Institute of Software Technology
Department of Programming Languages and Compilers
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis Nr. 3372

Visualization of Scheduling in Real-Time Embedded Systems

Peter Munk

Course of Study: INFOTECH

Examiner: Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder

Supervisor: Dipl.-Inf. Mikhail Prokharau

Commenced: July 17, 2012

Completed: January 16, 2013

CR-Classification: D.4.1, D.4.8, C.4.d, I.6.8.i

Abstract

Many embedded systems, especially real-time systems, are used in safety-critical applications such as cars and aircraft. The consequences of different scheduling algorithms for such systems have to be properly understood. Software which simulates scheduling processes supports the research and development of new scheduling policies. It may be used for educational purposes, as simulation and visualization enhance the understanding of the consequences of scheduling decisions. This thesis introduces new, flexible, and extensible discrete event-driven simulation software. The software focuses on but is not limited to scheduling policies primarily used in real-time and embedded systems. Contrary to most existing scheduling simulators, it synchronously simulates and visualizes the current simulation result. In order to inspect the current scheduling situation in closer detail, the software is designed to allow pausing the simulation automatically on the occurrence of specific points of interest or manually at an arbitrary time. The simulation model is specified by a human-readable file which is loaded by the software. During the simulation, elements are added to the simulation model and each element is visualized. The simulation meta-model is designed to support a variety of system configurations. It supports shared resources with several units as well as multiple processing units. A set of scheduling policies and resource access protocols for single-core systems as well as a basic multi-core protocol were implemented and demonstrate the functionality of the software.

Acknowledgement

I want to thank Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder for giving me the possibility to write my master thesis at the Institute of Software Technology, Department of Programming Languages and Compilers at the University of Stuttgart.

Furthermore I would like to thank my supervisor Dipl.-Inf. Mikhail Prokharau for his continuous supervision and insightful guidance with fruitful discussions, inspiring suggestions and valuable reviews throughout my work on this thesis.

A special thank you goes to Sandra Erne for both, proof-reading this work and always having a sympathetic ear.

I am profoundly grateful to my parents Roland and Mechthild Munk for their constant encouragement and support during the past 26 years.

It is my pleasure to express my gratitude to all the people who contributed, in whatever manner, to the success of this work.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Motivation

In 2010, more than 98% of all produced processors were used in an embedded context [BBB$^+$10]. These embedded computers are not visible to the user at first glance; however, their correct function is essential for the whole system to work properly. Many embedded systems are used in critical applications, where a failure would lead to severe consequences.

Apart from computational correctness, it is often required that the result is available within strict time constraints [BW01]. Such systems are referred to as real-time systems [BW01]. The verification of their correct function is of paramount importance if they are used in safety-critical environments. To guarantee compliance with timing requirements in systems with a limited number of processing units and resources shared by tasks, special scheduling policies are necessary.

In recent years, the number of processing units, or cores, per processor has increased not only in conventional, but also in the embedded environment [DB11]. Many restrictions which hold for systems with only a single core do no longer apply for multi-core systems. Thus a new set of problems arises, especially in the field of real-time scheduling.

To ease the research and development of new scheduling policies solving such problems, software tools which simulate the scheduling process and are not specific to only one policy have been created since the end of the 1990s [GBA$^+$97]. Such tools can also be used for educational purposes, where a simulation and graphical representation enhances the understanding of otherwise abstract consequences.

Some of the existing simulation tools already support system models containing shared resources and multiple cores or allow new scheduling policies to be added. However, almost all existing simulation tools are decoupled from a visual representation of their results. They either do not provide a visualization at all and assign this job to other software products, or they generate a visualization after the simulation result has been calculated. However, presenting a visualization synchronously to the simulation relieves the user of the need to specify the time and duration of the part of interest as long as the simulation can be paused for closer examination at any time. Additionally, a “live” representation showing the scheduling decisions at original speed or in slow motion is more demonstrative than a static visualization.

1.2 Objective

The aim of this work is to develop a software to simulate and visualize arbitrary scheduling processes, especially in real-time embedded systems. The input elements of the simulation and their properties have to be file-oriented in a format that is easily maintained by humans.

The software has to support periodic and sporadic tasks, shared resources, and the following scheduling policies:

- cyclic executive
- fixed-priority preemptive with
  - simple priority inheritance
  - transitive priority inheritance
  - Original Ceiling Priority Protocol (OCPP)
  - Immediate Ceiling Priority Protocol (ICPP)
- Eclipse Modeling Framework (EMF)

Apart from an appealing and informative graphical representation, the simulation and the visualization of the simulation results have to be in synchronization. It must always be possible to pause the simulation, inspect or discuss a certain event like a scheduling decision in detail, and continue afterwards. In addition, the simulator has to recognize such events itself and notify the user of them. Such points of interest could be a deadline miss or a priority inversion while accessing a shared resource. Of course, the user has to be able to disable any of these automatic pause conditions at any time in order to concentrate on a specific detail.

A very important requirement is the extensibility of the software to be built. It must be possible to define an arbitrary number of cores, tasks, and shared resources. Furthermore, it has to be possible to add new scheduling algorithm implementations to the simulation software. To support such extensions, a software architecture of high quality is imperative.

In order to support multiple platforms, the chosen programming language is Java.

1.3 Structure

The remainder of this thesis is structured as follows. In order to establish a common notation, the required terms are defined in chapter 2. The project management and software engineering methods applied in this thesis are introduced in chapter 3. An overview of existing simulation and visualization software of scheduling in real-time systems is given in chapter 4. Before designing the program in chapter 6, existing frameworks and libraries from which the software could benefit are evaluated in chapter 5. Chapter 7 deals with implementation details of the software. Some key aspects and the functionality of scheduling policies implemented during the development of this project are presented in chapter 8. Before concluding the work and discussing topics of future research in chapter 10, the validation and test of the program are described in chapter 9.

2 Definitions

This section introduces and defines terms in order to establish a common notation. These elements include embedded systems, real-time systems, tasks, cores, resources, and schedulers.

2.1 An Embedded System

An *embedded system* is an information processing system encapsulated in a fixed context, dedicated towards a predefined functionality, and mostly not directly visible to the user [Mar06]. Typical contexts where embedded systems find application in are transportation systems like cars, trains, or aircraft as well as production and process control systems. Telecommunication equipment like Internet routers as well as consumer electronics like smart phones, dish washers, or TVs often contain embedded systems, too.

2.2 A Real-Time System

Many embedded systems must meet real-time constraints. Such systems are called *real-time systems*. According to Burns and Wellings, a real-time system is defined as “any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified delay” [BW01, p. 2].

In consequence, the correctness of a real-time system depends not only on the logical correctness of the computation, but also on the time at which these results are calculated [BW01].

For example, if the air bag electronic control unit (ECU) of a car does not respond to crash-indicating sensor signals within a given time span, i. e. the *deadline*, it might have fatal consequences for the passengers. If the signal decoder in a digital TV does not decode the input stream within a certain time, the output might be disruptive and unpleasant to watch.

The former is a typical example of a *hard* real-time system. If such a system misses deadlines, the consequences are serious and hazardous. The latter is an example of a *soft* real-time system, where occasionally missing a deadline can be accepted [BW01].

2.3 Task

The functionality of a real-time system is provided by the software controlling the system's hardware and peripherals. Typically, several jobs are handled by a single embedded system, e. g. controlling the mechanical parts of a washing machine and displaying the left-over time of the washing program. In order to encapsulate the different jobs and enhance reuseability and maintainability, the jobs are modularized and each module is called a *task*.

In other words, "a task is a software entity or program intended to process some specific input or to respond in a specific manner to events conveyed to it" [Nis97, p. 180].

A task typically consists of a sequence of commands, which include executions and resource interactions. An execution command consists of an arbitrary sequence of instructions, whereas each instruction must be part of the processor's instruction set. A resource interaction takes place if a sequence of instructions is nested between a previous resource request command and a succeeding release resource command.

Tasks occur either periodically or sporadically [Liu00, p. 40]. Whilst periodic tasks repeat at fixed times like multiples of a timer interrupt, sporadic tasks arise at random times, like an interrupt from an external sensor [BW01, p. 433]. As mentioned in the previous section, a real-time environment imposes strict timing requirements which have to be met for a successful execution. These requirements are taken account for by assigning a deadline to a task [Liu00, p. 41]. Sporadic tasks are called aperiodic, if they have a soft or no deadline instead of a hard deadline [Liu00, p. 42].

During each periodic or sporadic instance of a task, different states are traversed. The states and transitions used in this thesis are an adaption of the basic state diagram for Ada tasks presented by Burns and Wellings in [BW07, p. 181]. The state diagram is depicted in figure 2.1 on the facing page.

The default state of a task is *non-existing*, which means that the scheduler is totally unaware of its existence. A task comes into existence at a predefined time if it is periodic, or as soon as an external event or interrupt occurs if it is sporadic. The creation of a task is usually performed by the operating system (OS), which allocates memory space to the task and loads its code. Subsequently, the task is in the state *created*. The scheduler is now responsible to set the state of the task to *ready* when the task is prepared to be executed. Depending on the scheduling policy, one of the *ready* tasks is selected to be executed. The state of this task is set to *running*. If there is a higher priority task to execute while the task is currently running, the scheduler might preempt the task and set its state to *ready* again. If the task requests access to a shared resource which is currently locked by another task, it cannot continue with its execution and is set to the *blocked* state by the scheduler. Once the shared resource is unlocked, the task is set to the *ready* state again. As soon as the task finishes its execution, its state is set to *terminated*. The scheduler has to decide when to delete the task totally, i. e. set its state to *non-existing*.

The transition from the state *blocked* to the state *ready* is required if a blocked task is waiting on a lower priority task which itself is preempted by an even higher priority task. According to the definition of direct blocking, the state of the blocked task is changed to *ready*, too. More information about direct blocking is given in section 8.8 on page 80.

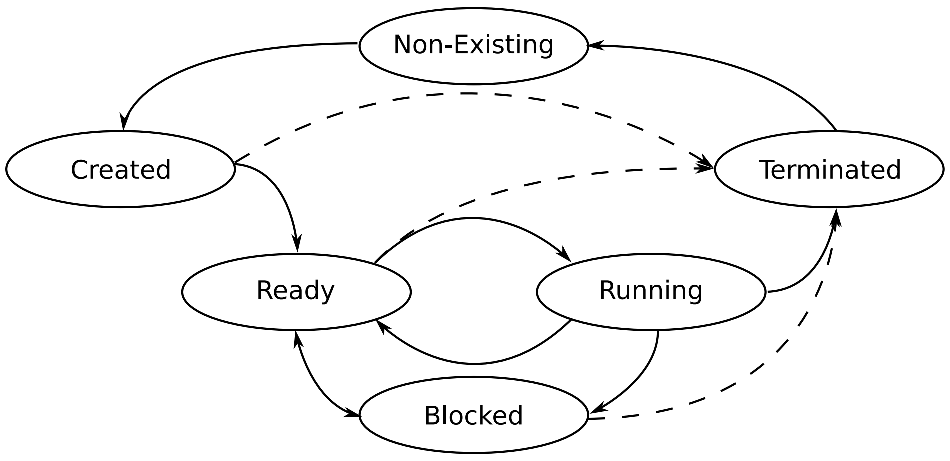


Figure 2.1: The state diagram of a task. Solid lines represent normal state changes, dashed lines represent state change in case of an exception.

In case an exception occurs or the task is "killed" from the outside, the task state has to be changed to *terminated*, independent of the state it was in before. In the figure 2.1, these transitions are indicated with dashed lines in contrast to the previously discussed, regular transitions.

2.4 Core

Since the invention of multi-core processors, it seems to be reasonable to use the term *core* for the processing unit of a system instead of the term *processor*.

When a task is in the state *running*, its commands are executed by the core it is scheduled on. If there are multiple tasks, at any given time only one task can physically be executed on one core. The functionality of the system might, however, require multiple tasks to be executed at the same time, e. g. a modern control unit of a combustion engine has to control the fuel injection and at the same time it has to provide diagnostic data. Given only one core, the only way to execute both tasks *concurrently* is to switch between the executions of both tasks. Therefore, "the term **concurrent** indicates potential parallelism" [BW07, p. 180]. Since the tasks might have deadlines, the scheduler has to take care that the switching is performed at reasonable times.

If more than one core is available in the embedded system, tasks can run truly in parallel. Such systems are referred to as multi-core systems. In general, as many tasks as there are available cores in a multi-core system can execute in parallel.

2.5 Resource

In contrast to a core, a *resource* does not execute the instructions of a task; however, it is necessary for the task to make progress [Liu00, p. 35]. Typical examples of resources are memory, semaphores, sensors, and actuators. When two or more tasks access one resource, the resource is called a *shared* resource and special attention is required. Suppose a task writes a calculation result to a memory location and is preempted by another task, which overwrites this memory location. Once the first task is running again, the further calculation is incorrect due to the overwritten value read from the memory. Such situations are called *race conditions*. Depending on the memory model of the programming language or platform, the consequences of a race condition might be completely undefined, which is why it has to be avoided under any circumstances in most cases.

That is why shared resources are guarded with constructs like semaphores and have impact on the scheduling policy, too [BW01]. In the example above, if the scheduler would not have preempted the first task because it had locked the memory or there had been a semaphore protecting the memory location, the race condition could have been avoided.

A shared resource can also contain multiple units, e. g. a printer pool with multiple printers [Liu00, p. 36]. In case there are fewer tasks than units, the access to the shared resource still has to be managed in order to ensure that two tasks never use the same unit simultaneously.

2.6 Scheduler

The term *scheduling policy* or *scheduler* was frequently used in the previous sections. In general, the scheduler is part of the OS and its main purpose is to determine which task is executed on which core at which time [Tan09]. Even if there is no OS at all, e. g. in very simple systems like a washing machine, a scheduling policy is still required to allow the system to contain more than one task.

3 Project Management and Software Engineering

This project includes the development and implementation of a new software product of significant size. In order to guarantee a successful and in-time delivery, the project has to be thoroughly managed. This chapter presents the project management and software engineering methods which were applied for that reason.

3.1 Project Management

Since the software is developed by only one person, heavy-weight project management methods like the classical plan-driven approach seem inappropriate, as they require too many resources.

Agile project management methods try to avoid time-consuming processes and can be combined with agile software development [Som10]. A well-known agile project management method is called *Scrum*. It establishes three phases: the planning phase, the sprint cycles, and the project closure phase.

During the planning phase, general objectives are assessed, and the software architecture design is created. Arbitrarily many sprint cycles follow this first phase, each of which has a typical duration of about two to four weeks. Within each cycle, the features to be developed are selected in cooperation with the customer and based on the results of the planning phase. These functionalities are then implemented and afterwards presented to the customer. If the customer is not satisfied with the result, the relevant work items are newly prioritized and become part of the next cycle. As soon as all features are implemented, the project closure phase is entered and the project is completed by writing the documentation and user manuals and by assessing the lessons learned of the project. More information about project management methods and Scrum is given in [Som10].

This project was managed by methods derived from Scrum. At the beginning, the general objectives, described in section 1.2 on page 14, were discussed. Regular meetings with the customer, i. e. the supervisor, were arranged, which included a presentation of the current results and a discussion of the requirements and features to be implemented next.

3.2 Software Engineering Methods

The initial requirements were formulated by just a few bullet points. Experience shows that some of these points might be subject to change during the development of the software, e. g.

because the formulation is ambiguous, or new requirements have to be added. To cope with these changes, special software engineering methods were applied.

3.2.1 Prototyping

One of the employed techniques is prototyping. A prototype is an initial version of a software used to demonstrate concepts and test design options [Som10]. The rapid, iterative development is used to discuss the requirements with the customer and to show the current progress. With the help of the prototype, the customer might get new ideas for requirements and recognize shortcomings in the product. Prototyping goes hand in hand with Scrum, the project management method applied for this project, which requires regular presentations and discussion of intermediate results.

3.2.2 Incremental Delivery

Incremental delivery is closely related to rapid prototyping. When a subset of the system functionality is ready, the current version of the software is delivered to the customer who will examine it. This way, the customer can decide about the importance of the different features. During the development of an increment, requirement change requests may arise but are saved for the next increment and not implemented directly. [Som10]

The incremental delivery approach has difficulties if the software replaces an existing system or is dependent on special facilities like data bases or special hardware [Som10]. Fortunately, that is not the case for the given project.

3.2.3 Continuous Integration

As a part of agile software engineering methods, continuous integration was carried out during the development. Continuous integration involves building the whole software frequently even after small code changes. The build process typically includes software tests and documentation generation. [Som10]

There exist different software tools to automate the build process, like Apache Maven, Hudson, CruiseControl or Apache Ant. Hudson and CruiseControl are both rather complex web-based services. Apache Maven's principle of convention over configuration imposes constraints on the folder hierarchy. This is why by using Apache Ant, which is also used internally by Eclipse, a very flexible solution has been chosen to automate the software building process.

4 Related Work

There already exist several projects to simulate or visualize a schedule of tasks. Among these, some concentrate on business projects or production schedules and therefore are not considered relevant. This chapter gives an overview of existing software for simulation and visualization of scheduling on OS level and presents an evaluation of its features.

4.1 Schedule Visualization Projects

Among the projects which focus on task schedules as they are typically generated by an OS, some only provide visualizations. The schedule itself has to be created by other means beforehand, e. g. by augmenting a scheduler to generate a trace or by simulation. Since the visualization is an important aspect of this work, some of these projects shall be discussed in the following.

4.1.1 Jedule

A program which presents an existing schedule graphically is called *Jedule*. It provides a visualization of given results in a custom Extensible Markup Language (XML) format. The XML file has to be created beforehand as explained above. Jedule aims to generate an overview of the whole schedule of a parallel program. It is written in Java and published under the GNU's Not Unix (GNU) General Public License (GPL).[HHS10]

4.1.2 Pajé

Pajé is a visualization tool for parallel applications running on distributed systems. Its software architecture consists of components connected in a data flow graph to facilitate the extension of Pajé. In order to visualize the schedule, an external tool is needed to generate a trace of the investigated application. This trace is then imported and passed to a simulator, which will produce thread states, simulate communications and generate concurrent primitives like semaphores which are finally displayed. The visualization can be inspected in detail, as it shows additional information about each event. Pajé is written in Objective C and is published under the GPL.[Ker00]

4.1.3 Summary

Other projects like the *Visual Trace Explorer (ViTE)* [CFJ$^+$12] or *gltracevis* [Lef09] have similar functions to the previously discussed programs.

Among the presented projects, some have interesting features as well as appealing and interactive graphical representations. However, none of them focuses specifically on a real-time and embedded environment.

4.2 Schedule Simulation Projects

None of the projects discussed in the previous section is able to generate the data to be visualized itself. Nevertheless, there are projects which mainly focus on simulation. In the following, some of these which concentrate on real-time scheduling shall be presented in the chronological order of their first publication.

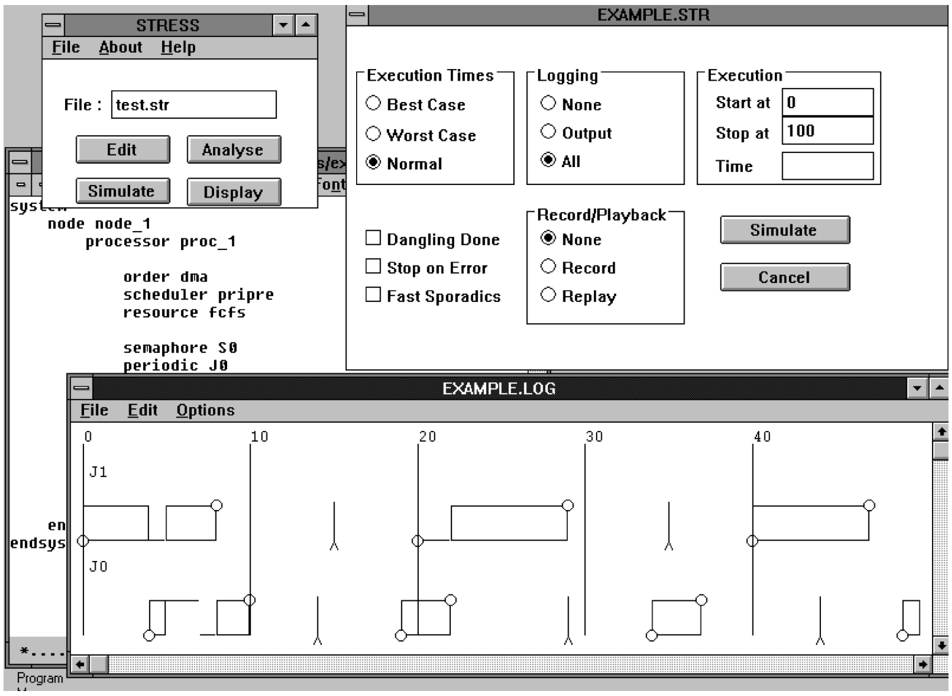
4.2.1 STRESS

STRESS is one of the first notable real-time simulation projects for arbitrary task sets. It focuses on analyzing and simulating the behavior of hard real-time safety-critical applications. STRESS consists of a closed simulation environment and provides its own programming language, also called STRESS. With this language, the system configuration including processors, networks to connect the processors, tasks, and even the behavior of the tasks as well as semaphores can be defined. The simulation environment allows tasks to control other tasks and thus it is possible to define one's own schedulers with STRESS. Nevertheless, it is not possible to define tasks with soft deadlines.[ABRW94]

Although written in 1994, it already provides a graphical user interface (GUI) and a graphical representation of a previously simulated schedule as visible in figure 4.1 on the facing page. However, compared to the visualization projects discussed in section 4.1 on the previous page, the GUI does not represent the state of the art anymore. Unfortunately, no newer publication nor any sources of the project are available.

4.2.2 GHOST

The *General Hard real-time Oriented Simulator Tool (GHOST)* is a real-time scheduling simulator. As one of the first of its kind, it supports hard, soft and no deadlines as well as resource allocation protocols. Tasks can be grouped to classes and each class can be handled by its own task scheduler. The superior class scheduler selects the class and invokes the respective task scheduler. Apart from a number of existing scheduling policies, new scheduling algorithms can be added by specifying them in C. GHOST also provides a graphical representation of the simulated schedule. The simulator can run in a time- or event-driven manner. However, GHOST only supports single-core systems and the schedule visualization



File : test.str

```
syst
    node node_1
        processor proc_1

            order dma
            scheduler pripre
            resource fcfs

            semaphore S0
            periodic J0
    en
endsys
```

Execution Times: ○ Best Case ○ Worst Case [✓ Normal]

Logging: ○ None ○ Output [✓ All]

Execution: Start at 0, Stop at 100, Time

☐ Dangling Done
☐ Stop on Error
☐ Fast Sporadics

Record/Playback: [✓ None] ○ Record ○ Replay

Figure 4.1: An overview of the STRESS GUI, taken from [ABRW94].

lacks detailed information compared to other projects. It is implemented in C and not publicly available.[GBA$^+$97]

4.2.3 RTSIM

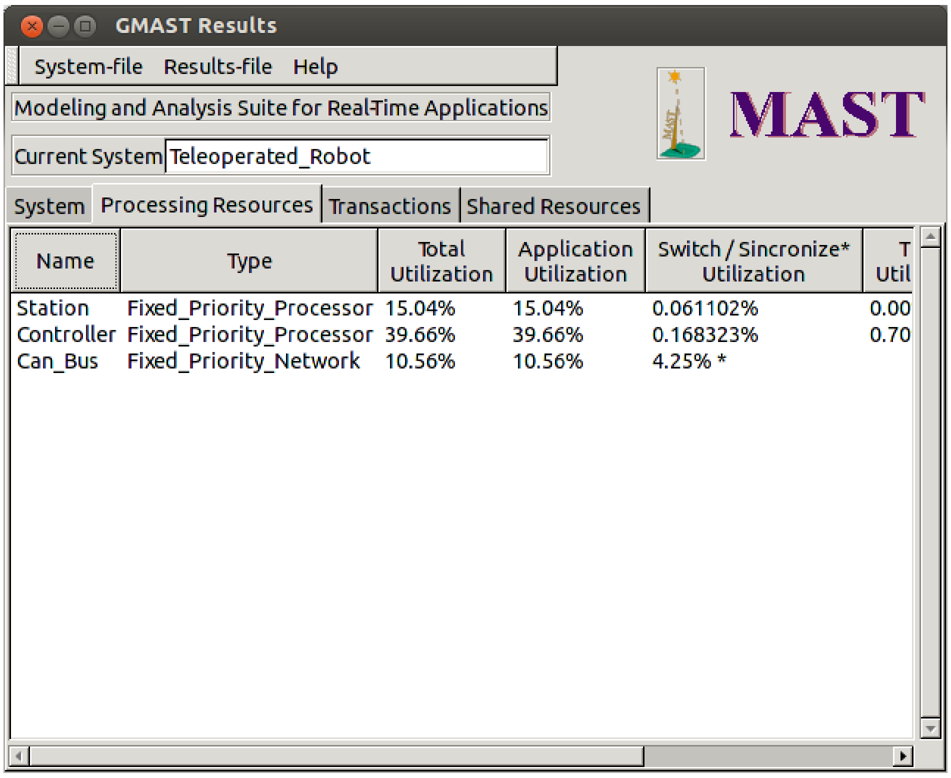
RTSIM is a framework to perform discrete event simulations of real-time control systems. These systems consist of one or more nodes connected by a network. RTSIM supports shared resources and comes with a GUI to specify the system to be simulated and to view the results. As a framework, the users can implement their own scheduler, too. Before the simulation starts, three description files, which were generated from the GUI or manually, are translated into C++ code and compiled and linked to the library. RTSIM is written in C++ and the GUI is implemented in Java. It is published under the GPL. [CBLL98] On the project's website, the Java GUI is no longer available but instead a C++ implementation of the GUI.[BL11]

4.2.4 FORTISSIMO

FORTISSIMO is an open real-time scheduling simulation framework implemented in C++. It offers a predefined software architecture for work load generation and task dispatching. In order to use the framework, a scheduling class has to be implemented and integrated into the framework. The result of the simulation is processed by so-called secretaries, which are informed whenever an event occurs and which generate statistical data as well as a graphical visualization. Although it supports task dispatching for multiple processors, it does not offer a possibility to define shared resources nor does it provide a GUI.[KAK00]

4.2.5 MAST

A comprehensive software to analyze different real-time schedulers is called *MAST*. It uses theoretical approaches to test the schedulability of a given task set, using appropriate heuristics for multi-core systems. The system model can be created within the software or via MAST-UML, a Unified Modeling Language (UML) meta-model which allows the usage of external UML modeling tools. Models are saved in a special-purpose text format or in an XML-based format.[HGGM01]



Modeling and Analysis Suite for Real-Time Applications

Current System: Teleoperated_Robot

Name	Type	Total Utilization	Application Utilization	Switch / Sincronize* Utilization	T Util
Station	Fixed_Priority_Processor	15.04%	15.04%	0.061102%	0.00
Controller	Fixed_Priority_Processor	39.66%	39.66%	0.168323%	0.70
Can_Bus	Fixed_Priority_Network	10.56%	10.56%	4.25% *	

Figure 4.2: The results of an example simulation by SIM-MAST.

SIM-MAST, depicted in 4.2, is an additional tool for simulating the behavior of existing models. However, this tool does not provide a graphical representation of the simulated results. MAST and SIM-MAST are written in Ada and use GTKAda for their GUIs. Both are published under the GPL.[Gon12]

4.2.6 RTsim

RTsim is a simulator for real-time task scheduling. Its main purpose is to support teaching. RTsim includes a variety of scheduling algorithms including priority ceiling protocols, and it supports single as well as multi-processor systems. The results are visualized by a Gantt chart and statistical values.[MMN01] Unfortunately, the source code is not available nor does the project's website provide any further documents or the program itself.[Man06]

4.2.7 VizzScheduler

Another scheduling visualization project is called *VizzScheduler*. It is part of a framework to develop and evaluate scheduling algorithms for the LogP cost model. The VizzScheduler provides a task graph diagram and Gantt chart of a given task set scheduled by an arbitrary scheduler. The scheduler is connected via the Java Debug Interface to update the visualizations whenever a breakpoint is passed. This allows a "live" view and a detailed controllability of a running scheduler.[LL01] The project is written in Java but unfortunately not open source and no longer maintained since 2002.

4.2.8 ARTISST

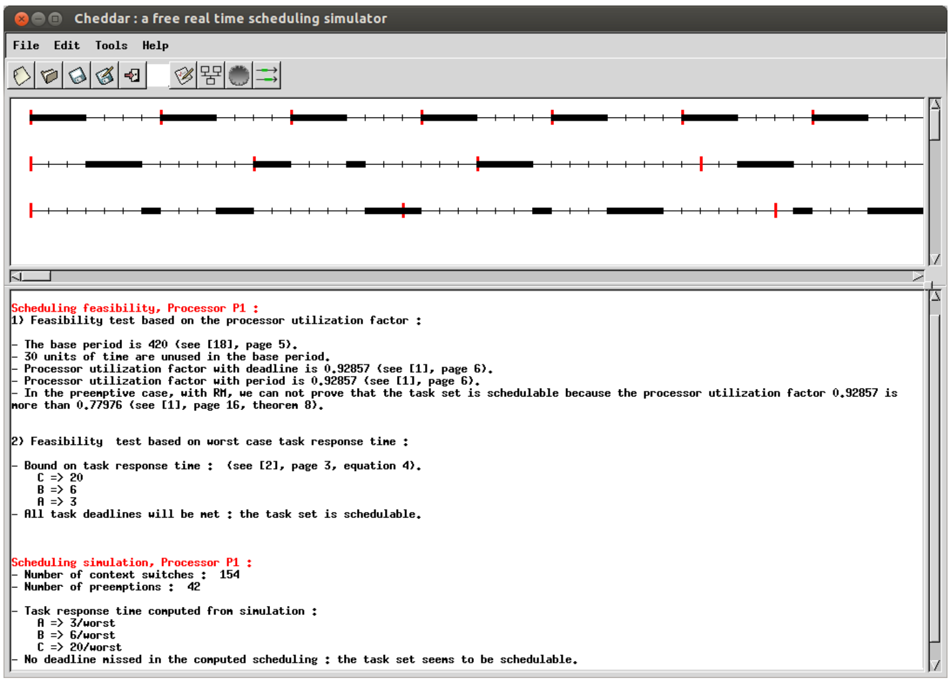
ARTISST is a Real-Time System Simulation Tool. It uses an event-driven simulation framework and allows to model the inner control flow of a task. To increase the simulation quality, the operating system costs such as context switching times are taken into account. The result of the simulation is presented as a chronogram or by statistical values. However, it neither supports multiple cores nor the handling of shared resources. ARTISST is implemented in C++ and open-source without a specific license yet.[DP02]

4.2.9 Cheddar

Similar to MAST, *Cheddar* can theoretically analyze real-time schedulers but is also capable to simulate and visualize them directly.

It comes with its own domain specific language (DSL) to model the system properties such as processors, tasks, and shared resources. However, it also supports the Society of Automotive Engineers (SAE) Architecture Analysis & Design Language (AADL), which has an Ada-like representation. Cheddar's own DSL is based on XML. Although AADL supports an interchange format based on XML, Cheddar imports and exports AADL directly. It also supports the definition of several processors, however, Cheddar currently supports only multi-core Rate Monotonic Schedulers (RMSs) with rather simple partitioning strategies.[Sin12]

For single-core processors, Cheddar's library offers a variety of fixed and dynamic priority schedulers like the RMS, the Deadline Monotonic Scheduler (DMS) and the Earliest Deadline First (EDF) scheduler. Other scheduling algorithms can be defined in its own, Ada-like DSL



```
Scheduling feasibility, Processor P1 :
1) Feasibility test based on the processor utilization factor :

- The base period is 420 (see [18], page 5).
- 30 units of time are unused in the base period.
- Processor utilization factor with deadline is 0.92857 (see [1], page 6).
- Processor utilization factor with period is 0.92857 (see [1], page 6).
- In the preemptive case, with RM, we can not prove that the task set is schedulable because the processor utilization factor 0.92857 is
more than 0.77976 (see [1], page 16, theorem 8).


2) Feasibility  test based on worst case task response time :

- Bound on task response time :  (see [2], page 3, equation 4).
    C => 20
    B => 6
    A => 3
- All task deadlines will be met : the task set is schedulable.



Scheduling simulation, Processor P1 :
- Number of context switches :  154
- Number of preemptions :  42

- Task response time computed from simulation :
    A => 3/worst
    B => 6/worst
    C => 20/worst
- No deadline missed in the computed scheduling : the task set seems to be schedulable.
```

Figure 4.3: The Cheddar software displaying a simulation of a RMS with three tasks.

which is interpreted by Cheddar at runtime. Like MAST, Cheddar is written in Ada and uses GTKAda for its GUI, as displayed in figure 4.3. It is published under the GPL.[SLNM04, SPD08, MSH11]

4.2.10 Realtss

Realtss is an open-source real-time scheduling simulator, which focuses on teaching and research. The application is written in the Tool Command Language (TCL) and new schedulers can be added by implementing them in TCL or C. It offers a GUI to configure the simulation and display statistical results. With the help of an external tool called *Kiwi*, the schedule can also be displayed graphically. Realtss supports soft and hard deadlines and shared resources accessed by mutexes. It is released under the GPL.[DBC07]

4.2.11 RTSSim

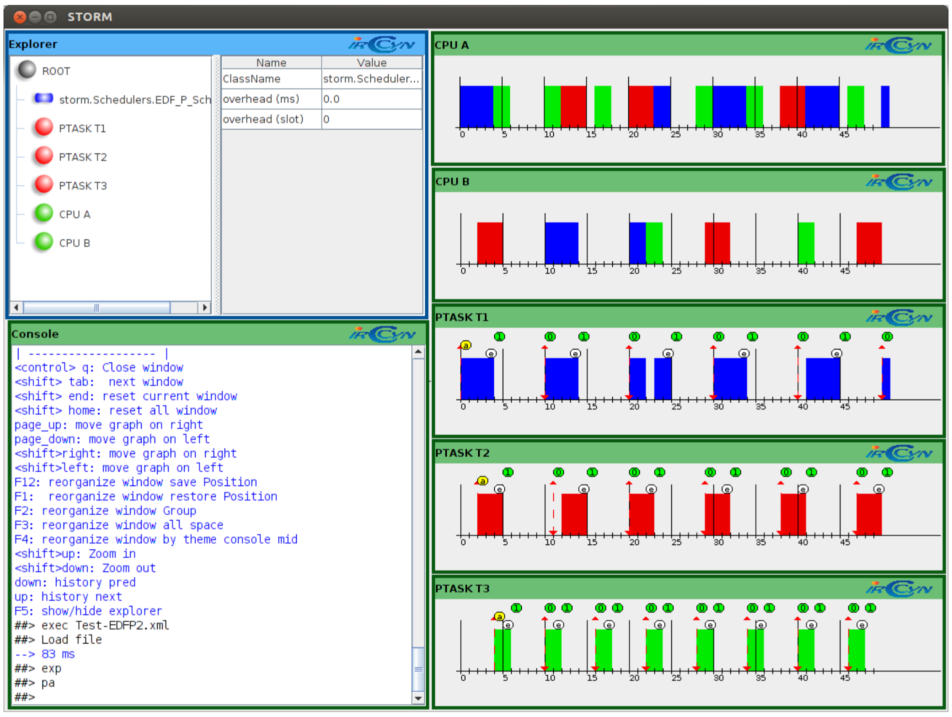
RTSSim is a simulation framework for embedded systems, which focuses on timing and resource usage. The simulation model is expressed in C code, such that each task is a C program executed within a sandbox environment. The model has to be compiled and linked to the framework. However, multiple cores are not supported and the only scheduling policy is preemptive fixed-priority which allows to change the task priorities at runtime. The simulated schedule can be visualized with the help of an external software called *Percepio Tracealyzer*.[Kra09]

4.2.12 Alea 2

Alea 2 is a simulation and visualization tool based on GridSim. GridSim itself is an event-based modular Grid simulation toolkit based on an event simulation library called SimJava. It allows to simulate distributed applications and generate various visualizations. The jobs to be scheduled, however, are read from a previously recorded trace. Alea 2 is written in Java and published under the Lesser GPL (LGPL).[KR10, Klu12] The visualization of the recorded trace can be played back in real-time, which offers a kind of "live" visualization. Unfortunately, the software focuses on distributed applications in large clusters and not on real-time systems.

4.2.13 STORM

STORM is a Simulation Tool for Real-time Multiprocessor scheduling, which supports multiple cores and shared resources. All entities like cores, resources, tasks, and schedulers are represented as Java classes, thus new elements can be added to STORM by adding new classes to the classpath. A simulation has to be specified by an XML file which contains the Java class names of the utilized entities. The simulator generates a list of events internally, which is used to derive statistical data as well as graphical visualizations of the schedule afterwards.



Name	Value
ClassName	storm.Scheduler...
overhead (ms)	0.0
overhead (slot)	0

```
| ------------------- |
<control> q: Close window
<shift> tab:  next window
<shift> end: reset current window
<shift> home: reset all window
page_up: move graph on right
page_down: move graph on left
<shift>right: move graph on right
<shift>left: move graph on left
F12: reorganize window save Position
F1:  reorganize window restore Position
F2: reorganize window Group
F3: reorganize window all space
F4: reorganize window by theme console mid
<shift>up: Zoom in
<shift>down: Zoom out
down: history pred
up: history next
F5: show/hide explorer
##> exec Test-EDFP2.xml
##> Load file
--> 83 ms
##> exp
##> pa
##>
```

Figure 4.4: The STORM GUI consisting of a command line and several information windows.

As shown in figure 4.4 on the preceding page, STORM provides a GUI which includes a command line to control the program. It is written in Java and is currently published under the Creative Commons License BY-NC-ND 2.0, which neither allows commercial use nor derived works.[UDT10, Uru12]

4.2.14 Schesim

Schesim is a scheduling simulator for real-time applications. In order to generate a more accurate result than other projects, the simulator allows to describe the control flow of the tasks as well as inter-task relationships such as shared variables. It includes a number of predefined single- and multi-core scheduling algorithms. Since its source code is available, it is generally possible to implement own schedulers, too, but Schesim's whole architecture must be understood.

The output of a simulation is a log file. It can be displayed graphically with the help of an external program called Trace Log Visualizer (TLV), an open-source application for Microsoft Windows. Schesim is written in Ruby and published under the Apache License 2.[MSHT12, MS12]

4.2.15 Summary

An overview of the presented projects is given in table 4.1 on page 32. Most of the programs first perform the simulation and generate the visualization afterwards or rely entirely on external software to display the schedule graphically.

Some of them, such as *Ghost*, *FORTISSIMO*, *ARTIST*, and *RTSSim*, do not support either shared resources or multiple cores, which both are necessary features as stated in section 1.2 on page 14. *Alea 2* and *RTSIM* support these features, but focus mainly on distributed systems, not on embedded systems with real-time constraints. *Schesim*, *Realtss*, *MAST*, and *STRESS* do not offer an internal visualization, or in case of *STRESS*, only a rather basic one. *Cheddar* and *STORM* support nearly all requirements, however, they do not visualize the schedule while the simulation is running, nor do they offer the possibility to pause the simulation at arbitrary points and show intermediate situations, a required feature according to section 1.2 on page 14.

It is generally possible to execute the simulation first and afterwards visualize its result in an animated way, allowing the user to pause the animation in order to concentrate on a specific detail. However, this would require to run a full simulation or to trace the whole execution of an application, save the results and hand them over to the visualization part. This is not an optimal solution, as it needs an expensive preparation of what otherwise could be shown immediately. Additionally, the user has to specify the time and duration of the part of interest to be simulated in advance.

Only *VizzScheduler* and *Alea 2* update the visualization while the simulation is running, but they are either not maintained anymore or focus on a different environment, namely distributed systems.

Furthermore, projects which are published under licenses such as the GPL require all software built upon or using parts of these projects to be again published under the GPL. As it is currently not intended to publish the source code of this project, such software is avoided. In contrast to the GPL, licenses such as the Eclipse license, the Apache license, the Massachusetts Institute of Technology (MIT) license, and the LGPL allow a commercial use of software containing parts under these licenses without publishing the source code.

Considering these problems, we decided to develop a new application which targets specifically on a controllable simulation of scheduling in a real-time environment with a simultaneous visualization of the current result.

Project	simulation	visualization	“live” simulation	shared resources	multiple cores	sporadic tasks	open-source	programming language
Alea2	—[1]	√	√	—	√	√	√	Java
ARTISST	√	√	—	—	—	√	√	C++
Cheddar	√	√	—	√	√	√	√	Ada
FORTISSIMO	√	—	—	—	√	√	√	C++
gltraceviz	—	√	—	√	√	√	√	C++
GHOST	√	√	—	—	—	√	—	C
Jedule	—	√	—	—	√	√	√	Java
MAST	√	—	—	√	√	√	√	Ada
Pajé	—[1]	√	—	√	√	√	√	Objective C
Realtss	√	√[2]	—	√	—	—	√	TCL
RTsim	√	√	—	√	√	—[3]	—[4]	*Unknown*[4]
RTSIM	√	√	—	√	√	√	√	C++
RTSSim	√	√[2]	—	√	—	√	—	C
Schesim	√	√[2]	—	√	√	√	√	Ruby
STORM	√	√	—	√	√	√	√	Java
STRESS	√	√	—	√	√	√	—	C
ViTE	—	√	—	√	√	√	√	C++
VizzScheduler	√	√	√	—	√	—	—	Java

[1] Simulation from a previously recorded trace
[2] With the help of external tools
[3] Only sporadic server available as scheduler
[4] Website does not contain any information

Table 4.1: An overview of existing real-time scheduling simulation projects and their properties.

5 Frameworks and Libraries

It is nearly impossible to write software for a modern desktop OS without using libraries and frameworks. They provide all interfaces necessary e. g. to display a GUI or to read and write files. In this chapter, different frameworks for the GUI and the serialization are presented and evaluated. Although there also exist several simulation frameworks for different programming languages, we decided to write the simulation part without the help of a library in order to have full control and to save the time required to understand the library itself.

5.1 Graphical Frameworks

A graphical toolkit which provides visual components and handles the communication with the OS eases the implementation of a visualization and provides essential routines to create a GUI. Two well-known frameworks for Java will be discussed in the following.

5.1.1 Swing

Swing is a common GUI toolkit for Java. In contrast to the Abstract Window Toolkit (AWT) and the Standard Widget Toolkit (SWT), Swing is a lightweight toolkit. This means that all visible components are not drawn by OS functions, but by the library itself. On the one hand, as a lightweight toolkit Swing is independent of the underlying OS, on the other hand the look does not fit into the rest of the user's desktop. Swing alleviates this by providing packages which imitate the look and feel of Microsoft Windows platforms and Linux with GIMP Toolkit (GTK+).[Ull12]

AWT is historically the first GUI toolkit for Java, and Swing still depends on AWT to some extent. Although AWT is thread-safe, Swing in general is not.[Ull12]

Swing is included in the Java Foundation Classes (JFC) together with other components, one of them being Java2D. Java2D provides methods to paint various two-dimensional graphics on the screen. The JFC are contained in the standard edition of the Java platform, which results in small file sizes of applications using Swing, as they do not need to contain the library.[Ull12]

5.1.2 Visualization Libraries for Swing

Although it is possible to create all components of the visualization directly with Swing and Java2D, there exist several frameworks which already provide useful graphical elements and convenient classes.

Prefuse

Prefuse is an information visualization toolkit built upon the Java2D framework and integrable into Swing. It strongly emphasizes interactiveness and is published under the Berkeley Software Distribution (BSD) license. [HCL05]

However, the interactive animations only apply to static data and do not support an evolving data model like a schedule.

Piccolo

Piccolo is a toolkit for structured 2D graphics which makes use of a scenegraph abstraction. A scenegraph maintains visual objects in a hierarchical structure and propagates manipulations throughout this structure. Piccolo has a rather long history: The project started in 1993 as Pad and was later rewritten under the name Jazz out of which Piccolo finally evolved. It makes use of the Java2D framework, but is also available as a .NET version. It focus on the zooming interface, which allows the user to scale the visible area. It is published under the BSD license.[BGM04]

However, apart form the zooming facility the Java2D toolkit already provides similar functions.

JUNG

The Java Universal Network/Graph Framework (JUNG) is a graph processing and visualization library. As the name already suggests, it concentrates on networks and graphs while pertaining relationships and links within them. It is published under the BSD license.[OFWB03]

Although a schedule can be represented as a graph, it is commonly visualized in a tabular manner, which is not supported by JUNG.

InfoVis

The *InfoVis* toolkit supports nine different visualization techniques, including scatter plots and time series. It is written in Java and uses an internal table structure, in which each column contains only objects of the same type. It runs on top of the Java 2D framework as well as the Agile 2D framework, whereas the latter makes use of the OpenGL Application Programming Interface (API) for hardware acceleration and therefore is faster in many situations. InfoVis is published under the MIT license.[Fek04]

However, none of the provided nine visualizations could be used directly to visualize a sequence of scheduled tasks, i. e. a schedule. Hence, the framework would have to be extended, which can cause comparability issues if the framework is updated and its API changes.

Timebars

Timebars is a framework to display a Gantt chart. It is available as a Swing and as an SWT library and published under the GPL or under a commercial license. [Kli09]

Although Gantt charts would be very useful for the project at hand, timebars focus only on business project schedules and calendars.

Summary

Other libraries like the *Visualization Toolkit (VTK)*, which is implemented in C++ and concentrates on three-dimensional visualizations [SML96], or the *Graphviz*, which is a graph visualization software with its own description language [GN00], offer huge functionality but focus on far too distant topics. Applications like *Processing*, which is a full-grown integrated development environment (IDE) to create interactive images and animations with its own programming language [RF07], or *Improvise*, which provides functions to create different interactive visualizations with a shared coordinate system [Wea04], are both open-source and implemented in Java. Yet, being self-contained programs with a different focus, their application to simulate and visualize a scheduling process seems inappropriate.

Among the discussed frameworks, none of them seems to add benefit compared to the costs of getting familiar with the framework. Therefore, a prototype is implemented using only Swing and Java2D in order to test the possibilities and for comparison against other GUI toolkits.

5.1.3 A Swing prototype

With the Swing and Java2D APIs the first visualization prototype was created as depicted in figure 5.1 on the following page.

The program uses a predefined task set with five tasks which are scheduled by a cyclic executive scheduler on a single core. The cyclic executive scheduler is a basic scheduling approach which executes tasks in a predefined way. In section 8.1 on page 71, the cyclic executive scheduler is

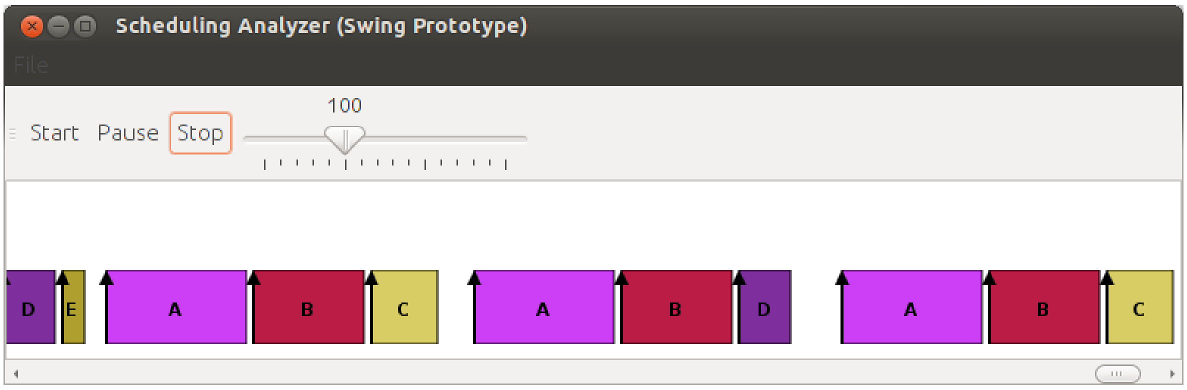


Figure 5.1: A prototype using the Swing and the Java2D API showing a cyclic executive with altogether five tasks.

explained in more detail. Implementing a more sophisticated scheduler within the prototype would not add any advantages for the comparison of visualization frameworks.

The application allows the user to start, pause and stop the scheduler. It provides a rather primitive visualization of the generated schedule which is updated continuously while the scheduler is running. To fit a longer time period on the screen, the schedule is displayed within a scrollable container which also provides zooming in and out by the mouse wheel as well as by GUI elements. Additionally, it supports the export of the generated schedule to an image file of the formats Portable Network Graphics (PNG), Joint Photographic Experts Group (JPEG) File Interchange Format (JFIF), Windows Bitmap (BMP) and Graphics Interchange Format (GIF).

5.1.4 Eclipse RCP and SWT

Rich Client Platform

The term *rich client* became popular in the early 1990s, when applications started to move from terminal clients to full-fledged solutions with GUIs. As the name suggests, these programs provided a “rich” and high-quality user experience by making use of the native user interface (UI) of the OS and supporting desktop metaphors like drag & drop and the system clipboard. [MLA10]

A Rich Client Platform (RCP) provides the middleware, upon which the business logic can be added. The contained frameworks and libraries to build the UI or to connect to databases accelerate the implementation of a new application. [MLA10]

Eclipse

Eclipse is a well-known software IDE for Java, which is maintained by the Eclipse Foundation, a non-profit open-source community. The IDE originates from IBM's *visual age*. Apart from Java, Eclipse supports many different programming languages. Each language is supported by a set of different tools. All tools are integrated into a generic platform, which happens to be the Eclipse RCP. So the Eclipse IDE is just another rich client application based on the Eclipse RCP. With version 3.0, interdependencies between tools and IDE were eliminated by introducing an OGSi-based runtime. More information about Eclipse is given in [MLA10].

OGSi

Open Services Gateway initiative (OGSi) is a standard which defines a module and service platform for Java. It also allows dynamic loading and unloading of these modules, so-called hot-plugging. The OGSi modules are called *bundles*, however, the Eclipse term for them is *plug-ins*. Both terms can be used interchangeably. A plug-in is self-describing, which means it contains a manifest file including information such as the plug-in's version and all dependencies on other plug-ins.

The implementation of the OGSi specification used by Eclipse is called *Equinox*, which also happens to be the reference implementation. With the help of Equinox, it is possible to start and stop software components without having to restart the whole application. This is achieved by assigning each plug-in a dynamic class loader. The OGSi framework collects the dependencies of each plug-in to allow the different plug-ins to collaborate.

A plug-in can declaratively define extension points, allowing other plug-ins to contribute the required information in form of basic values but also in form of complete classes. This feature facilitates future extensions of a product. In fact, all components of the Eclipse platform are implemented as plug-ins, even some parts of Equinox itself.

More information about the OGSi and Equinox is given in [MVA10, MLA10, Dau08].

SWT

The GUI of Eclipse is not based on Swing, but on the Standard Widget Toolkit (SWT). In contrast to Swing, described in section 5.1.1 on page 33, the SWT is a heavyweight GUI toolkit which means that it makes use of the widgets provided by the OS. This, of course, affects the portability of an application using the SWT. Fortunately, the toolkit is available on a variety of platforms including Linux and Microsoft Windows [MLA10]. The benefit of this method is a native look and feel for the user.

There exist a variety of plug-ins which build upon the SWT and provide higher abstraction layers to facilitate the programming.

JFace

The JFace plug-in is a GUI toolkit based on the SWT and provides more complex widgets, including different viewers, dialogs and Wizards. It also adds the Model–View–Controller (MVC) pattern to the SWT. [MLA10]

Graphical Editing Framework

The Graphical Editing Framework (GEF) provides all the functionality necessary to create visualizations of all kinds of models. It is intended to work with models generated by the EMF. However, it can also be used without the EMF if the model is specified with Plain Old Java Objects (POJOs). The GEF is built upon Draw2D, which is a standard 2D drawing framework based on the SWT [MDG$^+$04, p. 88].

The GEF allows the user to edit the graphical visualization and changes the underlying model accordingly. Although the editing part is not needed for the sole visualization, the usage of the GEF is still advantageous compared to a direct use of Draw2D or even the SWT, because it provides some handy features such as scrollable views and automatic updating of the visualization on model changes [RWC11].

Zest

The Zest plug-in is a visualization toolkit which is built upon the GEF. It provides a set of Eclipse visualization components and ready-made viewers for different purposes. However, Zest focuses only on graphs and is therefore of no use for this project.

5.1.5 An Eclipse prototype

The plug-in features of Eclipse explained in sections 5.1.4 on the previous page and the different graphical abstraction layers present a strong foundation for the given project. For example, each scheduler could be written as a dedicated plug-in and then be inserted into the existing RCP without having to compile the whole application again.

In order to compare the Swing and the Eclipse framework, another prototype is built as an Eclipse RCP using the GEF. Its GUI is displayed in figure 5.2 on the facing page. The prototype executes the same task set as the Swing prototype described in section 5.1.3 on page 35. It supports the same features like scrolling, zooming and exporting to an image file, in order to provide a high compatibility.

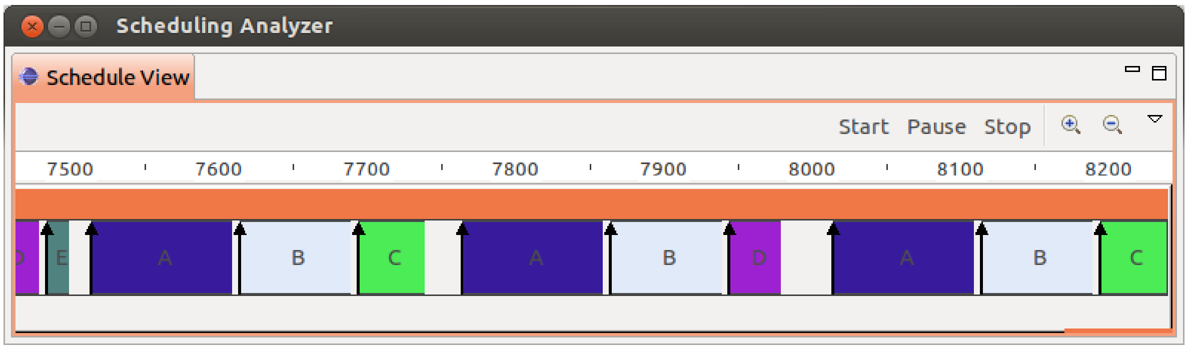


Figure 5.2: A prototype using the Eclipse RCP and the GEF showing a cyclic executive with altogether five tasks.

5.1.6 Comparison of Swing and Eclipse

Based on both prototypes, the Swing prototype described in section 5.1.3 on page 35 and the Eclipse prototype described in section 5.1.5 on the preceding page, a comparison of the different frameworks is performed in this section.

The implementation based on Swing compressed into a Java Archive (JAR) file is only 42.3 kB big. This is due to the fact that Swing and Java2D belong to the JFC which are already included in the standard edition of the Java platform; therefore, no other libraries are necessary. The Eclipse implementation exported as a stand-alone product is 18.9 MB big, that is around 500 times bigger than the Swing application. However, the four plug-ins containing all functionality are only 64.5 kB big, the rest is caused by the fact that the Eclipse core libraries as well as the GEF framework have to be included.

Although not all features of GEF are used by the prototype and the file size is unnecessarily extended, the bigger size comes with some advantages, too. For example, GEF already includes an MVC implementation and a zooming functionality. All these features have also been implemented in the Swing prototype, but required additional classes to be written. Functions like saving the generated presentation of the schedule as an image file require about the same amount of code in both implementations. Saving the schedule in scalable format like Scalable Vector Graphics (SVG) requires additional libraries in both solutions. In total, the Swing prototype is implemented in 1633 lines of code. This number is very similar for the Eclipse prototype, which is programmed in 1635 lines of code. These values were derived with the help of the metrics2 plug-in for the Eclipse IDE [SB12].

Swing, however, does not offer any way to load additional components at runtime the way it is possible with Eclipse and its plug-in-driven structure. This feature is essential to extend the simulator with new schedulers and therefore supports the decision to use the Eclipse framework.

5.2 Serialization Frameworks

As stated in section 1.2, one objective of the software is the file-oriented simulation input. The input of the simulator is a model defining all necessary entities such as tasks, cores, and resources.

Since this model has to be loaded by the simulator, the data interchange format must be readable by humans as well as by machines. There exist a variety of file types, including the XML, the JavaScript Object Notation (JSON), the YAML Ain't Markup Language (YAML), or even a simple comma-separated values (CSV) file, which are readable by both parties. However, since XML is likely to be known by most users and is supported by a large number of libraries, it seems to be the most appropriate file type to use.

Although Java provides libraries to parse XML and other document formats, the more elegant way to read a file and load a model into the program is to make use of serialization. Serialization is the process of saving an internal object hierarchy to an external data structure. This external data structure can be preserved beyond the end of the program execution and later remapped or deserialized to a new internal object hierarchy. In the end, the model can be saved and loaded without bothering how to generate or parse a file.

The Java language supports serialization natively. It facilitates saving all objects implementing the `Serializable` interface into a binary stream. This is also called marshalling or deflating. The binary stream can be saved into a file and later deserialized or unmarshalled to recreate the previously saved objects. More information about serialization is given in [Ull12].

However, this binary format is not directly readable for humans. Fortunately, there exist various options to serialize objects into the well-known XML format [Dau08]. In the following, different possibilities will be analyzed.

5.2.1 Java Bean Persistence

Java beans were originally invented to encapsulate many objects and therefore enhance the reusability. They are originally supposed to be manipulated visually with a special software tool.

Generally, a Java bean is a normal Java class which complies with some specific conventions, e. g. having a public constructor without any arguments and providing getter and setter methods for all internal properties following the default naming convention. Additionally, a Java bean has to implement the `Serializable` interface. Java beans should not be confused with Enterprise Java Beans (EJBs), the latter are part of Java Enterprise Edition. More information about Java beans is given in [Ull12].

If a class follows the conventions mentioned above, it is called a Java bean and can collaborate with the Java bean API. The Java bean API provides the classes `PropertyChangeSupport` and `PropertyChangeListener`, which allow to generate and to react to events. Other useful classes provided by the Java bean API are `XmlEncoder` and `XmlDecoder`. The usage of these classes in order to serialize a Java bean is presented in the listing 5.1. Exception handling and comments are omitted in all following code snippets to enhance readability.

Listing 5.1 Java code for XML serialization of a Java bean.

```
XMLEncoder enc = new XMLEncoder(new FileOutputStream(fileName));
enc.writeObject(objectToSerialize);
```

Listing 5.2 presents the generated XML file of the task set already used in sections 5.1.3 on page 35 and 5.1.5 on page 38. The listing reveals the rather simple way of the Java bean serialization. The XML structure is predefined, all data structures are broken up down to primitive types. It is not possible to exclude any reference or variable from the serialization.

Listing 5.2 Excerpt of the XML serialization of a Java bean.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_24" class="java.beans.XMLDecoder">
 <object class="de.unistuttgart.iste.ps.schedulinganalyzer.core.model.TaskSet">
  <void property="cores">
   <void method="add">
    <object class="de.unistuttgart.iste.ps.schedulinganalyzer.core.model.Core">
     <void property="id">
      <string>Core-1340194971174-2</string>
     </void>
     <void property="name">
      <string>Core A</string>
     </void>
    </object>
   </void>
  </void>
  <void property="tasks">
   <void method="add">
    <object class="de.unistuttgart.iste.ps.schedulinganalyzer.core.model.Task">
     <void property="deadline">
      <long>250</long>
     </void>
     <void property="id">
      <string>Task-1340194971174-3</string>
     </void>
     <void property="name">
      <string>A</string>
     </void>
     <void property="period">
      <long>250</long>
     </void>
     <void property="wcet">
      <long>95</long>
     </void>
    </object>
    ...
```

5.2.2 Java Architecture for XML Binding

Together with the sixth version of the Java Development Kit (JDK), the Java Architecture for XML Binding (JAXB) was released. It contains a compiler called *xjc* which is capable of generating Java classes from an existing XML Schema Definition (XSD) file. XSD is a format proposed by the World Wide Web Consortium (W3C) to define a set of rules a XML document has to follow in order to be considered valid. JAXB also contains a compiler for the inverse direction, i.e. to generate an XSD document from annotated Java classes, which is called *schemagen*.

The generated Java classes contain annotations to provide meta-information, e.g. the order of properties inside the XML document. Using these annotations, it is possible to augment the generated classes with references and variables which will not be serialized. However, since a new compilation of the XSD file would overwrite manually added annotations, this is not recommended.

The JAXB API provides functions to serialize and deserialize the generated Java classes. The generated XML document will be a valid instance of the XSD file. The source code to serialize previously generated Java classes is presented in listing 5.3.

Listing 5.3 Java code for XML serialization with JAXB.

```java
JAXBContext context = JAXBContext.newInstance(ObjectToSerialize.class);
Marshaller marshaller = context.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE );
marshaller.setProperty(Marshaller.JAXB_ENCODING, new String("UTF-8"));
marshaller.setProperty(Marshaller.JAXB_SCHEMA_LOCATION, new String("TaskSet.xsd"));
marshaller.marshal(objectToSerialize, new File(fileName));
```

Listing 5.4 on the next page shows the generated XML file of the same sample set of tasks used in section 5.2.1 on page 40. The XML namespace can be set in the header of the XML schema, which is then transformed by the *xjc* compiler into a Java package. For example the namespace `http://example.org/test` will be transformed into the Java package `org.example.test`.

The generation and reaction to events, which are necessary to update the GUI when the scheduler runs and creates the schedule, is problematic. While it works flawlessly with Java beans, JAXB tries to include any referenced `PropertyChangeSupport` object. These objects are not serializable and therefore an exception is thrown. There is the possibility to create classes from an XSD file with integrated event support using an *xjc* plug-in, but this plug-in requires the whole project to be built with Maven. Another solution is to implement the event handling manually and exclude it from serialization by an annotation, but as stated before, this modification will be overwritten on each recompilitation of the XSD. More information about the JAXB is given in [Dau08, SN09, ME07].

Listing 5.4 XML serialization with JAXB.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<TaskSet xmlns="http://www.iste.uni-stuttgart.de/ps/schedulinganalyzer/core/
    generatedModel" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id=""
    xsi:schemaLocation="TaskSet.xsd">
    <core name="Core 1" contextSwitchingTime="" active="false" id=""/>
    <task deadlineType="HARD" wcet="100" period="250" deadline="250" name="A"/>
    <task deadlineType="HARD" wcet="80" period="250" deadline="250" name="B"/>
    <task deadlineType="HARD" wcet="50" period="500" deadline="500" name="C"/>
    <task deadlineType="HARD" wcet="40" period="500" deadline="500" name="D"/>
    <task deadlineType="HARD" wcet="20" period="1000" deadline="1000" name="E"/>
</TaskSet>
```

5.2.3 Eclipse Modeling Framework

EMF is a modeling framework and code generation facility [SBPM11], intended to be used together with GEF. A model can be created with the help of a graphical editor based on GEF. Moreover, EMF can extract a model from annotated Java classes, from UML diagrams, as long as they export the diagram to XML Metadata Interchange (XMI), or from XMI directly, which is also used to save the model internally.

XMI is a data interchange format based on XML and specified by the Object Management Group (OMG) [Obj11]. It focuses on sharing models and although it is mainly used in software development, it is a very useful format to describe a simulation model, as it allows links between objects in the same or in different files.

The meta model used by EMF to specify the model structure is called *ecore*. Since *ecore* is itself modeled with EMF, it is actually a meta-meta model. A EMF model can be exported as XSD file, as UML model via XMI, or as annotated Java code. Some information required to export a model, like the namespace and the folder where the output files shall be located, are not saved in the *ecore* model but in another model called *generator model*. This *generator model* is structured in a very similar way to an EMF model.

As explained above, EMF is able to convert an *ecore* model to Java source code. Therefore, EMF's Java code generator creates an Eclipse plug-in containing an interface and a class implementing the interface for each class of the model. The interface is inherited from `EOBject`, an *ecore* equivalent of `java.lang.Object`. The class `EOBject` itself inherits from `Notifier` and thus provides a notification mechanism based on the observer design pattern. Objects that want to be notified of changes, i. e. the observers or listeners, are called adapters in EMF. In contrast to the standard observer design pattern specified in [GHJV95], the observed class has no notion of being observed. To observe changes within a hierarchy of objects, there also exists a convenient adapter class `EContentAdapter`, which can be subclassed in order to be notified of any change within that hierarchy.

Similar to JAXB, the generated Java classes are annotated. However, in contrast to JAXB, they are expected to be merged with user-written code. Any variable without the `@generated` annotation is neither serialized nor overwritten on regeneration of the source code.

With the help of the `Resource` class provided by EMF, generated classes can easily be serialized into and restored from XMI documents. This feature is the main reason why EMF is used. The code generation itself is just a handy feature during the development of the specification of the simulation model, but unessential once the specification is stable.

The source code to serialize the objects of previously generated Java classes is presented in listing 5.5. As different objects can be serialized in different XMI files references each other across documents, a `ResourceSet` can contain more than one `Resource`.

Listing 5.5 Java code for XML serialization with EMF.

```
ResourceSet resourceSet = new ResourceSetImpl();
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put("xmi", new
    XMIResourceFactoryImpl());
Resource resource = resourceSet.createResource(URI.createFileURI(fileName));
resource.getContents().add(objectToSerialize);
resource.save(null);
```

The XMI formatted output of a simulation model containing one core, once resource and once task requesting and releasing resource is shown in in listing 5.6. More information about EMF is given in [Dau08, SBPM11].

Listing 5.6 XML serialization with EMF.

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:model="http://model/1.0">
  <model:SystemModel name="Test" description="" >
    <task name="a" deadlineType="NONE" deadline="170" priority="1" offset="0"
        repetitions="1">
      <command xsi:type="model:Execution" duration="10"/>
      <command xsi:type="model:RequestResource" resource="/0/@resource.0" />
      <command xsi:type="model:Execution" duration="40"/>
      <command xsi:type="model:ReleaseResource" resource="/0/@resource.0" />
      <command xsi:type="model:Execution" duration="10"/>
    </task>
    <core name="Core"/>
    <resource name="V" priority="4"/>
  </model:SystemModel>
</xmi:XMI>
```

5.2.4 Summary

Apart from the discussed, there exist various third-party XML serialization libraries like Simple, XStream, Castor, or Zeus [Ull12, ME07]. However, since the Java code generated from an EMF model is perfectly integrated into Eclipse, supports serialization to and deserialization from XMI as well as contains a notification mechanism to update the visualization, there is no more benefit these libraries could add.

6 Design

An evaluation of frameworks and libraries was conducted in chapter 5. The selected frameworks determine the interfaces of the software architecture. This chapter presents the conceived software design.

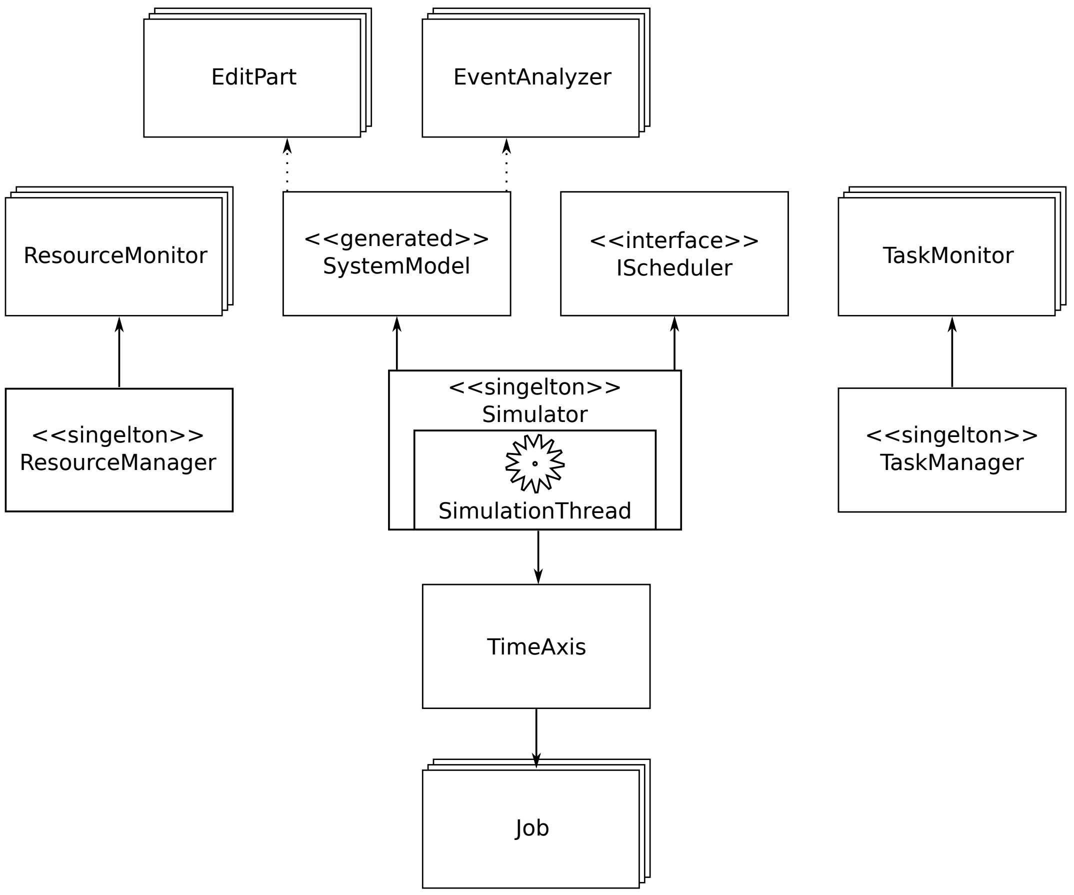


Figure 6.1: An overview of software design; boxes represent classes, solid arrows represent references and dotted arrows represent notifications.

An overview of the simulation architecture is given in figure 6.1. The image only provides a coarse overview and therefore does not follow the UML standard. In the following, all parts of the architecture will be discussed in closer detail, starting with the design of the system model.

Afterwards, the architecture of the simulator itself is described, before the visualization design as well as exception handling aspects are finally discussed.

6.1 System Model

Besides the visualization, one of the main aims of this work is the development of a simulator. According to SHANNON, simulation is “the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and / or evaluating various strategies for the operation of the system.” [Sha98, p. 7] Therefore, the model is an essential part of a simulation. It is an abstraction of a system and has to contain all information necessary to deduce the simulation result. It must not oversimplify the system nor be too detailed, as the calculation complexity as well as the quality of the results depend on the abstraction level of the model. [Sha98]

One objective introduced in section 1.2 on page 14 is the simulation of arbitrary scheduling processes. Each scheduling process is simulated based on its own simulation model, defining a specific set of entities such as tasks, resources, cores, and scheduling policies. All these elements have to be defined in an unambiguous and precise way to be handled by a common simulation kernel.

For this reason, a *meta-model* was developed. It specifies the way a simulation model and all its elements have to be described. In order to provide a maximum of extensibility and to allow the definition of features like multiple cores and different scheduling policies, the meta-model must be very flexible and foresee future requirements. Therefore, it has to be designed very carefully. The meta-model is called *system model*.

The topmost element of the system model is the class `SystemModel`. It has two properties, a name and a description, both of type String, which are useful to provide additional, textual information about the model to the user. Additionally, the system model contains multiple `Task`s, `Resource`s and `Core`s. All of these elements are derived from the interface `IElement` and add further, element-specific attributes. A UML representation of the model is depicted in figure 6.2 on the next page. The system model is defined with EMF and its source code generated by this framework. This way, the resulting classes are serializable and contain a notification mechanism as explained in section 5.2.3 on page 43.

Besides tasks, resources and cores, the scheduling policy is an essential component of the simulation model. However, different scheduling policies should be applicable to the same simulation model. Therefore, the scheduling policy is not contained in the system model, but referenced by the simulator at runtime.

An instance of the system model is called simulation model. It specifies the exact number of tasks, cores and resources as well as their detailed properties. The simulation model also serves as interface between the simulation and the visualization part. During the simulation, elements of the model are changed and new elements are added. The visualization is generated based on the simulation model and its elements. This will be discussed in detail in section 6.4 on page 58.

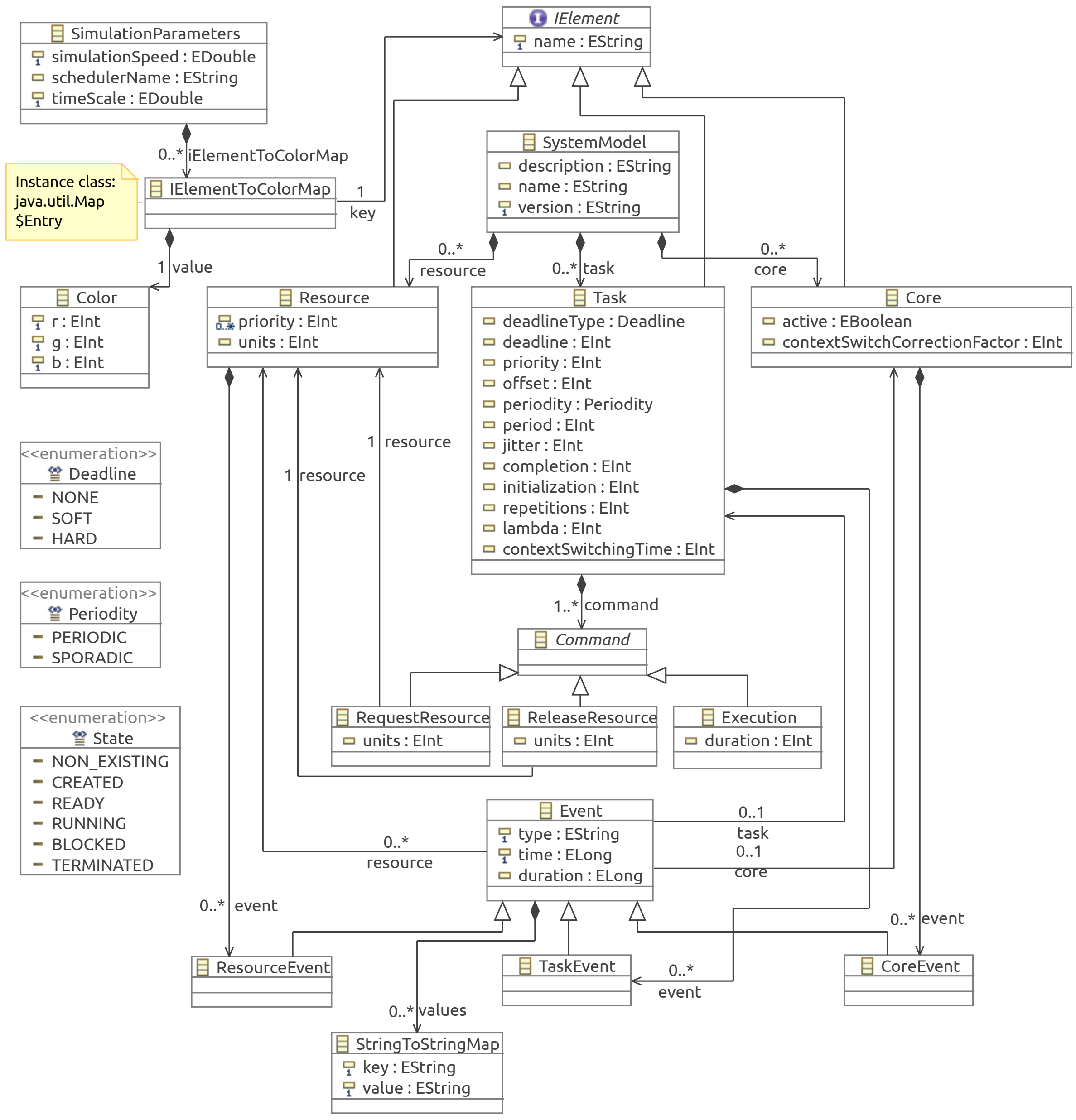


Figure 6.2: The UML class diagram of the system model.

6.1.1 Tasks

Each task is specified by a variety of parameters, which are explained in table 6.1 on the following page. With the help of these parameters, a simple, infinitely repeating periodic task with a hard deadline equaling its period of 100 ms and a priority value of 5 would be defined by `deadlineType=HARD, deadline=100, period=100, priority=5`. In contrast, a sporadic task with a soft deadline of 20 ms, a minimum inter-arrival time of 1000 ms, a priority of 10, an initial offset of 100 ms which requires a context switch time of 2 ms and repeats exactly 2 times with random delays added to the minimum inter-arrival time distributed according to a Poisson distribution with lambda equaling 50 ms would be defined by `deadlineType=SOFT, deadline=20, period=1000, priority=10, offset=100, repetitions=2, lambda=50, contextSwitchingTime=2`. As mentioned in section 2.3 on page 18, sporadic tasks are called aperiodic if they have a soft or no deadline instead of a hard deadline. Since the parameter set allows to specify the type of the deadline of a sporadic task to be hard, soft, or not existing, we do not differentiate between between aperiodic and sporadic tasks in the following.

Property	Default value	Description
`deadlineType`	`NONE`	`NONE`, `SOFT` or `HARD` deadlines are supported
`deadline`	`0`	The time of the relative deadline
`priority`	`0`	The task's basic priority
`offset`	`0`	Time offset of the first creation
`periodity`	`PERIODIC`	`SPORADIC` or `PERIODIC` tasks
`period`	`0`	Time of the task's period or minimum inter-arrival time if sporadic
`jitter`	`0`	The maximum variation of a process release time in both directions (earlier and later)
`completion`	`0`	Duration of the completion phase, for future use
`initialization`	`0`	Duration of the initialization phase, for future use
`repetitions`	`-1`	The number of repetitions of the task's creation, -1 represents infinity
`lambda`	`1`	The lambda parameter of the Poisson distribution used for sporadic task
`contextSwitchingTime`	`0`	The time necessary for a context switch *before* the task starts executing

Table 6.1: An overview of the task parameters.

Commands

As mentioned in section 2.3 on page 18, a `Task` contains arbitrarily many `Commands`, whereas a `Command` is an abstract class which is inherited by the classes `Execution`, `RequestResource`, and `ReleaseResource`.

An execution command has a duration value which equals the worst case execution time (WCET) of a sequence of instructions, containing conditional branches and loops. The sequence has to be split at every resource interaction instruction, as these are modeled by `RequestResource` and `ReleaseResource` commands.

By means of an ordered sequence of these commands, the total execution of a task can be modeled. The abstraction level is low enough to support resource operations and high enough not to include all instructions a real task might contain. This helps to speed up the simulation and is sufficient to create a scheduling visualization. However, it is not possible for a task to finish earlier than its WCET, to self-suspend or to interact with other tasks other than via shared resources. Future versions might extend the system model at this point.

6.1.2 Cores

A core is a part of a processor. The notion of a processor is not transferred into the system model, since processors are typically interconnected by a network, which adds timing complexity and is out of scope of this thesis. However, the system model might contain an arbitrary number of cores. Note that a physical network interface can be modeled by a shared resource, which has to be locked by a task from time to time to check whether new messages have arrived or to send messages.

The model parameters of a core are presented in table 6.2. The parameter `active` is specified, because each core of modern multi-core processors can be totally deactivated to save energy [JG04]. Future scheduler implementations might want to simulate this behavior.

Property	Default value	Description
`active`	`false`	Option to deactivate a core, for future use
`contextSwitchCorrectionFactor`	`0`	A correction factor of task context switch timings

Table 6.2: An overview of the core parameters.

The parameter `contextSwitchCorrectionFactor` is added to the task's `contextSwitchingTime`. It is allowed to be negative, but the result of the addition will be set to 0 if it is negative.

Modern multi-core processors contain several layers of caches. In most cases, the lowest cache is linked to exactly one core, whereas higher level caches are shared between different cores. More information about caches is given in [Tan09]. When a task is switched from one core to another, depending on the level of the cache containing its data, the context switching time differs. With the help of the `contextSwitchCorrectionFactor` parameter, this fact can be accounted for.

6.1.3 Resources

A resource or each of its units can only be accessed exclusively by one task at a time, as defined in section 2.5 on page 20. An overview of the parameters describing a shared resource is given in table 6.3 on the next page.

Since some resource access protocols require a resource to specify a ceiling priority, a resource contains the field `priority`. For resources with multiple units, depending on the number of tasks locking the resource and the available number of units left, the ceiling priority of a

Property	Default value	Description
`priority`	empty list	A list of priority ceiling values in the order of available units
`units`	`1`	The number of units

Table 6.3: An overview of the resource parameters.

resource has to be adapted and is therefore called *dynamic ceiling priority*. This is modeled by an array of ceiling priority values ordered by the number of available units, such that if one unit of the resource is locked, the first array element is the actual ceiling priority. If the resource is not locked by any task and all units are available, a priority lower than the priority of the lowest priority task is defined to be the actual ceiling priority. In general, as many ceiling priority values as units are necessary. More information about resources with multiple units is given in [Liu00].

6.1.4 Events

Tasks, resources, and cores all reference arbitrarily many `Events`, which are generated during the simulation. An event could be the creation of a task, the running or blocked state of a task, the task being scheduled on a core, or a resource being locked by a task. A graphical representation of each event is displayed on a time axis. Thus events serve as an interface between the simulation and the visualization. For each model element, there exists a special subclass of the general event class to allow a different visualization depending on the type.

An event is of a specific type defined by the field `type`. Different types allow a different representation of e. g. a deadline or a running event. The `type` field is of type `String`, so new types can be added by future extensions without changing the model. Apart from `time` and `duration`, each event contains a map `values`, which assigns textual keys to textual values. With the help of this map, it is possible to add arbitrary payload to an event, e. g. the number of units locked in case of a resource locking event. This additional information can also be visualized. In contrast to predefined fields, a map has the advantage that new information can easily be added without changing the model.

6.2 Simulation Parameters

Some information such as the proposed scheduling policy, which will should used to schedule the simulation model, and other simulation and visualization parameters are not a direct part of the system model. However, they should also be contained within the same file to be available when the simulation model is loaded.

For that reason, the class `SimulationParameters` is modeled with EMF. If contains a field to indicate the name of the scheduler which shall be preselected when the user opens the simulation model file. During the simulation, the parameter `schedulingSpeed` is multiplied by the timing values contained in the simulation model and thus allows to speed the simulation up or to slow it down. The user is allowed to edit this value after the simulation model is loaded. The same holds for the parameter `timeScale`, which is multiplied by the graphical representation of the simulation time and the duration of events and thus allows to stretch or shrink the visualization on the time axis.

Additionally, the simulation parameters contain a map assigning each implementation of the `IElement` interface a `Color` value. The color is defined by RGB values. This map allows to define the color of each task, resource, and core. If no color is defined for an element, the color is chosen from a predefined set of colors. In case more elements than predefined colors exist, the color is chosen randomly depending on the type of the element; tasks are typically represented by bright colors, resources by dark colors, and cores by shades of gray.

6.3 Simulator

The simulator is the central element of the software, as it drives the simulation by causing the simulation model to be modified. As such, the simulation kernel needs to be accessible from different classes, e. g. from a scheduler or from visualization parts. Therefore, the class `Simulator` is designed according to the singleton design pattern defined in [GHJV95]. This restricts the number of instances of the class to exactly one and allows other classes to access this object without holding a reference to it.

Other classes might need to be notified about changes of the simulation model, e. g. in order to update the visualization. Therefore, the `Simulator` is designed as the observable class of the observer design pattern [GHJV95]. All observing classes have to implement the `SimulatorListener` interface and register as a listener of the `Simulator`. From then on, the class will be notified of changes of the simulation model.

6.3.1 Discrete Event Simulation

In general, simulation can be either static or dynamic [BG07]. Since time plays an essential role in the scheduling process, the simulation of this work has to be dynamic. Dynamic systems can be simulated continuously or discretely [BG07]. Note that the simulation time is independent of the real-world time for both types of dynamic simulation.

Continuous simulations typically make use of differential equations to calculate the system state at any time [BG07]. For the simulation of a scheduling process, it is also possible to define a minimum time frame, e. g. 1 ms, and simulate step by step. One advantage of such a simulation would be a very smooth visualization, as a column of constant width is added to the time axis after each simulation step. However, this type of simulation has the disadvantage that the minimum time interval is fixed, which is why events with a shorter duration cannot be simulated. Additionally, the visualization has to be updated after each step, which is computationally inefficient especially for models with a large number of elements. It is sufficient to update the visualization only as soon as the simulation model is changed.

This idea is the basis of the discrete, event-driven simulation. A discrete event simulation (DES) is comprised of a model, events, a calendar, and the current simulation time. The DES operates as follows: Events are located at specific times in a calendar, i. e. the time axis. Since it is possible that more than one event is located at the same time on the time axis, real parallelism can be simulated. Simulation events are not to be confused with `Events`, the former are part of the simulation whereas the latter are part of the system model. As soon as the simulation time reaches the time of an event, the event is triggered. It changes the simulation model and optionally adds new, future events to the time axis. After the event has finished executing, it is deleted from the time axis. The simulation time proceeds from one event on the time axis to the next. In contrast to a continuous simulation, all moments in simulation time in between are omitted. The simulation time "jumps" along the time axis and it is sufficient to update the visualization only at these moments, since it is impossible for the simulation model to be changed in the meanwhile. Therefore, the discrete, event-driven simulation is in general faster than the continuous simulation. More information about DES is provided in [BG07].

The simulation kernel of this project is designed in a dynamic, discrete event-driven manner.

6.3.2 Simulation Thread

In order to decouple visualization and simulation and to allow the simulation to wait if necessary, both parts have to run in separate threads. This is why the class `Simulator` contains a private inner class `SimulationThread`, which extends the class `java.lang.Thread`, as indicated by a cog and the nested boxes in the overview figure 6.1 on page 45. The private inner class cannot be accessed from outside except from the class it is contained in, which provides methods like `start()`, `play()`, `pause()`, and `stop()` to control the state of the `SimulationThread`. This design improves the encapsulation and safety, as the thread cannot be accessed from outside in an unintended way. It also represents the logical grouping of both classes.

As explained in the previous section, the simulation time is independent of the real-world time. However, since the visualization is synchronous to the simulation, running the simulation as fast as possible is not intended. Instead the simulation shall run in real-world time if possible. This "live" representation is more demonstrative for the user. For that reason, as soon as all events at the current simulation time are triggered and the simulation is about to "jump" to the next time step, the simulation thread shall wait for the same amount of real-world time that is passed in simulation time. With the help of the parameter `schedulingsSpeed`, introduced in section 6.2 on page 50, the waiting times can be adjusted and the simulation itself accelerated or decelerated. Linking the real-world time to the simulation time does not only result in a "live" view of the schedule, but is also necessary for the visualization thread to process and display the changes.

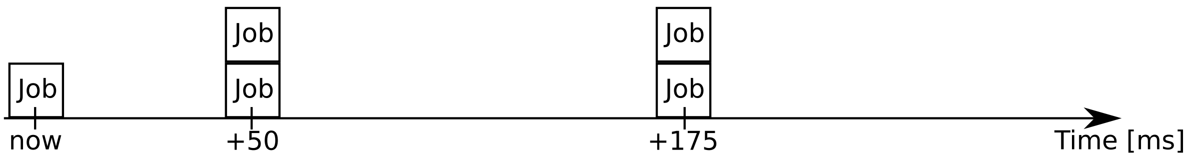
The simulation thread is also responsible for the creation of the initial events on the time axis to allow the simulation to start. For that reason, the function `setup()` is called before the thread starts. It generates an event for each task at the time the task is created for the first time.

6.3.3 Time Axis and Jobs

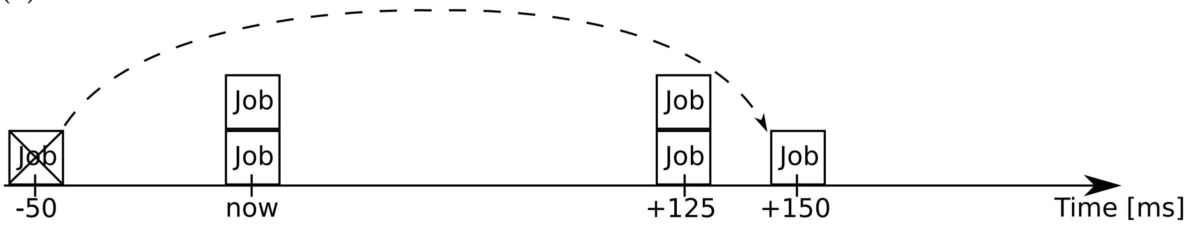
In section 6.3.1 on page 51 the components of a dynamic, discrete event-driven simulation are enumerated. All of them are part of the `Simulator` class, which also contains the simulation time represented by a simple data field. The calendar and the events which cause the simulation time to increase are explained in this section.

The calendar is represented by the class `TimeAxis`. It contains a map, linking a point in time to a list of events, represented by the abstract class `Job`. Note that `Event`s are usually generated by `Job`s and are part of the system model. Typically, jobs change the state of a task, create a deadline event for a task, or call the scheduler when a tasks wants to interact with a resource. Each implementation of the `Job` interface can specify whether a dispatch call is necessary after it has been executed or not. This way a job which just adds a deadline event does not cause the scheduler to interrupt the running task unnecessarily.

Jobs can be added to the time axis with the function `addJob(long time, Job job)`. This function ensures that no job is added at a simulation time lying in the past. Additionally, it directly executes jobs, which are added at the current simulation time. The simulation thread calls the function `executeJobs(long now)` to trigger all jobs at the current simulation time. Once a job was triggered, it is deleted from the time axis. Thereupon, the simulation thread calls `getNextTimeStep()` to determine the next simulation time and to calculate the real-world time the simulation will wait. If no more jobs are contained in the time axis, `getNextTimeStep()` will return the maximum value of `long`, and the simulation ends. The class `TimeAxis` additionally offers some methods to manipulate jobs, e. g. it is also possible to delete future jobs from the time axis. This is necessary if a task is preempted and thus the job ending the execution of the task is no longer at the correct time.



(a) At the current simulation time.



(b) After the job was executed, generated a new job and the simulation time proceeded.

Figure 6.3: An example of a time axis at two successive time instants.

An example of the time axis at two consecutive simulation times is shown in figure 6.3. The job at the current simulation time in figure 6.3a is triggered by the simulation thread and creates a new job 200 ms later. After all jobs at the current simulation time have been triggered, it is deleted from the time axis and the next simulation time is determined to be 50 ms later. Therefore, the simulation thread will wait for 50 ms before it triggers the two jobs at the new simulation time.

6.3.4 Scheduler Interface

A UML class diagram of the scheduler interface `IScheduler` is presented in figure 6.4. Each combination of a scheduling policy and resource access protocol has to implement this interface. However, with the help of object orientation and inheritance, different resource access protocols can be added to the same scheduling policy by overriding the resource interaction methods. A scheduler implementation is added via Eclipse's extension point mechanism, facilitating the addition of new policies without having to recompile the whole software.

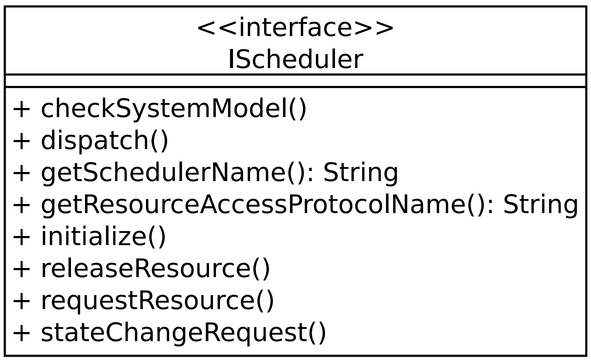


Figure 6.4: The UML class diagram of the scheduler interface.

The simulation thread assures that the function `initialize()` is called before the actual simulation starts. Once the state of a task needs to be changed, e. g. because it was created, the scheduler is informed via the function `stateChangeRequest(long time, Task task, State newState)`. It is up to the scheduler implementation to drive the tasks, i. e. it has to fulfill the request and change the task's state to *created.* It is also the scheduler's duty to change the state from *created* to *ready* over the course of time.

The simulator calls the function `dispatch(long time)` whenever a new scheduling decision is necessary. Again, it is the scheduler's job not only to select the task to be executed, but also to preempt the running task if necessary and to trigger the execution of the selected task. The class `TaskMonitor` provides useful methods for these jobs; it will be discussed in the following.

The both functions `requestResource(long time, Task task, Resource resource, int units)` and `releaseResource(long time, Task task, Resource resource, int units)` are called whenever a task requests or releases a resource, respectively. In the former case, the scheduler has to decide whether access is granted and the task locks the resource or the task is blocked. In the latter case, the scheduler has to unlock the resource and change the state of waiting tasks if necessary.

Similar to the class `TaskMonitor`, the class `ResourceMonitor` provides the required functions for these jobs, it will also be discussed in the following.

An alternative design of the scheduler interface would just inform the scheduler about a task having changed its state. The dispatch routine could just select and return the task to execute, relying on the simulator to preempt the currently running task as well as to start the execution of the selected task. Furthermore, a resource request could just be granted or denied by the scheduler, leaving the lock or unlock operation to the simulator. Although these design alternatives would require less work from the point of view of the scheduler implementation, they are also less flexible. A scheduler might have to postpone the creation of a task or perform additional steps when dispatching. This is why these alternatives were rejected in order to provide a maximum of flexibility for future scheduling protocols.

System Model Check

There is no guarantee for a system model instance, i. e. a simulation model to be valid. It might contain tasks without any execution command or tasks which lock more units of a resource than the resource contains. The validity might also depend on the scheduling policy or the resource access protocol.

For this reason, after a simulation model has been loaded from a file and the scheduling policy is selected, the function `checkSystemModel(SystemModelCheckReporter reporter, SystemModel systemModel)` is called. It is the job of the scheduler to inspect the given simulation model and report all findings via the function `addFinding(Severity severity, String message)` of the `SystemModelCheckReporter` instance. The enumeration `Severity` contains `INFO`, `WARNING`, `ERROR`, and `FATAL`. If a fatal error is reported, the user is not allowed to start the simulation.

6.3.5 Task Monitor

As explained previously, the `dispatch(long time)` routine of the scheduler interface not only has to select the task to run, but also has to start its execution and preempt the currently running task if necessary. Changing the state of a task is a basic functionality, which is why the simulation kernel provides adequate methods.

Such methods are defined in the `TaskMonitor` class. Each `Task` is linked to a `TaskMonitor` object via the `TaskManager`, which is a singleton class maintaining a map of tasks and their monitors. A task as part of the system model merely contains a set of parameters and its source code is generated. To calculate the remaining execution time in case of a preemption or to keep track of the current state of the task, additional fields are necessary, which do not belong to the system model and should not be serialized. Therefore, providing methods to control the state of the task by a separate monitor object and not by the task object itself helps to separate between the model element and other implementation details and leads to a sound design.

The `TaskMonitor` offers functions to change the state of a task and ensures that transitions between states are allowed according to the state diagram presented in figure 2.1 on page 19.

In case the state of a task is changed from *running* to *ready*, the `TaskMonitor` object stores the elapsed execution time and reloads it when the task continues to run. It also delays the start of the execution of a task according to the context switching time parameters. Most importantly, the `TaskMonitor` generates events introduced in section 6.1.4 on page 50.

6.3.6 Resource Monitor

Similar to a `TaskMonitor`, a `ResourceMonitor` object is linked to a `Resource` object via the `ResourceManager`. The `ResourceMonitor` provides methods to lock and unlock a resource, it keeps track of the usage of the resource, and monitors the number of free and occupied units. It also maintains a list of tasks which are waiting for the resource. Most importantly, it generates all necessary events.

6.3.7 Event Analyzer

The system model source code generated by EMF includes a notification mechanism as explained in section 5.2.3 on page 43. This notification mechanism can be used to detect interesting events in the simulation model and notify the user or pause the simulation if such an event occurs, as requested by an objective presented in section 1.2 on page 14.

The abstract class `AbstractEventAnalyzer` is notified whenever elements of the simulation model are changed or new elements are added. Each subclass of `AbstractEventAnalyzer` implements the abstract method `modelChanged(Notification notification)` and thus is able to detect events like priority changes, system ceiling changes, and deadline misses.

The user has the option to select each `AbstractEventAnalyzer` subclass as well as to change its notification state with the help of a GUI. If a subclass of `AbstractEventAnalyzer` is selected, it is notified of simulation model changes. Once it detects an event it searches for, depending on the notification state, the finding is only logged or the simulation is paused and the user is immediately informed.

`AbstractEventAnalyzer` subclasses are added by Eclipse's extension point mechanism similar to scheduler implementations. The `type` field of an `Event` is of type `String`, which enables future scheduler implementations to create new types of events without changing the model. Future `AbstractEventAnalyzer` subclasses might trigger these new events.

6.3.8 Operation of the simulator

The cooperation of the presented classes is explained by a UML sequence diagram. The diagram in figure 6.5 on the facing page shows the start of a simulation and all involved classes with a simulation model containing only one task with one execution command.

The `Simulator` offers a `start()` function which is called when the user clicks on correspondent GUI elements. This function initializes local variables like the simulation time, instantiates the `TimeAxis` and the `SimulationThread`, and initializes the scheduler by calling `initialize()`. To

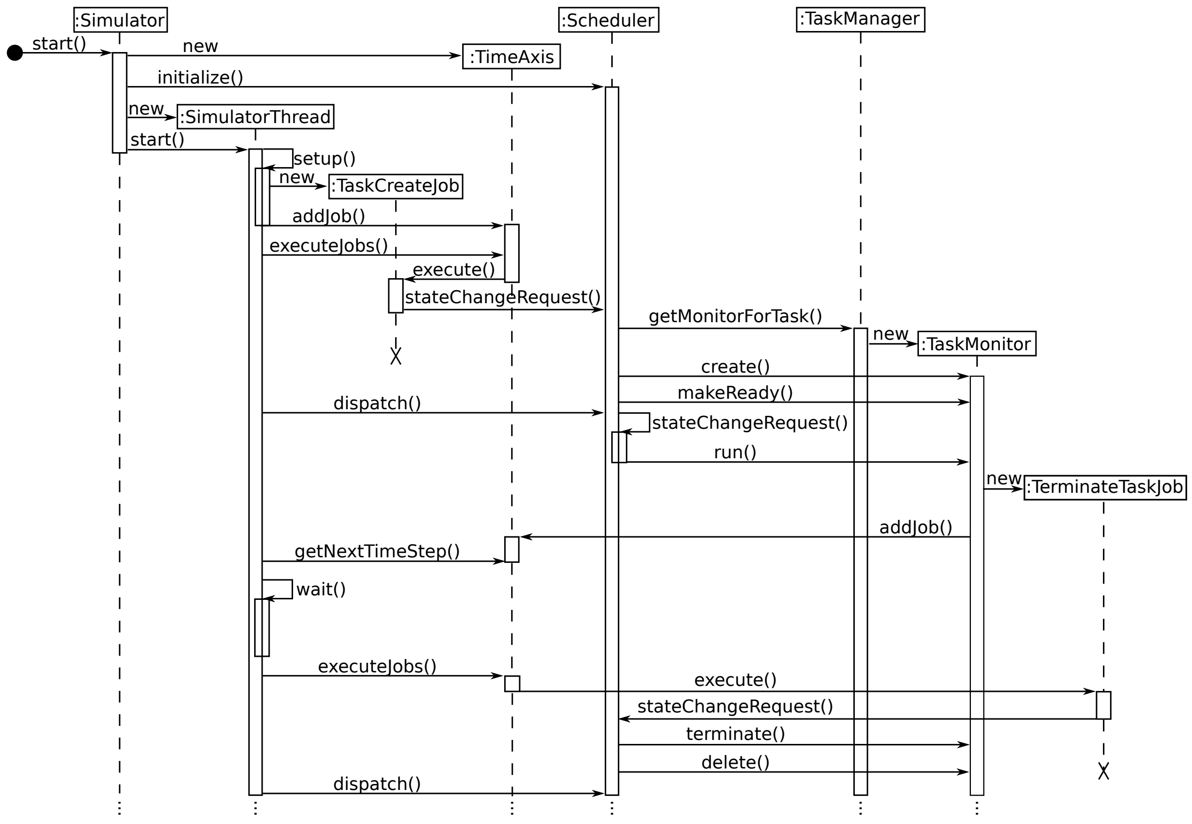


Figure 6.5: A UML sequence diagram of the start of a simulation with only one task. Answers of synchronous calls are omitted on grounds of of clarity and readability.

start a simulation, input stimuli are necessary. For the given system, these are the initial creations of the tasks. Therefore, as soon as the `SimulationThread` is started, it calls its own `setup()` routine and generates a `TaskCreateJob` for each task. All `TaskCreateJob`s are added to the `TimeAxis` at the time when they should be triggered.

The `SimulationThread` now enters its main loop and calls the `executeJobs()` function of the `TimeAxis`. In case no task is created at simulation time 0, this function does nothing. The `SimulationThread` will be informed of the time of the first job via `getNextTimeStep()`, wait for this time, and call `executeJobs()` again. However, under the assumption that one task is created at the current simulation time 0, the `TimeAxis` executes the `TaskCreateJob`, which informs the scheduler about this state change request via `stateChangeRequest()` and optionally adds a new `TaskCreateJob` to the `TimeAxis` if the task is repeating.

It is now up to the scheduler to decide what happens with the task. In most cases, it will change the task's state to *created* immediately. This can be achieved by making use of the `TaskMonitor`, which the scheduler will get from the `TaskManager`. The `TaskManager` maintains a map linking `Task`s to `TaskMonitor`s. In case the `TaskMonitor` does not exist already, it is created by the `TaskManager` and returned to the scheduler, which calls `create()` to change the state of the task to *created*. Depending on the scheduling policy, the scheduler changes the state from *created* to *ready* by calling `TaskMonitor`'s `makeReady()` function subsequently.

After all calls have been returned, the `SimulationThread` continues by calling the `dispatch()` method of the scheduler, because the `TaskCreateJob` changed the task's state and thus dispatching is required. The scheduler selects the task to run and starts its execution by calling its own `stateChangeRequest()` method, requesting the *running* state. In case another task was currently running, the scheduler would have to set its state to *ready* by the same means. Calling `stateChangeRequest()` effectively calls the `run()` function of the `TaskMonitor`, which calculates the duration the task was in state *ready*. For the time the task was in *ready*, a new `Event` is added to the `Task`. Assuming the `Task` has only one `Execution` command, the `TaskMonitor` creates a new `TerminateTaskJob`, which is added to the `TimeAxis` at the time the task will have finished its execution.

The `SimulationThread` determines the next simulation time step by calling the `getNextTimeStep()` function of the `TimeAxis`. To allow the visualization to update and to generate the feeling of a "live" system, the thread will wait in real-world time for the time skipped by the simulation time multiplied by the `schedulingSpeed` parameter of the `SimulationParameters`. In case the simulation was paused in between, the simulation thread will wait here until the simulation is continued.

After the waiting time is over, the simulation time is updated and the main loop starts from the beginning by calling the `executeJobs()` function of the `TimeAxis`. As there were no other `Job`s added to the `TimeAxis`, the `TerminateTaskJob` is located at the current simulation time and is executed. It calls the `stateChangeRequest()` method of the scheduler, which makes use of the `TaskMonitor`'s `terminate()` function to change the tasks state to *terminated*. The `TaskMonitor` calculates the duration the task was in *running* state and adds a respective `Event` to the `Task`. Depending on the scheduling policy, the scheduler subsequently changes the task's state from *terminated* to *non-existing* by a similar call.

Since there are no more `Job`s to be executed, the `SimulationThread` continues by calling the schedulers `dispatch()` function. Assuming that there are no more tasks, the scheduler simply returns the call. The `SimulationThread` determines the next simulation time step by calling the `getNextTimeStep()` and waits until the task is created again. The main loop ends as soon as the `Simulator`'s `stop()` function is called or `getNextTimeStep()` returns the maximum value of `long`, which indicates that there are no more `Job`s on the `TimeAxis`.

6.4 Visualization

Different visualization frameworks were compared and GEF was chosen to be the most appropriate in section 5.1 on page 33. GEF already provides helpful classes such as a scrollable visualization pane. It also imposes the MVC design pattern.

6.4.1 Edit Parts

According to the MVC design pattern, a model element is linked to its view, i. e. its graphical representation via a controller class. These controller classes are called *edit parts* in GEF. Therefore, for each element of the system model an edit part class extending `AbstractGraphicalEditPart` exists. The model hierarchy is translated to edit parts by overriding the `getModelChildren()` method of this class.

The abstract function `createFigure()` has to return an object implementing the `IFigure` interface, which is a general interface for a visual element in Draw2D. The Draw2D framework provides different geometrical primitives like polygons, rectangles, textual labels, and so on. All these elements implement the `IFigure` interface, which itself can contain arbitrarily many `IFigure` objects as children. Therefore, complex graphics are realizable by creating hierarchies of these basic elements.

6.4.2 Main View

The default window of an Eclipse RCP application is subdivided in so-called *views*. In order to be notified of changes of the simulation model, the main view containing the visualization implements the `SimulatorListener` interface and registers as an observer of the `Simulator`. This way, it gets notified whenever the simulation time increases and is able to update the visualization. The transfer of the simulation model to its visualization is performed by routines of the GEF and the edit parts explained in the previous section.

6.5 Exception Handling

An advantage using Java and the Eclipse RCP is that it exception handling features are already included. One of them is another view, called “Error Log”, which contains a list of status messages. With the help of the class `StatusManager` new messages can be added to this list. Is also supports different severity levels such as information, warning, and error. All exceptions which arise during the simulation, i. e. in the main loop of the `SimulationThread`, are caught and reported by the `StatusManager`.

In order to differentiate between general Java exceptions and errors of the scheduler implementation, e. g. incorrect changing the state of a task, the class `SimulationException` is introduced. In order to serve as a Java exception, it inherits from `java.lang.Exception`. A `SimulationException` is potentially thrown by methods of the `TaskMonitor` or the `ResourceMonitor` classes, since they are intended to be used by the scheduler to change the state of a task or a resource. The `SimulationException` contains a field to save a message providing detailed information about the exception. This message and the stack trace, which contains the list of methods that were called when the exception occurred, provide a detailed picture of what happened.

7 Implementation

In the previous chapters, different software libraries and frameworks were evaluated and the design of the simulation and visualization software was introduced. Based on these results, this chapter will present different aspects of the implementation of the application, which is named Simulation And Visualization Of Real-time Scheduling (SAVORS).

As shown in section 6.3.4 on page 54 of the previous chapter, implementations of scheduling policies and resource access protocols are merely plugged into the existing software and interact with the software, especially with the simulation kernel, via a predefined interface. For this reason, the details of implementing different schedulers are presented in the following chapter.

7.1 Eclipse RCP Specifics

This section introduces specific aspects that are attended by the Eclipse RCP, upon which the simulation and visualization software is built. These include UI elements, their connection to the internal functions, and the deployment of the software.

7.1.1 Plug-in Hierarchy

As an Eclipse RCP application, the software is composed of different plug-ins. An overview of the plug-ins developed for this work and their interdependencies is given in figure 7.1.

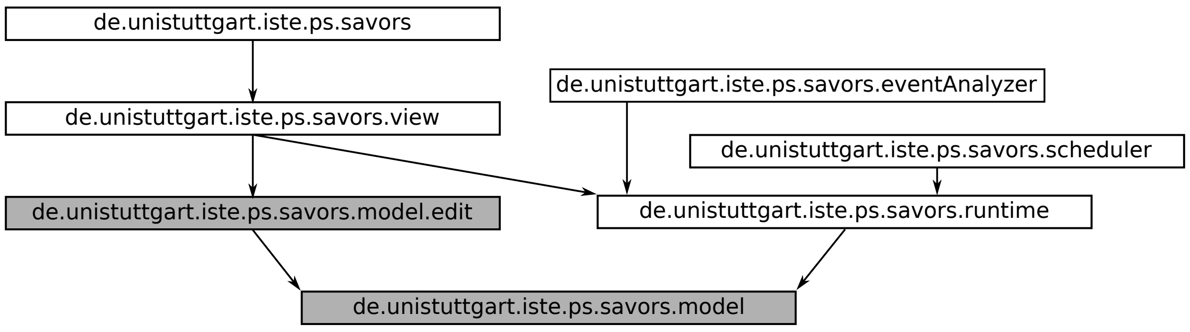

de.unistuttgart.iste.ps.savors
de.unistuttgart.iste.ps.savors.eventAnalyzer
de.unistuttgart.iste.ps.savors.view
de.unistuttgart.iste.ps.savors.scheduler
de.unistuttgart.iste.ps.savors.model.edit
de.unistuttgart.iste.ps.savors.runtime
de.unistuttgart.iste.ps.savors.model

Figure 7.1: An overview of the developed plug-ins and their dependencies. The source code of the shaded plug-ins is automatically generated.

The `de.unistuttgart.iste.ps.savors` plug-in contains some classes defining the Eclipse RCP application and the product definition which will be explained in the following. The `eventAnalyzer` and `scheduler` plug-ins contain the initial set of schedulers and event analyzers. The shaded plug-ins are automatically generated by the EMF from the system model. The `view` and `runtime` plug-ins will be described in section 7.2 and 7.3, respectively.

7.1.2 Commands and Handlers

In section 5.1.4 on page 37, the Eclipse extension point mechanism is introduced as part of the OGSi specification, allowing plug-ins to add further information and functionalities to other plug-ins.

The plug-in `org.eclipse.ui` defines the extension point `org.eclipse.ui.commands`. *Commands* are associated with UI elements such as menus, tool bar items, or key bindings. All these UI elements are declaratively defined with the help of extension points like `org.eclipse.menus` and `org.eclipse.ui.bindings`.

In order to link a command to a class containing the function to be executed when the associated UI is used, a *handler* is required. A handler is again defined with an extension point, namely `org.eclipse.ui.handlers`, and links a handler class to the command. Additionally, it is possible to activate a handler depending on the state of a variable. This allows one UI element to be sufficient to start, pause, and continue the simulation. This element is linked to the `de.unistuttgart.iste.ps.savors.commands.SimulationControlCommand`, which again is referenced by three different handlers, each of which is exclusively enabled depending on the state of a simulation. The state is determined by three Boolean variables, which are defined by the runtime plug-in and updated by the class `Simulator`.

7.1.3 Preferences

Eclipse provides a preference dialogue which is subdivided in pages. Fortunately, it can be extended and own preference pages can be added by extending the `org.eclipse.ui.preferencePages` extension point. Each page is defined by a name and a class extending the abstract class `FieldEditorPreferencePage`, which provides useful methods to create GUI elements, as presented by listing 7.1. The first parameter of the constructor of the `BooleanFieldEditor` is the preference node identifier, the second parameter is the text displayed to the user with the ampersand indicating the next character to be the mnemonic character, and the last parameter is the parent SWT element.

Listing 7.1 Creating and adding a Boolean preference node.

```
addField(new BooleanFieldEditor("ID", "&Preference", getFieldEditorParent()));
```

Preferences are saved by a `PreferenceStore` internally. Each plug-in can specify an `Activator` class, which is called as soon as the plug-in is loaded. The `Activator` class of each plug-in provides a method returning an instance of the `PreferenceStore`, as long as the `Activator` inherits from `AbstractUIPlugin`. Each preference node needs a unique identifier. In order to define the initial value of the preference nodes, a class extending `AbstractPreferenceInitializer` has to be added to the `org.eclipse.core.runtime.preferences` extension point. Its method `initializeDefaultPreferences()` is called whenever the preferences need to be initialized. More information about Eclipse preference features is contained in [Dau08].

7.1.4 Scheduler and Event Analyzer Extension Points

New scheduler and event analyzer implementations should be added as separate plug-ins. To be informed about the existence of new plug-ins, the Eclipse extension point mechanism is used. The `de.unistuttgart.iste.ps.savors.runtime` plug-in defines two extension points in its plugin.xml file as shown in listing 7.2.

Listing 7.2 Extension point definition for scheduler and event analyzer implementations.

```
<extension-point id="de.unistuttgart.iste.ps.savors.scheduler" name="Scheduler" schema="
    schema/de.unistuttgart.iste.ps.savors.scheduler.exsd"/>
<extension-point id="de.unistuttgart.iste.ps.savors.eventAnalyzer" name="EventAnalyzer"
    schema="schema/de.unistuttgart.iste.ps.savors.eventAnalyzer.exsd"/>
```

The linked XSD documents were generated with the help of a user dialogue provided by the Eclipse Plug-in Development Environment (PDE). They define what information is required by the extension point. The scheduler extension point requires exactly one Java class implementing the `IScheduler` interface. The event analyzer extension point requires one Java class extending the `AbstractEventAnalyzer` class and two Boolean variables, specifying whether the event analyzer should be selected initially and if it should notify the user in case of a detected event by default.

In the `Activator` class of the same plug-in defining the two extension points, the source code of listing 7.3 will collect all elements of the scheduler extension point and add an instance of the specified class to the `SchedulerManager`. In the same way the elements of the event analyzer extension point are loaded and added to the `EventAnalyzerManager`.

Listing 7.3 Loading elements of the scheduler extension point.

```
for (IConfigurationElement element : extensionRegistry.getConfigurationElementsFor(
    SCHEDULER_EXTENSION_POINT_ID)) {
  Object o = element.createExecutableExtension("class");
  if (o instanceof IScheduler) {
      SchedulerManager.addScheduler((IScheduler) o);
  }
}
```

7.1.5 Deployment

The set of plug-ins an Eclipse application consists of is defined in a product definition file. This file is an XML document which also allows to define the product's branding details like the launcher icon and the splash screen to be displayed while the application is started. It is contained in the plug-in `de.unistuttgart.iste.ps.savors`.

Adding every single plug-in to the product definition might become cumbersome, as all dependencies have to be taken care of. So-called *feature* definitions at an intermediate level between the product definition and the plug-ins provide relief. A feature is again defined by an XML document, specifying a set of plug-ins. In contrast to product definitions, it may also depend on other features. Therefore, instead of a long list of plug-ins, the product definition can also contain merely a few features. In order to use the Eclipse update mechanism Equinox Provisioning Platform (P2), a feature-based product definition is mandatory [Dau08].

With the help of the *Product Configuration Editor*, a special dialogue which is part of the PDE, the product definition file can easily be edited. This dialogue provides a reference to the *Eclipse Product Export Wizard*, another dialogue helping to generate the application. If the so-called *delta-pack* is installed within Eclipse, it is possible to export the product for various platforms, including Microsoft Windows, GNU Linux and Apple Mac OS X.

The building process internally relies on Apache Ant scripts. It is possible to create an own automated software building process with Apache Ant scripts by calling these internal scripts. An own building process can automatically create a JavaDoc documentation and invoke a coding style checking tool before building the software itself.

7.2 Visualization

The plug-in `de.unistuttgart.iste.ps.savors.view` contains all UI related classes like the handler classes described in section 7.1.2, the edit parts introduced in section 6.4.1, and the views which are added to the main window of the Eclipse RCP application. The main view displays the simulation result synchronously to its generation. It is implemented in the class `ScheduleView`, which extends the abstract class `org.eclipse.ui.part.ViewPart` and is added to the `org.eclipse.ui.views` extension point.

7.2.1 Element Overview

The `ScheduleView` contains a `ScrollingGraphicalViewer`, which is a part of the GEF and displays the graphical representation of the simulation model. After each update of the visualization, the horizontal scrollbar of the `ScrollingGraphicalViewer` is set to the rightmost position in order to show the current simulation time. However, an overview of the model elements like tasks, cores, and resources should stay at the left side to show which row of events belongs to which element.

This can be achieved by two different approaches. First, the `ScheduleView` inserts a layer which is separated in two parts. The left part contains the element overview and the right part contains the `ScrollingGraphicalViewer`. The splitting is achieved by making use of SWT's layout options.

The element overview has to be horizontally scrollable to contain all model elements independent of the size of the window. As the right part containing the events is scrollable, too, both scrollbars are linked in a way that if one is moved, the other one follows immediately.

As explained in section 6.4.1 on page 59, there exists a subclass of the class `AbstractGraphical-EditPart` for each model element, which generates the graphical representation of the element with the help of Draw2D. Although it is generally possible to make use of these edit parts for the element overview on the left side, it is not intended by the GEF that a model element is linked to more than one edit part. However, this would be necessary in order to show two different representations of the same model. Therefore, the overview on the left side is entirely modeled with SWT and updated only when a new model is loaded.

The second approach relies entirely on the GEF. In this solution, the edit part for the `SystemModel` displays the element overview on the left side by means of Draw2D functions. The same class also creates a scrollable container for the events next to the overview and sets the scrollbar to the rightmost position each time the simulation model is updated.

Although Draw2D is based on SWT, and SWT makes use of the UI functions provided by the OS as mentioned in section 5.1.4 on page 38, Draw2D provides its own scrollbar independent of the OS. Unfortunately, this also means that if the image in the `ScrollingGraphicalViewer` is zoomed in or out, the Draw2D scrollbar also grows or shrinks. Additionally, when exported as a raster image file, only the currently visible part inside the Draw2D scrollbar is depicted, whereas in the first solution, the whole time axis is saved. For this reason, the first approach is implemented.

7.2.2 Raster Graphics Export

Exporting the visualization of the resulting schedule as an image file is advantageous for adding comments or for including it into other documents. Therefore, a class `SaveImageCommandHandler` is set as a handler for the respective command. It opens a dialogue allowing the user to select the path and file name for the image file.

Since saving the image might take some time, the `SaveImageCommandHandler` contains an inner class of type `org.eclipse.core.runtime.jobs.Job`. A job runs in a separate thread and the progress is reported to the user in a special window provided by Eclipse.

Saving the image to a file makes use of SWT classes like `Image`, `GC`, `Graphics`, and `ImageLoader`. Basically, the topmost edit part containing all other edit parts of the visualization paints into an image buffer, which is saved to the image file. The following raster image file types are supported:

- PNG
- JPEG
- BMP
- Tagged Image File Format (TIFF)

- GIF

As explained in the previous section, the element overview is separated from the rest of the visualization. In order to add the element overview to the image, the class `SidebarUtil` returns an `IFigure` object which is constructed in a similar way as the element overview of the `ScheduleView`. This way is less complex than combining the element overview, implemented as a combination of SWT `Control` objects, with the image of the simulation result based on `IFigure` objects. Both `IFigure` objects are drawn side by side into the same image buffer.

7.2.3 Vector Image Export

In contrast to raster image formats, a vector image is described by geometrical primitives like polygons, rectangles, and textual labels and not pixel by pixel. This has the advantage that even at high magnification levels, the vector image does not lose quality and stays sharp. A common document format for vector images is SVG, which is standardized by the W3C. For more information about SVG please refer to [DDG+11].

Since the visualization of the simulation result is built of basic graphical elements, a vector image export seems obvious. The GEF internally supports exporting to the SVG format. It uses the Apache Batik library, which transforms Java2D graphics to SVG documents and is published under the Apache license. Since Java2D is based on AWT and thus is not compatible with Draw2D and SWT, the Draw2D graphics have to be converted. However, classes that transform Draw2D to Java2D images like the `org.eclipse.gmf.runtime.draw2d.ui.render.awt.internal.svg.export.GraphicsSVG` are not accessible from outside the framework. A bug report already describes this problem and requests making these internal classes public [Dut10]. Unfortunately, this has not happened up to the present date, thus the only way apart from rewriting all code or modifying the framework itself is to copy the respective internal classes.

Since this solution is far from ideal, the SVG export functionality is contained in an additional plug-in and therefore can easily be detached from the software.

7.2.4 Tool Bar Contribution

The user should be able to quickly change the speed of the simulation and the scale of the time axis. Both parameters are controlled by fields inside the `SimulationParameters` class, as explained in section 6.2 on page 50.

Therefore, the main tool bar containing control elements to control the simulation, to change the scheduling policy, and to activate event analyzers is extended by a combo box for each parameter. A combo box is a UI element which provides a predefined set of values and lets the user specify an own value.

A class implementing the abstract class `WorkbenchWindowControlContribution` has to implement methods returning an SWT control element like a combo box and can be added to the `org.eclipse.ui.menus` extension point. Implementing the `SimulatorListener` interface, the class is informed of changes of the parameter in case a new simulation model file is loaded. Since the `Simulator` is designed according to the singleton design pattern, changes made by the user via the combo box can easily be forwarded to the simulation kernel.

7.2.5 Property View

If the user clicks on a graphical element, detailed information like the priority and period of a task or the start time of an event as well as its duration should be presented. Eclipse provides the class `org.eclipse.ui.views.PropertySheet`, which is a view containing a table of the name and the value of each property of an element.

Fortunately, as the model is generated with the EMF as explained in section 5.2.3 on page 43, not only Java code implementing the model but also an "edit" plug-in containing classes to describe the model, so-called *item providers*, can be generated. Furthermore, the EMF can generate another plug-in, the "editor" plug-in, which provides an entire Eclipse RCP application with a UI allowing the user to edit the model. The item providers are used by the editor plug-in to determine the name and the representation of each model element and its properties.

The generated code of the editor plug-in served as a reference for adding a property view for the simulation model elements to the software. The data of the `PropertySheet` view has to be provided by a `IPropertySheetPage`. The EMF provides an `ExtendedPropertySheetPage`, which implements this interface and determines the name of each property with the generated item provider classes of the edit plug-in. This is why the plug-in containing the views depends on the generated edit plug-in as shown in figure 7.1 on page 61.

The class `ScheduleView` contains the `ScrollingGraphicalViewer`, which finally displays the simulation model. In order to update the property view when the user selects a visual element, the `ScheduleView` has to implement the `ISelectionProvider` interface and listen to selection changes of the viewer object. When the user clicks on an element in the visualization, the `ScrollingGraphicalViewer` returns an object of type `IStructuredSelection`, which contains a list of the selected edit parts. Since the `ExtendedPropertySheetPage` only handles elements of the `SystemModel`, the selection has to be transformed to contain the respective model elements.

In order to link the property view to the `IPropertySheetPage`, Eclipse makes use of the adapter design pattern introduced in [GHJV95]. An adapter class helps to provide a common interface for two classes, which otherwise would be incompatible. The `ScheduleView` contains the method `getAdapter(Class type)`, which returns the `ExtendedPropertySheetPage` object as an implementation of the `IPropertySheetPage` interface. When the property view is visible to the user, it is notified when the selection changes. Thereupon, the selection provider, in this case the `ScheduleView`, is queried for an object implementing the `IPropertySheetPage` interface via the adapter pattern. The returned `ExtendedPropertySheetPage` is then used to determine how the properties of the selected object should be displayed. More information about showing properties of model elements is given by [MDG+04].

7.3 Simulation Kernel

The `de.unistuttgart.iste.ps.savors.runtime` plug-in contains all classes necessary to run the simulation. Apart from the `Simulator`, all classes extending the abstract `Job` class, the `TimeAxis`, the `TaskMonitor`, and the `ResourceMonitor` as well as the `AbstractEventAnalyzer` and the `IScheduler` interface are part of this plug-in. Additionally, the `SimulationException` is defined in runtime plug-in.

7.3.1 Simulator

The class `Simulator` provides methods to start, pause, continue, and stop the simulation, which is driven by the inner class `SimulationThread`. These methods also update the state of the variables enabling the different handlers for the simulation control command, as explained in section 7.1.2 on page 62.

The inner class `SimulationThread` contains the main loop and thus drives the whole simulation. The code of the main loop is presented in listing 7.4 on the facing page. Whenever the simulation time increases, all registered observers of the `Simulator` are notified.

7.3.2 Time Axis and Jobs

The `TimeAxis` class contains a map linking a simulation time to a list of `Job` objects. If a `Job` object is added to the `TimeAxis`, it first checks whether the simulation time is not in the past or equals the current simulation time. In the former case, a `SimulationException` is thrown, in the latter case the job is executed immediately. Otherwise the job is added to the map at the given time.

New `Job` objects might be added during the execution of another `Job` object. This also means that the `TimeAxis` is currently iterating over the list of `Job` objects and modifying this list leads to a `ConcurrentModificationException`. Therefore, these new `Job` objects are saved in another list and added afterwards.

Each `Job` has to implement the function `isDispatchNecessary()` returning a Boolean value. All return values of the `Job` objects with a common simulation time are combined with the *OR* operator after their execution. Hence only one `Job` object requesting the call of the dispatch routine is sufficient for the simulation kernel to call the scheduler.

7.3.3 Task Monitor

As stated in section 6.3.5 on page 55, each `TaskMonitor` object is linked to exactly one task via the singleton class `TaskManager`.

The `TaskMonitor` is implemented as a state machine and provides methods to change the state of the task. In case the state transition is not allowed according to the state transition diagram

Listing 7.4 The main loop of the simulation thread, exception handling is omitted to enhance readability.

```java
setup(); // create the initial task creation jobs
while (!stopped) {
  // execute the generated jobs
  dispatchNecessary = timeAxis.executeJobs(globalTime);

  if (dispatchNecessary) {
     scheduler.dispatch(globalTime);
  }

  long nextTime = timeAxis.getNextTimeStep();

  // test if there is anything to do at all
  if (nextTime == Long.MAX_VALUE) {
      // Stop simulation, as there are no further events.
  } else

  // update our listeners
  for (final SimulatorListener listener : listeners) {
      listener.simulationTimeUpdated();
  }

   try {
      if (paused) {
         synchronized (this) {
            while (paused)
               wait();
         }
      }

      // "live" feature: wait in real-world time for the simulation time span.
      synchronized (this) {
          final long timeToWait = (long) ((nextTime - globalTime) / getSpeedFactor());
          if (timeToWait > 0) {
              wait(timeToWait);
          }
      }
  } catch (final InterruptedException e) {
      // nothing, we've been woken up from being paused
  }

  globalTime = nextTime;
}
```

depicted in figure 2.1 on page 19, a `SimulationException` is thrown. Otherwise the respective `Event` objects are created and added to the simulation model.

Once a task is in state *running*, all `Commands` of the task are executed in the order of their definition in the system model. The `TaskMonitor` saves the current command number and the executed time in case a running task is preempted. Once the task is resumed, it continues with the same command and the remaining execution time. If the current command requests or releases a resource, the respective methods of the `IScheduler` interface are called. If the scheduler implementation allows the task to lock the resource, it has to set the state of the task to *running* again, in order to execute its next command. Additionally, a list of locked resources and a list of resources blocking the task are maintained. The current implementation does not allow two consecutive execution commands, as this would add a superfluous call of the dispatch method of the `IScheduler` interface and the two execution commands can easily be combined to one.

7.3.4 Resource Monitor

Similar to the `TaskMonitor`, each `ResourceMonitor` is linked to exactly one resource via the singleton class `ResourceManager` as explained in section 6.3.6 on page 56.

The `ResourceMonitor` offers methods to lock and unlock a resource. The function `lock(long time, Task lockingTask, int units)` returns `true` if enough free units are available. The `ResourceMonitor` also provides a method returning the dynamic ceiling priority, which depends on the number of available units as described in section 6.1.3 on page 49. Note that the lock function does not check the task's priority, as this is the duty of the scheduler implementation and differs between the resource access protocols. The `ResourceMonitor` creates all necessary `Event` objects and adds them to the simulation model. Additionally, a list of tasks locking the resource and a list of tasks waiting on the resource are maintained.

7.3.5 Event Analyzer

The runtime plug-in contains the `EventAnalyzerManager` and the `AbstractEventAnalyzer` classes. The former saves the selection and notification state of each event analyzer with the help of Eclipse preference nodes. The latter has to be inherited by every event analyzer. Each subclass has to implement the `modelChanged(Notification notification)` method, which will be called whenever the simulation model is changed. The `Notification` object is part of the EMF and contains the new or changed object of the simulation model. By checking whether the new object is of type `Event` and by comparing the type of the `Event`, e. g. a deadline event can be triggered for. If such a deadline event occurs and the same task is not in terminated state yet, a deadline was missed and the user can be informed, using the `reportIncident(Severity severity, String message, boolean show)` method of the `Simulator` class.

In some cases, it might be necessary to request additional parameters from the user once the event analyzer is activated. Therefore, each subclass of `AbstractEventAnalyzer` has to implement the `configure(Shell parentShell)` method, which is called when the event analyzer is selected. The `Shell` class is part of the SWT and necessary to create and display UI elements. Note that in case the event analyzer is selected initially, the same method is called with the parameter set to `null`, as the user should not be bothered with event analyzer configuration details before the application is fully loaded.

8 Scheduling Policies

This chapter presents different scheduling policies and resource access protocols and how they are implemented as part of the set of initially available schedulers. Except for the last section, all described scheduling policies are defined for a single-core environment and their properties only hold within such systems.

The last but one section explains the procedure of adding further scheduling policies to an existing installation of the software. A partitioned multi-core scheduling policy serves as example and is implemented this way. Therefore, it is not available by default. Each scheduler has to implement the `IScheduler` interface and has to be added to the `de.unistuttgart.iste.ps.savors.scheduler` extension point as described in section 7.1.4 on page 63.

8.1 Cyclic Executive

Under the *Cyclic Executive* scheduling policy, the order of execution of the periodic tasks is determined once before the execution. As the resulting schedule is precomputed, it belongs to the class of *off-line* scheduling policies [Liu00, p. 77]. A time window of constant length, the *major cycle*, is subdivided into *minor cycles*. Each minor cycle contains a set of tasks in such way that all tasks are executed within their period. The major cycle is repeated infinitely.

The implementation of a Cyclic Executive is rather straightforward. The dispatch routine selects and executes the task in the sequence of the fixed schedule or simply waits for the next minor cycle to begin. Once the end of the predefined sequence, i. e. the major cycle is reached, it is started again from the beginning.

The disadvantage of the Cyclic Executive is that the construction of the schedule is a bin-packing problem, which is known to be NP-hard [BW01]. This problem needs to be solved for each set of tasks, and it is not possible to add tasks or modify the parameters of existing tasks without having to compute a new solution. Additionally, sporadic tasks are hard to incorporate into this schedule. More information about the Cyclic Executive is given in [BW01, Liu00].

The Cyclic Executive scheduler implementation is written for a set of tasks defined by a specific simulation model, thus it has to exclude all other simulation models. The implementation makes use of the `checkSystemModel(SystemModelCheckReporter reporter, SystemModel systemModel)` method of the `IScheduler` interface, which is called after a new simulation model is loaded but prior to the simulation. If the name of a simulation model and the contained tasks are not as expected, the `reporter` is notified about a finding of fatal severity. Hence, the user is not allowed to start the simulation and has to select another scheduling policy for the simulation model.

8.2 Fixed-Priority Preemptive Scheduler

The fixed-priority preemptive scheduling policy belongs to the class *on-line* scheduling policies [Liu00, p. 78]. This means that, in contrast to the Cyclic Executive, it makes its scheduling decisions at run-time. Only the priority value, which is assigned to each task, is static and determined before the execution. It will later be extended by a dynamic priority value, which is required by some resource access protocols.

The dispatch routine selects and executes the task with the highest priority among all tasks in *ready* state. A higher priority is represented by a higher integer value of the `priority` field, which is according to [BW01, p. 470] but in contrast to [Liu00, p. 166]. If a task with a higher priority than the currently running task becomes ready, the running task is preempted, i. e. set in *ready* state, and the higher-priority task starts running. In case there are no tasks in *ready* state or the currently running task has a higher priority, it continues execution. If there are no tasks in *ready* state and no task is currently executing, the dispatch routine simply does nothing. Additional information about the fixed-priority preemptive scheduling policy is given in [BW01, Liu00].

The fixed-priority preemptive scheduler implements a default resource access protocol. If a task requests a shared resource but fails to lock it, its state is set to *blocked.* As soon as the resource is unlocked again, the state of the blocked task is set to *ready* again. After each resource interaction the dispatch routine is called, since the state of a task might have been changed and a scheduling decision is necessary.

8.2.1 Task Creation Delay

The previously introduced fixed-priority preemptive scheduling policy is designed for single-core systems. The system model does not restrict a task from having a longer WCET than its period. Thus, it is generally possible for the same task to create a new instance while a previous instance of the same task is currently running. As explained in section 2.4 on page 19, on systems with only one processing unit, tasks can only be executed concurrently, not really in parallel. Hence, for a task to start executing while another instance of the same task is running, and “the release of a process will be delayed until any previous releases of the same process have completed” [BW01, p. 498]. For this reason, the `stateChangeRequest(long time, Task task, State newState)` method checks whether a task which requests its state to be set to *created* is currently in the *non-existing* state. If that is not the case, the state change is deferred until the current instance of the task is terminated. Since more than one creation request could arise while the same instance of the task is still not in *terminated* state, the number of deferred requests is monitored.

If a periodic, infinitely repeating task with a period shorter than the sum of the durations of its execution commands is defined in a single-core system, the core utilization is greater than one and a schedule is infeasible. However, if the number of repetitions is limited or if a sporadic task with a minimum inter-arrival time shorter than the duration of its execution is defined, it might still be possible to derive a schedule using the described mechanism.

8.2.2 Rate-Monotonic and Deadline-Monotonic Scheduler

The fixed-priority preemptive scheduling policy is implemented as an abstract class and serves as super class for the RMS and DMS implementations. The difference between these two protocols is the assignment of the task priorities. The RMS requires a task with a shorter period to be assigned a higher priority. Additionally, the deadline of a task has to equal its period. In contrast, the DMS requests a task with a shorter deadline to be assigned a higher priority and allows arbitrary periods.

Since the priority values are defined by the user in the simulation model, the only difference in the implementation of the two scheduling policies is in the `checkSystemModel( SystemModelCheckReporter reporter, SystemModel systemModel)` function, where a correct priority assignment is verified. If the assignment is found to be incorrect, the priority values are overwritten by correct values and the user is notified.

8.2.3 Priority Queues

The tasks in *ready* state are maintained by the class `PriorityQueues`. As soon as two tasks have the same priority, some kind of resolution mechanism is required. As will be explained in section 8.7.3 on page 78, some resource access protocols demand a First In First Out (FIFO) regime for such cases. For this reason, the class `PriorityQueues` contains a queue under FIFO regime for each priority level and provides methods to access the highest priority task and to add tasks at the beginning or at the end of these queues.

8.3 Fixed-Priority Non-Preemptive Scheduler

The implementation of a non-preemptive fixed-priority scheduler is very similar to the preemptive fixed-priority scheduler described in the previous section. In contrast to the preemptive version, the currently running task is not allowed to be preempted by any other task, even if the other task has a higher priority. The preemptive fixed-priority scheduler is in general more reactive, thus it usually is preferred to the non-preemptive version [BW01, p. 470].

The non-preemptive fixed-priority scheduler contains the same basic resource access protocol as was implemented by the preemptive scheduler. However, assuming that no task self-suspends, all shared resources a task requests are free since a task cannot be preempted once it has started running.

8.4 Simple Priority Inheritance

Under the fixed-priority preemptive scheduling policy, a high-priority task c might be blocked because it requests a resource currently locked by a low-priority task a. This behavior is called *priority inversion*. Moreover, tasks with a priority between the priority of tasks c and a can preempt task *a* and thus increase the time task *c* is blocked. As the highest-priority task has the shortest deadline under DMS and under RMS regime, the priority inversion effect needs to be minimized to schedule all tasks within their deadlines. More information about priority inversion is given in [BW01, Liu00].

One method of limiting the described effect is the use of the *simple priority inheritance* protocol. Under priority inheritance regime, the priority of a low-priority task *b* is promoted and becomes equal to the priority of the high-priority task *c* for the duration while task *b* blocks task *c* [BW01, Liu00, SRL90]. The *simple* priority inheritance protocol limits this inheritance to one step, i. e. if task *b* is blocked itself by lower-priority task *a*, the priority of task *a* is *not* promoted. The transitivity is limited only for educational reasons. Transitive priority inheritance prevents priority inversion in a larger number of cases.

For this protocol, the priority of a task has to be dynamic [BW01]. In order to maintain the original priority value, which is part of the simulation model, the active priority is saved by the `TaskMonitor` object.

The simple priority inheritance protocol is implemented as an abstract class inheriting from the fixed-priority preemptive scheduler and overwriting the `requestResource()` and `releaseResource()` methods. If locking a resource fails, the task which currently holds the lock of the resource inherits the priority of the requesting task. In case the resource contains multiple units and is locked by more than one task, the task with the highest priority among the locking tasks is promoted [Liu00, p. 315]. Once the promoted task releases the resource, its priority value is reset to its previous value. Note that if the priority of a task is raised, the task is added to the tail of the FIFO queue of its new priority value, whereas if the priority is reset from a higher priority, the task is added to the head of its new FIFO queue [Liu00].

Similar to the fixed-priority preemptive scheduler, this protocol is implemented as an abstract class, in order to be inherited by an RMS and a DMS version.

8.5 Transitive Priority Inheritance

As the name suggests, the transitive inheritance protocol extends the simple priority inheritance protocol by transitivity. The priority inheritance routine is implemented recursively. If a task is blocked by a lower-priority task, the lower-priority task inherits the blocked task's priority. If the lower-priority task is blocked itself, the task which blocks the lower-priority task also inherits the new priority value. This is continued until a task blocking a higher-priority task is able to execute. In the end, the whole chain of tasks blocked on each other including the last, executable task inherit the first task's priority.

Neither the simple nor the transitive priority inheritance protocol prevents deadlocks [BW01]. In case task *b* requests a resource, it will be blocked by task *a* if task *a* currently holds the lock of the same resource. However, if task *a* requests another resource currently locked by task *b*, task *a* is blocked itself by task *b*. According to the protocol, the priority of task *a* is promoted and set equal to the priority of task *b*. Due to transitivity, as task *a* is blocked, too, the priority of task *b* has to be promoted as well. Without further measures, this would lead to an infinite recursion in the scheduler. By saving a list of tasks which have already been promoted at the current simulation time, the scheduler implementation is able to prevent the infinite recursion. In this case, the user is notified about the detection of a deadlock and the simulation is stopped.

The class implementing the transitive inheritance protocol is an abstract subclass of the fixed-priority preemptive scheduling policy implementation. It is extended by an RMS and a DMS version.

8.6 Original Ceiling Priority Protocol

The *Original Ceiling Priority Protocol (OCPP)* was presented by Sha, Rajkumar, and Lehoczky in 1990 under the name *Priority Ceiling Protocol* [SRL90]. Burns and Wellings define it by the following set of rules:

1. "Each process has a static default priority assigned (perhaps by the deadline monotonic scheme)." [BW01]
2. "Each resource has a static ceiling value defined, this is the maximum priority of the processes that use it." [BW01]
3. "A process has a dynamic priority that is the maximum of its own static priority and any it inherits due to it blocking higher-priority processes." [BW01]
4. "A process can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource (excluding any that is has already locked itself)." [BW01]

The OCPP ensures that if a shared resource is locked by task a and blocks a higher-priority task b, no other resource that could block b is allowed to be locked by any task other than a [BW01]. Hence, transitive blocking cannot occur and a task is blocked not more than once on a lower-priority task. Additionally, this protocol prevents deadlocks. More information about OCPP is given in [BW01], in [Liu00] under the name Basic Priority-Ceiling Protocol and in [But11, SRL90] under the name Priority Ceiling Protocol.

The OCPP is implemented as a subclass of the simple priority inheritance protocol, as it needs to access the method calculating the new, promoted priority for the task currently blocking one or more higher priority tasks.

The `requestResource()` method is overridden in such a way that as soon as a task requests a resource R, it is checked whether resource R has enough units available. If not, the task is blocked and the task with the highest priority among the tasks locking resource R is promoted.

If there are enough free units available, the task's active priority needs to be higher than the current system priority ceiling in addition. According to rule 4, this means that the task's priority has to be higher than the priority ceiling of any resource currently locked, except the resources locked by the task itself. If task's active priority is higher than the current system priority ceiling, the task locks the requested resource R and the system priority ceiling value is updated.

If the task fails the second test and its active priority is lower than or equal to the current system priority ceiling, the task is blocked by the highest priority ceiling resource Q. According to the priority inheritance rule, the priority of the task with the highest priority among the tasks locking Q is promoted. Note that resource Q could, but does not have to equal resource R, which was actually requested by the task. Finally, the dispatch routine is called.

Somewhat less complex is the overridden `releaseResource()` method. Once a resource is released by a task, the state of all tasks waiting on the released resource is set to *ready*. As mentioned above, tasks which are blocked because of failing the second condition are waiting on the highest priority ceiling resource. Therefore, the tasks which are set to *ready* state again did not necessarily request the released resource. In case the task's priority was promoted while it locked the resource, it's priority value is reset. After the system priority ceiling is updated, the dispatch routine is called.

Similar to the previous protocols, the OCPP implementation is sub-classed to derive a DMS and an RMS version.

8.7 Immediate Ceiling Priority Protocol

The *Immediate Ceiling Priority Protocol (ICPP)* is a modification of the OCPP. It has the same worst case behavior, but is generally more easy to implement [BW01]. It is also known as Priority Protect Protocol in POSIX, Priority Ceiling Emulation in Real-Time Java [BW01], Ceiling Priority Protocol in Ada [Liu00] and Highest Locker Protocol [COF12].

8.7.1 ICPP definition

The ICPP is defined by Burns and Wellings by the following set of rules:

1. "Each process has a static default priority assigned (perhaps by the deadline monotonic scheme)." [BW01]
2. "Each resource has a static ceiling value defined, this is the maximum priority of the processes that use it." [BW01]
3. "A process has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked." [BW01]

Another very similar specification, given by McCormick, Singhoff and Hugues, is based on the following rules:

1. "Each task has a static and a dynamic priority. The static priority is assigned according to rules such as Rate Monotonic or Deadline Monotonic." [MSH11]

2. “Each shared resource has a priority. This priority is called a *priority ceiling* and its value is equal to the maximum static priority of all the tasks which use the shared resource.” [MSH11]
3. “The scheduler always selects the ready task with the highest dynamic priority. The dynamic priority is equal to the maximum of the task’s static priority and the ceiling priorities of any resources the task has locked.” [MSH11]

In contrast to the OCPP described in the previous section, the priority of a task locking a resource is immediately promoted to the resource’s priority ceiling, not only in case the task blocks a higher priority task by a locked resource.

8.7.2 Unspecified Behavior

The property of OCPP that a task can be blocked only once is reinforced for the ICPP, as “a process will only suffer a block at the very beginning of its execution. Once the process starts actually executing, all the resources it needs must be free.” [BW01]

However, applying the rules presented above with a sample set of tasks and resources, it can be shown that this statement does not hold in general as described in the following.

Consider a situation with one resource R and three tasks a, b, and c, in which task a has the lowest priority, task b has a higher priority than task a, and task c has the highest priority. Task a and task b both request the resource R during their execution. According to rule 2 of both definitions, the priority ceiling of resource R equals the priority of task b, since task b is the task with the highest priority accessing R. Furthermore, task a will start its execution before task b, and task b starts before task c.

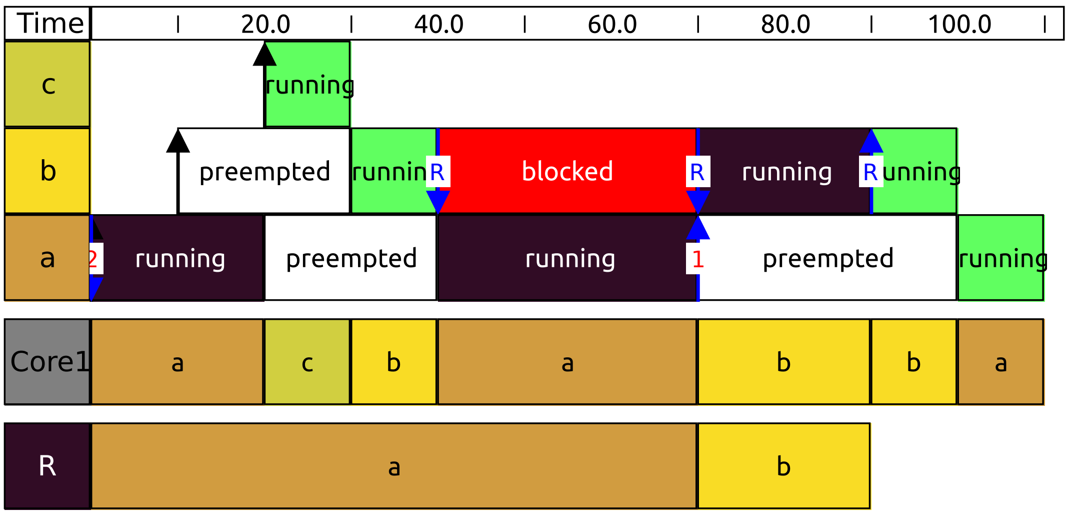


Figure 8.1: Simulation result under an underspecified version of ICPP, as task b is blocked after its execution started.

The result of the simulation of the described task set is depicted in figure 8.1 on the preceding page. Task b is blocked at time 40. This clearly violates the property that a task will only suffer a blockage at the very beginning of its execution. So what happened? At the beginning, task a is the only task in *ready* state, hence it starts executing and locks resource R. According to rule 3, its active priority is immediately promoted to the priority ceiling of resource R, i. e. 2. At the time the state of task b is set to *created*, it has the same active priority as task a and therefore is not allowed to preempt task a. As soon as task c is set to *ready*, it preempts task a, as its active priority is higher. As the highest-priority task, task c finishes without being interrupted anymore. At the time task c terminates, there are two *ready* tasks, a and b, with the same active priority of 2. The rule set of Burns and Wellings as well as the rule set of McCormick, Singhoff and Hugues provide no rule for such a case, thus the task to be scheduled is unspecified. If task b is chosen arbitrarily as shown in figure 8.1 on the previous page, it will be blocked by task a as soon as it requests resource R. If task a was chosen, all properties of the ICPP would hold since task a would be preempted by task b after it has released resource R. Task b would execute without being blocked or preempted and task a would finally finish its execution.

Note that for the OCPP, it is generally possible for two tasks to have the same priority, too. However, according to rule 3 in the definition of OCPP presented in section 8.6 on page 75, a task only inherits another task's priority if it blocks that task. Therefore, it is impossible for two tasks to be *ready* and have the same priority at the same time, as one of the tasks is blocked by the other one.

8.7.3 Extended ICPP Definition

With the Ceiling-Priority Protocol and the Stack-Sharing Priority-Ceiling Protocol, Liu describes two different specifications of the ICPP [Liu00]. The former is implemented by Ada 2005 [Liu00, TDB$^+$07], whereas the latter is derived from Baker's protocol [Liu00].

The Stack-Sharing Priority-Ceiling Protocol is defined by a set of rules rather different from the ones presented in section 8.7.1 on page 76, because it is developed from a different approach and out of a different motivation. Nevertheless, the Ceiling-Priority Protocol and the Stack-Sharing Priority-Ceiling Protocol produce the same schedule for all tasks as long as a task never self-suspends. [Liu00]

The Ceiling-Priority Protocol is described similarly to the definitions of section 8.7.1 on page 76, namely as follows:

1. "*Scheduling Rule:*" [Liu00]
   a) "Every job executes at its assigned priority when it does not hold any resource. Jobs of the same priority are scheduled on the FIFO basis." [Liu00]
   b) "The priority of each job holding any resource is equal to the highest of the priority ceilings of all resources held by the job." [Liu00]

2. "*Allocation Rule:* Whenever a job requests a resource, it is allocated the resource." [Liu00]

Rule 1.a) of the Ceiling-Priority Protocol defines the order of tasks with the same priority, which was unspecified by the previous definitions. "Jobs of the same priority are scheduled on the FIFO basis." [Liu00, p. 303] It is very important to note that if a task is preempted by a higher priority task, "it is added at the head of the ready queue for its active priority." [TDB$^+$07, p. 521].

For the sample task set presented in the previous section this means that at the very beginning task a is added to the tail of the queue of priority 1. As soon as it locks resource R and inherits its priority ceiling, it is removed from the queue of priority 1 and added to the tail of the queue of priority 2. At the time task b is set to *ready*, it is added to the tail of the queue of priority 2. When task c starts running and preempts task a, it is added to the head of the queue of priority 2 again, which will cause task a to be continued after task c has terminated. Therefore, the simulation of the same task set as described in section 8.7.2 on page 77 with a FIFO regime for tasks of same priority leads to the correct result, as presented in figure 8.2.

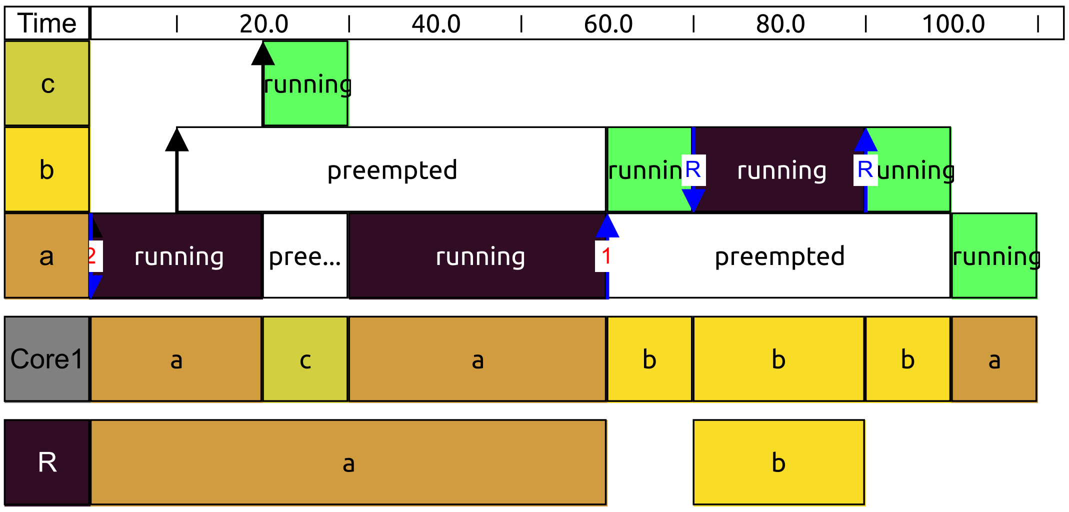


Figure 8.2: Simulation result under a sufficiently specified version of the ICPP.

Comparison of Tasks and Jobs

The reader may have noticed that the definition of the Ceiling-Priority Protocol is based on *job*s, not on tasks. These jobs must not be confused with the simulator jobs on the time axis mentioned in section 6.3.3 on page 53. According to [Liu00, p. 26], "each unit of work that is scheduled and executed by the system [is called] a job" and "a set of related jobs which jointly provide some system function [is called] a task".

Similar to the definition of a task given in section 2.3 on page 18, jobs are executed on processing units [Liu00, p. 26]. Furthermore, jobs can periodically or sporadically become available for execution, are assigned relative deadlines, and might require shared resources for specific time intervals [Liu00, p. 27, 38, 56]. These points indicate a great similarity between Liu's definition of a job and the definition of a task given in section 2.3.

The definitions are different for the period parameter. Liu's jobs only contain release times and the period is defined by the task the job is contained in. However, this means that the period parameter of a task can be expressed by multiple release times of a job.

Furthermore, jobs can have dependencies on each other, which "constrain the order in which they can execute" [Liu00, p. 42]. Fortunately, if there is only one processing unit, an *effective* release time and an *effective* deadline can be derived, which incorporates these precedence constraints [Liu00, p. 65-67]. Hence interdependencies between jobs can be neglected for single-core scheduling protocols like the ICPP by deriving effective release times and deadlines.

As long as interdependencies can be neglected and the period parameter can be replaced by multiple release times, the rules based on jobs are valid for tasks as defined in section 2.3 as well. More information about the different concepts of tasks is given in [Liu00, p. 40, 57].

8.7.4 ICPP Implementation

The ICPP is implemented as an abstract subclass of the fixed-priority preemptive protocol implementation.

The `requestResource()` method is overridden to realize the rules presented above. Once a task locks a requested resource, its priority is immediately promoted to the maximum priority ceiling of all resources it currently locks. Since the ICPP guarantees that all resources a task requests are free once it starts executing, an exception is thrown if a resource could not have been locked successfully.

As no task is ever blocked under the ICPP, the overridden `releaseResource()` method does not need to change the state of any task to *ready*. Instead, it merely unlocks the resource and updates the active priority of the releasing task.

The ICPP class is inherited by a DMS and an RMS version.

8.8 Differentiation of Blocking States

The section will provide information about a differentiation of blocking types like direct, inheritance and ceiling blocking. Although the application of the different blocking states does not affect the resulting sequence of scheduled tasks, i. e. the schedule, in most cases, there are certain differences especially in the visualization which have to be considered.

8.8.1 Direct blocking

The general definition of *blocking* according to LIU is:

> "When the scheduler does not grant η_i units of resource R_i to the job requesting them, the lock request $L(R_i, \eta_i)$ of the job fails (or is denied). When its lock request fails, the job is *blocked* and loses the processor. A blocked job is removed from the ready job queue. It stays blocked until the scheduler grants it η_i units of R_i for which the job is waiting for. At that time, the job becomes *unblocked*, is moved backed to the ready job queue, and executes when it is scheduled." [Liu00, p. 279]

In contrast to this definition, according to which a task is only blocked if it is denied to lock a resource, a different definition for *direct* blocking is given by several authors: "A higher-priority job J_h is said to be *directly blocked* by a lower-priority job J_l when J_l hold some resource which J_h requests and is not allocated." [Liu00, p. 282] "A job J is said to be blocked by the critical section $z_{i,j}$ of job J_i if J_i has a lower-priority than J but J has to wait for J_i to exit $z_{i,j}$ in order to continue execution." [SRL90, p. 1176] "If a process is waiting for a lower-priority process, it is said to be **blocked**." [BW01, p. 486] So, according to the direct blocking definition it is sufficient to have to wait on another task for a task to be directly blocked.

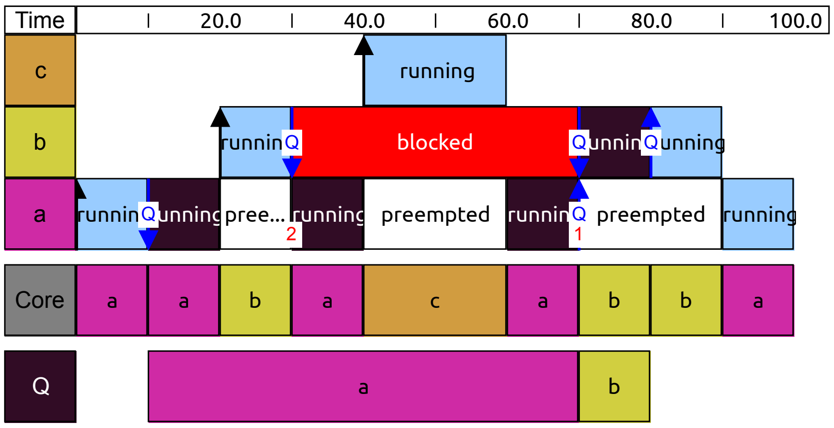
The direct blocking definition does not explicitly include what happens when a low-priority task blocking a middle-priority task is preempted by a high-priority task. However, example 2 of [SRL90, p. 1179] and figure 13.8 of [BW01, p. 492] clearly indicate that since the low-priority task is preempted itself, the middle-priority, blocked task is no longer blocked, but also preempted. For this reason, a transition from *blocked* to *ready* in the task state diagram depicted in figure 2.1 on page 19 is required. Preempting the blocked task results in a different duration of the *blocked* state in contrast to the general definition, which defines that the blocked task "stays blocked until the scheduler grants it η_i units of R_i for which the job is waiting for" [Liu00, p. 279] as illustrated in figure 8.3 on the following page.

LIU's theorem 8.2 for OCPP states that "when resource accesses of preemptive, priority-driven jobs on one processor are controlled by the priority ceiling protocol, a job can be blocked for at most the duration of one critical section." [Liu00, p. 295] The theorem was proven by SHA et al. in [SRL90]. This property does not hold in the schedule generated under the general blocking definition presented in figure 8.3a on the next page. The low-priority task a locks the resource Q for 30 time units, but the higher-priority task b is blocked by a for 40 time units. Therefore, the theorem only holds if the definition of direct blocking including the preemption of directly blocked tasks is applied.

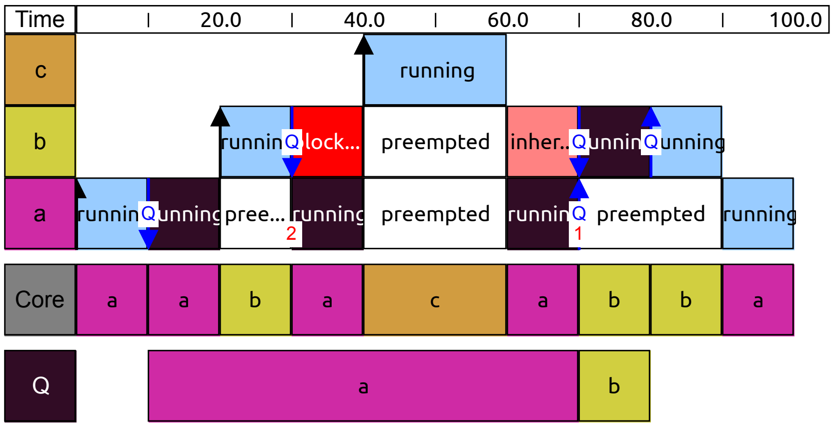
To provide a maximum compatibility, the user has the option to select the blocking definition for schedulers based on the fixed-priority preemptive policy in the preference dialogue.

Simple Priority Inheritance With Direct Blocking

As stated in section 8.4 on page 73, the simple priority inheritance protocol is not transitive. If a task is blocked by a lower priority task, the lower-priority task inherits the priority of that



(a) Blocking according to the general definition.



(b) Direct blocking.

Figure 8.3: A simple set of three tasks and one resource scheduled with DMS and OCPP for the comparison of the different blocking definitions.

task. If the lower-priority task is blocked itself, it will *not* promote its blocking task. Therefore, compared to the transitive priority inheritance protocol, the implementation is less complex. As explained above, the definition of direct blocking implies that if a medium-priority task b is blocked by a low-priority task a while a high-priority task d preempts a, not only the state of task a, but also the state of task b is set to *ready*. If task d continues to run and eventually gets blocked by task b, according to the simple priority inheritance protocol, only the priority of task b shall be promoted. Since task b is in state *ready* and now has the same active priority as d, the dispatch routine will select and execute task b, which tries to access a resource still blocked by a. Even without a recursive priority inheritance routine, task a now inherits the priority of b and d, respectively.

This shows how the simple priority inheritance protocol unintentionally got transitive due to a different blocking definition. It also implies that the resulting schedule of the simple priority inheritance protocol is not independent of the blocking definition. The discussed effects are visualized in figure 8.4 on page 88.

However, it can be shown that the simple priority inheritance protocol under the direct blocking definition does not equal the transitive priority inheritance protocol in general. In case task *d* of the previous example starts directly with the resource request command, it is immediately blocked. This time, task *b* is not in *ready* state when it inherits the priority of *d*. Therefore, task *b* is unavailable for the dispatch routine. So it does not try to lock the resource again, which is why the priority of task *a* is not promoted. The transitive priority inheritance protocol would promote the priority of task *a* independent of the state of task *b*.

In a nutshell, the simple priority inheritance protocol is defined on top of the general blocking definition and thus does not work correctly with the direct blocking definition. In order to verify that the simple priority inheritance protocol is only simulated under the general blocking definition, a notification will be presented if the user selected this protocol and the direct blocking definition.

8.8.2 Inheritance blocking

Apart from direct blocking, SHA et al. define another blocking type called *push-through* blocking for the priority inheritance protocols. This blocking type is also known as *inheritance* blocking [Liu00]. A medium priority task *b* is push-through or inheritance blocked if it is not allowed to execute because a low-priority task *a* locks a resource and inherits the priority of a high-priority job *c* waiting on this resource. This way, task *b* cannot block the high-priority task *c* indirectly. [SRL90]

Since task *b* has a lower active priority than task *a* after the latter inherited task *c*'s priority, it will not be selected by the dispatch routine even if it stays in the *ready* state. For that reason, changing the state of the inheritance blocked task from *ready* to *blocked* does not affect the resulting sequence of scheduled tasks. Similar to direct blocking, the user has the option to choose whether the state shall be changed to *blocked* if a task is inheritance blocked or not.

8.8.3 Ceiling blocking

For the OCPP, a third blocking type called *ceiling* blocking is introduced by SHA et al.. A high-priority task *c* is ceiling blocked by a low-priority task *a* if task *a* currently locks a resource with a priority ceiling higher than the priority of task *c*, and task *c* requests another resource [SRL90]. Ceiling blocking is also called *avoidance* blocking, since it is necessary to avoid deadlocks [Liu00].

Ceiling blocking is similar to inheritance blocking as a task is not allowed to execute because its priority is not high enough. Again, if the state of a ceiling blocked task is not changed to *blocked* but stays *ready*, the resulting schedule is the same. For that reason, the user has the option to choose whether the state shall be changed to *blocked* or not if a task is ceiling blocked.

As discussed in section 8.7, a task can only be blocked once under the ICPP. According to rule 3 of the ICPP definition given by Liu, "whenever a job requests a resource, it is allocated the resource" [Liu00, p. 301]. Therefore, it is impossible for a task to be directly blocked under the ICPP. Since a task locking a resource immediately inherits the resource's priority ceiling, the only possible blocking type is inheritance blocking.

Note that, although Burns and Wellings do not mention any of the additional blocking states explicitly, the examples 13.7, 13.8 and 13.9 for the discussed resource access protocols given in [BW01] show inheritance and ceiling blocked tasks in state *blocked*.

8.9 Earliest Deadline First

Apart from the fixed-priority scheduling policies presented in the previous sections, there exist *dynamic-priority* protocols like the Earliest Deadline First (EDF) scheduling policy. Under EDF regime, the order of the execution of the runnable tasks is determined by their absolute deadlines. In contrast to the relative deadline, which is saved as a parameter of a task in the simulation model, the absolute deadline is computed at runtime when an instance of a task is created. The task with the closest absolute deadline is selected and executed. According to the EDF scheduling policy, the priority of a periodic task might change between two instances. However, the priority of an instance of a task is fixed from its creation until its termination, since the absolute deadline of this instance is constant. The EDF scheduling policy is optimal on preemptive uniprocessors, because it successfully schedules any set of periodic, independent tasks with a processor utilization U up to 100%. More information about EDF is given in [LL73, BW01, Liu00].

The implementations of the EDF policy and the fixed-priority preemptive scheduler are similar. The priority queues presented in section 8.2.3 on page 73 are used with the shortest absolute deadline instead of the task's priority. The highest-priority task, i. e. the task with the shortest absolute deadline of the set of tasks in *ready* state, is compared to the currently running task. If the absolute deadline of the highest-priority task is closer, the currently running task is preempted.

In the definition of the EDF scheduling policy in [LL73, p. 55], the case that the currently running task and the task with the shortest absolute deadline among the set of ready tasks have exactly the same deadline is not mentioned. According to [SRS98, p. 17], if two tasks have the same deadline, the task to be executed can be chosen arbitrarily. Figure 6.4 of [Liu00, p. 118] contains such a case and shows that the currently running task is preempted. For the sake of compatibility, the EDF implementation also preempts the running task in case of equal deadlines.

The EDF scheduler implementation contains a default resource access protocol. If a task requests a resource but fails to lock it, the task is set to *blocked* state. Its state is set to *ready* as soon as the resource is released again. After each resource interaction, the dispatch routine is called, since the state of a task might have been changed.

8.10 Stack Resource Protocol

The EDF scheduling policy suffers an analogy to priority inheritance considering shared resources and blocking. If a resource is currently locked by a task, another task requesting the same resource is not able to execute, even if it has a closer deadline. This effect is called *deadline inversion* [BW01].

In 1991, Baker presented the *Stack Resource Protocol (SRP)*, a resource access protocol for the dynamic-priority EDF scheduler [Bak90]. The SRP influenced the development of the ICPP [BW01], although the SRP uses so-called *preemption levels* instead of priority values to support dynamic scheduling protocols. Preemption levels capture the possibility that one task preempts another task. A valid preemption level assignment for periodic tasks is based on the relative deadline: the closer the deadline, the higher the preemption level. Similar to static priorities, the preemption level assignment is fixed and needs to be derived only once before execution. Each resource is assigned a preemption ceiling value, i. e. the highest preemption level of all tasks requesting the resource. Additionally, the system preemption ceiling always equals the highest preemption ceiling of all resources currently locked. If all resources are unlocked, the system preemption level is defined to be lower than the lowest preemption level of all tasks. Further details about the SRP are given in [Bak90, Liu00].

According to Liu, there exist a basic and a stack-based version of the SRP, similar to the two versions of the ICPP described in section 8.7.3 on page 78. The SRP is implemented according to the rules of the stack-based version and as a subclass of the EDF scheduler implementation. The priority field of the task and resource specification in the simulation model is used to save the preemption level.

The `dispatch()` function is overridden, since under the SRP, a task is blocked from starting execution until its preemption level is higher than the current system ceiling. Once a task started running, it is added to the shared runtime stack. As soon as the task terminates, it is deleted from this stack again. The task added last to the shared runtime stack is compared to the task with the closest absolute deadline. If the task in *ready* state has a closer absolute deadline than the task on the shared runtime stack, it is only allowed to start execution in case its preemption level is higher than the current system preemption ceiling. Otherwise, the currently running task continues its execution.

The `requestResource()` method is overridden, since the SRP guarantees that "whenever a task requests a resource, it is allocated the resource" [Liu00, p. 311]. The `releaseResource()` method is also overridden, as each resource interaction requires the system preemption ceiling to be updated. In contrast to the ICPP implementation, the system preemption ceiling is not calculated each time but added to and removed from a stack according to Baker's implementation considerations [Bak90].

The SRP definition contains a priority inheritance rule. Fortunately, "the preemption test has the effect of imposing priority inheritance; that is, an executing job that holds a resource modifies the system ceiling and resists preemption as though it inherits the priority of any jobs that might need that resource. Note that this effect is accomplished without modifying the priority of the job." [But11]

8.11 Adding a Scheduler

New scheduling policies and resource access protocols can be added to an existing installation of the software as additional plug-ins. The class containing the new scheduler has to implement the `IScheduler` interface and needs to modify the simulation model. Therefore, the new plug-in will depend on the `de.unistuttgart.iste.ps.savors.runtime` and the `de.unistuttgart.iste.ps.savors.model` plug-ins, which have to be present in Eclipse. Additionally, the class has to be added to the `de.unistuttgart.iste.ps.savors.scheduler` extension point, as explained in section 7.1.4 on page 63. Eclipse offers comfortable methods to create and export a new plug-in as well as to add an entry to an extension point.

With the Equinox Provisioning Platform (P2), Eclipse includes a full-fledged update mechanism. However, P2 requires the maintenance of a repository of plug-ins and increases the size of the software [Dau08]. Therefore, the decision was made against this framework and in favor of updating the software manually. Adding the newly developed plug-in into an existing installation of the application requires two steps.

1. The JAR file containing the plug-in has to be copied into the “plugins” subdirectory of the installation location.
2. The configuration file “bundles.info” of the `SimpleConfigurator`, which is located in the “configuration/org.eclipse.equinox.simpleconfigurator” subdirectory of the installation location has to be edited. A new line needs to be added, specifying the ID and version of the new plug-in, its location as well as its start level and whether it should start automatically. The start level should be the same as for the other plug-ins, i. e. 4, and autostart should be set to false.

8.12 Multi-Core Scheduling Policies

All scheduling policies introduced in the previous section are defined only for a single-core environment. However, as indicated in section 1.2 on page 14, the software should support multi-core environments and respective scheduling policies.

The simulation supports only *homogeneous* multi-core systems, in which the rate of execution of all tasks is the same on all processing units [DB11]. This is due to the duration of an execution command being specified by its WCET as mentioned in section 6.1.1 on page 48. If the cores have different rates of execution, the duration will also depend on the core the task is running on.

Furthermore, intra-task parallelism is not supported, which means that at any given time, a task can execute on one core at most [DB11, p. 4].

Even with these restrictions, there exist various approaches to solve the two main problems [DB11]:

1. Find a core to execute every task [DB11].

2. Find a priority for each task to determine the order of execution [DB11].

The solution to the first problem, the allocation problem, can be subdivided into three possible classes [DB11]:

1. Tasks are allocated to a core and no migration is permitted. [DB11].
2. Each instance of a task may execute on different cores, however, a single instance is fix allocated to a core [DB11].
3. A single instance of a task can migrate on different cores, e. g. after a preemption [DB11].

For the second problem, the priority assignment can be static or dynamic similar to single-core scheduling policies.

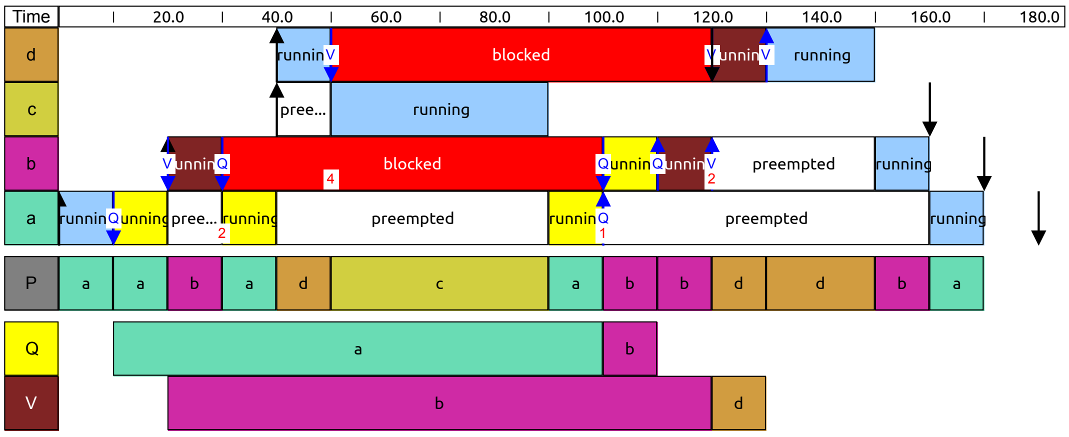
More information about how these problems can be solved is given in [Liu00, DB11].

8.12.1 Partitioned Multi-Core Scheduler

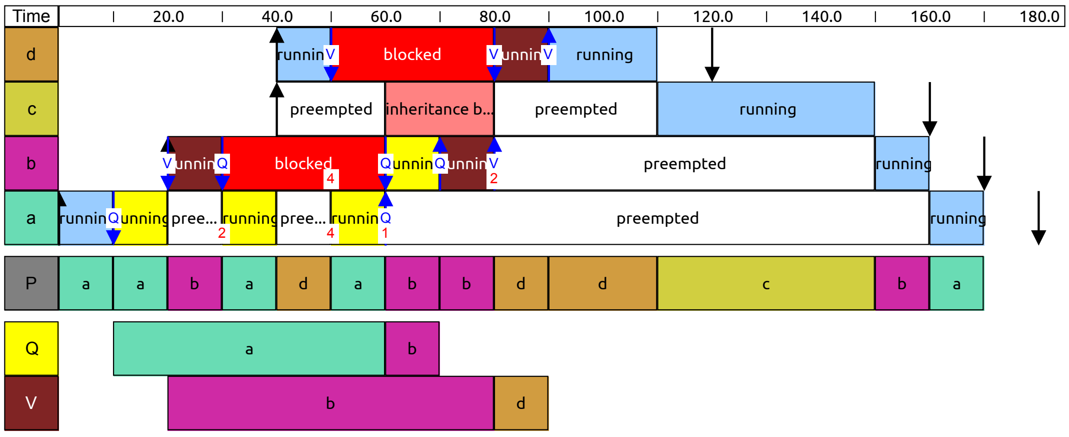
Scheduling policies which allocate tasks to a core without allowing migration are referred to as *partitioned* [DB11]. The problem of finding an optimal task assignment is NP-hard [Liu00, p. 339]. In most case, heuristics like first fit (FF), next fit (NF), and best fit (BF) are used, which find suboptimal solutions in reasonable amount of time [DB11, Liu00]. Once the allocation problem is solved, the set of tasks per core can be scheduled under the well-known scheduling policies introduced in the first sections of this chapter [DB11, p. 14]

A partitioned fixed-priority preemptive multi-core scheduler was created as a plug-in to be added to an existing installation of the software using the mechanism described in section 8.11 on the preceding page. The assignment of tasks to cores has to be solved by the user in beforehand and is included in the simulation model. Each set of tasks per core is scheduled under the DMS with the default resource access protocol.

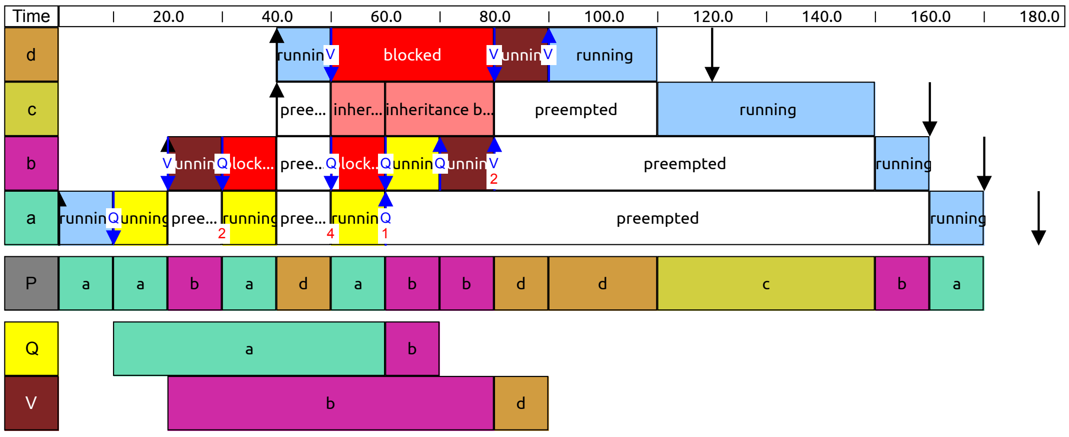
The scheduler makes use of the DMS implementation, which is part of the `de.unistuttgart.iste.ps.savors.scheduler` plug-in. For each core an instance of this scheduler is created. Depending on the task's name, the core to execute on as well as the respective DMS implementation are selected. All state change and resource requests are forwarded to the respective DMS implementation. The dispatch call is forwarded to all DMS implementations, as the event responsible for the dispatch call is unknown.



(a) The simple priority inheritance protocol under general blocking regime is not transitive.



(b) The transitive priority inheritance protocol under general blocking regime.



(c) The simple priority inheritance protocol under direct blocking regime is transitive.

Figure 8.4: An example schedule presenting the effects of different blocking definitions on the simple priority inheritance protocol.

9 Validation and Tests

Validation and tests are important aspects in software engineering. The former is concerned with showing that the program works as expected and the latter is necessary to discover defects and undesired system behavior [Som10]. Therefore, both are mandatory for a system to fulfill all requirements and to be reliable and stable. This chapter presents how the entire software is validated and tested.

Software is usually tested at different layers, starting from unit tests and ending with comprehensive system tests. An overview about software testing is given in [FLS07, Som10].

9.1 Unit Testing

Unit testing is the process of testing each program component and its functions separately. Functions can be tested by calling them with different sets of parameters, especially with parameters representing corner cases, and comparing the returned result with the expected value. An object is tested by testing all its methods, by modifying and checking all its attributes, and by putting the object in all possible states by simulating events that cause state changes [Som10, p. 211]. Unit tests are typically performed as so-called *white-box* tests. This type of test methods have access to the source code and the internal structures of the program. Detailed information about unit testing and white-box test methods is given in [Som10].

With JUnit, there exists a comfortable environment for conducting unit tests in the Java programming language [Som10]. Each class is tested in a separate environment, while the framework logs the test results [Som10].

For Eclipse RCP applications, which are based on the Eclipse platform, ordinary JUnit tests might not be runnable since the component to be tested may require services of the Eclipse platform. For this reason, the Eclipse PDE provides a special version of JUnit, which starts a complete instance of the Eclipse RCP application in a separate Java Virtual Machine (JVM) and executes all test cases inside this environment. For more information about this PDE version of JUnit, please refer to [SJB08].

As explained in section 6.1 on page 46, the EMF generates the source code of the system model. Furthermore, it is able to generate a test plug-in for the generated classes. However, this test plug-in contains only a stub test case for each model element with no actual tests inside. Since the system model elements merely contain attributes and no methods, the tests themselves are rather simple.

A test suite based on the PDE version of JUnit was developed. It contains tests for the most important classes of the simulation kernel and for each event analyzer. The `TimeAxis` is tested by checking whether `Jobs` added at specific time instances are later returned properly and their deletion is performed correctly. Various methods of the `Simulator` are tested, e. g. it should not be possible to start the simulation without a simulation model to be present. It is checked whether the `TaskMonitor` prevents nonexistent state transitions. Additionally, the `TaskMonitor` has to create various `Events` and `Jobs` whenever a task successfully changed its state, which is also tested. Furthermore, the correct support of multi-unit resources by the `ResourceMonitor` is tested.

Each subclass of the `AbstractEventAnalyzer` is tested by adding respective events to the simulation model and checking for the event analyzer to trigger.

9.2 System Testing

In contrast to unit tests, system tests are usually performed as so-called *black-box* tests. Such tests do not consider the internal structure of the program. The main aim of system tests is to validate that all requirements are met. For this reason, the completely integrated system with all subcomponents is tested. More information about this topic is contained in [FLS07, Som10].

By making use of software engineering methods like prototyping and incremental delivery as described in section 3.1 on page 21, discrepancies between the requirements and the software were checked regularly in meetings with the customer, i. e. the supervisor.

In order to check the correctness of the software, especially the simulation, various simulation models were extracted and translated from examples given in [BW01], [Liu00], [SRL90], and [Bak90]. The simulation results of these models were compared to the sample schedules given in these references. Defects and shortcomings in the implementation of the scheduling policies and resource access protocols could be detected and fixed this way. Some important insights gained by these tests are covered in the sections 8.7.2 and 8.8 of the previous chapter.

In case the resulting schedule was determined to be correct, the simulation model as well as the simulation result serve as reference for new versions of the program. A new version should produce the same results as the previous versions if no functional parts of the program have been changed. This test is referred to as *regression test* [FLS07]. To decrease the costs of testing, the regression tests are automated with the help of the JUnit framework.

Furthermore, test cases for the activation and deactivation of GUI elements, the differentiation of the different blocking types, the manual and automatic pausing of the simulation, the effects of different visualization preferences, and the recognition of faulty simulation models were defined. Faulty simulation models either contain malformed XMI syntax, specify generally unsupported situations like consecutive execution commands, or are incompatible to the selected scheduling policy. An example for the latter case is an incorrect priority assignment for the RMS.

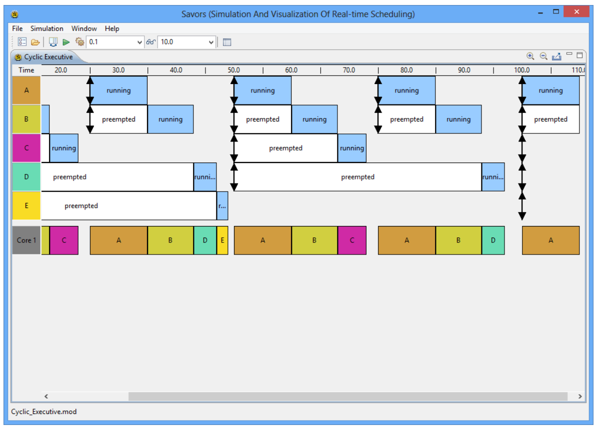
All mentioned test cases are described in a tabular manner, specifying each step to be performed and the expected behavior. The tests are executed by following these steps and comparing the result to the expectation, as described in [FLS07].

As mentioned in section 1.2 on page 14, an advantage of using Java is its support of multiple platforms without recompilation. Using Eclipse and SWT slightly restricts the number of platforms, as explained in section 5.1.4 on page 37. The software was tested on different platforms, as shown in figure 9.1 on the following page.

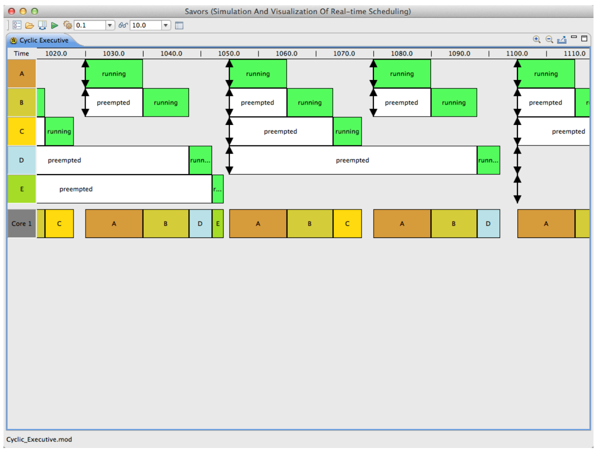
9.3 Code Conventions Check

Making use of coding style conventions enhances the readability and improves software maintenance [KND+99], which is an important objective as mentioned in section 1.2 on page 14. For this reason, the source code is written under the Code Conventions for the Java Programming Language given in [KND+99].

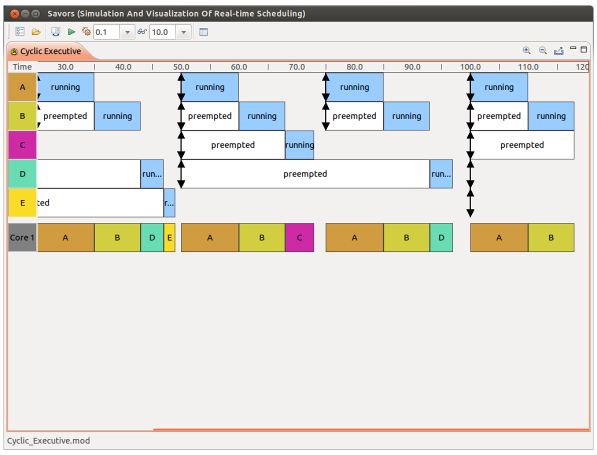
In order to automate the checking of the coding style conventions and to enforce their correct application, the code convention checking software tool *checkstyle* is employed. It supports the Code Conventions for the Java Programming Language and can be added to the Eclipse IDE or be invoked by an Apache Ant task [Bur12]. The automated software building process described in section 7.1.5 on page 63 makes use of this software tool to generate a documentation of coding style violations in the source code.


Savors (Simulation And Visualization Of Real-time Scheduling)
File Simulation Window Help
Cyclic Executive
Time
A
B
C
D
E
Core 1
running
preempted
Cyclic_Executive.mod

(a) Microsoft Windows 8 Professional N 32-bit.



(b) Apple OS X Lion 10.7.5 (11G63) 64-bit.



(c) Ubuntu 12.10 with Linux 3.5.0-17 64-bit.

Figure 9.1: Testing the software on different platforms.

10 Conclusion and further research

10.1 Conclusion

In the presented work, task scheduling simulation and visualization software was designed, implemented and tested. The software focuses on but is not limited to scheduling policies primarily used in real-time and embedded systems.

First of all, the functionalities and current state of existing projects were researched and summarized. It could be shown that none of these projects is available under an appropriate license, focuses on real-time and embedded systems, and at the same time is able to generate a graphical representation of the current simulation result while the simulation is running. The synchronous visualization of the simulation result relieves the user of the need to specify the time and duration of the part of interest. The software offers the possibility to pause the simulation at arbitrary points of interest in order to inspect intermediate scheduling situations.

Different visualization frameworks for Java were compared with the help of prototypes. Based on this comparison, Eclipse and the GEF were selected to generate the graphical representation of the simulation result. Apart from the visualization, loading the file-based simulation model can be facilitated with the help of another framework. A comparison amongst different approaches led to the EMF, which is not only able to serialize and deserialize objects, but additionally capable of generating the source code of these objects from a UML model.

The software architecture design is based on the previously selected frameworks. Special effort was put into the design of the simulation meta-model, which needs to provide a high degree of flexibility to allow modeling a variety of system configurations. The simulation supports multiple processing units, periodic and sporadic tasks with soft, hard, and no deadlines as well as shared resources with multiple units. Each of these elements can contain so-called events, which are created in the course of the simulation. With the help of the EMF, the simulation model is loaded from an XMI formatted file. This format is based on the well-known XML, and is therefore likely to be easily edited by humans. Furthermore, it extends the XML to allow links between elements. Such references are useful to handle cases such as a task requesting a specific resource.

The simulation kernel of this project is designed in a dynamic, discrete event-driven manner. The simulation time is increased stepwise whenever the simulation model and its elements are modified. In order to create a more realistic impression of the running scheduling process, the simulator waits in real-world time for the elapsed simulation time. Together with the pausing function, this feature is helpful for understanding the scheduling algorithm.

The simulation kernel executes the simulation in a separate thread. A special type of exception is introduced, which allows the scheduler implementation and the simulation kernel to provide additional information in case a problem occurs. All exceptions that occur while the simulation is running are logged and depending on their severity, the user is additionally notified directly.

The scheduler implementation has to trigger all necessary actions to drive the simulation. The software supports the scheduler by providing functions which monitor the state of tasks and resources and add respective events to the simulation model. In order to support various scheduling policies and resource access protocols, the simulator does not restrict a direct modification of the simulation model.

In addition to the initially required scheduling policies, a cyclic executive scheduler, a non-preemptive fixed-priority scheduler, an EDF scheduler with the SRP, and a partitioned multi-core DMS were implemented. The last-mentioned multi-core scheduler demonstrates that the software potentially supports scheduling protocols for multi-core environments.

The simulation model does not only contain the configuration of the system to be simulated. All results generated during the simulation are added to the simulation model in the form of events. With the help of the GEF, a graphical representation for each element of the simulation model is generated. The resulting visualization of the simulation result is updated whenever the simulation time increases and thus always shows the current state. All visual elements provide insight into their properties with the help of an additional window.

The user is allowed to pause the simulation at any time in order to inspect a specific scheduling decision in closer detail. Furthermore, the simulation model is automatically scanned for specific events, on the occurrence of which the simulation pauses itself and optionally notifies the user.

Eclipse provides an extension mechanism which is used by the software to allow the addition of further scheduler implementations. The same mechanism is also used to offer the addition of new event analyzing functions, which pause the simulation in case a specific event has occurred. The extension mechanism allows to upgrade an existing installation of the software with additional functionality.

The result of the simulation can be saved as an XMI file, as a bitmap, and as a vector image. If the XMI file is reopened with the software, all previously calculated information is visualized immediately.

The software is tested with unit and system tests. The correct function of the scheduling policy implementations is checked by comparing text book examples with the simulation result. In order to enhance the readability and improve the maintenance of the source code, it is written under coding style conventions. The compliance with these conventions is checked as part of the automatic software building process. A documentation of the source code is generated with the JavaDoc software as part of the automated software building process.

10.2 Further Research

The current version of the simulation and visualization software already allows simulating a multitude of scheduling policies and resource access protocols for single-core systems. Apart from adding further scheduling policies like *round robin* and *least slack time first*, there still exist different interesting opportunities and possibilities to extend and enhance the software.

It was shown that the current version of the software supports basic scheduling protocols for multi-core environments. However, more complex multi-core protocols require clusters of cores or tasks. An extension of the simulation meta-model could include such clusters. With the help of the EMF, the source code of these new elements could be automatically generated. Unfortunately, in order for the new elements to have a graphical representation, a visualization based on the GEF would have to be implemented manually. The existing scheduling protocols would have to be adapted, too.

The current version of the simulator does not support intra-task parallelism, since it is impossible in a single-core environment. However, as soon as there exist several processing units, multiple instances of the same task could be executed in parallel on different cores. One possible solution to this problem is to change the simulation meta-model. Currently, the meta-model is based on tasks according to the definition given in [BW07]. Switching to the definition of tasks and jobs according to [Liu00] would allow the same task to have two jobs which could both execute on a different core in parallel. Furthermore, it would increase the compatibility to a variety of scheduling policy definitions which are based on jobs. Unfortunately, all existing scheduling policies would have to be adapted and as jobs are defined to have interdependencies, the implementation would be rather complex.

The current simulation meta-model does not allow tasks to self-suspend. This could be changed by defining a self-suspend command. Unfortunately, the implemented scheduling policies would have to be adapted as well in order to support the new command.

By adding an initialization and completion phase to the task state model, system configurations with start-up and finalization phases could be simulated. Similar to the previous extension possibilities, the changes in the simulation meta-model would be rather simple, however the changes in the scheduler implementations would be complex.

In order to relieve the user from editing the simulation model externally via XMI formatted files, a GUI could be added, offering functions to add and remove elements of the simulation model and to change their properties.

A theoretical response time analysis could be performed by the software. Thus, the user could be informed about the general schedulability of the simulation model prior to the simulation.

Bibliography

[ABRW94] N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings. STRESS: A Simulator for Hard Real-Time Systems. *Software: Practice and Experience*, 24(6):543–564, 1994.

[Bak90] T. P. Baker. A Stack-Based Resource Allocation Policy for Realtime Processes. In *Proceedings of the 11th Real-Time Systems Symposium*, pp. 191–200. 1990.

[BBB+10] S. Bakhkhat, F. Böde, M. Brucke, K. Degen, C. Ebert, I. Einsiedler, C. Gouma, F. Grunert, R. Möllers, E. Trenew, J. Niehaus, K. Renger, S. Richter, S. Rupp, J. Salecker, R. Stein, O. Winzenried, S. Ziegler. Eingebettete Systeme – Ein strategisches Wachstumsfeld für Deutschland. Technical report, BITKOM, 2010. URL `http://www.bitkom.org/de/themen/54926_62539.aspx`.

[BG07] L. G. Birta, A. Gilbert. *Modelling and Simulation: Exploring Dynamic System Behaviour.* Springer, London, UK, 2007.

[BGM04] B. B. Bederson, J. Grosjean, J. Meyer. Toolkit Design for Interactive Structured Graphics. *IEEE Transactions on Software Engineering*, 30(8):535–546, 2004.

[BL11] C. Bartolini, G. Lipari. RTSIM, 2011. URL `http://rtsim.sssup.it`.

[Bur12] O. Burn. Checkstyle 5.6, 2012. URL `http://checkstyle.sourceforge.net`.

[But11] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Real-Time Systems Series. Springer New York, NY, USA, 3rd edition, 2011.

[BW01] A. Burns, A. J. Wellings. *Real-Time Systems and Programming Languages: Ada 95, real-time Java and real-time POSIX.* Addison-Wesley, Harlow, UK, 3rd edition, 2001.

[BW07] A. Burns, A. Wellings. *Concurrent and Real-Time Programming in Ada.* Cambridge University Press, New York, NY, USA, 2007.

[CBLL98] A. Casile, G. Buttazzo, G. Lamastra, G. Lipari. Simulation and Tracing of Hybrid Task Sets on Distributed Systems. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, RTCSA'98, pp. 249–256. IEEE Computer Society, Washington, DC, USA, 1998.

[CFJ+12] K. Coulomb, M. Faverge, J. Jazeix, O. Lagrasse, J. Marcoueille, P. Noisette, A. Redondy, C. Vuchener. ViTE (Visual Trace Exlporer), 2012. URL `http://vite.gforge.inria.fr`.

[COF12] A. Carminati, R. S. de Oliveira, L. Friedrich. Implementation and Evaluation of the Synchronization Protocol Immediate Priority Ceiling in PREEMPT-RT Linux. *Journal of Software*, 7(3), 2012.

[Dau08] B. Daum. *Rich-Client-Entwicklung mit Eclipse 3.3.* dpunkt-Verlag, Heidelberg, Germany, 2008.

[DB11] R. I. Davis, A. Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4):35:1–35:44, 2011.

[DBC07] A. Diaz, R. Batista, O. Castro. Realtss: a Real-Time Scheduling Simulator. In *4th International Conference on Electrical and Electronics Engineering*, ICEEE'07, pp. 165–168. 2007.

[DDG$^+$11] E. Dahlström, P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers, J. Watt. Scalable Vector Graphics (SVG) 1.1. Technical report, W3C, 2011. URL `http://www.w3.org/TR/SVG/`.

[DP02] D. Decotigny, I. Puaut. ARTISST: an Extensible and Modular Simulation Tool for Real-Time Systems. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '02, pp. 365–372. Washington, DC, USA, 2002.

[Dut10] M. Dutoo. Bug 327563: Export Workflow to SVG Image, 2010. URL `https://bugs.eclipse.org/bugs/show_bug.cgi?id=327563`.

[Fek04] J.-D. Fekete. The InfoVis Toolkit. In *Proceedings of the 10th IEEE Symposium on Information Visualization*, InfoVis'04, pp. 167–174. IEEE Press, Piscataway, NJ, USA, 2004.

[FLS07] K. Frühauf, J. Ludewig, H. Sandmayr. *Software-Prüfung: eine Anleitung zum Test und zur Inspektion.* vdf-Hochschulverlag, Zurich, Switzerland, 6th edition, 2007.

[GBA$^+$97] F. S. Giorgio, G. C. Buttazzo, P. Ancilotti, S. Superiore, S. Anna. GHOST: A Tool for Simulation and Analysis of Real-Time Scheduling Algorithms. In *Proceedings of the IEEE Real-Time Educational Workshop*, RTEW'97, pp. 42–49. Montreal, Canada, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Boston, MA, USA, 1995.

[GN00] E. R. Gansner, S. C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.

[Gon12] M. González. MAST, 2012. URL `http://mast.unican.es`.

[HCL05] J. Heer, S. K. Card, J. A. Landay. Prefuse: a Toolkit for Interactive Information Visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI'05, pp. 421–430. ACM, New York, NY, USA, 2005.

[HGGM01] M. G. Harbour, J. G. García, J. P. Gutiérrez, J. D. Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, ECRTS'01, pp. 125–134. IEEE Computer Society, Los Alamitos, CA, USA, 2001.

[HHS10] S. Hunold, R. Hoffmann, F. Suter. Jedule: A Tool for Visualizing Schedules of Parallel Applications. In *Proceedings of the 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pp. 169–178. IEEE Computer Society, Washington, DC, USA, 2010.

[JG04] R. Jejurikar, R. Gupta. Dynamic Voltage Scaling for Systemwide Energy Minimization in Real-Time Embedded Systems. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, ISLPED '04, pp. 78–81. ACM, New York, NY, USA, 2004.

[KAK00] T. Kramp, M. Adrian, R. Koster. An Open Framework for Real-Time Scheduling Simulation. In *Proceedings of the 15th IPDPS Workshops on Parallel and Distributed Processing*, IPDPS'00, pp. 766–772. Springer-Verlag, London, UK, 2000.

[Ker00] P. B. J. Chassin de Kergommeaux, B. Stein. Pajé, an Interactive Visualization Tool for Tuning Multi-Threaded Parallel Applications. *Parallel Computing*, 26(10):1253–1274, 2000.

[Kli09] P. Kliem. Jaret Timebars Component (1.30), 2009. URL `http://jaret.de/timebars/documentation/timebars.html`.

[Klu12] D. Klusáček. Alea - GridSim based Grid Scheduling Simulator, 2012. URL `http://www.fi.muni.cz/~xklusac/alea/`.

[KND+99] P. King, P. Naughton, M. DeMoney, J. Kanerva, K. Walrath, S. Hommel. Code Conventions for the Java Programming Language. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 1999. URL `http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html`.

[KR10] D. Klusáček, H. Rudová. Alea 2 – Job Scheduling Simulator. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools'10. ICST, 2010.

[Kra09] J. Kraft. RTSSim - A Simulation Framework for Complex Embedded Systems. Technical Report, Mälardalen University, Västerås, Sweden, 2009.

[Lef09] L. Lefever. 3D Visualization of Scheduling, Threads and Synchronization Primitives, 2009. URL `http://code.google.com/p/gltracevis`.

[Liu00] J. W. S. Liu. *Real-Time Systems.* Prentice Hall, Upper Saddle River, NJ, USA, 2000.

[LL73] C. L. Liu, J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[LL01] W. Löwe, A. Liebrich. VizzScheduler - A Framework for the Visualization of Scheduling Algorithms. In *Proceedings of the 7th International Euro-Par Conference on Parallel Processing*, Euro-Par'01, pp. 62–66. Springer-Verlag, London, UK, 2001.

[Man06] A. Manacero Jr. Real Time Systems, 2006. URL `http://www.dcce.ibilce.unesp.br/spd/rtsim/english`.

[Mar06] P. Marwedel. *Embedded System Design.* Springer, Dordrecht, The Netherlands, 2006.

[MDG$^+$04] B. Moore, D. Dean, A. Gerber, G. Wagenknecht, P. Vanderheyden. *Eclipse Development Using the Graphical Editing Framework And the Eclipse Modeling Framework.* IBM International Technical Support Organization (ITSO), Riverton, NJ, USA, 2004.

[ME07] B. McLaughlin, J. Edelson. *Java & XML: Solutions to Real-World Problems.* O'Reilly, Beijing, China, 3rd edition, 2007.

[MLA10] J. McAffer, J.-M. Lemieux, C. Aniszczyk. *Eclipse Rich Client Platform.* Addison-Wesley, Upper Saddle River, NJ, USA, 2nd edition, 2010.

[MMN01] A. Manacero Jr., M. B. Miola, V. A. Nabuco. Teaching Real-Time with a Scheduler Simulator. In *31st ASEE/IEEE Frontiers in Education Conference*, volume 2, pp. 15–19. 2001.

[MS12] Y. Matsubara, Y. Sano. Schesim: A flexible Scheduling Simulator for Real-Time Applications, 2012. URL `http://en.schesim.org`.

[MSH11] J. W. McCormick, F. Singhoff, J. Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada.* Cambridge University Press, Cambridge, UK, 2011.

[MSHT12] Y. Matsubara, Y. Sano, S. Honda, H. Takada. An Open-Source Flexible Scheduling Simulator for Real-Time Applications. In *Proceedings of the 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC'12, pp. 16–22. IEEE Computer Society, Washington, DC, USA, 2012.

[MVA10] J. McAffer, P. VanderLei, S. Archer. *OSGi and Equinox: Creating Highly Modular Java Systems.* Eclipse Series. Addison-Wesley Professional, Bosten, MA, USA, 2010.

[Nis97] N. Nissanke. *Realtime Systems.* Prentice-Hall, Upper Saddle River, NJ, USA, 1997.

[Obj11] Object Management Group, Needham, MA, USA. *OMG MOF 2 XMI Mapping Specification Version 2.4.1*, 2011. URL `http://www.omg.org/spec/XMI/2.4.1/PDF`.

[OFWB03] J. O'Madadhain, D. Fisher, S. White, Y.-B. Boey. The JUNG (Java Universal Network/Graph) Framework. Technical report, School of Information and Computer Science, University of California, Irvine, CA, USA, 2003. URL `http://www.datalab.uci.edu/papers/JUNG_tech_report.html`.

[RF07] C. Reas, B. Fry. *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press, Cambridge, MA, USA, 2007.

[RWC11] D. Rubel, J. Wren, E. Clayberg. *The Eclipse Graphical Editing Framework (GEF)*. Addison Wesley, Upper Saddle River, NJ, USA, 2011.

[SB12] F. Sauer, G. Boissier. Eclipse Metrics Plugin, 2012. URL `http://metrics2.sourceforge.net`.

[SBPM11] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Upper Saddle River, NJ, USA, 2nd edition, 2011.

[Sha98] R. E. Shannon. Introduction to the Art and Science of Simulation. In *Proceedings of the 30th Simulation Conference*, WSC'98, pp. 7–14. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[Sin12] F. Singhoff. The Cheddar project: a free real time scheduling analyzer, 2012. URL `http://beru.univ-brest.fr/~singhoff/cheddar`.

[SJB08] H. Sippel, M. Jastram, J. Bendisposto. *Die Eclipse Rich Client Platform: Entwicklung von erweiterbaren Anwendungen mit RCP*. Software und Support Verlag, Munich, Germany, 2008.

[SLNM04] F. Singhoff, J. Legrand, L. Nana, L. Marcé. Cheddar: a Flexible Real Time Scheduling Framework. In *Proceedings of the Annual ACM SIGAda International Conference on Ada: The engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies*, SIGAda'04, pp. 1–8. ACM, New York, NY, USA, 2004.

[SML96] W. J. Schroeder, K. M. Martin, W. E. Lorensen. The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization. In *Proceedings of the 7th Conference on Visualization*, VIS'96, pp. 93–102. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[SN09] M. Scholz, S. Niedermeier. *Java und XML: Grundlagen, Einsatz, Referenz*. Galileo Press, Bonn, Germany, 2nd edition, 2009.

[Som10] I. Sommerville. *Software Engineering*. Pearson, Boston, MA, USA, 9th edition, 2010.

[SPD08] F. Singhoff, A. Plantec, P. Dissaux. Can We Increase the Usability of Real Time Scheduling Theory? The Cheddar Project. In *Proceedings of the 13th Ada-Europe international conference on Reliable Software Technologies*, Ada-Europe'08, pp. 240–253. Springer, Berlin, Germany, 2008.

[SRL90] L. Sha, R. Rajkumar, J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[SRS98] J. A. Stankovic, K. Ramamritham, M. Spuri. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[Tan09] A. S. Tanenbaum. *Moderne Betriebssysteme.* Pearson Studium, Munich, Germany, 3rd edition, 2009.

[TDB+07] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Plödereder, P. Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1.* Springer, Secaucus, NJ, USA, 2007.

[UDT10] R. Urunuela, A. Déplanche, Y. Trinquet. STORM: a Simulation Tool for Real-Time Multiprocessor Scheduling Evaluation. In *IEEE Conference on Emerging Technologies and Factory Automation*, ETFA'10, pp. 1–8. Bilbao, Spain, 2010.

[Ull12] C. Ullenboom. *Java ist auch eine Insel, Das umfassende Handbuch.* Galileo Press, Bonn, Germany, 10th edition, 2012.

[Uru12] R. Urunuela. STORM Introduction, 2012. URL `http://storm.rts-software.org`.

[Wea04] C. Weaver. Building Highly-Coordinated Visualizations in Improvise. In *Proceedings of the IEEE Symposium on Information Visualization*, pp. 159–166. IEEE Computer Society, Austin, TX, USA, 2004.

All links were last followed on January 11, 2013.

Glossary

AADL
: The AADL was designed by the SAE and provides a textual and graphical modeling notation for embedded real-time systems. It provides constructs for software as well as for hardware elements of such system. Additionally, it includes a standardized interchange format based on the XML.

Ant
: Apache Ant is a Java-based automated software building tool, which uses XML documents to save the configuration.

Draw2D
: Draw2D is a lightweight toolkit for displaying graphical components. It is built on top of SWT.

Gantt chart
: Gantt charts were developed by Henry Gantt. They display tasks and their chronological order in an horizontal bar graph.

GTK+
: GTK+ is an open-source graphical toolkit written in C. It serves as framework for the creation of GUI elements on multiple platforms.

GTKAda
: GtkAda is an graphical toolkit for the Ada programming language based on GTK+.

Java2D
: The Java2D API is a set of classes providing functions for the creation of advanced 2D graphics. It supports basic elements like polygons, textual labels, and raster images.

JavaDoc
: JavaDoc is a software tool which generates a HyperText Markup Language (HTML) documentation of Java source code based on comments in the code.

JUnit
: JUnit is a unit testing framework for Java.

LogP
LogP is a model for parallel computation. Each letter of the word LogP represents one parameter describing the machine. L is the latency of communication, o is the message overhead, g is the gap required between message operations and P is the number of processing units.

mnemonic
On most platforms, the mnemonic character of a UI element appears underlined. If the user presses a key sequence that matches the mnemonic, the UI element is activated.

.NET
Microsoft's .NET framework provides a class library and an application virtual machine. It enhances the interoperability of different programming languages.

OpenGL
OpenGL is a software framework for developing portable, interactive 2D and 3D graphics applications.

RGB
A color model in which red, green, and blue light are added together to define a color.

Wizard
A wizard consists of several GUI elements to guide the user on a specific task.

XML Schema
An XML schema definition is used to express a set of rules an XML has to obey in order to be considered valid. It is developed by the W3C.

Acronyms

AADL	Architecture Analysis & Design Language
API	Application Programming Interface
ARTISST	ARTISST is a Real-Time System Simulation Tool
AWT	Abstract Window Toolkit
BF	best fit
BMP	Windows Bitmap
BSD	Berkeley Software Distribution
CSV	comma-separated values
DES	discrete event simulation
DMS	Deadline Monotonic Scheduler
DSL	domain specific language
ECU	electronic control unit
EDF	Earliest Deadline First
EJB	Enterprise Java Bean
EMF	Eclipse Modeling Framework
FF	first fit
FIFO	First In First Out
GEF	Graphical Editing Framework
GHOST	General Hard real-time Oriented Simulator Tool
GIF	Graphics Interchange Format
GNU	GNU's Not Unix
GPL	GNU General Public License
GTK+	GIMP Toolkit
GUI	graphical user interace
HTML	HyperText Markup Language
ICPP	Immediate Ceiling Priority Protocol

IDE	integrated development environment
JAR	Java Archive
JAXB	Java Architecture for XML Binding
JDK	Java Development Kit
JFC	Java Foundation Classes
JFIF	JPEG File Interchange Format
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
JUNG	Java Universal Network/Graph Framework
JVM	Java Virtual Machine
LGPL	Lesser GPL
MIT	Massachusetts Institute of Technology
MVC	Model–View–Controller
NF	next fit
OCPP	Original Ceiling Priority Protocol
OGSi	Open Services Gateway initiative
OMG	Object Management Group
OS	operating system
P2	Equinox Provisioning Platform
PDE	Plug-in Development Environment
PNG	Portable Network Graphics
POJO	Plain Old Java Object
RCP	Rich Client Platform
RMS	Rate Monotonic Scheduler
SAE	Society of Automotive Engineers
SAVORS	Simulation And Visualization Of Real-time Scheduling
SRP	Stack Resource Protocol
STORM	Simulation Tool for Real-time Multiprocessor scheduling
SVG	Scalable Vector Graphics
SWT	Standard Widget Toolkit
TCL	Tool Command Language
TIFF	Tagged Image File Format

TLV	Trace Log Visualizer
UI	user interface
UML	Unified Modeling Language
ViTE	Visual Trace Explorer
VTK	Visualization Toolkit
W3C	World Wide Web Consortium
WCET	worst case execution time
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Definition
YAML	YAML Ain't Markup Language

Declaration

I declare that this thesis is the solely effort of the author. I did not use any other sources and references than the listed ones. I have marked all contained direct or indirect statements from other sources as such. Neither this work nor significant parts of it were part of another review process. I did not publish this work partially or completely yet. The electronic copy is consistent with all submitted copies.

Stuttgart, January 16, 2013