \mathit{labels}(S) \ \text{ for } \ \textbf{op}\ N == S \in m^*
$$

$$
\begin{array}{ll}
\mathit{labels}(s; S) & = \mathit{labels}(s) \cup \mathit{labels}(S) \\
\mathit{labels}(\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2) & = \mathit{labels}(S_1) \cup \mathit{labels}(S_2) \\
\mathit{labels}(\textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}) & = \mathit{labels}(S) \\
\mathit{labels}(t!x.m(d)) & = \{t\} \\
\mathit{labels}(\_) & = \emptyset
\end{array}
$$

$$
\mathit{methods} \ = \ \bigcup \mathit{methods}(S) \ \text{ for } \ \textbf{op}\ N == S \in m^*
$$

$$
\begin{array}{ll}
\mathit{methods}(s; S) & = \mathit{methods}(s) \cup \mathit{methods}(S) \\
\mathit{methods}(\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2) & = \mathit{methods}(S_1) \cup \mathit{methods}(S_2) \\
\mathit{methods}(\textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}) & = \mathit{methods}(S) \\
\mathit{methods}(t!x.m(d)) & = \{m\} \\
\mathit{methods}(\_) & = \emptyset
\end{array}
$$

Figure 6.7: Helper functions in encoding global declarations. An underscore represents other possibilities as parameter.

Given a minimum d_n, a maximum d_x and an initial value d_i for deadline values, the global declarations *Dec* are defined as follows. The global declarations *Dec* consists of the following elements. For easier reading, we put them in a list. Treating each item as a set, *Dec* is formally defined as their union. The functions *IntT*, *IntL* and *IntM* below produce unique integer values, starting from zero (*IntL* starts from 1) and incrementing by one every time they are called.

- global constants. The # function returns the number of elements in a set.
  `const int MSG =` $Max(\ \#(\mathit{methods} \setminus \mathit{tasks}), \#(\mathit{tasks})\ )$`;`
  `const int nObj =` $\#(a^*) + 1$`;`
  `const int LBL =` $\#(\mathit{labels})$`;`
- the global clock used by all method automata and the deadline variable.
  `clock c; meta int[`d_n, d_x`] deadline;`
- a unique number for each task and subtask.
  $\{$`const int` t `=` $\mathit{IntT}()$`;` $\mid t \in \mathit{tasks}\}$
- a unique number for each label, and a boolean array.
  $\{$`const int` l `=` $\mathit{IntL}()$`;` $\mid l \in \mathit{labels}\}$
  `bool labels[LBL+1][nObj];`
- an array for each variable, either a bool or an int.
  $\{\mathit{type}\ \mathit{name}$ `[nObj];` $\mid \mathit{name} : \mathit{type} \in v^*\ \}$
- a bool to indicate method completion, helping scheduler decide whether to issue a reply signal.
  `bool complete[nObj];`

- a unique number for each method called on the objects provided as arguments.
  {`const int` m `=` *IntM*`();` | $m \in$ *methods* \ *tasks* }

- the channels used by the scheduler.

```
chan delegate [MSG+1][nObj];
chan invoke [LBL+1][MSG+1][nObj][nObj];
urgent chan start [MSG+1][nObj];
chan finish[nObj];
chan wait[LBL+1][nObj];
urgent chan resume[LBL+1][nObj];
chan reply[LBL+1][nObj];
```

- code to help the scheduler start the tasks correctly.

```
bool isEnabled (int msg, int self) {
  {if (msg == n) return en;      | for all (n, en) ∈ starts }
  return true;
}
```

6.1.3 End-to-end Deadlines

When calling a method with a deadline and waiting for the response later in the method, we are, in fact, enforcing an end-to-end deadline on the method call. This deadline must be enforced on the behavioral interface for the arrival of the reply signals (with the assumption that only individually schedulable objects will be used together). This is crucial to the schedulability of the caller object. If no such restriction on the arrival of the reply signal is enforced, the caller may be provided with the reply too late, therefore it will miss the deadline required by the method that is waiting for the reply. This restriction is reflected in the extended compatibility check (cf. Section 6.2).

In this thesis, we did not consider any network delays which may contribute to the end-to-end deadlines of method calls. Network delays can be added by means of another automaton modeling the network or by means of a coordination languages like Reo [4] extended with Quality-of-Service analysis [5].

MutEx Example. As another example for the translation algorithm, we use the `MutEx` example. We repeat the Creol code to make reading and comparison easier. Figure 6.9 shows the automaton generated for the method reqL from Figure 6.8. The automaton for reqR would be similar. Since reqL has two release points, it results in three tasks; one for the method itself, and one for each possible subtask generated by releasing the processor. Figure 6.10 shows the function modeling the enabling conditions for `MutEx`. Since the enabling condition for the main tasks is true, it is handled as the default case in this function.

```
class MutEx (left: Entity, right: Entity) begin
  var taken : bool
  op initial ==
     duration(1,1); taken := false
  op reqL ==
     duration(1,2); await !taken;
     duration(4,4); taken := true; l ! left.grant(10); await l?;
     duration(2,3); taken := false
  op reqR ==
     duration(1,2); await !taken;
     duration(4,4); taken := true; r ! right.grant(10); await r?;
     duration(2,3); taken := false
end
```

Figure 6.8: The Creol code for `MutEx`.

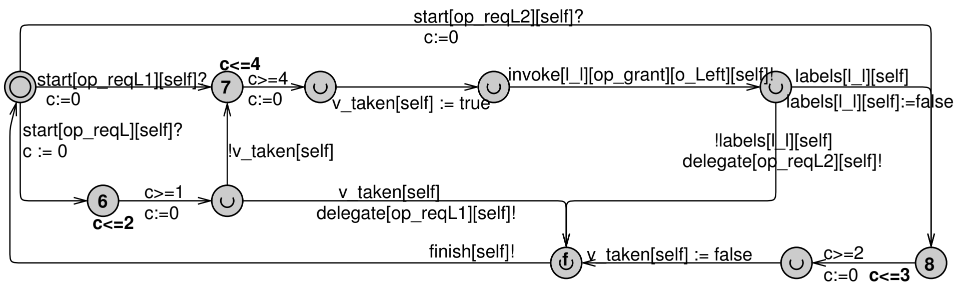


Figure 6.9: The automaton generated for the method reqL above; reqR is similar to reqL. The enabling conditions for reqL, reqL1 and reqL2 are true, !taken and l?, respectively.

```
bool isEnabled (int msg, int self) {
    if (msg == reqL1) return !v_taken[self];
    else if (msg == reqL2) return labels[l_l][self];
    else if (msg == reqR1) return !v_taken[self];
    else if (msg == reqR2) return labels[l_r][self];
    return true;    // other tasks
}
```

Figure 6.10: The function capturing enabling conditions for `MutEx` tasks.

6.2 Checking Compatibility in Full Creol

Once schedulability of individual objects is established, these objects can be put together to form a system. Then we should check if the usage of the objects in the system is compatible with their behavioral interfaces.

A complication in forming a system is making methods send reply signals. If when calling a method, a label t is assigned to the method call, the called method should set the variable corresponding to this label, namely labels[l_t], to true and send a reply signal by synchronizing on the reply channel when the method finishes. This is done by the scheduler as explained in previous sections. In individual object analysis, reply signals for outgoing messages should be captured in the behavioral interface; this will include an upper bound for receiving the replies in time.

When testing for compatibility, we should also check if the correct replies to methods are received. To do so, the objects need to synchronize with the test case on reply channel (as well as invoke and delegate channels). In the case of sending messages, the test-case checks the compatibility of the deadline values. For replies, we check if the correct variable (corresponding to the correct label) is set to true. Compatibility thus ensures that replies arrive in time with respect to the end-to-end deadline requirements (cf. Section 6.1.3).

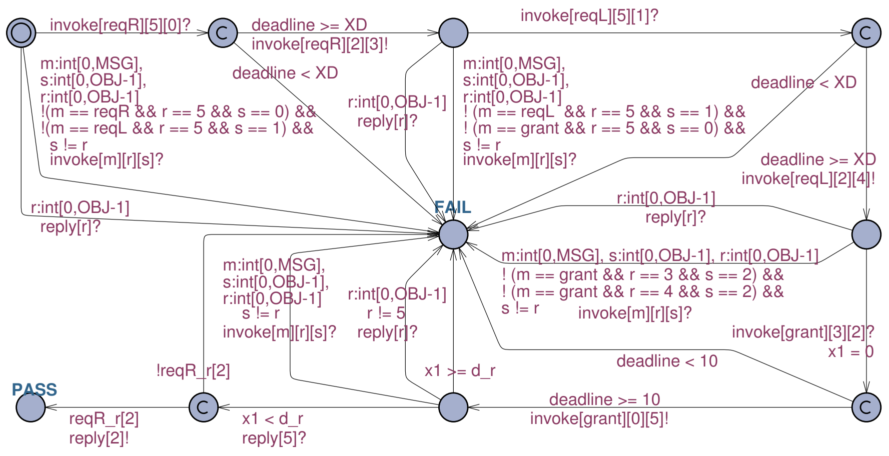


Figure 6.11: Test case for compatibility including reply to messages

Figure 6.11 shows a test case for checking the compatibility of a system using the `MutEx` object. In this system, the deadline for grant method is assumed to be 10 in the behavioral interface of the Entity class. The test case does not show the **Inconc** location (cf. Chapter 5). Any message or reply not foreseen in the behavioral interfaces will lead to **Fail**.

6.3 Checking Conformance between Real-Time Creol and Timed Automata[2]

In this section, we explain an alternative approach to schedulability analysis of Creol models. First we explain the overall methodology for the schedulability analysis of a Real-Time Creol model. We consider open systems and model the real-time pattern of incoming messages in terms of a timed automaton (the behavioral interface of the Creol model). Next we develop on the basis of sequence diagrams, which describe the observable behavior of the Creol model, automata abstractions of its overall real-time behavior. We analyze the schedulability of the product of this abstraction and the given behavioral interface (in for example UPPAAL). To ensure that the analysis results carry over to the Creol model, we define conformance between the Real-Time Creol model and its timed automaton abstraction with respect to the given behavioral interface in terms of inclusion of the timed traces of observable actions. This alternative approach is useful when the timed automata semantics presented before is not abstract enough, or if a Creol class contains dynamic behavior like object creation (which is not supported by the automatic automata generation algorithm).

More specifically, let $\mathbf{C}$ denote a Creol model, i.e., a set of Creol classes, $\mathbf{B}$ a timed automaton specifying its behavioral interface and $\mathbf{A}$ a timed automata abstraction of the overall behavior of $\mathbf{C}$. We denote by $O(\mathbf{A} \parallel \mathbf{B})$ the set of timed traces of observable actions of the product of the timed automata $\mathbf{A}$ and $\mathbf{B}$. The set of timed traces of the timed automaton $\mathbf{B}$ we denote by $T(\mathbf{B})$. Further, given any timed trace $\theta \in T(\mathbf{B})$, the Creol class $\mathit{Tester}(\theta)$ denotes a Creol class which implements θ (see, for example, the class *Tester* in Figure 6.13). This class simply injects the messages at the times specified by θ. We denote by $O(\mathbf{C}, \mathit{Tester}(\theta))$ the set of timed traces of observable actions generated by the Real-Time Maude semantics of the Creol model $\mathbf{C}$ driven by θ. We now can define the conformance relation $\mathbf{C} \leq_{\mathbf{B}} \mathbf{A}$ by

$$O(\mathbf{C}, \mathit{Tester}(\theta)) \subseteq O(\mathbf{A} \parallel \mathbf{B}),$$

for every timed trace of observable actions $\theta \in T(\mathbf{B})$.

In this section we illustrate a method for testing conformance by searching for *counter-examples* to the conformance in terms of our running example. Note that a counter-example to the above conformance consists of a timed trace $\theta \in T(\mathbf{B})$ such that $O(\mathbf{C}, \mathit{Tester}(\theta)) \setminus O(\mathbf{A} \parallel \mathbf{B}) \neq \emptyset$. We explain this section with the help of the thread-pool example. The Creol code for thread-pools is defined in Section 2.3.2 and the automata models are defined in Section 3.4; we use the parallel threads model. An intricacy in this section is the different formal semantics of the two levels, namely, Creol's semantics in rewrite logic and time automata.

[2]This section is an improvement and extension of the results published in [15].

6.3.1 Testing Conformance

For testing conformance, we take a trace from the abstract automata model and augment it with ready set information, thus showing allowed behavior at each step along the given trace; then we check whether the concrete Creol model stays within the allowed behavior when restricted to the particular trace. A complication in computing ready sets is the nondeterminism present at both levels.

To generate a test case, we first obtain a timed trace $\theta = (t_1, a_1), \ldots, (t_n, a_n)$ by simulating the abstract timed automaton model $\mathbf{A}$ together with the behavioral interface $\mathbf{B}$. To this end, we add a dummy automaton with a fresh clock `global` and an integer `time` which is incremented every time unit. This way we can find the absolute time interval in which every action in the trace has happened. In order to be able to search for a counter-example to conformance, we augment the trace with ready sets of observable actions w.r.t. the behavioral abstraction of the Creol model.

Definition 6.3.1 (Ready Sets). *Given a trace $\theta = (t_1, a_1), \ldots, (t_n, a_n)$, the ready set R_i for each step i is defined as follows:*

$$R_i = \{b \mid \exists t_b \leq t_i \ . \ (t_1, a_1), \ldots, (t_{i-1}, a_{i-1})(t_b, b) \in O(A \parallel B)\}$$

We further define ready traces of θ with respect to $A \parallel B$ as:

$$O_R(\theta) = \{(t_1, a_1), \ldots, (t_{i-1}, a_{i-1})(t_b, b) \mid b \in R_i \wedge t_{i-1} \leq t_b \leq t_i\}$$

□

The ready set R_i is the set of all possible actions b for which there exists $t_b \leq t_i$ such that the trace $\theta_b = (t_1, a_1), \ldots, (t_{i-1}, a_{i-1})(t_b, b)$ is an observable trace in $A \parallel B$. In presence of nondeterminism, this definition takes every trace that partly matches θ into account. This is different from the normal definition of ready sets that record the deviating actions only along the main trace.

6.3.2 Generating a Test Case

Suppose `a_f` is a flag denoting whether the observable action a has occurred in the interval between t_{i-1} and t_i. The observable action a is in R_i if the following timed reachability property holds:

"`E<>` t_{i-1} `<= global && global <` t_i `&& a_f`",

Instead of checking this property directly for every action, we encode it into one automaton as explained below (see Figure 6.12 for an example). This way, we avoid the need to add flags like `a_f` for every observable action and to go deep in the model to set it true when the corresponding action happens.

The algorithm to construct the automaton in Figure 6.12 for generating ready sets is as follows. Given a trace $\theta = (t_1, a_1), \ldots, (t_n, a_n)$, we first create a linear timed automaton T_θ with the locations $L = \{l_i \mid 1 \leq i \leq n+1\}$. By going from

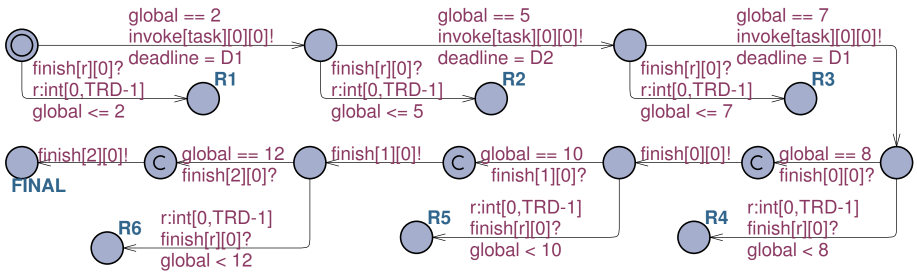


Figure 6.12: Generating ready sets.

l_i to l_{i+1}, this automaton should ensure that action a_i happens at time $x == t_i$. This is done differently for inputs and outputs. Since the abstract Uppaal model **A** (i.e., excluding the behavioral interface) is input-enabled, the input actions only need to inject the task at the required time; namely with a transition from l_i to l_{i+1} with an `invoke` action. This transition should provide the required deadline. The output action `finish` is, however, produced by a task and consumed by the scheduler. To intercept this action, this automaton first mimics the scheduler by accepting the action, i.e., `finish?`, and then it mimics the task by issuing `finish!`.

We add to T_θ a location R_{ij} for each time interval between t_{i-1} and t_i and for each observable output action $o_j \in O(\mathbf{A} \parallel \mathbf{B})$, with one transition from l_i to R_{ij} with a guard `global` $\leq t_i$ accepting the output action o_j; if a_i is the same as o_j, this transition is guarded by `global` $< t_i$. In our example, there is only one observable output action namely `finish`, but since a task can be taken by different threads, the `finish` action can be issued by different threads; therefore, the transitions for receiving this action should allow any thread identity `r` between 0 and `TRD-1`.

Finally, the reachability of the location R_{ij} implies that o_j must be included in the ready set R_i. We observe that in our example, only `R3` and `R4` are reachable; this is due to the possibility of finishing the first task instance in the interval $[7, 8]$. The consequent task instances can finish in the intervals $[10, 11]$ and $[12, 13]$, therefore, their completion does not contribute to an action in the ready sets. The test case including the ready sets and deadlines is:
$(2, \mathit{invoke}(D1), \{\})$ $(5, \mathit{invoke}(D2), \{\})$ $(7, \mathit{invoke}(D1), \{\mathit{finish}_0\})$ $(8, \mathit{finish}_0, \{\mathit{finish}_1\})$
$(10, \mathit{finish}_1, \{\})$ $(12, \mathit{finish}_2, \{\})$

With a test case generated as explained above, we can find counter-examples to conformance. To do so, we use the test case as a driver for the Creol class and check whether the observable output behavior of the Creol class, given the input behavior of the test case, is included in the ready set at each step. The theorem below shows the soundness of this testing method.

Theorem 6.3.2 (Soundness). *Any action in $O(\mathbf{C}, \mathit{Tester}(\theta))$ that is not in $O_R(\theta)$ is a counter-example to refinement.*

```
class Tester (mut:ResourcePool) begin
   op run ==
      duration(2,2);
      !mut.invoke(6);
      duration(3,3); // 5−2 = 3
      !mut.invoke(6);
      duration(2,2); // 7−5 = 2
      !mut.invoke(6)
end
```

Figure 6.13: Tester Class

The proof follows from the definition of ready sets such that every action possible in $A \parallel B$ given a prefix of θ is included in the corresponding ready set. Therefore, any other action at that given point is a counter-example to conformance.

6.3.3 Executing a Test Case in RT-Maude

Executing a test case amounts to injecting the inputs at the right times and looking for the right outputs at the right times. The system is input-enabled, so it accepts all the inputs. If the system under test cannot produce the expected output at the right time, the test fails. If along the test execution, the system under test can do an observable action that is neither the expected output nor in the ready-set, it is a counter-example to conformance. If the system can produce all expected outputs and no counter-example is found, the test passes in the sense that we are more confident that refinement holds and that the Creol model is schedulable. Notice that a counterexample to refinement does not necessarily imply non-schedulability in itself, but it shows an execution path that is likely to miss a deadline. We demonstrate this with the test-case from the previous subsection, repeated below:
$(2, \mathit{invoke}(D1), \{\})$ $(5, \mathit{invoke}(D2), \{\})$ $(7, \mathit{invoke}(D1), \{\mathit{finish}_0\})$ $(8, \mathit{finish}_0, \{\mathit{finish}_1\})$ $(10, \mathit{finish}_1, \{\})$ $(12, \mathit{finish}_2, \{\})$

We encode the input behavior given in the test-case as a complementary class that calls the methods of the model under test at the required times. For our running example, the code for the trace from previous subsection is given in Figure 6.13 (assuming $D1 = D2 = 6$).

By generating one instance of the ResourcePool class (with size 3 which gives us a schedulable Uppaal model, cf. Section 4.3.2) and one instance of the Tester class, we can check the output behavior of the Creol model against the test-case with consecutive search commands in Real-Time Maude as shown in Figure 6.14. The command **tsearch** searches for a requested configuration from an initial one within a time bound; for example, the first **tsearch** command starts the search from the initial configuration of the model init and looks for a configuration in which a finish method is sent and this search is bounded to 2 time units.

tsearch [1] {init} $\longrightarrow^*$ {Conf1 finish(T)} **in time** $\leq$2
If this search is successful, then we have a counter-example; otherwise, we continue with the following search:
tsearch [1] {init} $\longrightarrow^*$ {Conf1 invoke(2)} **in time** $\leq$2
If this search is not successful, then the test fails; otherwise, if Maude answers C1 $\rightarrow$ Conf1 then we continue with the following search:

tsearch [1] {C1 invoke(2)} $\longrightarrow^*$ {Conf2 finish(T)} **in time** $\leq$3
If this search is successful, then we have a counter-example; otherwise, we continue with the following search:
tsearch [1] {C1 invoke(2)} $\longrightarrow^*$ {Conf2 invoke(5)} **in time** $\leq$3
If this search is not successful, then the test fails; otherwise, if Maude answers C2 $\rightarrow$ Conf2 then we continue with the following search:

tsearch [1] {C2 invoke(5)} $\longrightarrow^*$ {Conf3 invoke(7)} **in time** $\leq$2
If this search is not successful, then the test fails; otherwise, if Maude answers C3 $\rightarrow$ Conf3 then we continue with the following search:

tsearch [1] {C3 invoke(7)} $\longrightarrow^*$ {Conf4 finish(8)} **in time** $\leq$1
If this search is not successful, then the test fails; otherwise, if Maude answers C4 $\rightarrow$ Conf4 then we continue with the following search:

tsearch [1] {C4 finish(8)} $\longrightarrow^*$ {Conf5 finish(T)} **in time** <2
If this search is successful, then we have a counter-example; otherwise, we continue with the following search:
tsearch [1] {C4 finish(8)} $\longrightarrow^*$ {Conf5 finish(10)} **in time** $\leq$2
If this search is not successful, then the test fails; otherwise, if Maude answers C5 $\rightarrow$ Conf5 then we continue with the following search:

tsearch [1] {C5 finish(10)} $\longrightarrow^*$ {Conf6 finish(T)} **in time** <2
If this search is successful, then we have a counter-example; otherwise, we continue with the following search:
tsearch [1] {C5 finish(10)} $\longrightarrow^*$ {Conf6 finish(12) **in time** $\leq$2

Figure 6.14: Executing the test-case for the thread-pools

In our case, the only observable output action is `finish`. To find a counter-example along this trace, we need to check whether a `finish` action can happen when it is not expected in the ready set, i.e., before time 2, between 2 and 5, between 8 and 10, or between 10 and 12. For each search command, we need to specify as time bound the duration since its start configuration, e.g., to search from C2 which is at time 5, we only need to search for another 2 time units to reach time 7. In the semantics of real-time Creol given in Chapter 2, each action is given a time-stamp which is written in parentheses in this figure. It is possible to write a meta-level Maude script to automate the consecutive execution of these search commands, such that each search starts from the resulting configuration of the previous one.

Local delarations of a scheduler automaton

The declarations below are generic and are used for any scheduler. The only model dependent part is the length of the queue in the first line.

```
const int MAX = 7;   // queue length; the only non-generic definition
int[0,MSG]    q  [MAX+1]; // one extra; see shift()
int[0,OBJ-1]  s  [MAX];    // sender
int[0,MAX-1]  ca [MAX];    // clock assigned
int[0,LBL]    l  [MAX];    // labels in sending messages; 0 = no label
int[0,MAX]   ret [MAX];    // return point for sync calls: syncORblock()

clock x[MAX];                    // queue clocks
int[MD,XD]  d[MAX];              // task deadlines queue
int[0,MAX]  counter[MAX];        // number of tasks assigned to a clock

int[0,MAX]  run=0;       // currently running task
int[0,MAX]  active=0;    // currently active task (may be blocked)
int[0,LBL]  res=0;       // resume a blocked local sync call
int[0,MAX+1]  tail;      // tail == MAX+1 means queue overflow
int[0,LBL]  rl;          // resume label

void initialize(){
   q[0] = op_init;
   d[0] = INIT_DEADLINE;    // defined globally
   s[0] = self;
   ret[0] = MAX;   // not a sync call
   q[1] = op_run;
   d[1] = RUN_DEADLINE; // defined globally
   s[1] = self;
   ret[1] = MAX;   // not a sync call
   ca[0] = 0;   // clock x[0] is assigned
   ca[1] = 1;   // clock x[1] is assigned
   counter[0] = 1;
   counter[1] = 1;
   l[0] = 0;
   l[1] = 0;
   tail = 2;
}

void insertInvoke(int m, int snd, int lbl){
   int c, i;
   if (tail < MAX) {
      c = MAX;
      for (i = 0; c == MAX; i++) {
        if (counter[i] == 0) {
          c = i;
      } }
      q[tail] = m;
      s[tail] = snd;
      l[tail] = lbl;
      ret[tail] = MAX;
      ca[tail] = c;
      x[c] = 0;
      d[c] = deadline;
      counter[c] = 1;
      // if all messages in queue are disabled, choose this one now
      if (run == MAX) run = tail;
   }
   tail++;
}

void insertDelegate(int msg){
   if (tail < MAX) {
      q[tail] = msg;
      s[tail] = self;   // delegation is a self call
      l[tail] = 0;   // no labels for delegation
      ret[tail] = MAX;
      ca[tail] = ca[run];
      counter[ca[tail]] ++;
   }
   tail ++;
}

void invokeReply(int lbl){
   // If no method is running...
   if (run == MAX){
      if (active != MAX){  // a blocked call has higher priority
         if (rl == lbl){ res = lbl; rl := 0; }
      }
      else         // select the first enabled method
         for(run=0;run<MAX && !isEnabled(q[run], self);run++);
} }

void syncORblock(int lbl){
   int i;
   run = MAX;      // block current method
   rl = lbl;
   if (labels[lbl][self]){ // continue if reply has already arrived
```

```
        res = lbl;
        rl := 0;
        return;
    }
    for (i=0; i<tail && run == MAX; i++){
        if (l[i] == lbl && s[active] == self){ // local sync call
            ret[i] = active;
            run = i;
            rl = 0;
    } }
    res = 0;
}

int contextSwitch(int next){
    int prev = run;
    if (ret[run] == MAX) {  // not a sync call
        run = next; // start the next method
    } else {     // in case of sync call
        res = l[run];  // resume ...
        active = ret[run];
        run = MAX;  // ... do not start a new method
    }
    return prev;
}

void shift(int a) {
    counter[ca[a]] --;
    if (counter[ca[a]] == 0)
      d[ca[a]] = MD;  // not assigned
    tail--;
    while (a < tail && a < MAX-1) {
        q[a] = q[a+1];
        ca[a] = ca[a+1];
        s[a] = s[a+1];
        l[a] = l[a+1];
        ret[a] = ret[a+1];
        a++;
    }
    q[a] = 0;    // not assigned
    ca[a] = 0;  // not assigned
    s[a] = 0;    // not assigned
    l[a] = 0;    // not assigned
    ret[a] = 0; // not assigned
}
```

Bibliography

[1] Gul Agha. The structure and semantics of actor languages. In *Proc. the REX Workshop*, pages 1–59, 1990.

[2] The Almende research company. `http://www.almende.com/`.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] Farhad Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14:329–366, 2004.

[5] Farhad Arbab, Tom Chothia, Sun Meng, and Young-Joo Moon. Component connectors with QoS guarantees. In Amy Murphy and Jan Vitek, editors, *Coordination Models and Languages*, volume 4467 of *LNCS*, pages 286–304, 2007.

[6] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Proc. Formal Methods for the Design of Computer, Communication, and Software Systems*, volume 3185 of *LNCS*, pages 200–236, 2004.

[7] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2003.

[8] Dave Clarke, Einar Broch Johnsen, and Olaf Owe. Concurrent Objects à la Carte. In Dennis Dams, Ulrich Hannemann, and Martin Steffen, editors, *Correctness, Concurrency, Compositionality: Essays in honor of Willem-Paul de Roever*, volume 5930 of *LNCS*, pages 185 – 206, 2010.

[9] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[10] Costas Courcoubetis and Mihalis Yannakakis. Minimum and maximum delay problems in real-time systems. *Formal Methods in System Design*, 1(4):385–415, 1992.

[11] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proc. ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[12] Frank de Boer, Tom Chothia, and Mohammad Mahdi Jaghoori. Modular schedulability analysis of concurrent objects in Creol. In *Proc. Fundamentals of Software Engineering (FSEN'09)*, volume 5961, pages 212–227, 2009.

[13] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In Rocco de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer-Verlag, March 2007.

[14] Frank S. de Boer, Immo Grabe, Mohammad Mahdi Jaghoori, Andries Stam, and Wang Yi. Modeling and analysis of thread-pools in an industrial communication platform. In *Proc. 11th International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *LNCS*, pages 367–386. Springer, 2009.

[15] Frank S. de Boer, Mohammad Mahdi Jaghoori, and Einar B. Johnsen. Dating concurrent objects: Real-time modeling and schedulability analysis. In *Proc. 21st International Conference on Concurrency Theory (CONCUR'10)*, volume 6269 of *LNCS*, pages 1–18, 2010.

[16] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.

[17] Immo Grabe, Mohammad Mahdi Jaghoori, Bernhard Aichernig, Christel Baier, Tobias Blechmann, Frank de Boer, Andreas Griesmayer, Einar Broch Johnsen, Joachim Klein, Sascha Klüppelholz, Marcel Kyas, Wolfgang Leister, Rudolf Schlatte, Andries Stam, Martin Steffen, Simon Tschirner, Xuedong Liang, and Wang Yi. Credo methodology. Modeling and analyzing a peer-to-peer system in Credo. In Einar Broch Johnsen and Volker Stolz, editors, *Proceedings of the 3nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS'09), ICTAC 2009 satellite workshop*, 2010. to appear.

[18] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 373–382, New York, NY, USA, 1995. ACM.

[19] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. In *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 77–117, 2008.

[20] Carl Hewitt. Procedural embedding of knowledge in planner. In *Proc. the 2nd International Joint Conference on Artificial Intelligence*, pages 167–184, 1971.

[21] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[22] Mohammad Mahdi Jaghoori and Tom Chothia. Timed automata semantics for analyzing Creol. In *Proc. Foundations of Coordination Languages and Software Architectures (FOCLASA'10)*, EPTCS 30, pages 108–122, 2010.

[23] Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia, and Marjan Sirjani. Task scheduling in Rebeca. In *Proc. Nordic Workshop on Programming Theory (NWPT'07)*, 2007. extended abstract.

[24] Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia, and Marjan Sirjani. Schedulability of asynchronous real-time concurrent objects. *J. Logic and Alg. Prog.*, 78(5):402 – 416, 2009.

[25] Mohammad Mahdi Jaghoori, Delphine Longuet, Frank S. de Boer, and Tom Chothia. Schedulability and compatibility of real time asynchronous objects. In *Proc. RTSS'08*, pages 70–79. IEEE CS, 2008.

[26] Claude Jard, Thierry Jéron, and Pierre Morel. Verification of test suites. In *Proc. Testing Communicating Systems (TestCom 2000)*, pages 3–18, 2000.

[27] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, 2007.

[28] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1-2):23–66, 2006.

[29] Christos Kloukinas and Sergio Yovine. Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In *Proc. ECRTS'03*, pages 287–294. IEEE CS, 2003.

[30] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *Model Checking Software, 11th International SPIN Workshop*, volume 2989 of *LNCS*, pages 109–126, 2004.

[31] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[32] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[33] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[34] Jennifer McManis and Pravin Varaiya. Suspension automata: A decidable class of hybrid automata. In David Dill, editor, *Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 105–117. Springer, 1994.

[35] José Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[36] Bertrand Meyer. *Eiffel: The language.* Prentice-Hall, 1992.

[37] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[38] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1–2):161–196, June 2007.

[39] Barbara Paech and Bernhard Rumpe. A new concept of refinement used for behaviour modelling with automata. In *Proc. FME '94*, pages 154–174. Springer-Verlag, 1994.

[40] Wolfgang Reisig. *Petri nets: an introduction.* Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[41] Bernhard Rumpe and Cornel Klein. Automata describing object behavior. In *Object-Oriented Behavioral Specifications*, pages 265–286. Springer, 1996.

[42] David P. L. Simons and Mariëlle Stoelinga. Mechanical verification of the IEEE 1394a root contention protocol using Uppaal2k. *STTT*, 3(4):469–485, 2001.

[43] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundamamenta Informaticae*, 63(4):385–410, 2004.

[44] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[45] Ingrid Chieh Yu, Einar Broch Johnsen, and Olaf Owe. Type-safe runtime class upgrades in Creol. In *Proc. FMOODS'06*, volume 4037 of *LNCS*, pages 202–217. Springer-Verlag, June 2006.

Samenvatting

In dit proefschrift hebben wij nieuwe technieken ontwikkeld voor het ontwerp van gedistribueerde software, d.w.z. software waarvan de verschillende onderdelen op verschillende machines worden uitgevoerd. Dergelijke software is van vitaal belang voor een diversiteit aan maatschappelijke toepassingen die in het algemeen uit een groot aantal complexe processen bestaan en een hoge mate van Quality of Service vereisen. Echter, de State of the Art in parallel en gedistribueerd programmeren wordt (nog steeds) bepaald door programmeertalen als Java die ondanks hun populariteit niet geschikt zijn voor een optimaal gebruik van de onderliggende hardware.

Uitgangspunt van dit proefschrift is de fundamentele observatie dat de Quality of Service van (gedistribueerde) software in hoge mate afhangt van de manier waarop processen worden geselecteerd voor uitvoering op de verschillende machines. Deze selectie wordt in de implementatie van de huidige programmeertalen overgelaten aan de onderliggende besturingssystemen (operating systems) en valt daarmee geheel buiten de controle van een specifieke toepassing.

Wij hebben onderzocht hoe dit aspect van de implementatie op een hoog abstractieniveau formeel beschreven kan worden en daarmee gecontroleerd. Het belangrijkste resultaat van dit proefschrift is een (prototype van een) nieuwe programmeertaal die de programmeur in staat stelt zelf te bepalen en te analyseren hoe de gedistribueerde processen voor uitvoering geselecteerd worden op basis van deadlines die de Quality of Service garanderen.

Dit opent ook de weg naar andere nieuwe en veelbelovende toepassingen, zoals software voor de ondersteuning van de continue planning van logistieke processen in een dynamische omgeving en de ontwikkeling van software voor multicore architecturen. Nederlandse bedrijven (DEAL Services) en researchcentra (het Uppsala Programming for Multicore Architectures Resrearch Center UPMARC) hebben hier al belangstelling voor getoond.

Abstract

The software today is distributed over several processing units. At a large scale this may span over the globe via the internet, or at the micro scale, a software may be distributed on several small processing units embedded in one device. Real-time distributed software and services need to be timely and respond to the requests in time. The Quality of Service of real time software depends on how it schedules its tasks to be executed. The state of the art in programming distributed software, like in Java, the scheduling is left to the underlying infrastructure and in particular the operating system, which is not anymore in the control of the applications.

In this thesis, we introduce a software paradigm based on object orientation in which real-time concurrent objects are enabled to specify their own scheduling strategy. We developed high-level formal models for specifying distributed software based on this paradigm in which the quality of service requirements are specified as deadlines on performing and finishing tasks. At this level we developed techniques to verify that these requirements are satisfied.

This research has opened the way to a new approach to modeling and analysis of a range of applications such as continuous planning in the context of logistics software in a dynamic environment as well as developing software for multi-core systems. Industrial companies (DEAL services) and research centers (the Uppsala Programming for Multicore Architectures Resrearch Center UPMARC) have already shown interest in the results of this thesis.

Curriculum Vitae

Mohammad Mahdi Jaghoori was born in Mashhad, Iran, on March 22, 1982 (Farvardin 2, 1361, Iranian calendar). He finished school in Mashhad and moved to Tehran in 1999 for university, where he started software engineering at the University of Tehran, Faculty of Engineering. After 4 years, he graduated and was accepted at Sharif University of Technology for Master's study. He finished his Master's in 2005. He continued his research for another six month in Tehran at the school of computer science, IPM. At the beginning of 2006, he moved to Mashhad for a short period and gave lectures at the University of Sadjad for one semester. Since November 2006, he started his PhD at the national Dutch center of mathematics and computer science, CWI, in Amsterdam. For three years, he worked as part of the European project *Credo*. He moved at his fourth year to the Leiden Institute for Advanced Computer Science (LIACS), Leiden University. This PhD thesis is the result of four years of research in The Netherlands.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications.* Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures.* Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability.* Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions.* Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments.* Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression.* Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages.* Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations.* Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics.* Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems.* Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication.* Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and*

Empirical Studies about the UML. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering,

در اینجا به عنوان حسن ختام، خداوند منان را به خاطر بزرگترین و شیرین‌ترین اتفاق زندگی‌ام که مربوط به اواخر دوران دکترایم می‌باشد شکرگزارم. و از همراهی‌ها و شکیبایی‌های همسرم کمال تشکر را دارم و امیدوارم که بتوانم پاسخگوی مهربانی‌هایش باشم. همچنین از خانواده همسرم و بار دیگر از خانواده خودم تشکر می‌کنم که چنین روزهای خوشی را برای ما فراهم کردند.

محمد مهدی جاقوری

آذر ۱۳۸۹

تشکر و قدردانی

در ابتدا مثل همیشه خدا را شکرگزارم که سایه لطف بیکران خود را بر سر ما افکنده، و همواره آرامش بخش و هدایتگر است. سپس از خانواده‌ام، به خصوص پدر و مادرم کمال تشکر و قدردانی را دارم که پیوسته حامی و پشتیبان من بوده و هستند.

بر خود لازم می‌دانم از سرکار خانم دکتر مرجان سیرجانی که در دوران دکترا همانند دوره‌های گذشته (کارشناسی و کارشناسی ارشد)، راهنمای من در زمینه‌های علمی و جنبه‌های عملی زندگی بوده‌اند نهایت تشکر را داشته باشم. همچنین از همراهی‌های جناب آقای دکتر فرهاد ارباب و سرکار خانم دکتر حمیده افسرمنش که علاوه بر مباحثات علمی به عنوان دو ایرانی، همواره سختی دوری از خانواده را بر من آسان نموده‌اند و از جناب آقای دکتر محمد رضا موسوی که به خصوص ماه‌های اول حضور در هلند، نقش موثری در آشنایی من با شرایط زندگی و کار داشته‌اند سپاسگزارم.

هرچند در دو سال اول زندگی‌ام در هلند، با ایرانیان زیادی در ارتباط نبودم اما پس از گذشت مدت زمانی با دوستان ایرانی زیادی آشنا شدم که دشواری‌های دوری از وطن و خانواده‌ام را برایم قابل تحمل نمودند. در اینجا به نشانه تشکر از همراهی‌ها و ساعات خوشی که در کنار هم داشتیم به اسامی تعدادی از دوستان اشاره می‌کنم: سارا رمضانی، وحید هاشمیان، اندیشه کمالی، سعید محمدیان، بهناز چنگیزی، میلاد نیکویی، نرگس ضرابی، شایان شاهند، کامران مسعودی، بهار مهمانی، سانلی فائز، کاملیا سجادی، ناصر آیت، آیلار سلطانی، جعفر تنها، لیلا شعیبی، حمید عباسی، سعید اسلامی، دوستان خارج آمستردام: حسین رحمانی، بهروز نوبخت، حامد یزدانی، حمید صدقی و اعظم مین باشی و تعداد بسیار زیادی دوستان دیگر. به علاوه شایسته است از دوستانی که به طور موقت در هلند بوده‌اند نیز یاد کنم: محمد ایزدی و فائزه آیت، محمد مهدیزاده، فاطمه قاسمی و ابوالحسن شمسایی و بهمن پوروطن. همچنین از خاطره‌انگیزترین روزهای زندگی‌ام در خارج از کشور سفرهایی بود که به همراه همسفران خوبی همچون مرتضی منعمی زاده، علی مرجوی، علی جان نثاری و حسین حجت داشته‌ام.

Universiteit Leiden

زمان در خدمت شما

امکان‌سنجی زمان‌بندی سرویس‌های توزیع شده با در نظر گرفتن زمان

پایان نامه

جهت اخذ مدرک دکترا از دانشگاه لایدن

محمد مهدی جاقوری

تاریخ دفاع

۲۹ آذر ماه ۱۳۸۹