Jiotto – A Java Framework Implementing
the Giotto Semantics

Diplomarbeit
zur Erlangung des akademischen Grades
Diplom-Ingenieur
an der Naturwissenschaftlichen Fakultät
der Universität Salzburg

eingereicht von
Markus Amersdorfer

Betreuer
Ao. Univ. Prof. Mag. Dr. Helge Hagenauer

Salzburg, im März 2004
Abstract

Jiotto is a Java framework implemented using the Real-Time Specification for Java. Jiotto’s ideas are based on the Giotto project: This is an approach to embedded real-time programming which – above all – separates a program’s functionality and timing from the underlying platform.

Jiotto offers a way to develop programs in pure Java while still following the Giotto paradigm. Jiotto’s RTSJ-compliance provides the developer with a broad range of possibilities such as various runtime environments or hardware platforms specifically designed for RTSJ.

After an overall introduction in chapter 1, chapter 2 covers general topics concerning real-time systems.

The ideas as well as the components of Giotto are explained in chapter 3.

Chapter 4 deals with Java related topics, especially considering its real-time capabilities.

The design of the Jiotto framework and the corresponding development process leading to it are presented in chapter 5.

Simulating the control of an elevator and thus pointing out the applicability of Jiotto to real-world problems is shown in chapter 6.

The document concludes with final thoughts and comments in chapter 7.
Acknowledgements

First of all, I want to thank my supervisor Ao. Univ. Prof. Dr. Helge Hagenauer for his ongoing support in many scheduled and non-scheduled meetings we held. This thesis would not have been possible without him.

Many thanks to my friends from Mülln, at subnet and at the University of Salzburg for all those discussions on relevant and irrelevant topics about both work and life.

Thanks to all the developers of LyX\(^1\) and \LaTeX\(^2\), who made it possible for this document to be created in a most sophisticated way. Thanks to the developers of the free operating system Linux\(^3\), the GNU project\(^4\) as well as the Debian\(^5\) community for their great work. Thanks also to the creators of the awesome programming language Java\(^6\), the Real-Time Specification for Java\(^7\) as well as the Java-IDE Eclipse\(^8\). This document as well as the jiotto package were created based on especially all these projects.

Sincere thanks to my family for their support, not only throughout my studies at the university.

Most importantly, I want to thank my girlfriend Ursula Berghammer for helping me through and standing by me in those countless occasions over the last years. Thank you for your love and all this great time we share together!

Markus Amersdorfer
Salzburg, March 2004

\(^{1}\text{http://www.lyx.org/}\)
\(^{2}\text{http://www.latex-project.org/}\)
\(^{3}\text{http://www.kernel.org/}\)
\(^{4}\text{http://www.gnu.org/}\)
\(^{5}\text{http://www.debian.org/}\)
\(^{6}\text{http://java.sun.com/}\)
\(^{7}\text{http://www.rtj.org/}\)
\(^{8}\text{http://www.eclipse.org/}\)
# Contents

1 Introduction  
1.1 Giotto ........................................ 1  
1.2 Jiotto ....................................... 2  

2 Real-Time Systems  
2.1 Definition of a Real-Time System ................. 3  
2.2 Real-Time Operating Systems .................... 4  
   2.2.1 Real-Time Scheduling .................... 4  
2.3 Real-Time Programming Models ................... 7  
   2.3.1 Synchronous Model ....................... 7  
   2.3.2 Scheduled Model ......................... 8  
   2.3.3 Timed Model ............................. 8  

3 Giotto  
3.1 The Ideas Behind Giotto ........................... 9  
3.2 The Embedded Machine .......................... 10  
3.3 Components .................................. 10  
   3.3.1 Ports ................................... 11  
   3.3.2 Drivers .................................. 11  
   3.3.3 Tasks ................................... 12  
   3.3.4 Modes ................................... 12  
3.4 Giotto Micro Steps .............................. 13  
   3.4.1 Discussion of the GMS Sequence .......... 15  
3.5 More Specific Topics ............................ 16  
3.6 Giotto in Practice ............................. 16   

4 Java and Real-Time  
4.1 Java's Real-Time Incompatibilities ............... 17  
   4.1.1 Performance ............................. 17  
   4.1.2 Determinism and Memory Management ........ 18  
   4.1.3 Thread Model and Scheduling .............. 18  
   4.1.4 Device Drivers ........................... 19  
   4.1.5 Asynchronous Flow of Control ............. 19  
4.2 Approaches to Real-Time Java ..................... 20
Chapter 1

Introduction

1.1 Giotto

Usual computer programs such as word processors or web browsers “only” have to perform their tasks with no further strict constraints concerning available time, memory or other such resources.

While it might be annoying if a text editor lags for a few seconds every once in a while, this does not render the program unusable. However, considering more critical applications such as a car’s anti-blocking system, everything changes: Nondeterministic behaviour must not be tolerated in this environment – delaying computations for a few seconds every now and then might have disastrous consequences.

Such systems with precisely specified timing requirements are called real-time programs (see chapter 2 for more information on this topic). Furthermore, as such applications most always do not run on a standard PC but rather on specific hardware platforms as part of a bigger system with a different prime function (examples are the ABS of a car or the temperature control of a greenhouse), a lot of them can be classified as embedded systems usually.

Nowadays, most control software of embedded real-time systems has to be rewritten from scratch if its environment changes, for example due to a new car model being released. Giotto is a project developed at the University of Berkeley, which – among other things – represents an approach to improve re-usability of code, software components and application structures. Most importantly, introducing a new programming language based on code-execution through a “virtual machine”, Giotto separates a system’s timing and functionality from the underlying platform. (See chapter 3 for a more detailed explanation of Giotto’s design and ideas.)
CHAPTER 1. INTRODUCTION

1.2 Jiotto

This document presents Jiotto, a Java framework implementing the Giotto semantics. Giotto’s original ideas, structures and components are used to create a framework, which enables Java programmers to easily develop real-time applications with the above-mentioned advantages.

To achieve this goal, Jiotto has been implemented to comply with the Real-Time Specification for Java (RTSJ). This not only enhances Java with real-time capabilities and thus offers the needed deterministic behaviour. Being an officially approved standard, RTSJ makes it possible for Jiotto to be used with a broad range of RTSJ compatible compilers and hardware platforms.

Jiotto is compatible with Giotto. Giotto applications can be and most always are graphically modelled; figure 3.1 on page 11 shows a basic application model that represents a software’s functionality and timing. It is generally possible to use such Giotto application models as they are for Jiotto as well, and vice versa. Furthermore, a Giotto application’s native code such as task functions, possibly implemented in C, can be integrated in Jiotto as well by using the Java Native Interface (JNI).

Developing a Giotto system involves writing platform specific code in external languages such as C or Oberon. While creating such code is of course necessary with Jiotto too, the RTSJ standard used as a basis for Jiotto offers mechanisms to do so in “Java only”: Through the use of raw memory access as well as asynchronous event handlers and happenings, interrupt aware device drivers can be created with RTSJ. This way, an overall Jiotto system may be developed using Java as its sole programming language, offering Java’s usual advantages of rapid development and solid programs.\(^1\)

Another objective of this thesis is to evaluate RTSJ’s capabilities and suitability concerning the domain of embedded control systems with hard real-time constraints.

\(^1\)If needed or where applicable, "native code" written in languages such as C can be included in Java too.
Chapter 2

Real-Time Systems

2.1 Definition of a Real-Time System

As both Giotto and Jiotto are situated in the “real-time” domain, it is necessary to specify this term more closely. The following is a definition given by Burns and Wellings [3]:

any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified delay.

The crucial thing about this definition is to “respond within a specified delay”. As has already been mentioned in the introductory chapter 1, usual programs such as text editors do not fail their purpose simply because of nondeterministically lagging for a few moments in time. Nevertheless, with real-time programs, this exactly is the case: The system has to react within a given time frame, always and in a deterministic way.

Determinism is another important keyword when dealing with real-time systems. Citing Dibble [4]:

In the real-time field, the term determinism means that timing is predictable at the precision required by the problem without heroic effort. [...] Without a deterministic operating system and processor, the analyst cannot even predict whether an event will reach the event handler before its deadline, much less whether the event handler will complete a computation before the deadline.

An important classification of real-time systems uses the terms hard real-time and soft real-time. With hard real-time systems, it is necessary to present the result within the given time frame. Missing a deadline is a catastrophic event and the overall system has failed. Examples are the flight control system of an aircraft and the anti-blocking system of a car. Soft real-time systems on the other hand may miss a deadline every now and then. While of course this
should not happen on a regular basis, the system may still function properly though. Video streaming is an example for a soft real-time application, as a missing frame every now and then is probably acceptable. However, losing for example five frames in a row can not be tolerated anymore.

Giotto was designed for “embedded control systems with hard real-time constraints” [10]. Consequently, this paper will concentrate on this field of real-time systems with corresponding exemplary applications such as controlling a helicopter’s autopilot or a car’s anti-blocking system. Chapter 6 shows a simulation of Jiotto controlling an elevator.

2.2 Real-Time Operating Systems

For real-time programs to be able to meet their deadlines, they have to be designed and implemented in a specific, real-time compatible way (see sections 2.1 and 2.3). Nevertheless, real-time capable applications alone do not suffice. Apart from the software actually performing the task, the platform beneath, consisting of the hardware and the operating system, has to be “real-time” too.

While the hardware is assumed to be deterministic and reliable enough to meet the expectations\(^1\), the topic of real-time operating systems (RTOS) will be dealt with in more detail in this section.

A RTOS has to be able to perform its tasks of managing hardware and software resources predictably and within a specified time frame. To achieve this goal effectively, it should be pre-emptible, which means that not only application code but also the kernel code can be interrupted by an external event.

2.2.1 Real-Time Scheduling

Due to multitasking, most modern computer systems can handle and – apparently simultaneously – execute several processes at a time. This is accomplished by rapidly switching between active processes. The scheduler poses a vital part of a general-purpose operating system as it decides, which one of the runnable processes may use the CPU next to have its code executed. When taking the requirements of real-time applications into account, scheduling algorithms become even more important.

The most basic classification differentiates between static and dynamic scheduling. While a dynamic scheduling algorithm takes information into account that is only available at run-time, a static one does not do that and decides before the actual code execution or only depends on other static information such as process priorities.

---

\(^1\)With safety-critical real-time systems, fault tolerance is an important issue in the overall design and implementation process. Dealing with concepts such as detecting and handling failures, errors or faults in real-time systems is outside the scope of this thesis. See the corresponding literature for information on this topic, for example Kopetz [21]. Note that fault tolerance is not restricted to hardware issues.
2.2. REAL-TIME OPERATING SYSTEMS

2.2.1.1 Earliest Deadline First (EDF) Scheduling

With this dynamic scheduling approach, the process with the nearest deadline is executed first. Notice: Be aware of some problems with this most basic deadline-related scheduling approach such as those Dibble [4] mentions as “total failure”.

This means, while generally offering better CPU utilisation (up to 100% [25]) than the following FPS algorithm, the latter is typically easier to implement and its behaviour in overloaded situations is more predictable.

2.2.1.2 Fixed Priority Scheduling (FPS)

Fixed Priority Scheduling (FPS) is a static scheduling algorithm where all processes have static priorities. Depending on them, the scheduler knows the importance of every process and can prefer the higher ranked over the lower ranked ones. Citing Burns and Wellings [3]:

In real-time systems, the 'priority' of a process is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity.

As usual, if processes may be interrupted during their execution phase (for example as a higher priority process becomes runnable and thus needs the CPU), the scheduling scheme is known to be pre-emptive.\(^3\) If a process can retain control over the CPU until it has finished or gives it away voluntarily, it is called non-pre-emptive.

Assuming a pre-emptive FPS scheme, the question arises of what happens if a second process becomes runnable which has the same priority as the one already owning the CPU. Two widely used policies for this case are FIFO and Round-Robin (RR):

With FIFO, a process owns the CPU until it blocks or completes its calculations and thus “voluntarily” sets the CPU free.\(^4\) When using Round-Robin (RR), the CPU is equally shared between all processes with the same priority. (A process is pre-empted as soon as it has consumed its available time slice.\(^5\))

Rate Monotonic Priority Assignment (RMS)  With this priority assignment scheme applicable for periodically executing processes, each one has a

\(^2\)Advantages and disadvantages of these algorithms can for example be found in Burns and Wellings [3] in more detail.
\(^3\)Pre-emptive FPS is required by the RTSJ.
\(^4\)The RTSJ reference implementation (see section 4.4.1) in combination with Linux as the operating system uses a pre-emptive FIFO based FPS scheme by default.
\(^5\)Silberschatz et al. [31] mentions such a time quantum to range from 10 to 100 milliseconds usually.

Of course, the best value depends on many factors such as the problem to be solved and the platform used. Making the time quantum shorter results in better reactivity on the one hand, but leads to more context switches and thus lower overall throughput on the other hand.
priority that relates to its time period: The shorter the period, the higher the priority. Though CPU utilisation is rather low (for the general case with a large set of processes it is about 70%), it is shown in Liu and Layland [25] that – assuming fixed priority scheduling – “such a priority assignment is optimum in the sense that no other fixed priority assignment rule can schedule a task set which cannot be scheduled by the rate-monotonic priority assignment”. This is valid for periodic, independent tasks with a constant run-time and where the deadline equals the end of the period.6

**Priority Inversion** Though priority inversion is a quite simple situation, it might be devastating for real-time systems. In essence, priority inversion describes a situation, where a high priority process is not able to run because it has to wait for a low priority process to finish first.

A typical setup involves three tasks with distinct priorities each: Assume the lowest priority process \(C\) is currently working and already using some shared resource \(SHM\). Next, the highest priority process \(A\) gets ready to work. However, as it also needs exclusive access to \(SHM\), it has to wait for \(C\) to leave its critical section. The actual problem now arises if the medium priority process \(B\) gets ready to run. Assuming it does not use \(SHM\) and due to having a higher priority than \(C\), the medium priority process gets access to the CPU and may use it for an undefined amount of time. The priority inversion has occurred: A lower priority process (\(B\)) is preferred over a higher priority one (\(A\)).

Two widely used solutions for this problem are the priority inheritance protocol and the priority ceiling protocol which both depend on changing process priorities at run-time.

**Priority Inheritance Protocol** Priority inheritance7 solves the above-mentioned problem by temporarily assigning process \(A\)’s priority to process \(C\). This way, \(C\) will not be pre-empted by \(B\) and thus can finish executing its critical code as soon as possible in order to set free the shared resource.

While this approach may work in many situations, Sha et al. presented a better one8 solving some problematic issues of plain priority inheritance such as possible deadlocks: The priority ceiling protocol.

**Priority Ceiling Protocol** Burns and Wellings [3] list the following characteristics to define a priority ceiling protocol:

- Each process has a static default priority assigned (perhaps by the deadline monotonic scheme).

---

6Jiott (see chapter 5) fits into this environment perfectly and uses rate monotonic scheduling for the task threads.
2.3. REAL-TIME PROGRAMMING MODELS

- Each resource has a static ceiling value defined, this is the maximum priority of the processes that use it.
- 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.

The run-time behaviour is described by Kopetz [21] as follows:

The priority ceiling of a semaphore is defined as the priority of the highest priority task that may lock this semaphore. A task $T$ is allowed to enter a critical section only if its assigned priority is higher than the priority ceilings of all semaphores currently locked by tasks other than $T$. Task $T$ runs at its assigned priority unless it is in a critical section and blocks higher priority tasks. In this case, it inherits the highest priority of the tasks it blocks. When it exits the critical section, it resumes the priority it had at the point of entry into the critical section.

2.3 Real-Time Programming Models

To conclude the description of real-time fundamentals, this section presents basic real-time programming models as described in Kirsch [19].

The following three different models are visualised in figure 2.1 (which is based on a figure to be found in Kirsch’s document).

2.3.1 Synchronous Model

The idea behind a synchronous model is that the real-time system reacts to external events. Furthermore, every corresponding calculation it performs happens in logically “no time at all”: At the same time the system receives data from the environment, it can already present the corresponding answer.

As in reality anything to be performed needs some time of course, this zero-time concept can (only) be approximated by returning the result before the environment’s next event occurs. Though being within a specified time frame,
one typical characteristic of a synchronous system is that the exact point of
time when the result is available may vary.

### 2.3.2 Scheduled Model

The *scheduled model* is more intuitive to a usual software developer: A real-
time program may consist of several processes which consume time and are
scheduled by the OS’ scheduler when to be executed or not. The result has to
be available before a specified deadline, but again the exact point of time when
this happens may vary.

Quoting Kirsch [19]:

> The downside of the scheduled model is that it is not compositional
> with respect to value- or time-determinism. In general, the com-
> position of scheduled processes results in real-time behavior of the
> scheduled processes that is different from the real-time behavior of
> the processes when running individually. The problem [lies] in situ-
> ations such as priority inversion or deadlock [...].

While the latter problem may be solved by concentrating on *time safety* (which
means to use “timed semaphores” that may lock access to shared resources only
within a specified time frame) instead of the usual *space safety*, such a system
still was not time-deterministic.

### 2.3.3 Timed Model

A system based on a *timed programming model* interacts with the environment
in a more deterministic way than the scheduled model specifies. A calculation’s
result is always available at a specific point in time. Most always, the corre-
spending process will finish before this given time. Nevertheless, in this case it
has to wait and must present the outcome not until the defined time has ar-
rived. From an outside point of view, the process may be working all the time
until the result is available. (During the phase before the anticipated response,
the environment does not care whether the process has already physically fin-
ished its calculations or not, as long as it returns the result on time eventually.)

While checking for a program to be *time-safe* is the task of the compiler,
this is difficult and “may not always be feasible at compile-time. If the timed
program is indeed late, a runtime exception may be thrown [...]” [19].

Giotto (see chapter 3) is a programming language following the timed model,
but it also uses ideas of the synchronous and the scheduled model. Another
well-known approach to the timed model is the *Time-Triggered Architecture*
developed at the Technische Universität Wien [21].

---

A program is *time-safe* if there is enough real time available to be able to complete calculations on time. Kirsch [19]: “Time safety depends on performance, utilization, and scheduling scheme.”
Chapter 3

Giotto

3.1 The Ideas Behind Giotto

The following quotations from Henzinger et al. [11] present a good overview of Giotto:

Giotto is a [...] design methodology for implementing embedded control systems on platforms of possibly distributed sensors, actuators, CPUs, and networks. Giotto is based on the principle that time-triggered task invocations plus time-triggered mode switches can form the abstract essence of programming control systems. [...] Giotto supports the automation of control system design by strictly separating platform-independent functionality and timing concerns from platform-dependent scheduling and communication issues. The time-triggered predictability of Giotto makes it particularly suitable for safety-critical applications with hard real-time constraints. [...] Giotto is a programming language that aims at distributed hard real-time applications with periodic behavior, such as control systems. A typical control system periodically reads sensor information, computes control laws, and writes the results to actuators.

The usual design process for embedded applications includes – above all – the two steps modelling and implementation of the software. Timing and functionality are part of the system’s model and, from the environment’s point of view, characterise it explicitly. Giotto’s idea is to separate the system’s behaviour from its implementation: The outside world does not care whether the overall apparatus consists of one or many CPUs or whether it is distributed or not. The only vital thing is that the system reacts the way it is supposed to by returning the correct results (“functionality”) at the right time (“timing”). With Giotto, the underlying platform, consisting of the hardware and an operating system, can be changed without altering the system’s behaviour.
3.2 The Embedded Machine

Giotto uses a virtual machine, the so-called Embedded Machine or \textit{E} machine. Similarly to the well-known process of the Java programming language, Giotto source code is first compiled into platform independent \textit{E} code (code executed by the Embedded Machine). The \textit{E} code specifies especially the system’s timing. In a second phase, the platform specific characteristics such as hardware performance and the operating system’s scheduler are taken into account to check for “time safety” of the \textit{E} code.

A detailed explanation of the \textit{E} machine and \textit{E} code can be found in Henzinger and Kirsch [12].

3.3 Components

Having described the more practical parts of Giotto, Giotto’s formal components are presented in this section. The original definition of the language can be found in Henzinger et al. [10].

A Giotto program generally consists of different \textit{modes}, which have similarities to the states of a state machine. Each mode usually holds several \textit{tasks}. A task is the basic functional unit in Giotto. As long as the system is in a distinct mode, the corresponding tasks are executed periodically. By switching between the modes and thus changing the set of tasks to be executed, the system can alter its overall behaviour. The concept of \textit{drivers} is used for tasks to be able to communicate with each other, enable the system to process sensory environmental inputs as well as control actuators. A driver transfers values between so-called \textit{ports}, which are the containers of data values. Figure 3.1 on the next page shows a basic Giotto model consisting of a mode $m_0$ with one task $t_0$, the corresponding input port $i_0$ is updated by the driver $d_0$ with the value of the output port $o_0$. The mode has a period of 100 time units (typically milliseconds), the task’s frequency is 2; this means that the task is started every 50 ms. The function of the task determines the result that is finally made available via the output port. A task function may process a lot of data, but it may also be as simple as to increment the input value by one and return the result.

One key aspect of Giotto is that it makes use of all three programming models presented in section 2.3: While the execution of drivers follows the \	extit{synchronous} approach and thus happens in logically zero time, the tasks represent \textit{scheduled} computation. Finally, “the periodic invocation of tasks, the reading of sensor values, the writing of actuator values, and the mode switching are all triggered by real time” [10].
3.3. COMPONENTS

3.3.1 Ports

Ports are used for the overall communication and have a specific value type assigned. The current value is kept until it is explicitly updated, for example by a driver. Furthermore, each port has an initial value. Possible types of ports are:

- **Actuator ports**
- **Sensor ports**
- **Task input ports**
- **Task output ports**
- **Task private ports**

Within these types of ports, their names have to be distinct.

While sensor ports are updated by the environment, all others are updated by the Giotto system itself: Actuator and input ports are updated by drivers, output and private ports by tasks.

Output ports may also be classified as mode ports. A set of mode ports consists of all of the mode’s output ports and thus is specific to each mode. Note that mode ports do not allocate distinct memory on their own, but rather use exactly the same memory as the corresponding output ports do.

3.3.2 Drivers

The following types of drivers are used to transfer values each from a specific set of source to destination ports:

- **Task driver**: Updates input ports with values from sensor or mode ports.
- **Actuator driver**: Updates actuator ports with values from mode ports.

\[^1\text{Note that constant values are valid sources too.}\]
• **Mode driver:** Updates mode ports with values from sensor or mode ports.

Besides the sets of source and destination ports, drivers also have a function \( h \) defined. (Though formally being defined as a mathematical function, in practice, the driver function boils down to transferring the source value to the destination port as it is, without changing it.) Furthermore, each driver has a guard associated, which is evaluated before the execution of the driver function: The guard evaluates the values of the driver source ports. If it returns true, the function is performed – otherwise it is not.

### 3.3.3 Tasks

Tasks are periodically executed code blocks performing the actual work of computing the system's responses according to its state and the current environmental situation. A task’s functional code is executed within a certain time period, specified as a frequency in relation to the mode's period. Once started, a task cannot be terminated prematurely – it simply returns the results by updating its output ports at the response time (see the “Timed Model” in figure 2.1 on page 7).

Besides its function, a task is defined to consist of input ports (which must be distinct for all tasks) and output ports (which may be shared among tasks, as long as the sharing tasks are not invoked in the same mode). Furthermore, tasks can have a set of private ports, which define an internal state.

#### 3.3.3.1 Giotto vs. Jiotto

When developing with the original Giotto, the actual code of a task has to be implemented natively in external languages such as C or Oberon. With Jiotto (see chapter 5), it is possible to develop the overall system in Java only – both the functional code of tasks as well as the equivalent to the Giotto code defining the software model.

### 3.3.4 Modes

A Giotto program is a collection of Giotto modes. According to environmental conditions and corresponding calculations, the Giotto program can switch between modes in order to change its overall behaviour or state it is in.

Besides a positive mode period \( \pi \) (most always given as milliseconds), a mode also consists of mode ports: This set of ports equals to the set of this mode's task output ports and is especially used when switching between modes. Furthermore, a mode specifies sets of the following entity types:

- All of a mode's task invocations are dealt with at a given frequency. Each task invocation has both a task and a task driver associated, with the driver's function being executed if the corresponding guard evaluates to true.
3.4 Giotto Micro Steps

The Giotto Micro Steps (GMS) are a vital part of the overall Giotto design. During the execution of a Giotto program, these steps are executed by the E machine at regular intervals depending on the mode the system currently is in:

The program configuration\(^2\) is updated accordingly every \(\frac{\pi[m]}{\omega_{\text{max}}[m]}\) time units with \(\pi[m]\) being the mode period of \(m\) and \(\omega_{\text{max}}[m]\) being the least common multiple of the mode frequencies\(^3\) of \(m\).

The GMS are executed in the following order:

1. Update task output and private ports
2. Update actuator ports
3. Update sensor ports
4. Update mode
5. Update mode ports
6. Update mode time
7. Update task input ports
8. Update active tasks
9. Advance time

\(^2\)Above all, a program configuration is defined by the current mode, valuations of all ports and a set of active tasks.

\(^3\)The mode frequencies consist of all the task frequencies, the actuator frequencies and the mode-switch frequencies.
Figure 3.2: Relations between Giotto Components
The GMS are covered in more detail in section 5.2.1.5 as part of the description of the *giotto* class hierarchy.

### 3.4.1 Discussion of the GMS Sequence

With the execution sequence of the GMS as defined by Henzinger et al. [10] and as shown above, the application developer needs to bear in mind that the actuator ports are updated prior to a possibly performed mode switch. Depending on the implementation, when a mode switch is performed, this sequence may result in a flow of information to the actuators about the system's current situation that is delayed by up to one mode round.

In a given mode, a distinct actuator port must not be updated by several actuator drivers at the same time. As, in practice, drivers simply transfer values according to a fixed schedule (or do not perform any transfers at all if the guard evaluates to false), an actuator port can only be updated with the current value of a distinct port or with a constant value. Thus, in order to update an actuator port with *different* values, there are the two possibilities to either use a different driver in a different mode, or to have a task function calculate the value for the actuator port and make it available to the driver via an output port.

Using different drivers in different modes introduces the delayed flow of information, as the system eventually switches to a new mode in GMS 4 but has already updated the actuator port with a – meanwhile old – value in GMS 2. If the given actuator was updated in the target mode at a frequency of 1, the flow of information to the physical environment lags one mode round behind the software system's state. While real-world applications often use very short mode periods (or high driver evaluation frequencies) to decrease the response time, this might not be a problem after all, but it should be kept in mind when developing a Giotto application.

With the second approach of having a task function calculate the value for an actuator, it is possible to circumvent the delayed flow of information at the cost of duplicating the mode driver's guard condition and include the case of a possibly performed mode switch in the task function. A task thus is aware of the conditions under which a mode switch will or will not be performed and can calculate the corresponding actuator control values accordingly.

Note: The Jiotto application *Elevator2* presented in chapter 6 uses mode periods of only 50 ms. Assuming an elevator that needs 10 seconds to move the cabin by one floor, which equals a distance of 3 meters, the cabin has an average velocity of 0.3 meters per second. A real-world elevator also slows the cabin down when approaching the target floor, resulting in an assumed velocity of only 0.05 meters per second. This means that, with mode periods of 50 ms and driver evaluations with a frequency of 1, the cabin would be positioned correctly within 2.5 millimetres of the target floor, which is an acceptable accuracy.
CHAPTER 3. GIOTTO

Nevertheless, the Elevator2 system still uses the second approach described above. While this introduces duplication of the mode drivers’ guards’ conditions in the task functions, it simplifies system design by updating movement-related actuators in movement-related modes only.

3.5 More Specific Topics

Giotto also has the concept of different levels of annotations. With this, the developer can help the compiler “in finding a feasible scheduling function in difficult situations” [10], such as having to solve a Giotto scheduling problem for distributed platforms. One aspect that needs to be available is the worst-case execution time (WCET) of the functional code.\footnote{Kopetz [21] defines the WCET of a task to be “an upper bound for the time between task activation and task termination. It must be valid for all possible input data and execution scenarios of the task [...]”. Determining the WCET, which heavily depends on the platform being used and often is a non-trivial task, is outside the scope of this thesis.}

For a more indepth exposure to the Giotto language and components definition, including the above topic, refer to Henzinger et al. [10, 11] as well as the Giotto web page [8].

3.6 Giotto in Practice

The Giotto Development Kit (GDK), available from the Giotto web page [8], above all includes a Java-implemented compiler that compiles Giotto source code into E code, an implementation of the E machine and some examples. Furthermore, for testing purposes, it enables the developer to write and use functionality code in Java.

E machine implementations used for real-world applications above all include the following ones:

- E machine for a custom RTOS on a StrongARM embedded processor used to control an autonomously flying helicopter [13].
- E machine for Linux.
- E machine for Lego Mindstorm robots where the E machine is part of the OS kernel [11].

A project that is heavily based on Giotto is MoDECS [27]. Its principal challenge is to extend the Giotto methodology and tools to cope with distributed platforms.
Chapter 4

Java and Real-Time

Java is a programming language developed by Sun Microsystems. It was designed to meet the needs of portability and code re-usability, rapid application development, security as well as reliability, which are at least partially made possible through the language's simplicity.

Java is not only heavily used for general-purpose web- and user-applications, but has also gained ground in the embedded hardware market – especially cell-phones with Java support are widely spread nowadays. The industrial trend to adopt the language is most importantly made possible due to the different flavours of Java. Besides the Java 2 Platform Standard Edition (J2SE), among others, there is also the Micro Edition (J2ME), which is an optimised Java runtime environment specifically designed for hardware such as smart-cards, cell-phones and similar.

Despite this success on the embedded market, usual Java programs are not real-time capable. The next sections deal with the problems that standard Java has and presents according methods of resolution.

4.1 Java’s Real-Time Incompatibilities

4.1.1 Performance

The typical criticism concerning Java is its lack of performance. While the early versions of Java really did perform poorly in many areas, this has dramatically improved with more up-to-date releases — especially due to the introduction of and advancements in the field of just-in-time (JIT) compilation as well as Sun’s Hotspot JVM. Further possibilities to overcome performance problems exist. Besides the fact that hardware performance gets cheaper every day, it is possible to have Java programs compiled to native code and thus circumvent

---

1 Runtime environments, development kits and all kinds of documentation including the “Java Language Specification” itself can be found at Sun’s official Java web page [15]. A great resource on the language is Bruce Eckel’s book “Thinking in Java” [5].
the need of byte code interpretation. Furthermore, there exists specialised hardware, which for example is able to execute Java byte code directly.

Nevertheless, further improvements are still possible. Even Dibble admits some performance drawbacks of Java: "For now, that shortcoming has to be accepted: Java is slower than the alternatives, and it requires a daunting amount of memory to run a trivial program." Thus, performance may not be a strength of Java on the one hand, but on the other hand, it is something a real-time programmer may not need necessarily.

### 4.1.2 Determinism and Memory Management

The Garbage Collector (GC) is responsible to “automatically” free previously used memory again, which is not needed anymore. As Java has no dedicated method to explicitly perform this action manually, the GC is an integral part of memory management for most every Java implementation.

Garbage collection is a complex topic. In general, the GC is nondeterministically executed in cases where the system is idle, the user application has requested GC (note that in Java this is considered a user suggestion and the JVM is not required to immediately act on its behalf), or when there is not enough free memory available but new objects are to be created.

There are different GC algorithms, each having certain advantages and disadvantages. To point out possible problems of GC in combination with a real-time environment: Assuming the simple “mark and sweep” algorithm to perform GC, it cannot be pre-empted and resumed afterwards. “Mark and sweep” starts with a root set and marks every referenced object as “alive”. Having finished this procedure for all objects, every object without the “live” flag set is garbage and the according memory can be freed. The problem for the GC when being pre-empted is that the application could have performed operations, which might finally lead to the GC erroneously freeing objects that are still in use. This means, in order to correctly collect unreferenced memory, the GC has to finish its run and must not be pre-empted. As a consequence, the actual application has to be suspended for as long as the GC runs, which might be a period of up to tenths of a second or even several seconds.

Though there are “incremental” and other advanced GC algorithms, the problem usually boils down to the following statement to be found in Dibble [4]: “Without a processor dedicated to garbage collection, the JVM cannot guarantee that garbage collection will not disrupt the timing of any code that includes object creation.”

### 4.1.3 Thread Model and Scheduling

There are several problems with Java’s thread model and the way threads are scheduled. By default, Java only offers 10 priorities, which is not flexible
4.1. JAVA’S REAL-TIME INCOMPATIBILITIES

enough to for example perform efficient rate monotonic scheduling. Furthermore, the priorities are only used as “suggestions” to the scheduler. Quoting the Java Language Specification [9]:

Every thread has a priority. When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.

While this may increase reactivity for general-purpose interactive processes, it is clearly not tolerable for priority based real-time scheduling. In addition, Java offers no mechanism to avoid priority inversion.

Furthermore, as Brosgol and Dobbing [2] point out, there is “no guarantee that priority is used for selecting which thread is awakened by a \texttt{notify()}, or which thread awakened by a \texttt{notifyAll()} is selected to run”.

4.1.4 Device Drivers

Device drivers usually access memory via pointers to primitive data types. Unfortunately, default Java lacks these capabilities. In addition, the hardware is often controlled through the use of interrupts, which Java has no access to either.

4.1.5 Asynchronous Flow of Control

With real-time systems, it is sometimes necessary to transfer the flow of control from one thread to another due to some external event such as a hardware interrupt. (The intuitively related method \texttt{interrupt()} can not be used to service real hardware interrupts: If the method is invoked on a blocked thread that has for example invoked \texttt{wait()}, it is woken up. Nevertheless, in case the thread has been running normally, all that happens due to the method invocation is that its interrupted flag is set, but no explicit exception is thrown which would notify the thread about the interrupt.)

Though Java initially had some threading methods dedicated to asynchrony, most of them have been deprecated\(^2\) or are not to be used due to insecure behaviour.

\(^2\) See “Why Are \texttt{Thread.stop}, \texttt{Thread.suspend}, \texttt{Thread.resume} and \texttt{Runtime.runFinalizersOnExit} \texttt{Deprecated}?” at \url{http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html}
CHAPTER 4. JAVA AND REAL-TIME

4.2 Approaches to Real-Time Java

Based on requirements for real-time extensions to Java specified in a paper published by NIST\(^3\), two groups accordingly defined the following two Java-oriented real-time specifications:

- “Real-Time Core Extensions” by the Real-Time Java Working Group, technical committee of the J-Consortium [28].
- “Real-Time Specification for Java (RTSJ)” by the Real-Time Java Expert Group (RTJEG) [1].

For a comparison in more detail of these efforts, see Brosgol and Dobbing [2]. The following two quotations serve as a good summary:

The J-Consortium has focused on defining real-time “core” facilities external to a JVM, similar to services provided by a traditional RTOS, whereas the RTJEG has defined an API that needs to be supported within a JVM implementation.

[...]

Kelvin Nilsen has summarized this distinction as follows: “The RTSJ makes the Java platform more real-time, whereas the Core Java specification makes real-time more Java-like.”

There exists a complete reference implementation for RTSJ that makes it possible to implement and execute real-time Java applications. Especially for this practical purpose, the remainder of this document concentrates on this approach. More information on the Core Extensions can be found at the official homepage, summaries are given for example in Brosgol and Dobbing [2] or Schoeberl [30].

4.3 Real-Time Specification for Java (RTSJ)

The Real-Time Specification for Java (RTSJ) extends the original Java platform and its API. It has been developed under the Java Community Process and is referenced as Java Specification Request JSR 1 [16]. The specification, reference implementation (RI) and technology compatibility kit (TCK) are available or referenced to from the web page [29]. The specification is also available in print [1].

While being widely compatible with standard Java in the sense that RTSJ implementations should be backwards compatible to non-real-time Java programs, an implementation following the usual Platform/JVM/Byte Code architecture will have to make RTSJ specific adjustments to the JVM (see the RI, section 4.4.1).

Java’s problems with real-time topics have been discussed in section 4.1. The following sections deal with how the RTSJ addresses these problems in order to make Java real-time capable and present the reference implementation. Further details about the RTSJ can be found in external resources, for example [1, 4, 20].

4.3.1 Performance

As described in previous chapters, “real-time” actually is about determinism and predictability, and these requirements can be met by the RTSJ. Whether high performance is essential or not depends on the field of application.

4.3.2 Determinism and Memory Management

With the nondeterministic side effects of garbage collection (GC) and Java’s inability to free previously used memory otherwise, making the language real-time capable requires quite significant enhancements.

The two mainly used memory types for most kind of data in Java are stack and heap. As discussed in Eckel [5], the fixed-size primitives such as int or boolean as well as object references are placed on the much more efficient stack, whereas usual objects are placed on the “general-purpose pool of memory” heap.

While, due to the nature of the stack, the memory for primitive inline variables is automatically freed again after leaving their scope, this is not the case for objects placed on the heap. Here, the GC has to free unreferenced memory. Every time an object is created using the new keyword, the application consumes additional heap memory. In case there is none left, the GC is activated and tries to find some.

The RTSJ solves this problem by introducing the new memory areas immortal memory and scoped memory.

**Immortal Memory** The easiest way to circumvent garbage collection is to simply not need its functionality by never releasing any memory that has been allocated once. All objects placed in the immortal memory exist as long the application is running. There is no way to release used memory again.

**Scoped Memory** Memory allocated from a scoped memory is automatically freed at the end of the scope.

By not needing GC, these new memory areas not only solve the problem of new possibly blocking the application for an undetermined time period. Furthermore, it is also possible for code that does not reference objects in the heap to pre-empt the GC without it having to start all over again – real-time processes therefore can be treated in preference to any Java system code.
CHAPTER 4. JAVA AND REAL-TIME

These advantages do not come for free, though. As not simply all objects are placed on the heap as is done with standard Java, there have to be additional rules to allow references from one memory area to another. Especially due to the volatile nature of scoped memory, these are partially quite restrictive: While objects on the heap or immortal memory are always referenceable from any memory area⁴, references to objects in a scoped memory are for example never possible from the heap or immortal memory. Bearing in mind that scoped memory areas can be nested, “scoped objects” can be referenced to from within the same or inner scopes.⁵

4.3.2.1 Usage of Non-Heap Memory

Immortal memory is shared among all threads of an application. The method ImmortalMemory.instance() returns a reference to this memory area.

A scoped memory object with a specified memory size has to be created before it can be used to hold “scoped objects”. The following code creates a scoped memory object with linear allocation time and a size of 8 kilobytes:

```
LTMemory scopedMem = new LTMemory(8192, 8192);
```

The scopedMem object itself is allocated from the current memory area, which can be any of heap, immortal memory or another scoped memory.

The most typical ways of allocating from these memory areas are to either enter them from within application code and thus temporarily change the default memory area to the new one, or to create a real-time thread that has a specific memory area set explicitly at construction time.

A problem with these new memory areas is the introduction of possible memory leaks: The developer has to bear in mind that at least every new operation, when performed on immortal memory, consumes memory which can not be freed anymore. When repeatedly used, this can lead to insidious memory leaks. The code block shown in algorithm 1 on the facing page intends to enter a scoped memory and temporarily allocate a new object from it. While this actually is part of what happens, if the current memory area is immortal memory, a memory leak is produced due to the way the enter method is used: Every time it is invoked, a new Runnable object is created, which’s run() method is then executed with the scoped memory as its default memory area. The problem is the creation of the Runnable object, which is allocated from the surrounding immortal memory area every time.

The solution to this problem is to create an object of a specific class once only and then reuse this object as shown in algorithm 2 on the next page.

---

⁴An exception are references to objects on the heap from within a NoHeapRealtimeThread, this is discussed in section 4.3.3.

⁵Dibble [4] discusses in detail specific conditions, which have to be met when using nested and shared scoped memory areas.
4.3. REAL-TIME SPECIFICATION FOR JAVA (RTSJ)

Algorithm 1: Possible memory leak in RTSJ

```java
LTMemory scopedMem = new LTMemory(8192, 8192);
for (int i = 0; i < times; i++) {
    scopedMem.enter(new Runnable() {
        public void run() {
            Integer run = new Integer(i);
            System.out.println("This is run " + run);
        }
    });
}
```

Algorithm 2: How to prevent memory leaks in RTSJ

```java
class MyLogic implements Runnable {
    int x;
    public void run() {
        Integer run = new Integer(x);
        System.out.println("This is run " + run);
    }
}
MyLogic logic = new MyLogic();
LTMemory scopedMem = new LTMemory(8192, 8192);
for (int i = 0; i < times; i++) {
    logic.x = i;
    scopedMem.enter(logic);
}
```
Extensively using anonymous inner classes can make the code hard to read sometimes, so it may be advisable to explicitly use named classes not only to prevent memory leaks as has just been described.

### 4.3.3 Thread Model and Scheduling

The RTSJ requires an implementation to support *pre-emptive fixed priority scheduling*, other schedulers may be implemented and can be integrated via a specific API. There have to be at least 28 distinct real-time priorities, additionally to the 10 default priorities of the standard Java. Real-time threads can be used concurrently to non-real-time threads, with the former having access to specific real-time features such as scheduling or release parameters as well as memory specifications. (See section 4.3.7.1 for more information of synchronisation specifics such as priority inversion avoidance protocols.)

Classes implementing the `Schedulable` interface and which are thus able to be run by a scheduler, besides `AsyncEventHandler` (see section 4.3.4), are `RealtimeThread` and its subclass `NoHeapRealtimeThread`.

#### 4.3.3.1 RealtimeThread

`RealtimeThread` extends Java's `Thread` class and essentially introduces the following additional features – different constructors offer the possibility to explicitly set subsets or all of them at construction time via references to objects of suitable types:

**Scheduling Parameters** With the default FPS scheme, this usually is an object of type `PriorityParameters` and holds the process’ priority.

**Release Parameters** This object holds all timing related information such as when to kick-off the thread's `run()` method after invoking `start()`, the thread's periodicity, values for cost and deadline as well as the corresponding overrun and miss handlers.

**Memory Parameters** Attributes such as possible memory consumption can be limited.

**Memory Area** The default memory area can be specified at construction time already. All new allocations from within the thread's `run()` method – if not changed explicitly – use this area by default.

As indicated by the release parameters' description, real-time threads can be *periodic*. “Hello Periodic Real-Time World” is a simple periodic real-time program, see algorithm 3 on page 27.

Note: The priorities or periods of threads can be changed dynamically at run-time.\(^6\)

---

\(^6\)This is used by Jiotto for the task threads as well as the control thread (see chapter 5).
4.3. REAL-TIME SPECIFICATION FOR JAVA (RTSJ)

4.3.3.2 NoHeapRealtimeThread

A NoHeapRealtimeThread is not allowed to use heap memory in any way. It can neither allocate objects from heap nor even reference such. This enables the thread to pre-empt even the garbage collector. NoHeapRealtimeThread is thus an essential part of the RTSJ as, in combination with immortal and scoped memory, it makes it possible to have deterministic behaviour of threads.\(^7\)

A NoHeapRealtimeThread object itself must not be allocated from heap either. To create a no-heap object, either immortal or scoped memory has to be entered first.

4.3.4 Asynchronous Events

A feature, which is missing in standard Java but that is often needed in embedded real-time programming is the usage of interrupts and signals. While the J2SE and similar versions do use events as part of the AWT API used for GUIs, this is still not flexible and general enough for many cases.

The RTSJ introduces asynchronous events with each having one or more asynchronous event handlers (AEH) being associated. This enables usage of external interrupts and signals as well as internal events (through invocation of fire()) for Java programs. In combination with physical memory and raw memory access (see section 4.3.6), development of device drivers is hereby possible. Note that when using usual AEH, events lead to an implicit start of threads for the AEH. As this might pose a performance bottleneck, the application’s response time can be improved by using bound AEH, which use distinct threads that are dedicated for specific events and have to be created only once.

Asynchronous events are covered in detail in Dibble [4]. The following quotation shows the most important fields of application:

- If a thread (or AEH) misses its deadline, the scheduler can fire an AIE [Asynchronously Interrupted Exception].
- If a thread (or AEH) overruns its CPU budget, the scheduler can fire an AIE.
- The physical memory allocator can use async event handlers to notify callers when memory is inserted and removed.
- The PeriodicTimer and OneShotTimer classes fire async event handlers when they expire.

4.3.5 Asynchronous Flow of Control

The RTSJ implements Asynchronous Transfer of Control (ATC) by throwing an AsynchronouslyInterruptedException (AIE) into another thread. If the

\(^7\)Nevertheless, it is still possible for no-heap threads to communicate with heap-using threads, see section 4.3.7.1.
thread is in a method that throws AIE and thus declares itself to be “interruptible”, the exception is dealt with immediately. Methods that do not throw AIE as well as synchronized blocks, though, defer the ATC until a section is reached which allows propagation of the interrupt. Blocked threads due to invocation of wait(), sleep(), join() or an I/O operation that throws InterruptedIOException will be unblocked.

Although ATC is a powerful feature, it is complicated to get to know it as well as to learn how to use it correctly.

### 4.3.6 Physical Memory and Raw Memory Access

Besides providing the basis for device driver development, the physical memory and raw memory access classes bring the idea of pointers to the Java world and enable the language to use shared memory with other tasks. Note that objects must not be mapped to raw memory, but access is only available with primitives using bit-wise operations.

### 4.3.7 Further Features

#### 4.3.7.1 Synchronisation

In order to avoid priority inversion, the RTSJ requires the priority inheritance protocol to be present for all synchronized objects. Further protocols, especially priority ceiling emulation, are explicitly allowed.

Wait-free queues enable synchronisation without locking. This way, communication between heap-using and no-heap threads is possible, without the latter ever being blocked due to garbage collection that might interrupt the former.

#### 4.3.7.2 High-Resolution Time

With HighResolutionTime and the corresponding subclasses AbsoluteTime, RelativeTime, and RationalTime, the RTSJ also features high-resolution time with accuracy of nanoseconds.

### 4.3.8 Hello Real-Time World

Algorithm 3 on the next page shows a periodic real-time version of “Hello World”.

### 4.4 RTSJ Implementations

RTSJ compatible implementations include:

import javax.realtime.*;
public class HelloPeriodicRTWorld {
    public static void main(String[] args) {
        /* Priority: Minimum real-time priority + 1 */
        int prio =
            PriorityScheduler.instance().getMinPriority() + 1;
        PriorityParameters prioP = new PriorityParameters(prio);

        /* Period: 200ms, 0ns. Start: Immediately */
        RelativeTime period = new RelativeTime(200, 0);
        RelativeTime start = new RelativeTime(0, 0);

        /* Release parameters for the periodic thread: */
        ReleaseParameters perioP = new PeriodicParameters(
            start, period, null, null, null, null);

        /* Create the periodic thread: */
        RealtimeThread rt = new RealtimeThread(prioP, perioP) {
            public void run() {
                int i = 1;
                do {
                    System.out.println(
                        "Hello periodic RT world! Period: " + i);
                    i++;
                } while (waitForNextPeriod() && (i <= 10));
            }
        };

        /* Start the periodic thread: */
        rt.start();
    }
}

Algorithm 3: Hello Periodic Real-Time World
JamaicaVM Aicas’ JamaicaVM implements most of the RTSJ and features an incremental, real-time capable garbage collector. See section 4.4.2.

jRate jRate [18] uses ahead-of-time compilation and is implemented as an extension to GNU’s Java compiler.

4.4.1 Reference Implementation (RI)

TimeSys offers the RTSJ Reference Implementation (RI), which is freely available at their web page [32]. Besides the javax.realtime package necessary to develop RTSJ programs, it includes an RTSJ compliant Java virtual machine that is based on J2ME. It can be run on top of the Linux operating system (see section 4.4.1.1 and chapter A). An external Java compiler converting Java source code into class files is still necessary, with Sun’s J2SE version 1.2 being preferred, but later versions of javac work fine as well usually.

The RI has not been designed with high performance or little memory usage in mind. It thus serves well for experimentation purposes, but it is not meant to be used as a drop-in replacement for commercial products. When executing applications, the Java byte code is always interpreted only, there is no ahead-of-time or just-in-time compilation used.

As mentioned in section 2.2.1, pre-emptive FIFO fixed priority scheduling is used by default.

See chapter B for more details on the RI.

4.4.1.1 Linux

While not being part of the primary scope of this thesis, some topics concerning the real-time capabilities of Linux that are relevant to Jiotto are covered in this section. Further information on “Linux and Real-Time” can be found in chapter A.

Priority Inversion TimeSys [32] offers several versions of their real-time Linux operating system. While there is also a free version with features such as full pre-emptibility and a constant-time scheduler, especially the priority inversion avoidance mechanisms priority inheritance and priority-ceiling emulation protocol are only available in TimeSys’ commercial products.

Numbers of Priorities TimeSys’ RTSJ RI offers the real-time priorities 11 through 265. In an RTSJ application, thread priorities are typically set in relation to the values returned by the methods getMinPriority() and getMaxPriority() of the class PriorityScheduler.8

8Dibble points out that these getter methods are the preferred way of getting the minimum and maximum values and that the constants PriorityScheduler.MIN_PRIORITY and MAX_PRIORITY are “legacy features”, which should not be used. See http://cio.nist.gov/end/emaildir/lists/rtj-discuss/msg00098.html.

Typical usage of these getter methods is as follows:
4.4. RTSJ IMPLEMENTATIONS

As the scheduling of the threads is performed by the Linux kernel itself and the priorities are set by standard POSIX calls, the OS has to be able to take account of this number of priorities. A default Linux 2.4 supports real-time priorities of values up to and including 99. All higher values are treated as usual non-real-time threads. This means that, when running on a default Linux kernel, a `RealtimeThread` using the RI's minimum priority (or any value up to and including 99) is preferred over higher priority threads, including those using the RI's maximum priority.

This problem can be solved by using either the TimeSys Linux kernel, which supports up to 512 priorities, or a patched Linux kernel in order to have it accept a wider priority range. With the patch shown in section A.3 applied to a default Linux, the kernel maps all priorities from 100 through 265 to the priority 99.

4.4.1.2 Jiotto

`Jiotto` has been run successfully on both TimeSys Linux/GPL (which is based on Linux 2.4.7) and a vanilla Linux 2.4 (for example 2.4.24) with the priorities patch applied.

4.4.2 JamaicaVM

One of the key characteristics of Aicas’ `JamaicaVM`\cite{14} is its pre-emptible, deterministic, hard real-time capable garbage collector. Every time new memory is allocated, the GC performs only a few machine instructions and collects 32 bytes of memory (though other configurations are possible too). The advantage of this GC is that it offers real-time behaviour for all threads; a strict separation into non-real-time and real-time code is thus not inherently necessary. Note that, while heap is available for and usable by all threads, scoped and immortal memory are supported as well. On the one hand, the VM can save time by not having to perform some run-time checks such as that `NoHeapRealtimeThread`s are not allowed to access heap memory in any way. On the other hand, the garbage collector poses additional overhead when creating objects.

The JamaicaVM software package offers an application’s Java byte code to be interpreted directly using the command `jamaicavm`, or be ahead-of-time compiled to native code using the command `jamaica`. The latter offers numerous optimisation-related options and should most probably be used for performance and memory consumption analysis as well as a product’s final deployment.

The current version is JamaicaVM 2.2, which supports most of the features specified in the RTSJ.

```
((PriorityScheduler) Scheduler.getDefaultScheduler()).getMinPriority()
```
4.4.2.1 Jiotto

Due to some minor bugs concerning changing a thread’s period at run-time, it has not been possible yet to correctly execute a complete run of Jiotto’s more complex applications using the JamaicaVM. Workarounds would for example include using AsyncEventHandlers, which meant changing some of the CT’s core code of how to implement periodicity. The JamaicaVM should be fixed shortly after this thesis document has been finished, so there is no need to adapt the Jiotto code accordingly.

4.5 Ravenscar-Java (RJ)

While the RTSJ is a very complete real-time specification and offers a lot of flexibility, it is also quite complex. This complexity not only introduces the risk of implementation errors, but also makes corresponding applications difficult to analyse.

The high integrity profile Ravenscar-Java (RJ) [22] is based on the RTSJ, but – following the philosophy of the Ravenscar Profile for Ada – it removes features that are hard to perform timing and functional analyses on. Besides generally offering more efficiency at run-time, the resulting subset is aimed to allow a predictable computational model, making programs more analysable and thus more dependable. This is needed for systems, where failure can cause loss of life or other significant damage.

The most important characteristics of Ravenscar-Java are:

**Execution Phases** RJ applications are divided into the two execution phases *initialisation phase* and *mission phase*. During the initialisation phase, all non-time-critical actions such as allocating scoped memory objects and real-time threads from the immortal memory are performed. As soon as the application is set up correctly, the mission phase begins and the program is executed normally.

**Memory Management** Garbage collection is not supported by a RJ runtime environment. Thus, if the usual heap memory is supported at all by the RJ implementation, its usage equals to the one of RTSJ’s immortal memory. While the RTSJ defines several types of scoped memory areas, RJ only allows linear time scoped memory areas (LTMemory), and their usage is restricted to not being nested or shared between Schedulable objects.

**Scheduling and Threading Model** A RJ implementation needs to support pre-emptive fixed priority scheduling with at least 28 distinct real-time priorities. Schedulability analysis can be performed pre-run-time, thus overrun and deadline-miss handlers as well as feasibility checks are not required. Threads shall not be created by extending RealtimeThread,
and usage of AsyncEventHandler is disallowed too. Instead, the two new classes PeriodicThread (based on NoHeapRealtimeThread) and SporadicEventHandler (based on BoundAsyncEventHandler) are to be used.

**Synchronisation** In order to guard all synchronized operations, the priority ceiling protocol is required.

**Asynchronous Transfer of Control** ATC is very hard if not impossible to analyse in advance, so this RTSJ feature is completely disallowed.

Several basic RTSJ or standard Java classes such as java.lang.Thread, RealtimeThread and NoHeapRealtimeThread are redefined to offer limited functionality only. Dynamic class loading in the mission phase is to be avoided.

Generally, RJ applications are valid RTSJ programs, though there might be some aspects which have to be considered when executing a RJ program in a default RTSJ runtime environment: For example, RTSJ does not require the priority ceiling protocol to be supported. Furthermore, RJ defines some new classes such as PeriodicThread, which are probably not available on a default RTSJ implementation.

### 4.5.1 Jiotto and Ravenscar-Java

#### 4.5.1.1 Where Jiotto uses RJ

RJ's basic characteristics overlap with the ideas of Giotto to provide distinct timing and functionality.

Jiotto uses a predefined set of modes with associated tasks that are executed periodically. According to environmental and temporal changes, the system may switch deterministically from one mode to another. With all required objects such as task threads, ports and similar known in advance, RJ's applications structure with the two execution phases can be implemented easily.

A lot of RJ rules and ideas can be found being applied to Jiotto, among others are:

- java.lang.Thread is not used directly at all. All thread objects are based on NoHeapRealtimeThread, so no garbage collection mechanism may interrupt the application at any given time during the mission phase.
- No schedulable objects are created during the mission phase.
- There is no ATC or any other thread aborting mechanism used.
- Correct timing is achieved through the control thread's periodicity only (no timers are used).
- Recursive method invocations are avoided.
There are comments for most every non-trivial code. Classes, methods and fields are documented using both Javadoc compatible and usual in-line comments.

Code blocks are explicitly surrounded with \{ and \}, even in case where the block consists of only a single statement.

Variables and object references are initialised in the constructors.

### 4.5.1.2 Jiotto is not RJ compatible

While direct usage of `NoHeapRealtimeThread` is not explicitly disallowed, RJ's `PeriodicThread` is the preferred way of working with threads. Nevertheless, the sole usage of `PeriodicThread` hinders RJ's applicability to an implementation of the Giotto paradigm due to the following reasoning:

When implementing Giotto tasks as independent threads as is done in Jiotto, they are executed periodically within one mode. However, in general, when switching to another mode, the set of tasks changes. In cases where tasks of the previous mode are not part of the target mode, these previously active tasks must not be executed anymore until the system enters the old mode again – and temporarily “stopping” a periodically executing thread is not possible. Technically, a solution could be to implement the task threads by assigning distinct thread objects to each mode. One task would thus be implemented by different thread objects in different modes. Every such thread could use an active field, which had to be `true` in order for the thread to perform its function, whereas if it were `false`, the thread would do nothing and thus indirectly have `waitForNextPeriod()` be invoked immediately. However, with the Giotto semantics, there arise the following problems: Giotto allows one task to be present in different modes. With the implementation mentioned above, different thread objects would have to implement the same task function. When switching to a target mode that holds at least one task that can also be found in the previous mode, the task’s state (i.e. at least its private ports) would have to be transferred explicitly from the “old” thread to the “current” one, posing an additional run-time overhead. Similarly, all threads would be running all the time, which means that threads, which’s Giotto tasks are not active, would be scheduled nonetheless. This not only, again, posed additional run-time overhead, but made fixed priority based scheduling at least complicated to be performed correctly.

With distinct tasks being associated to one thread each, RJ’s demand to not allow changing scheduling characteristics during the mission phase cannot be followed, as a task thread’s period and priority might have to be changed due to a mode switch. Furthermore, due to possible mode switches again, the

---

Note that RJ's `PeriodicThread` performs the call to `waitForNextPeriod()` transparently. Similarly, Jiotto's `Task` transparently introduces periodicity: It implicitly invokes a `wait()` after executing the `TaskFunction.f()` method.
control thread’s period has to be changed too in order to adapt the intervals of execution of the Giotto Micro Steps.

In addition, in order to simplify and clarify thread handling (see reasoning above), Jiotto invokes `wait()` at the end of a task function.\(^\text{10}\)

Similarly, although loop control is constrained by the RJ as for example `continue` and `break` statements are disallowed, Jiotto makes use of these standard Java features.

\(^{10}\text{Note that usage of unbound `sleep()` and `wait()` as well as `notify()` and `notifyAll()` is disallowed by RJ.}\)
Chapter 5

Jiotto

Jiotto is a Java framework implementing the Giotto semantics. It is based on the Real-Time Specification for Java (see section 4.3) and thus offers the deterministic behaviour that is needed within a real-time environment. Jiotto is largely compatible with Giotto\(^1\), which makes it possible to use the same application models for both Giotto and Jiotto. Such a model represents a software's functionality and timing from an external point of view and thus is the key characteristic of an overall application. Due to the use of Java and the RTSJ, it is not only possible to develop a new Jiotto system in “Java only”. A corresponding Giotto system’s possibly already existing functional code, for example implemented in C or similar languages, can be integrated too via the Java Native Interface (JNI).

Jiotto is available as a complete Java package, which can be used in combination with any RTSJ compliant Java virtual machine. The overall structure, specifics concerning the class hierarchy as well as the development process leading to the final package are explained in detail in this chapter.

5.1 Overview

One of the integral parts of a Jiotto system is the control thread (CT). From the user's point of view, it is the most important interface to the jiotto package, as it provides factory methods that create components such as modes or ports.\(^2\) Internally, however, the CT does much more: It performs a lot of sanity checks before the mission phase begins in order to ensure the correctness of the user's application model, and it periodically executes the Giotto Micro Steps.

The only “active” objects implemented as threads are the CT and the tasks: The former directly extends NoHeapRealtimeThread; the latter's class inher-

---

\(^1\) It is currently not possible in Jiotto to perform mode switches while tasks are logically still running.

\(^2\) Jiotto is a package, which enables others to create the final applications according to the Giotto paradigm. In this document, the term “user” denotes anyone “using” the package and thus can most always be thought of as developers of an overall Jiotto system.
5.1. OVERVIEW

```java
public class Initialiser extends RealtimeThread {

    public Initialiser() {
        super(new PriorityParameters(
            ((PriorityScheduler)
                Scheduler.getDefaultScheduler())
                .getMaxPriority()),
        null,
        null,
        ImmortalMemory.instance(),
        null,
        null);
    }
}
```

Algorithm 4: Initialiser

its from the Jiotto class `ReusableNoHeapRealtimeThread`. All other components, especially modes, drivers and ports, are “passive” objects that offer the corresponding functionality on demand. It is thus clear that Jiotto is not subject to any possibly available garbage collection mechanism, as all threads can not access heap in any way but rather must use immortal or scoped memory.\(^3\)

Alongside the overall Jiotto model, the tasks’ and the driver guards’ functions have to be implemented by the user. Jiotto offers interfaces to be used accordingly.

Based on Ravenscar-Java’s ideas, Jiotto also uses two execution phases:

**Initialisation Phase** Any Jiotto application extends the `Initialiser` class, shown in algorithm 4. It is based on Ravenscar-Java’s class `Initializer`.

According to the application model, which the user provides as the body of the `Initialiser`'s `run()` method, all base objects are created and initialised:

- The control thread instance is created.
- Objects of user classes implementing the task functions and driver guards are created.
- Ports, tasks, drivers and modes are created, partially using the “user space objects”.

After the set-up procedure, the initialising thread dies and thus automatically hands over control of execution to the CT.

---

\(^3\)In fact, the implementation of the `jiotto` package itself only uses immortal memory. Of course, scoped memory can be used in user space code though.
MISSION PHASE  Having the system’s highest priority (and all task threads having lower ones), the CT controls the overall application. It not only periodically executes the Giotto Micro Steps (GMS) when necessary and thus assures the application’s correct execution. The CT also handles the task threads’ priorities: While the actual scheduling is performed by the platform beneath\(^4\), the CT sets the threads’ priorities according to the rate monotonic priority assignment scheme. After all “administrative” work has been performed, the CT invokes `waitForNextPeriod()` and blocks until the next execution of the GMS.

Usually, the next highest priority threads are the tasks, which automatically execute their functions. The execution order is based on the thread priorities.

As soon as the CT’s period is over, it becomes runnable again and runs through the GMS, possibly executing drivers, performing mode switches or restarting task threads.

5.2 Class Hierarchy

5.2.1 ControlThreadSingleton

Among others, the control thread (CT) offers a similar functionality as Giotto’s “E machine” does. It serves as the application’s upmost instance, periodically executes, on a “time-triggered” basis, the GMS and updates the overall program configuration accordingly: Mode switches are performed, drivers are invoked to update port values, and so on. Furthermore, by offering factory and other control methods, the CT serves as a mediator between the user application and the jiotto package.

5.2.1.1 General Implementation Details

The CT always runs with the system’s maximum priority and is implemented as the class `ControlThreadSingleton`, which directly extends RTSJ’s `NoHeapRealtimeThread`.

As indicated by the class’ name, it is based on the Singleton design pattern (see section C.2.1), which matches Jiotto’s needs perfectly: There shall be of course only one instance controlling the program flow, and the user shall not be able to create any further such objects.

5.2.1.2 User Application’s Object Creation

Ports  When creating a port object such as an input port, the application runs code similar to the following one:

\(^4\)In the reference setup with TimeSys’ RI, the Linux operating system performs the scheduling.
ControlThreadSingleton ct =
    ControlThreadSingleton.getReference();

/* Create an InPort with ID "0", the initial
 * integer value "42" and a description: */
InPort i0 = (InPort)
    ct.createPort(InPort.class, 0, 42, "InPort-0");

As usual, the CT serves as an interface between the user application and the
jiotto package. Nevertheless, it does not perform the object creation itself,
but rather delegates this task to PortFactorySingleton using its method
create(Class, int, String).

This multi-level approach has several advantages: If an additional value-
type for ports such as boolean was introduced, nothing had to be adapted
in PortFactorySingleton concerning the port creation process. (The ini-
tialisation of the new port object with the correct init-value is performed in
ControlThreadSingleton.createPort() using Port’s overloaded meth-
ods Port.setInitValue(). Adding a new overloaded createPort() method
to ControlThreadSingleton is the only change necessary concerning the
port creation process.)

Besides being responsible for creating the ports, PortFactorySingleton
also keeps track of them. With the createPort() methods and the port con-
structors all having a package access modifier, and the only publicly available
way of creating ports being via the ControlThreadSingleton instance as
described above, it is guaranteed that the user cannot create any port objects
that the PortFactorySingleton does not know of.

ControlThreadSingleton.createPort() reads the exact port class to
be created as the method’s first parameter. While the new object returned is of
this specific port type, of course, the method returns it as a general Port
object. While this introduces the need of a typecast to be performed by the user,
one overloaded method was preferred over several type specific ones (such as
createInPort() or createOutPort()): Every port has a specific type of
values associated, which is determined by Jiotto via the initial value provided
by the user (see section 5.2.2.1). The Java language’s overloading mechanism
guarantees the correct createPort() method to be invoked automatically.
Assuming \( n \) different types of ports and \( m \) distinct types of port values, us-
ing dedicated creation methods for each of the port types would result in \( n \cdot m \)
creation methods in total to be present in ControlThreadSingleton (due to
the necessary overloading of the methods concerning the ports’ initial values).
With this complex code management in mind, the additional typecast, which is
performed only once during the initialisation phase, is considered a good trade
off.
Drivers, Tasks and Modes  The basic idea of the object creation process of drivers, tasks and modes is the same as it is for ports, though the CT does not need to initialise these objects in any way. This makes the corresponding methods such as createMode() public access points that hide the real implementation in the classes DriverFactorySingleton, TaskFactorySingleton and ModeFactorySingleton.

For more information on the factories, see section 5.2.7.

5.2.1.3  run() and go()

As is common with threads, the execution flow is controlled by the CT’s run() method, which basically does nothing but to invoke the GMS and afterwards block again by calling waitForNextPeriod(). The CT is then woken up by the RTSJ system at the exact point in time when the next period is due.

While Java threads are usually started by invoking the public start() method, the CT has to perform several initialisation steps beforehand and thus offers the method go() to be invoked instead by the user application. This method performs the following functionality:

- Register each mode’s tasks’ output ports as mode ports.
- Invoke doSanityChecks(). This performs many checks that all Giotto and Jiotto rules are obeyed, including the following: Comparison of the numbers of ports, drivers and tasks created vs. specified; correctness of driver transfers’ source and destination ports. (A complete list of checks performed is available in the API documentation.)
- Calculate each mode’s period for the CT’s execution of the GMS. (This ahead-of-time calculation improves performance in the mission phase and can be performed in advance as the mode characteristics do not change during the run-time of the application.)
- Set the system’s start mode; set this mode’s task threads priorities; set the CT’s period correctly accordingly (see section 5.2.1.4).
- Possibly start optional sensors threads, which can be used for simulation purposes (see API documentation for ControlThreadSingleton. addSensorsActuatorsFunction()).
- Kick off the mission phase by invoking the CT’s start() method.

5.2.1.4  setPeriod()

A Giotto program generally consists of several modes, which represent the different fields of activity. Each mode $m$ has a specific period $\pi[m]$ associated,

$^5$Directly invoking ControlThreadSingleton.start() results in a JiottoViolationException to be thrown.
which is the overall time in milliseconds a complete run of one round takes. (The mode’s tasks are executed periodically according to their frequencies in relation to the mode’s period.) The Giotto Micro Steps (see sections 3.4 and 5.2.1.5) are executed every \( \pi[m]/a_{\text{max}}[m] \) time units.

During the mission phase, the only task the CT performs is to execute the GMS repeatedly. The definition of when to run the GMS clearly shows that the exact time period the CT has to wait between the times at which the program configuration is updated, depends on the mode the system currently is in. This means that, due to a mode switch, the CT’s period has to be changeable at run-time.

In Jiotto, every time a mode switch is performed, the method `setPeriod()` is invoked to adapt the CT’s period (and thus the time between the executions of the GMS) to the new mode. Note that, as the characteristics of a mode never change, the time between the GMS executions can be calculated during the initialisation phase already. Invocation of `setPeriod()` thus simply adapts the CT’s period to an already available value (`Mode.correspondingCTPeriod`).

### 5.2.1.5 Giotto Micro Steps (GMS)

The Giotto Micro Steps (GMS) are implemented as distinct methods of the class `ControlThreadSingleton`.

**GMS 1: Update task output and private ports** For all task invocations of which the tasks are logically finished at this point in time (and that are no “dummy-tasks”), it is verified that the tasks have already finished performing their functions by trying `currentTask.checkReady()` (see section 5.2.6 for more information on this method, inherited from class `jiotto.ReusableNoHeapRealtimeThread`). If everything is fine, the global `OutPort` objects are updated by transferring the values of the `TempOutPort` objects into them. This way the output ports are updated at the correct point in time. Note that the user’s implementation does not know anything about a differentiation between `OutPort` and `TempOutPort` objects, as it simply uses `ModePorts` (which is an interface both classes implement).

**GMS 2: Update actuator ports** For all actuator updates that need to be considered at this point in time, the driver guards are evaluated. If they return `true`, the driver functions and thus the updates of the actuator ports are performed according to the transfers defined by the application model.

**GMS 3: Update sensor ports** Sensor ports are defined to be updated nondeterministically by the environment at any given time. Both task drivers and mode drivers may use sensor ports as source ports for their value

---

6The `TempOutPort` ports were used by the user’s task functions to temporarily store the results.
transfers. While the Giotto Micro Steps have to be performed in logically zero time, this is clearly not possible in a real-world implementation, as any kind of work consumes at least some time. In order to guarantee that all drivers read the same sensor values and thus to achieve simulation of zero time, the values of the TempSensePort objects (which are updated by the environment directly) are transferred into the globally available SensePort objects (representing the Giotto sensor ports internally) to be used by later GMS. The actual sensor values may change during the sequential execution of the drivers without any nondeterministic consequences.

GMS 4: Update mode All mode switches that have to be considered at this point in time are evaluated. In case of more than one mode driver evaluating to true, a GiottoViolationException is thrown.\(^7\) If exactly one guard evaluates to true, the mode switch is performed: currentMode is set to the target mode, resetTaskThreadsPriorities() is invoked to set the task threads priorities according to the rate-monotonic priority assignment scheme, and the CT's period is updated by invoking setPeriod().

GMS 5: Update mode port If a mode switch has been performed in GMS 4, the corresponding driver's transfers are performed and thus the target mode's mode ports are updated.

GMS 6: Update mode time If a mode switch has been performed in GMS 4, the mode time is reset to 0.\(^8\)

GMS 7: Update task input ports For all non-dummy-tasks that will start another round of execution at this point in time, the task driver guards are evaluated and the drivers are possibly executed accordingly.

GMS 8: Update active tasks For all non-dummy-tasks that will start another round of execution at this point in time, the tasks' go() method is invoked (see section 5.2.6 for more information on this method, inherited from class jiotto.ReusableNoHeapRealtimeThread).

GMS 9: Advance time In Jiotto, the CT's period is added to the mode time to reflect the time that passes when the CT blocks, starting at the end of GMS 9 until the next execution of GMS 1.

\(^7\)The situation of several mode drivers evaluating to true was not deterministic, as the choice of the target mode to be entered depended on the order of driver execution only. Such a situation is an error in the application model or the corresponding implementation, both of which the user is responsible for and thus has to solve problem.

\(^8\)This is based on the fact that mode switches are currently not allowed in Jiotto while tasks are logically running. In case this was allowed, the mode time was set to 0 only if no active tasks were "transferred" to the target mode, otherwise the mode time was set to a specific value to place the system as near as possible to the end of a round of the target mode (see Henzinger et al. [10]).
A simple performance improvement of the GMS' implementation in Jiotto is accomplished by having GMS mark the tasks that are logically finished and have to be dealt with at the given point in time. In order to know whether or not to perform on them, succeeding micro steps then can evaluate the tasks' states by simply checking the logicallyFinished fields. This way, the actual calculations to determine whether a task has finished its run, have to be performed only once as part of GMS.

5.2.1.6 Task Threads Priorities

ControlThreadSingleton.MIN_TASK_PRIORITY and MAX_TASK_PRIORITY are set to the platform's minimum real-time priority plus one and the platform's maximum real-time priority minus one respectively. With TimeSys' RTSJ reference implementation and TimeSys' Linux kernel, this results in the priorities 12 and 264. By default, the lowest real-time priority is reserved for sensors threads (see section 5.2.10), the highest priority is used by the CT; all priorities in-between may be used to accomplish the scheduling of the task threads based on the rate-monotonic priority assignment. (Though the priority of a given task may change from one mode to another, the scheduling algorithm used in Jiotto is considered to be static, as defined in Liu and Layland [25]. The reason for this is that all tasks, which are not part of the current mode, are “inactive” and can be considered non-existent. The only set of tasks that are relevant to the scheduler are the ones of the current mode, and within a mode, the execution of these independent tasks is not only periodic, but the priorities are fixed and stay as they are. With the transition into another mode, the system can be considered to be a totally different and independent one with its own new set of tasks – and a distinct fixed priority scheduling.)

As part of GMS 4, the method resetTaskThreadsPriorities() sets the priorities of the task threads of the current mode's task invocations. The exact value equals a task's frequency added to MIN_TASK_PRIORITY - 1. On the TimeSys platform, this results in task frequencies of 253 at most.9

5.2.1.7 Miscellaneous

The CT keeps track of the different numbers of Giotto components to be created; note though that it does not manage the objects themselves.

---

9If higher frequencies were needed, the priority assignment could easily be re-designed while still being RMS compatible.

Note: When running Jiotto with the RTSJ RI on a vanilla Linux kernel, mind that all processes with priorities higher than 99 are treated as non-real-time processes by default. Even with the kernel patch from section A.2 being applied (which enables Jiotto to be executed correctly on this quasi-vanilla kernel), ControlThreadSingleton.MAX_TASK_PRIORITY was still falsely set to 264. Tasks with different levels of importance could run with the same priority effectively, which – even more importantly – could be equal to the CT's one. Depending on which platform Jiotto is executed on, this issue may have to be taken care of manually by adapting MAX_TASK_PRIORITY as well as the CT's priority.
CHAPTER 5. JIOTTO

5.2.2 Ports

The class hierarchy for all ports related classes (except for two exceptions and PortFactorySingleton) is shown in the UML based figure 5.1.

5.2.2.1 Port

The abstract class Port is the central class offering most of the functionality that all different kinds of ports have. Due to Port being an abstract class, it is obvious that only objects of the more specific subclasses such as InPort can be created. Furthermore, as Port is declared with the default package access modifier, the user cannot extend the class on their own.

A port’s exact value type is specified at construction time. The initial value is not passed along as a parameter to the constructor, but is rather set via the overloaded setInitValue() methods (invoked by the CT’s createPort() methods accordingly). This reduces necessary changes to a minimum when adding a new value-type.

There are getter methods for the value type and the port ID as well as public getter and setter methods for the port values.

5.2.2.2 Port Subclasses

Currently, the Port subclasses such as InPort and OutPort do not add any additional functionality to Port but simply extend this class. However, all classes also implement one or more interfaces (see section 5.2.4.4).

Though, of course, the subclasses of Port are public classes, the constructors have the default package access modifier in order to prevent the user from directly creating objects without using the CT’s create() methods. Furthermore, as the classes are declared final, the user cannot extend them in any way.

TempSensePort is used in Jiotto to enable all drivers to access the same sensor values during the execution of the Giotto Micro Steps (see section 5.2.1.5, GMS 3). Jiotto’s usage of TempOutPort guarantees that the output ports are updated at the correct points in time exactly (see section 5.2.1.5, GMS 1). Note that with both “temp” port classes, the port objects used are totally independent from the corresponding ones of the “official” ports. The necessary one-to-one relationship between a “temp” and an “official” port is achieved by giving both ports the same ID, the correct creation of the “temp” objects is performed automatically.

5.2.2.3 Interface ModePort

Mode ports are a mode’s tasks’ output ports. Being based on the same physical memory space, these are just two different names for the same port. While tasks update output ports, all kinds of drivers use mode ports as source and
5.2. CLASS HIERARCHY

Figure 5.1: Jiotto Class Hierarchy: Ports
some drivers even as destination ports. To comply with Giotto's syntax and semantics, the public interface `ModePort` was introduced as an interface implemented by the output port classes.

A more practical reason for this interface is its usage in `TaskFunction.f()`. While the user might think of the task function to use the output ports directly, the actual objects which’s values are changed are of the type `TempOutPort`. This class implements `ModePort` the same way `OutPort` does. The actual separation, which enables the CT to transparently update the real output ports at the right time, is hidden from the user this way.

### 5.2.2.4 Interface `PortMethods`

The interface `PortMethods` is – directly or indirectly – implemented by all kinds of ports. Java guarantees that the methods of the interface are available and implemented. `PortMethods` makes it easier for the user to develop Jiotto based applications, as typecasts in the driver guard functions for `ModePort` objects are not necessary this way.

### 5.2.2.5 Interfaces `DriverDstPort`, `DriverSrcPort` and Extending Ones

As an example, objects that are to be scheduled as destination ports for a task driver must implement the interface `TaskDriverDstPort`. This level of interfaces in the ports’ class hierarchy is used as parameter types of the specific drivers. `DriverDstPort` and `DriverSrcPort` are used by Jiotto internally to handle registering of such ports and objects in a centralised method, regardless of the specific driver type. Also, see section 5.2.4.2.

### 5.2.3 Tasks

#### 5.2.3.1 General Implementation Details

Tasks are implemented as threads and, besides the CT, thus are the only “active” objects in Jiotto.

Distinct tasks can be active in several different modes; the exact assignments are defined by the modes’ sets of task invocations that also specify which task drivers are to be used.\(^{10}\) Scheduling of the task threads is based on the pre-emptive fixed priority FIFO scheme, by default. A task thread is assigned its priority by the CT according to the rate-monotonic priority assignment scheme (see section 5.2.1.6).

Using threads for the tasks offers the following advantages:

- The thread scheduling can be performed by the real-time platform, Jiotto itself does not have to do anything but to assign the correct priorities (in

\(^{10}\)Note that while a task driver may be assigned to one task only, a given task may use several task drivers in different modes. It is thus possible to change the flow of information while leaving the manipulation of data unchanged.
5.2. CLASS HIERARCHY

The class hierarchy concerning the tasks related classes is shown in figure 5.2, the classes are described in detail in the following sections. (Note that Task is shown as being composed of a TaskFunctionImplementingClass here. While there are even further classes composing Task, the one implementing the interface TaskFunction is – structurally – the most important one and thus mentioned explicitly in the class diagram.)

5.2.3.2 Interface TaskFunction

As part of the application development process with Jiotto, the user has to implement the tasks' functions by creating custom classes that implement the TaskFunction interface. This interface consists of the declaration of the task function method \( f(InPort[], ModePort[], PrivatePort[]) \) only. The

---

11 Note that Jiotto not offering this possibility is not necessarily considered to be a disadvantage, as mode switches with logically running tasks can make an application model quite complicated as well as the application's exact behaviour hard to determine.
user does not have to take care of topics such as periodicity, as this is offered by Jiotto automatically.

5.2.3.3 Task

The public final class `Task` extends Jiotto’s `ReusableNoHeapRealtimeThread` (see section 5.2.6). Due to the constructor having the package access modifier assigned (and in combination with the class being `final`), it is guaranteed that the user can create task objects only via the CT.

`Task` is (indirectly) based on `NoHeapRealtimeThread` and has the immortal memory set as the default memory area for any allocations. This of course includes any allocations made by user code implementing `TaskFunction.f()`. Thus, to prevent memory leaks, scoped memory objects (for example implemented as object attributes) have to be used for any temporary objects created by the user code.

The three methods `addInPort(InPort)`, `addOutPort(OutPort)` as well as `addPrivatePort(PrivatePort)` are invoked by the user application to add the corresponding port to the task object.

`logic()` implements the super class’ abstract method and, every time it is invoked, simply executes the task function method (which is a task’s only variable code block).

Another important method is `setTaskPriority()`, which is invoked by the CT every time a new mode is entered where this task is active – including at the very beginning when entering the start mode. Mind to not use `java.lang.Thread.setPeriod()`, as this method is not aware of any real-
time priorities. Due to being declared `public final`, it cannot be hidden by Jiotto directly.

5.2.3.4 Periodicity of Tasks

Tasks are executed periodically within one mode, which made the use of RTSJ’s support for periodic threads an obvious choice. However, a Giotto implementation has the “problem” that a different set of tasks has to be active for each mode. With the system being in a specific mode, all of the application’s tasks, which are not invoked in this one but only in other modes, must not run. Thus, there is the need to be able to control the execution of tasks: They have to be “stoppable” and “re-startable” again. As a Java thread must not be re-started again once its flow of execution has reached the end of the thread’s `run()` method, it is not directly possible to re-use a normal thread. The solution is Jiotto’s abstract class `ReusableNoHeapRealtimeThread` (see section 5.2.6),

12Note that by default a task thread object is created having assigned the global minimum task priority available as `ControlThreadSingleton.MIN_TASK_PRIORITY`. In combination with `setTaskPriority()`, not allowing to raise the priority to the platform’s maximum, it is thus guaranteed that the CT always has a higher priority than any task thread does.
which offers this functionality.\footnote{Note that the re-usability works transparently in Jiotto: The user only has to implement the \texttt{TaskFunction} interface and create a task accordingly.}

### 5.2.3.5 Dummy Tasks

A dummy task can be created by passing \texttt{null} over to the CT when creating a task (instead of providing a reference to a \texttt{TaskFunction} implementing object). Such a task only serves the purpose of being able to add certain output ports to a mode with no other intent than to have them registered as mode ports. While these ports are not updated by any tasks in the corresponding mode, having them available as mode ports might still be necessary in case they are for example used by mode driver guards or mode driver functions.

### 5.2.4 Drivers

The class hierarchy for the most important drivers related classes (including the necessary \texttt{Port} interfaces) is shown in figure 5.3 on the next page. (Note that the specific \texttt{Driver} subclasses are shown as being composed of corresponding driver guards implementing classes here. While there are even further classes composing the specific drivers, those implementing the driver guard interfaces are – structurally – the most important ones and thus mentioned explicitly in the class diagram.)

Note that there is no driver function to be implemented by the user: In practice, a driver simply transfers the values – without changing them – into the target ports. (To comply with Giotto’s formal definition, the mathematical function performed is the “identity” function \( f(x) = x, \forall x \in X. \)) Thus the user only has to set up the correct sources and destinations, the driver’s “function” then is to perform the transfers accordingly without changing the values.\footnote{In Jiotto, mode drivers are allowed to have no transfers at all, resulting in a possibly performed mode switch without any mode ports changed. All other driver types need at least one transfer scheduled.}

#### 5.2.4.1 Driver

The public abstract class \texttt{Driver} is the base class for all kinds of drivers. Besides a reference to the driver’s guard object, \texttt{Driver} is also composed of the two-dimensional private array \texttt{Object[][] transfers}, created by the constructor as \texttt{new Object[numberOfTransfers][2]}. All transfers that a specific driver has to perform when being invoked are registered in this array by the user application via the \texttt{Driver} subclasses’ \texttt{transfer()} methods.

While these overloaded public methods act as an interface between a Jiotto driver and the user application, they simply pass on the request to the corresponding \texttt{Driver.transferGeneric()} methods:

\begin{verbatim}
transferGeneric(DriverSrcPort, DriverDstPort) sets the references
\end{verbatim}
Figure 5.3: Jiotto Class Hierarchy: Drivers
5.2. CLASS HIERARCHY

to the two parameter objects in the transfers array directly. As it is also possible to not only transfer from one port to another, but also to transfer constant values of types such as int into driver destination ports, there are methods such as transferGeneric(int, DriverDstPort).\(^\text{15}\) As the transfers array holds Object-references only and does not accept primitives to be stored in it directly, the transferGeneric() methods with a “primitive”-parameter create an object of the according wrapper type – for example Integer – and invoke the method transferGeneric(Object, DriverDstPort) to eventually add the transfer to be performed by the driver. 

evaluateGuard() is invoked by the CT and returns either true or false, according to the driver guard’s result. Depending on the return value, the CT invokes – or does not invoke – performFunction() that transfers the values, especially using Port’s getter and setter methods such as getValueInt().

5.2.4.2 Driver Subclasses

The Driver subclasses ActuatorDriver, ModeDriver and TaskDriver are final, the constructors have the package access modifier assigned and simply invoke the super constructor. The overloaded transfer() methods serve the purpose of accepting an explicit type of ports only and simply invoke the more general Driver.transferGeneric() methods. Therefore, the user for example only can pass along objects of type TaskDriverDstPort when adding a transfer to a TaskDriver.

5.2.4.3 Interface DriverGuard and Extending Interfaces

The interface DriverGuard only serves to classify more specific interfaces such as TaskDriverGuard as general driver guards.

Task and mode driver guards evaluate based on mode and sensor ports. The extending interfaces TaskDriverGuard and ModeDriverGuard therefore offer the method evaluate(ModePort[], SensePort[]). Nevertheless, an ActuatorDriverGuard may only access mode ports, thus the corresponding evaluate method has ModePort[] as its sole parameter.

5.2.4.4 Ports Related Interfaces

The ports related interfaces are implemented by specific Port subclasses according to which port type is valid for the different drivers. Due to Java’s strong type binding, this both enables Jiotto to catch programming errors at compile time already and makes application development easier for the user, as most

\(^\text{15}\)The ability to transfer (non-port) constant values is not explicitly allowed by the formal definition in Henzinger et al. [10]. Nevertheless, the paper mentions this usage elsewhere. Being a very useful option, it is included in Jiotto. (Note that, even when sticking to the formal definition exactly, the same functionality could be achieved by creating distinct port objects with the desired values as the port’s initial values, which were not allowed to ever be changed.)
IDE’s show falsely used types during the programming process already. Furthermore, the development of Jiotto itself is less error-prone as well.

See figure 5.1 on page 43 for the complete ports class hierarchy and Henzinger et al. [10] for the formal definition.

5.2.5 Modes

The class hierarchy for Mode is shown in figure 5.4, again indicating the most important classes composing Mode. (Note: Though the hierarchy seems to be quite simple, Mode is one of the largest classes in Jiotto.)

5.2.5.1 Mode

While Mode is a public class, the constructor has the package access modifier assigned. In Jiotto, a mode is a “passive” object simply holding a specific type of information, which is periodically used by the CT to correctly execute the overall application.

A crucial responsibility of each Mode object is to keep track of the composing classes by using the arrays taskInvocations, actuatorUpdates and modeSwitches.

Design: 5.4.5

The three methods addActuatorUpdate(), addModeSwitch() as well as addTaskInvocation() add objects of the corresponding types to these arrays.\(^\text{16}\)\(^\text{17}\) The latter also ensures that the task threads of task invocations with different frequencies (which have different “deadlines” and thus are differently important) get distinct priorities assigned by using the rate-monotonic priority assignment scheme. See section 5.2.1.6 and the API documentation for Mode.addTaskInvocation() for more details on how the priorities are calculated.

As indicated in section 5.2.1.4, Mode.correspondingCTPeriod holds the mode-specific period for the CT in which to perform the Giotto Micro Steps. The actual value is the return value of calculateCTPeriod(). This method

\(^{16}\)A mode must have set at least one actuator update, mode switch or task invocation. While it is not allowed for all three sets to be empty in one mode, it is possible to create modes holding objects of one type only. This means that, for example, modes may lack any actuator updates and task invocations and thus serve as intermediary modes that perform mode switches only.

\(^{17}\)Adding a dummy task is possible by using addTaskInvocationDummy().
5.2. CLASS HIERARCHY

is invoked by ControlThreadSingleton.go() as one of the last steps of the initialisation phase. Furthermore, it makes sure that no mode switches with logically running tasks are scheduled. (This condition is true if for every mode switch all task invocation frequencies are an integer multiple of the corresponding mode switch frequency.)

ControlThreadSingleton.go() also invokes registerModePorts() as one of the last steps of the initialisation phase. This method adds every task's output ports to the mode's set of modePorts. In combination with the interface ModePort, this enables the correct implementation of the Giotto semantics.

Besides all this, Mode holds many getter and setter methods as well as further methods performing several consistency checks during the initialisation phase.

5.2.5.2 ActuatorUpdate

The ActuatorUpdate class is implemented as defined by Giotto: It holds a reference to an actuator driver and a given frequency at which it possibly is to be invoked, depending on the result of the driver guard.

5.2.5.3 ModeSwitch

ModeSwitch is composed of a target mode, a mode driver and the corresponding frequency at which to possibly perform the mode switch.

5.2.5.4 TaskInvocation

TaskInvocation specifies a task, a task driver and a frequency. Furthermore, the class holds the status information logicallyFinished, which is used to improve performance of the GMS invocations.

5.2.6 ReusableNoHeapRealtimeThread

ReusableNoHeapRealtimeThread is an integral part of the overall jiotto package, as it makes it possible for task threads to repeatedly perform the corresponding task function in a thread-safe way. In contrast to a usual periodic thread of RTSJ, this is done on demand rather than automatically. The basic structure is shown in algorithm 5 on the next page. \[Design: 5.4.4\]

5.2.6.1 logic()

While the run() method handles most of the re-usability issues, the abstract method logic() must be implemented in extending subclasses to hold the functionality that is to be run when being requested.

18Note that a task does not have a specific driver associated all the time: Due to mode switches, the driver updating a given task's input ports may change. However, mind that a task driver can be assigned to one task only.
public abstract class ReusableNoHeapRealtimeThread extends NoHeapRealtimeThread {

    private volatile boolean keepAlive = true;
    private volatile boolean runIsFinished = false;

    public abstract void logic(); /* The payload */

    public final void run() {
        while (keepAlive) {

            /* Perform the payload */
            logic();

            /* Block until being woken up by go() or
             * quit if end() has been invoked */
            try {
                synchronized (this) {
                    runIsFinished = true;

                    while (runIsFinished) {
                        wait();
                    }
                    if (!keepAlive) {
                        return; /* end() has been invoked */
                    }
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    public final void go() throws ReusableThreadStillWorkingException {
        if (runIsFinished && isAlive()) {
            /* Wake up the thread for another round */
            synchronized (this) {
                runIsFinished = false;
                notifyAll();
            }
        } else if (isAlive()) {
            throw new ReusableThreadStillWorkingException(
                    this + " is still performing its logic!");
        } else {
            if (!keepAlive) {
                start(); /* First time only */
            } else {
                throw new IllegalThreadStateException(
                        "Can not revive dead thread " + this);
            }
        }
    }
}

Algorithm 5: Essence of ReusableNoHeapRealtimeThread
5.2. CLASS HIERARCHY

5.2.6.2 run()

run() is crucial to the class’ functionality in that it manages re-usability. The basic idea is to run logic() from within a loop and then block using wait() afterwards. If the thread is woken up due to invocation of end(), keepAlive is set to false and the thread stops once and for all, else it invokes logic() another time and blocks again.

Note that wait() is invoked as long as runIsFinished is true. As this field is set to false only if go() is invoked, the re-usable thread is guaranteed to block until it is explicitly requested to run logic() another time.

As with usual threads, run() must not be invoked directly, of course. Starting the thread can be done by invoking either start() or go().

5.2.6.3 go()

go() is invoked every time the logic() shall be run again. The condition to be met for this case is that both runIsFinished is true (i.e. the thread has finished its previous invocation of logic()) and java.lang.Thread.isAlive() returns true (i.e. that the thread has been started already but has not died yet).

If the previous condition is false and in case isAlive() is true, go() throws a ReusableThreadStillWorkingException in order to notify the invoking part of the program that the thread is still running logic() (or at least has been at the time of the condition of the first if clause). Note that runIsFinished is not checked here at all: While it might be intuitive to check for this variable being false (as it has already been checked for being true just before), this would introduce a possible race condition. (See the comments in the implementation for more details on this situation.)

5.2.6.4 checkReady()

The checkReady() method offers a way to check if the thread object has already finished or if it is still running logic(). In case of the latter, a ReusableThreadStillWorkingException is thrown. In Jiotto, this method is invoked as part of GMS 1. If the exception is thrown, the CT knows that the corresponding task thread missed its deadline.

5.2.6.5 end()

d() can be invoked to have the thread die: It sets keepAlive to false and notifies the thread via go().

If end() was to be invoked for some reason, it has to be ensured that there are no open file handles or unfreed memory blocks in the implementation of logic(). However, a Jiotto application will not end a task thread at any time.
5.2.6.6 ReusableThreadStillWorkingException

The ReusableThreadStillWorkingException is thrown by the methods go() and checkReady(), indicating that the thread is still running logic(). In contrast to all other exceptions of the jiotto package, this one is not a run-time exception but a checked exception extending java.lang.Exception directly. The invoking part of the program thus must deal with this situation.

Note that, due to Giotto’s nature of time-triggered predictability that Jiotto clearly inherits, and when taking the worst-case execution times of the (independent) tasks into account, a deadline-miss should never occur. Thus, Jiotto’s CT should never have to catch the exception.

5.2.7 Factories

For each of the Giotto components (which are drivers, modes, ports and tasks), there is a -FactorySingleton class constructing the corresponding objects and keeping track of them. The factories are the only classes in the Jiotto system that have direct access to all objects created of a specific type (whereas for example modes only hold references to those OutPort objects that are registered within the mode as mode ports).

5.2.7.1 The Factory Method Design Pattern

One of the characteristics of the full Factory Method design pattern (see section C.2.2) is the idea of using a dedicated method that takes care of object creation, instead of having to perform this task directly using something like new Object(). While the overall object creation, initialisation and object usage process implemented in Jiotto does not completely qualify as a full Factory Method, crucial ideas of this pattern are used, as described below.

The initially mentioned -FactorySingleton classes all offer a create() method each, which return a new object of the type needed. While the object itself is as specific as possible, the type returned by these methods is of a more general one, usually: DriverFactorySingleton.create() creates TaskDriver objects, among others, but returns these objects being up-casted to the super type Driver. Nevertheless, the way the new object is used disqualifies Jiotto as implementing the pattern completely, as the initiating part of the program does care of the exact type returned: In the example given, the user’s Jiotto application probably immediately down-casts the Driver object to a more specific TaskDriver.

Despite this fact, as the creational part of the overall process meets the factory method design pattern requirements, the naming schemes of the corresponding classes include the designation “Factory”.

54
5.2. Exceptions

Jiotto offers several exceptions for different situations, most of which extend java.lang.RuntimeException.\(^\text{19}\)

Some of the most important ones are:

- **GiottoViolationException** Thrown in case something is set up that does not conform to Giotto’s semantics. For example: Output ports are shared between tasks within the same mode.

- **JiottoViolationException** Thrown in case something does not conform to Jiotto regulations. For example: Mode switches are scheduled at points in time where tasks are logically still running.

- **ReusableThreadStillWorkingException** This is the only checked exception in the jiotto package. See section 5.2.6 for details.

For a complete list of all exceptions, see the API documentation.

5.2.9 Logging and Debugging

5.2.9.1 Logging Facilities for Java

With J2SDK 1.4, the Logging API\(^\text{20}\) has been introduced to Java. It offers a very flexible way to include log messages with different levels, the logging process can be controlled via an external configuration file, and more.

Log4j\(^\text{21}\) is available as a Java package and is developed as part of the Apache project. Its basic ideas and its usage are similar to Sun’s official approach, yet Log4j is often considered to be more powerful.

While these logging facilities both offer a lot of useful options, they can by default not be used for Jiotto when being run on top of the TimeSys platform: The RTSJ RI is based on Sun’s J2ME, version 1.2, and thus lacks some packages needed for the above mentioned projects.

Debugging and logging messages in Jiotto are available yet. They are turned on and off using package wide accessible static final boolean fields. While this approach has the disadvantage of having to recompile the jiotto package in order to change the logging behaviour, it is still easy to use and performs well, as if such a variable is set to false, Sun’s Java compiler javac completely removes any code that is not to be run. If debugging is disabled, the application thus suffers from no performance penalties at all.

---

\(^{19}\)The decision to hardly use checked exceptions in the jiotto package is based on the idea that the programmer often “wants to get work done” and deal with error handling some other time. Extensive use of checked exceptions often leads to (most probably bad) code such as catch (SomeException e) {}, with the consequence that such exceptions never show up and may be completely lost. Also, see introductory words of Eckel [6] on this topic.

\(^{20}\)http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/

\(^{21}\)http://logging.apache.org/log4j/
CHAPTER 5. JIOTTO

5.2.9.2 Debug

The jiotto package is divided into several sections, such as CT, MODES or BASE. Each class’ section is stated at the very beginning of the source file. Debugging messages can be enabled all at once or for different sections each (independently of the others). The class Debug holds several static final boolean fields. If set to true, logging output is turned on for the corresponding section.

The corresponding debugging data is spread throughout the Jiotto code as System.err.print() messages, surrounded by if clauses. The following exemplary code fragment is based on the CT’s doSanityChecks() method:

```java
if (Debug.CT) {
    System.err.println(
        "DEBUG CT:\tdoSanityChecks():\t" + "InPorts must not be shared among tasks ... ");
}
/* Perform check. If everything is
  * fine, print "ok" message similarly. */
```

5.2.10 Simulations

The development process of control systems most always includes simulations of the application and its behaviour. While the program itself may be fully implemented already, the testing often takes place in a completely simulated environment with simulated sensors and actuators.

The following sections present the Jiotto classes offering simulation functionality. The sample applications in chapter 6 use this approach to simulate the physical elevator to be controlled.

5.2.10.1 Interface SensorsActuatorsFunction

The method logic(TempSensePort[], ActuatorPort[]) is declared in the interface SensorsActuatorsFunction. It can be used to implement the logic of a minimum-priority thread that simulates changes in the environment by updating sensor ports, possibly based on values of actuator ports.

ControlThreadSingleton.addSensorsActuatorsFunction() can be used to have one or more such simulation threads be added to the system and automatically started during the initialisation phase.

5.2.10.2 SensorsThread

SensorsThread extends NoHeapRealtimeThread with immortal memory set as the default memory area and the lowest real-time priority available in the system. Its run() method simply invokes SensorsActuatorsFunction.
5.2.11 Utilities

Util provides the methods \texttt{lcm(long, long)} and \texttt{gcd(long, long)}, which take two \texttt{long} arguments and return the least common multiple and greatest common divisor respectively. Note that \texttt{gcd()} is explicitly implemented non-recursively.

22

5.2.12 Discussion of Priority Inversion

The Jiotto implementation itself does hardly use any shared resources and is thus not vulnerable to priority inversion. While user-space code such as a task’s function could be programmed using shared resources, Jiotto is not intended to be used this way. All examples presented in this document use completely independent task threads and other code, which is not vulnerable to priority inversion or deadlocks, as no two distinct threads use a shared resource with both of them having write access to it. Furthermore, usage of \texttt{synchronized} is limited to two code blocks in \texttt{ReusableNoHeapRealtimeThread} only, with the current jiotto implementation guaranteeing that the corresponding methods are never invoked in a problematic way.

5.3 A Simple Jiotto Application

This section covers a simple Jiotto application to present the basic usage of the jiotto package.

5.3.1 The Application Model

The idea is to increment a value ten times in mode \textit{m}_0, switch to \textit{m}_1 and decrement five times, after which the application switches back to \textit{m}_0 again. The application's model is shown in figures 5.5, 5.6 and 5.7.

Both modes have the same period of 400 ms and hold one task each. While the task function of \textit{t}_0 increments the value provided via \textit{i}_0 by 1 and makes the output available via \textit{o}_0, \textit{t}_1’s task function decrements the value of \textit{i}_1 and

22Using recursions is not an option when working with time- and space-critical applications: For example, when using RTSJ’s immortal memory, recursions might easily consume a considerable amount of (irretrievable) memory.

23Examples for shared resources that are updated by one thread and read by another are the output and sensor ports. Mind that the sensor ports are updated by the (in the elevator examples of chapter 6 simulated) environment at any time, with the Jiotto system simply reading the most current value available. Concerning the output ports, GMS 1 performs a check to ensure that the task thread has already finished performing its function, before the values of the \texttt{TempOutPort} objects are transferred to the global \texttt{OutPort} ones.

24Note that Jiotto also comes with the package \texttt{samples}, which – among other simple applications – holds the template classes \texttt{S0*.java} that can be used as the basis of new user applications.
updates \( o_1 \) accordingly. \( t_0 \) is defined to be invoked with a frequency of 4 and thus is started every 100 ms. \( t_1 \), on the other hand, has a task invocation frequency of 1 and thus is started every 400 ms. The two task drivers \( d_0 \) and \( d_1 \) simply transfer the value of the output port to the input port, making it possible to steadily increment or decrement the current value.

\( d_2 \) has a mode switch frequency of 2, and its driver guard evaluates to \( \text{true} \) if \( o_0 \mod 5 = 0 \). \( d_3 \)'s mode switch frequency is 1, and the condition of its guard in order to evaluate to \( \text{true} \) is similar to the previous one: \( o_1 \mod 5 = 0 \). The task invocation and mode switch frequencies as well as the guards’ conditions and the mode periods all contribute to the desired behaviour of the application.

Note that the initial values of \( i_0 \) and \( i_1 \) are lost due to the executions of the task drivers \( d_0 \) and \( d_1 \) that update the input ports with the values of the corresponding output ports. Furthermore, the initial value of \( o_1 \) is lost too, as when switching from \( m_0 \) (which is the start mode) to \( m_1 \), the value of \( o_1 \) is updated by \( d_2 \) with the current value of \( o_0 \). The latter’s initial value thus determines all of the application’s subsequent values.
5.4 Development Process and Design Decisions

The following sections hold several design and implementation possibilities that were considered but discarded during the development process of Jiotto. (Throughout the document, there are links to most of the sections below using a marginal note with the keyword Design.) Changes to the class hierarchy made during the development of Jiotto are discussed in section 5.4.8.

5.4.1 The Control Thread’s Period

ControlThreadSingleton.setPeriod() (see section 5.2.1.4) is used to set the CT’s period at run-time to correspond to the changed conditions of a just entered mode.

As mentioned in section 4.5.1.2, changing the release parameters at run-time is not compatible with Ravenscar-Java. Other design possibilities, apart from implementing setPeriod() as described above, would have been as follows:

- Changing the CT’s period could be prevented if there was a dedicated control thread for all modes each. While these CTs could have fixed periods, this not only made the implementation of the package unnecessarily complex. Furthermore, it is not RJ compatible either as there was hardly a way to implement this correctly without Asynchronous Transfer of Control (ATC) (which again is not allowed by RJ).

5.3.2 The Code

The complete code is available in chapter D.

Figure 5.7: Sample Application: Mode Switch $m_1$ to $m_0$
• The CT’s period could be chosen as the least common multiple of all mode frequencies. In general, though, the CT’s period would drop dramatically and lead to the CT being woken up every single time unit. Clearly, this was not an acceptable solution.

5.4.2 Update Task Output Ports

Transferring the values from TempOutPort objects to the corresponding “official” OutPort ones is part of Giotto Micro Step 1 (see section 5.2.1.5) and is performed by the CT.

Besides from Giotto clearly stating this functionality to be performed by one of the GMS, having the tasks themselves perform these transfers would neither be practical nor even technically possible: The transfers have to be performed in logically no time. The CT has the highest priority of all processes and thus can easily pre-empt all others to transfer the values transparently. Nevertheless, it would not only be difficult to have the tasks act on the given point in time exactly. Even more, with the tasks themselves having different priorities, the correct timing of “updating all the tasks’ ports at once” could not be guaranteed, as a task was possibly simply not scheduled to run at the time it needed to.

5.4.3 Tasks as Threads

While tasks are implemented in Jiotto as distinct threads (see section 5.2.3), it was also considered to make the tasks usual passive objects with a known interface in order for the CT to run the task function as needed. While this would offer direct control to Jiotto over which task function would be performed when, this not only had the disadvantage of not using well-proven scheduling algorithms of the real-time platform. Furthermore, and most importantly, a full Giotto implementation was not possible this way, as the invocation of a Java method is not pre-emptible within one thread. As soon as the CT invoked a task function that needs \( n \) time units of physical execution time, other tasks that need to be scheduled every \( m < n \) time units would automatically miss their deadlines.

5.4.4 ReusableNoHeapRealtimeThread

When working with immortal memory, especially in combination with periodically executing applications such as Jiotto-based ones, it is vital to be able to re-use all objects needed during the run-time of the program. The class ReusableNoHeapRealtimeThread (see section 5.2.6) offers a well-defined and (externally) tested code base as a solution to this problem.

The functionality of this class could have been integrated into the Task class with the latter extending NoHeapRealtimeThread directly. However, for
example concerning the Giotto definitions, the re-usability concept for a thread has nothing to do with a task. Usage of a distinct class that `Task` extends separates responsibilities and clears up the design.

Throwing the `ReusableThreadStillWorkingException` in the method `ReusableNoHeapRealtimeThread.go()` is one possible way of notifying the invoking part of the program about the corresponding situation. In Jiotto, the CT then has to deal with the task that missed its deadline. While the current implementation simply prints the exception’s stack trace and exits the system, a real-world application will probably have to handle the failure situation by trying to undo any damage already caused due to missing the deadline and go on with the execution of the program. If this was not possible, the application should at least switch to an emergency mode and stop the system safely.\(^{25}\)

Note that, in case the application was designed to keep running, the implementation of the `ReusableNoHeapRealtimeThread` should be changed to return a – for example boolean – value instead of throwing an exception: The exception mechanism in Java is not only very slow compared to returning a primitive type, it also consumes valuable memory by creating a new object. With the application using immortal memory, this might pose a memory leak.

### 5.4.5 Arrays vs. HashMaps

A basic decision had to be made on how to store the different sets of objects, such as ports, a mode’s task invocations, a driver’s transfers, etc. Besides the classical arrays (which were chosen for the implementation of Jiotto), Java also offers different types of collections, such as Lists and Maps:

**Arrays** In Jiotto, all needed objects are known exactly in advance; during the mission phase, nothing changes dynamically. With this information based on the application model, the `jiotto` package can create the suitably sized arrays during the initialisation phase. As access to most of the objects such as ports is possible via their ID’s (which directly map to array indexes), this solution offers the best performance with practically no disadvantage. Furthermore, due to being strictly type-safe, type checking can be performed at compile time already, the code is generally less error-prone and less cluttered with typecasts.

**HashMaps** HashMaps constitute an interesting type of collection. Using a “key” (such as a port’s ID), it was easily possible to store and retrieve objects in a dynamical way so that it was not necessary to specify something like array sizes in advance. This flexibility would not come for free, yet: When adding objects to a hash-based collection, a “re-hashing” can occur.

\(^{25}\)Dealing with concepts such as detecting and handling failures, errors or faults in real-time systems is outside the scope of this thesis. See the corresponding literature for information on this topic, for example Kopetz [21].
While this process may be guaranteed to never occur under certain conditions\textsuperscript{26}, there was still the disadvantage of having to typecast all objects being retrieved from a collection\textsuperscript{27}. This would not only make code more complicated but also more error-prone, as type checking had to be done at run-time rather than at compile-time. Furthermore, the dynamic nature of collections, which also brings along a performance cost at run-time, is not needed in Jiotto.

Other types Again, the dynamic nature of lists is not needed in Jiotto, as is not the sorting of tree-based collections.

5.4.6 \textit{jiotto sub-packages}

In an early stage of implementing the Jiotto prototype, it was considered to create sub-packages such as \textit{jiotto.ports}. The necessary changes were quite drastic.

\texttt{ControlThreadSingleton} has initially been designed to hold the factory functionalities (and not only serve as an interface between the user and the factory classes as it is implemented now). Splitting the functionality into sub-packages such as \textit{jiotto.ports} or \textit{jiotto.drivers} also introduced the need for separate classes within these new packages offering the factory methods.

\textit{Reasoning:} Java offers the four access modifiers \textit{private}, \textit{protected}, \textit{public}, and the unspecified “package” one\textsuperscript{28}. Granting access to classes outside the local package is only possible for inheriting classes when using \textit{protected} or for every class when using \textit{public}. There is no way of specifying access rights based on package \textit{hierarchies} or for explicit “friendly” classes. The class \texttt{ControlThreadSingleton} is clearly placed in the base package \textit{jiotto} and thus is not part of any of the sub-packages \textit{jiotto.ports} or similar. For the CT to be able to create port objects such as \texttt{InPorts}, the constructors of these classes had to be \textit{public}\textsuperscript{29}. Hence, this would enable the user to create such objects without using the factory methods, which shall not be allowed, as the CT lacked control over the created objects this way.

With the resulting structure, it made no difference whether the user invoked \texttt{jiotto.ControlThreadSingleton.createPort()} or the factory method \texttt{jiotto.ports.PortFactory.create()}, since both methods actually “performed the same actions”. While this would have worked, note that it was not considered an advantage: One single interface to the overall Jiotto framework

\textsuperscript{26}Sun’s API documentation, \texttt{java.util.HashMap}: “If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.”

\textsuperscript{27}With J2SDK 1.5, \texttt{generics} will be introduced which will allow type-safe usage of collections. Nevertheless, J2SDK <= 1.4 based Java versions only know the \texttt{Object} type.

\textsuperscript{28}See \url{http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html} for detailed information on the access modifiers.

\textsuperscript{29}Usage of \texttt{protected} with the CT inheriting from any Giotto component class clearly is not an option.
5.4. DEVELOPMENT PROCESS AND DESIGN DECISIONS

for creating ports (and tasks, etc.) is the better choice.

In summary, clearing things up for both the Jiotto-user and the Jiotto-developer using sub-packages at the same time would introduce a considerable weakening of the OOP concept of encapsulation.

Note that, though Jiotto is available as the single package jiotto, the factory methods are nonetheless made available through distinct classes (see section 5.2.7).

5.4.7 Port Value Types

A Giotto implementation has to separate the output port objects, which may be updated by the task functions at any point in time of the tasks’ logical executions, from the output port objects that are updated by GMS 1 to make the change available to the overall application. When using Java primitives as it is implemented by jiotto now (see section 5.2.2.1), the values can easily be copied from one port to another.

The original design idea was to make Jiotto’s implementation of how to store the port values as flexible as possible by using a general object-based value type. However, it turned out that this is not possible or not reasonable:

- The two semantically related TempOutPort and OutPort objects cannot share a reference to the same value object (as a task function’s update of an output port would have immediate effect on the application wide output port in this case).

- Due to working with immortal memory, it is not possible to create new objects with the current value and have GMS 1 set the according reference in the global OutPort object; this obviously was a memory leak.

- Have the port value implement an interface that offered one or several getValue() methods (or similar) in order to enable the GMS 1 to easily transfer the value does not work either, as any method returns void or a value of a specific type. Returning an object reference would not help, as the two different ports must not use the same value object.

Thus, it is necessary to use Java primitives with different getter and setter methods. However, for this functionality, a distinct port value class is not advisable, as this also can be implemented in Port directly.

5.4.8 Class Hierarchy Development

Working on a project such as Jiotto, it is not only good OOP practice but it is even necessary to design a class hierarchy before starting to implement anything. As it turned out, the initial design of Jiotto’s class hierarchy has not changed much during the implementation process:

Note that in Java overloaded methods must have distinct parameter signatures. Different return values only is not enough.
Ports  The two classes `TempOutPort` and `TempSensePort` as well as the interfaces `ModePort` (implemented by `OutPort` and `TempOutPort`) and `PortMethods` (implemented by all ports) have been added, with the latter mostly serving as a convenience to the user.

Tasks  In order to separate different functionalities and to clear up responsibilities, `Task` has been changed not to extend `NoHeapRealtimeThread` directly, but rather inherit from `ReusableNoHeapRealtimeThread`.

Drivers  Driver type specific interfaces extending `DriverGuard` have been added to enable task and mode driver guards to evaluate on different sets of values than actuator driver guards do. The interface `DriverFunction`, though being correct according to the formal definition of Giotto, has been removed, because in practice a driver only transfers values as they are and does not change them on the way.
Chapter 6

Using Jiotto to Control an Elevator

This chapter presents two simulations of controlling an elevator, which points out the applicability of Jiotto to real-world problems. While the actual Jiotto program realistically reads sensors and updates actuators according to the application model (especially in the more advanced Elevator2 application), the physical world is simulated by additional threads.

6.1 Elevator

Elevator is based on a very similar demo application available in the Giotto Development Kit (GDK). The application model consists of five modes: idle processes the sensory data and decides, which floor the cabin has to move to next. If a new value is available, the system switches to closeDoors. The next mode, up or down, is entered according to whether the target floor is higher or lower than the current position of the cabin. Once the target floor has been reached, openDoors is entered, followed by a mode switch to idle again.

6.1.1 Application Model

The application model, shown in figures 6.1 and 6.2, is documented in detail in this section.

6.1.1.1 Ports

- SensePort $s_0$ ("Sensed Destination"): Updated by user-input at any given time when the system has nothing else to do (e.g. physical execution of tasks, GMS, etc.).

---

1Updating the sensor port $s_0$ is accomplished using a minimum-priority sensors thread.
Figure 6.1: Elevator: Modes, Actuator Updates and Task Drivers
Figure 6.2: Elevator: Mode Switches
Possible values: 0 - 9.
Initial value: 0 (same as o1).

- **OutPort o0 (“Target Floor”):** This value holds the current target floor to go to and is updated as part of the mode switch from *idle* to *closeDoors*. Being available as a mode port in *closeDoors*, *up* and *down*, all mode switches, task invocations and actuator updates can access this value, compare it to the value of o1 and act accordingly.
  Possible values: 0 - 9.
  Initial value: 0 (same as o1).

- **OutPort o1 (“Current Floor”):** While in reality the elevator’s current floor would also be a value of a sensor port\(^2\), in the given application the cabin moves by a whole floor during one mode round in each mode *up* and *down*. This change of the “physical” situation is accomplished by having tasks increment or decrement the value of o1 by 1.
  Possible values: 0 - 9.
  Initial value: 0 (same as s0 and o0).

- **OutPort o2 (“Doors”):** Similar to the situation of the cabin’s current floor, the real-world change is simulated by having tasks set the status of the doors to closed or open within one mode round of *closeDoors* and *openDoors* only.
  Possible values: 0 (closed), 1 (open)
  Initial value: 1 (same as a1).

- **OutPort o3 (“Value for Motor”):** The tasks *taskUp* and *taskDown* decide, what value is to be transferred into the motor-controlling actuator port in order to go up, down or stop moving. o3 serves as the intermediary storage port for this value.
  Possible values: -1 (down), 0 (stop), 1 (up)
  Initial value: 0 (same as a0).

- **ActuatorPort a0 (“Motor Actuator”):** Set to current value of o3.
  Possible values: -1 (down), 0 (stop), 1 (up)
  Initial value: 0 (same as o3).

- **ActuatorPort a1 (“Doors Actuator”):** Set to current value of o2.
  Possible values: 0 (closed), 1 (open)
  Initial value: 1 (same as o2).

- **InPort i0 (“Up0”):** Associated with task *taskUp*, set to current value of o1.
  Initial value: 0.)\(^3\)

\(^2\)Also, see Elevator2.

\(^3\)The brackets around “Init Value” mean that the initial value will never be used: It is updated by the task driver before being read the first time.
6.1. ELEVATOR

- **InPort** \( i_1 \) ("Up1"): Associated with task \( taskUp \), set to current value of \( o_3 \).
  (Initial value: 0.)

- **InPort** \( i_2 \) ("Down2"): Associated with task \( taskDown \), set to current value of \( o_1 \).
  (Initial value: 0.)

- **InPort** \( i_3 \) ("Down3"): Associated with task \( taskDown \), set to current value of \( o_3 \).
  (Initial value: 0.)

- **InPort** \( i_4 \) ("Close"): Associated with task \( taskClose \), set to constant value 1.
  (Initial value: 0.)

- **InPort** \( i_5 \) ("Open"): Associated with task \( taskOpen \), set to constant value 1.
  (Initial value: 0.)

6.1.1.2 Drivers

- **TaskDriver** \( d_0 \) ("TaskDriver Up"): \( o_1 \rightarrow i_0, o_0 \rightarrow i_1 \).
  Guard is true: Always.

- **TaskDriver** \( d_1 \) ("TaskDriver Down"): \( o_1 \rightarrow i_2, o_0 \rightarrow i_3 \).
  Guard is true: Always.

- **TaskDriver** \( d_2 \) ("TaskDriver Close"): \( 0 \rightarrow i_4 \).
  Guard is true: Always.

- **TaskDriver** \( d_3 \) ("TaskDriver Open"): \( 1 \rightarrow i_5 \).
  Guard is true: Always.

- **ActuatorDriver** \( d_4 \) ("ActuatorDriver a0"): \( o_3 \rightarrow a_0 \).
  Guard is true: Always.

- **ActuatorDriver** \( d_5 \) ("ActuatorDriver a1"): \( o_2 \rightarrow a_1 \).
  Guard is true: Always.

- **ModeDriver** \( d_6 \) ("idle-to-closeDoors"): \( s_0 \rightarrow o_0 \).
  Guard is true: If \( s_0 \neq o_1 \).

- **ModeDriver** \( d_7 \) ("closeDoors-to-up"): No transfers.
  Guard is true: If \( o_0 > o_1 \).

- **ModeDriver** \( d_8 \) ("closeDoors-to-down"): No transfers.
  Guard is true: If \( o_0 < o_1 \).

- **ModeDriver** \( d_9 \) ("(up|down)-to-open"): No transfers.
  Guard is true: If \( o_1 = o_0 \).

- **ModeDriver** \( d_{10} \) ("open-to-idle"): No transfers.
  Guard is true: Always.
6.1.1.3 Tasks

- **taskUp** (ID 0).
  
  Ports: \(i_0, i_1, o_1, o_3\).
  
  TaskDriver: \(d_0\).
  
  TaskFunction: The elevator is moved up to the next floor and the value for the motor is set accordingly.

  \[
  o_1 = i_0 + 1; \\
  \text{if } (o_1 < i_1) \ o_3 = 1; \\
  \text{else if } (o_1 == i_1) \ o_3 = 0; \\
  \text{else "Ooops"};
  \]

- **taskDown** (ID 1).
  
  Ports: \(i_2, i_3, o_1, o_3\).
  
  TaskDriver: \(d_1\).
  
  TaskFunction: The elevator is moved down to the next floor and the value for the motor is set accordingly.

  \[
  o_1 = i_2 - 1; \\
  \text{if } (o_1 > i_3) \ o_3 = -1; \\
  \text{else if } (o_1 == i_3) \ o_3 = 0; \\
  \text{else "Ooops"};
  \]

- **taskClose** (ID 2).
  
  Ports: \(i_4, o_2\).
  
  TaskDriver: \(d_2\).
  
  TaskFunction: The doors are closed.

  \[
  o_2 = i_4.
  \]

- **taskOpen** (ID 3).
  
  Ports: \(i_5, o_2\).
  
  TaskDriver: \(d_3\).
  
  TaskFunction: The doors are opened.

  \[
  o_2 = i_5.
  \]

- Four dummy tasks are used to add output ports as mode ports to specific modes, see figure 6.1.

6.1.1.4 Modes

- **idle** (ID 0, period 1000 ms).
  The elevator does nothing but to wait for a command where to move the cabin to next.
  User inputs (the floor the cabin shall move to) are accepted all the time.
in all modes. However, the elevator only reacts if the system is in the
idle mode. Only the value of sensor port \( s_0 \) at the time of evaluating the
mode switch to the mode \textit{closeDoors} is used, all other meanwhile possibly
entered commands are discarded.

- **Mode switch:**
  - Target mode \textit{closeDoors} (ModeDriver \( d_6 \), frequency 1): If the value
    of \( s_0 \) (where to go to) is different from the value of \( o_1 \) (the floor the
    cabin currently is in), switch to the target mode. As part of the
    mode switch, transfer the value of \( s_0 \) into \( o_0 \), which enables other
    modes to check whether or not the target floor has been reached
    already.

- **Task invocation:**
  - Dummy task only, registering \( o_1 \) as a mode port.

- \textit{closeDoors} (ID 1, period 1000 ms).
  On once a new target floor has been made available, the doors have to be
closed.

- **Actuator update:**
  - Invoke closing the doors (ActuatorDriver \( d_5 \), frequency 1): The
guard always evaluates to \texttt{true} and tells the doors to \textit{close} by
transferring the value of \( o_2 \) into \( a_1 \).

- **Mode switches:**
  - Target mode \textit{up} (ModeDriver \( d_7 \), frequency 1): If the target floor
    is higher than the cabin’s current floor.
  - Target mode \textit{down} (ModeDriver \( d_8 \), frequency 1): If the target
    floor is lower than the cabin’s current floor.

- **Task invocations:**
  - \texttt{taskClose} (frequency 1).
  - Dummy task, registering \( o_1 \) and \( o_2 \) as mode ports.

- \textit{openDoors} (ID 2, period 1000 ms).
  As soon as the target floor has been reached and the cabin has stopped,
the doors have to be opened.

- **Actuator update:**
  - Invoke opening the doors (ActuatorDriver \( d_5 \), frequency 1): The
guard always evaluates to \texttt{true} and tells the doors to \textit{open} by
transferring the value of \( o_2 \) into \( a_1 \).

- **Mode switch:**
CHAPTER 6. USING JIOTTO TO CONTROL AN ELEVATOR

- Target mode idle (ModeDriver $d_{10}$, frequency 1): The guard always evaluates to true.

  - Task invocation:
    - taskOpen (frequency 1).

- up (ID 3, period 1000 ms).
  Each round, the cabin is moved up by one floor.

  - Actuator update:
    - Move up or stop (ActuatorDriver $d_4$, frequency 1): Transfers the value of $o_3$ into $a_0$.

  - Mode switch:
    - Target mode openDoors (ModeDriver $d_9$, frequency 1): If the target floor has been reached, the system switches to openDoors.

  - Task invocations:
    - taskUp (frequency 1).
    - Dummy task, registering $o_0$ as mode port.

- down (ID 4, period 1000 ms).
  Each round, the cabin is moved down by one floor.

  - Actuator update:
    - Move down or stop (ActuatorDriver $d_4$, frequency 1): Transfers the value of $o_3$ into $a_0$.

  - Mode switch:
    - Target mode openDoors (ModeDriver $d_9$, frequency 1): If the target floor has been reached, the system switches to openDoors.

  - Task invocations:
    - taskDown (frequency 1).
    - Dummy task, registering $o_0$ as mode port.

6.1.2 Sensors Threads: Interactive vs. Random Mode

The Elevator application offers both an interactive mode, where the user can enter the next target floor manually, and a random mode, where the system sets the sensor value of $s_0$ to random values automatically. (This choice is available as a compile-time option by changing Elevator.java to add a different object using ct.addSensorsActuatorsFunction().) jiotto.SensorsActuatorsFunction, the interface implemented by the simulation thread classes, is discussed in section 5.2.10. The basic idea of

Note that such a thread runs at the lowest real-time priority possible: If the control thread or any task needs to do something, these components can pre-empt a sensors thread.
6.2. ELEVATOR2

elevator.SensorsFunctionInteractive is to continuously read lines from System.in and – if it is an integer – accept the first character as the new sensor value for $s_0$ and thus the next target floor.

6.1.3 Running the Application

```sh
# tjvm -Djava.class.path="." -Xbootclasspath=\
  ${JIOTTO}/lib/foundation.jar elevator/Elevator
security properties not found. using defaults.
Usage:
  First character of every line is the new floor
to go to. (The rest is discarded.)
  Valid values are 0 through 9.
  CTRL+D deactivates the elevator and exits the system.
  New floors to go to can be entered at any time,
  the latest is used.
Enter floor to go to:
  2
    |
    |
    XX   XX|
    XXXX   XXXX|
    XXXXXX   XXXXXX|
    XXXXXXXX   XXXXXXX|
    XXXXXXXXXXXXXXXXXXX|
      MOVED UP TO:  1
      MOVED UP TO:  2
    XXXXXXXXXXXXXXXXXXX|
    XXXXXXXX   XXXXXXX|
    XXXXXXX   XXXXXX|
    XXXX   XXX|
    XX   XX|
    |
```

At the beginning, the system is in mode idle, the cabin is in floor 0 and the doors are open. As soon as a new target floor is available (checked every 1000 ms), the system closes the doors – which is “graphically” shown by the next lines. Then, the cabin is moved to the target floor, and the doors are opened again.

6.2 Elevator2

The idea of this second and more realistic elevator is to have the Jiotto system update the actuators and have the environment react accordingly. (While in the previous version the actuators are updated, the cabin is simply moved by
one floor each mode round by having tasks set the \( o_1 \) mode port accordingly – the values of actuators are simply never read.)

To obtain the target floor, the same sensors thread is used as is also used for \textit{Elevator}. Nevertheless, a second \texttt{SensorsActuatorsFunction} is added as another minimum-priority thread, which reads the commands from the application’s actuator ports and reflects the continuous changes of the physical environment by setting float based sensor values. Every \( \text{EVERY\_MSECS} = 100 \) milliseconds the system spends in for example mode \textit{up}, the cabin is moved up by \( \text{CABIN\_MOVEMENT} = 0.1f \) of a floor height.

### 6.2.1 Changes Compared to \textit{Elevator}

The application model of \textit{Elevator2} is basically the same as the one of \textit{Elevator} (see section 6.1), with the differences being as follows:

- There are two more sensor ports \( s_1 \) (“Sensed Floor”) and \( s_2 \) (“Sensed Doors”) representing the current position of the cabin and the doors.
- There is another output port \( o_4 \) (“Value for Doors”).
- The value type of \( i_0, i_2, i_4 \) and \( i_5 \) has changed to be \texttt{float}.
- The possible values of \( a_1 \) (-1 (close), 0 (stop), 1 (open)) and its initial value (0 instead of 1) have changed.
- Some driver transfers have changed: \( d_0 \) transfers \( s_1 \rightarrow i_0 \) (instead of \( o_1 \rightarrow i_0 \)), \( d_1 \) transfers \( s_1 \rightarrow i_2 \) (instead of \( o_1 \rightarrow i_2 \)), \( d_2 \) transfers \( s_2 \rightarrow i_4 \) (instead of \( 0 \rightarrow i_4 \)), \( d_3 \) transfers \( s_2 \rightarrow i_5 \) (instead of \( 1 \rightarrow i_5 \)), \( d_5 \) transfers \( o_4 \rightarrow a_1 \) (instead of \( o_2 \rightarrow a_1 \)).
- The tasks \texttt{taskClose} and \texttt{taskOpen} have the new output port \( o_4 \) added.
- The mode periods of the four movement modes \textit{closeDoors}, \textit{openDoors}, \textit{up} and \textit{down} have changed considerably to only 50 milliseconds.
- The mode switch conditions for \( d_7 \) and \( d_8 \) have changed to also check for \( o_2 = 0 \) to evaluate to \texttt{true}.
- The mode switch condition for \( d_{10} \) has changed to check for \( o_2 = 1 \).
- There is another sensors and actuators function for the two new sensor ports: Depending on the actuator ports, it simulates the movement of both the cabin and the doors corresponding to the time the system spends in a specific mode. Simulation of changes is implemented by updating the sensor values accordingly.
- The task functions have changed considerably: While in \textit{Elevator}, the task functions moved the cabin and doors, in \textit{Elevator2} this is done by the
new sensors thread mentioned above. The task functions now simply calculate a new value for the actuators (based on the current sensor values) and present this result via the output ports $o_3$ and $o_4$ respectively.

Some of the changes are also shown in figure 6.3.

### 6.2.2 Running the Application

In contrast to the simple elevator application, $Elevator2$ moves the cabin as well as opens and closes the doors all continuously: Messages such as "Motor UP, cabin is at 0.1" as well as the status of the doors are printed by the sensors thread, "graphically" indicating the current status in the physical world. "MOVED UP TO: 1" and similar is output produced by the task function of $taskUp$ in order to log that the cabin has reached the first floor completely.

```bash
# tjvm -Djava.class.path="." -Xbootclasspath=
 ${JIOTTO}/lib/foundation.jar elevator2/Elevator2
[...]
Enter floor to go to:
2
```

Figure 6.3: $Elevator2$: Some of the Changes Compared to $Elevator$
CHAPTER 6. USING JIOTTO TO CONTROL AN ELEVATOR

| X X |
| XX XX|
| XXX XXX|
| XXXX XXXX|
| XXXXXXXXXX|

Motor UP, cabin is at 0.1
Motor UP, cabin is at 0.2
Motor UP, cabin is at 0.3
Motor UP, cabin is at 0.4
Motor UP, cabin is at 0.5
Motor UP, cabin is at 0.6
Motor UP, cabin is at 0.70000005
Motor UP, cabin is at 0.8000001
Motor UP, cabin is at 0.9000001
Motor UP, cabin is at 1.0000001

MOVED UP TO: 1

Motor UP, cabin is at 1.1000001
Motor UP, cabin is at 1.2000002
Motor UP, cabin is at 1.3000002
Motor UP, cabin is at 1.4000002
Motor UP, cabin is at 1.5000002
Motor UP, cabin is at 1.6000003
Motor UP, cabin is at 1.7000003
Motor UP, cabin is at 1.8000003
Motor UP, cabin is at 1.9000003
Motor UP, cabin is at 2.0000002

MOVED UP TO: 2

| XXXXXXXXXX |
| XXXX XXXX |
| XXX XXX |
| XX XX |
| X X |
| |
Chapter 7

Conclusion

Jiotto is an implementation of the Giotto semantics based on the Real-Time Specification for Java (RTSJ). With the RTSJ introducing the determinism necessary for real-time applications, the overall jiotto package enables the development of applications that are heavily based on the timed model. Jiotto brings the two worlds of Giotto and real-time Java together, adding valuable aspects and possibilities to each other.

With the realistic simulation of controlling an elevator in Elevator2, the applicability to real-world problems has been shown. This is completed by Java’s abilities to include native code via the JNI or even develop device drivers in Java itself with the help of the RTSJ.

The bulk of the jiotto package is implemented using standard-Java compatible code. The only core classes directly based on RTSJ functionality are:

- **Initialiser** Real-time thread with maximum priority; immortal memory.
- **ControlThreadSingleton** Periodic real-time thread with maximum priority; immortal memory.
- **ReusableNoHeapRealtimeThread** Real-time threads; immortal memory.
- **Task** Extending ReusableNoHeapRealtimeThread and using variable real-time priorities.

This little real-time specific code in the implementation of the total package shows that the RTSJ can make it easy for Java programmers to enter the embedded real-time world. Nevertheless, the environment the applications will run in as well as RTSJ specifics such as the correct usage of immortal and scoped memory have to be kept in mind; as is illustrated in this document, memory leaks and other Java non-typical characteristics might be introduced.

The shortest period possible in order to achieve correct execution for both a very simple periodic real-time thread as well as the full Elevator2 Jiotto application is 20 ms. This behaviour has been tested on and is the same for an
CHAPTER 7. CONCLUSION

Intel Mobile Pentium III running at 730 MHz and 1000 MHz as well as an AMD Duron 1300 MHz. While this hardware is nothing that would be used for deployed embedded systems nowadays, it suggests that not the hardware itself is the performance bottleneck here. These constraints are rather based on the RTSJ reference implementation, the Linux kernel used (which was not optimised for real-time applications with high-resolution timing) or a combination of both.

The following quotation from Dibble [4] compares real-time Java with C, being one of the most important and widely used programming languages: “Java puts programmer productivity before everything else. [...] If you feel that strongly about performance, use C or assembler, but a Java application would be designed, written, and debugged before the C application is coded. Real-time Java is designed with the same theme.” The applicability or non-applicability of real-time Java may thus depend on the problem to be solved. In any case, the RTSJ reference implementation (RI) most probably offers too little care for memory use and performance in comparison to a commercial product, but it is definitely well suited for experiments concerning the RTSJ. A more detailed discussion of the RTSJ and the RI can be found in chapter B.

Continuative work based on Jiotto might include:

- Investigate and analyse performance and memory footprint issues, including high-resolution timing in order to be able to decrease periods.
- Have Jiotto run on Java dedicated hardware such as JOP [17] or similar.
- Make Giotto support complete by allowing mode switches with logically running tasks.
- Implement a Giotto-to-Jiotto (and vice versa) parser in order to automatically translate Giotto to Jiotto code.
- Bring in fault tolerance.
- Consider worst-case execution times (WCET) of functional code.

Finally, I would like to mention some personal experiences I made while designing and implementing the jiotto package. Not having developed a library or Java package with a similar way of being used before, it was necessary to determine what the system is supposed to do. Once I had realised that development of an API is somewhat different than the usual application development, it proved quite useful to concentrate on finding the use cases: Designing a sample application in order to specify the demands to be met by jiotto helped a lot to dive into API programming.

Some decisions made cannot be classified as being correct or incorrect. For example, the jiotto package has been designed and implemented with many Ravenscar-Java oriented approaches in mind. One of them specifies that “all
user-defined classes must include constructors that initialise all internal variables and objects”. On the one hand, complying with this rule (as has been done when developing Jiotto) leads to class fields and objects with an explicit initial state. On the other hand, this approach is unusual to the Java world and thus quite error-prone if the developer does not keep it in mind all the time. Forgetting to perform the initialisation every now and then leads to an inconsistent implementation; in which case it probably might have been better to drop such a rule totally.

When running Jiotto applications, keep the platform characteristics in mind: Run the programs as Linux’ root user in order to get correct real-time scheduling, and see sections 5.2.1.6 and A.3 for more details on the interaction of Linux/RTSJ-RI and the jiotto package concerning the thread priorities.

Both the initial and detailed design of the class hierarchy as well as the early start to implement a prototype have proven to be good practice. While the former is necessary to get an idea of which classes there are, how they interact with one another and which objects handle what data, the latter was valuable to prevent me from “analysis paralysis”: It is usually not possible to know everything in advance. Having finished a detailed design, starting to work on the prototype revealed solutions for old as well as totally new problems shortly after, which I could not figured out easily before by simply thinking about the overall application.
Appendix A

Linux and Real-Time

As can be found in Silberschatz et al. [31], things required by an operating system to be able to provide at least soft real-time functionality include that “the system must have priority scheduling, and [the] real-time processes must have the highest priority”. This is true for Linux where two available scheduling algorithms are the priority based FIFO and Round-Robin (RR), and all real-time processes have higher priorities than any other normal process in the system.

Furthermore, Silberschatz et al. requires the system to have small dispatch latency. One possibility to achieve this is to make the entire kernel pre-emptible. (Also see section A.2 for more information on Linux 2.6.)

A.1 Linux: A Hard or Soft Real-Time OS?

The question of Linux being a hard or soft real-time operating system cannot be answered easily – and possibly also depends on specific requirements for a given product. With Linux being pushed on (or into) nearly all niches of the IT market, there is a lot of information available – also concerning its real-time capabilities. The fast production cycles with corresponding improvements make it even harder to evaluate current facts.

Nevertheless, Linux is real-time capable to some degree at least. Furthermore, besides the default kernel tree, there are many different flavours of Linux with several approaches of how to add (further) “real-time” to the kernel. Some of them are free and some are commercial, several claim to offer hard real-time services.

Two possible and technologically distinct approaches to a “real-time Linux” are:

- FSMLabs (RTLinux) and LynuxWorks (BlueCat RT) both use a small real-time operating system as the cores of their products. Linux itself is run as a thread to this OS.

- TimeSys (TimeSys Linux RTOS) uses changed kernel code and additional
modules not only to improve Linux’ predictability and make it “hard real-time”, but also to introduce additional features such as priority inversion avoidance mechanisms.

A dense list of real-time Linux products is for example available at [24] and can be used as a place to start further research on this topic. Most of the solutions currently available are still based on Linux 2.2 or Linux 2.4.

Process scheduling (including corresponding real-time topics) for the Linux 2.2 series is explained in detail in Bovet and Cesati.\(^1\)

### A.2 Linux 2.6

While, according to Silberschatz et al. [31], Linux is only a soft real-time operating system, especially due to lacking the feature of kernel pre-emption, this has changed with the latest release of this OS: As a new feature of the default Linux kernel tree, version 2.6 includes “pre-emption points” in order to stop a kernel task in favour of an important user application to be run instead.

Further improvements of Linux 2.6 over previous versions include the $O(1)$ scheduler: Processes are sorted into a queue as soon as they get ready to run. In order to decide, which process is to be executed next, the scheduler does not have to consider all runnable processes anymore, but can simply use the “first” one in the queue. This way, it only consumes a constant amount of time.

Additional features new to the 2.6 series that are relevant to the embedded real-time world can for example be found at White [23]. Quoting the author: “Version 2.6 is a great stride in the right direction.”

### A.3 Extend Accepted Priority Range

As mentioned in section 4.4.1.1, the priorities supported by a default Linux 2.4 kernel range up to 99. In order to be able to use .getMaxPriority() correctly with TimeSys’ RTSJ reference implementation (RI), one solution is to apply the patch shown in algorithm 6 to the Linux 2.4.24 kernel sources.\(^2\) All requested priorities within the range from 100 through 265 are mapped to the real priority 99. With an application only requesting priorities of up to 98, this ensures a correct behaviour of the RI. (Without this patch, all priorities above and including 100 are treated as those of normal non-real-time processes. This could lead to a possibly wrong behaviour of the RTSJ application as well as the RI itself. Having the application explicitly not use priorities above 98, leaves the vanilla Linux’ highest real-time priority 99 for the RI itself. Also, see section B.6.)

\(^1\)The corresponding chapter of Bovet and Cesati is available at http://www.oreilly.com/catalog/linuxkernel/chapter/ch10.html.
\(^2\)Note: Lines ending with a “\" are to be joined with the next one to form one line only.
Algorithm 6: Linux kernel patch to extend the accepted priority range
Appendix B

RTSJ and the Reference Implementation

The RTSJ Reference Implementation (RI) is freely available at TimeSys [32] and is briefly described in section 4.4.1. This chapter discusses further aspects of the RI and the RTSJ in general.

B.1 Hello Real-Time World

Once the software has been set up according to the descriptions in the README file, the “Hello Real-Time World” program shown in algorithm 3 on page 27 can be compiled and run as follows, with javac being Sun’s Java compiler and tjvm being the RTSJ-VM:

```
$ javac -classpath /opt/rtsj/lib/foundation.jar HelloPeriodicRTWorld.java
# tjvm -Djava.class.path=.
    -Xbootclasspath=/opt/rtsj/lib/foundation.jar HelloPeriodicRTWorld
security properties not found. using defaults.
Hello periodic RT world! Period: 1
Hello periodic RT world! Period: 2
Hello periodic RT world! Period: 3
Hello periodic RT world! Period: 4
Hello periodic RT world! Period: 5
Hello periodic RT world! Period: 6
Hello periodic RT world! Period: 7
Hello periodic RT world! Period: 8
Hello periodic RT world! Period: 9
Hello periodic RT world! Period: 10
```
Note that the program has to be run as root in order to be able to use Linux’ real-time scheduling capabilities. (When being run by a normal user, the application threads will be scheduled as usual ones: no preferences over other threads in the system are granted and thus at least the timing will most probably not be correct.)

B.2 Non-Heap-Threads and Exceptions

In Java, exceptions that are thrown by a thread’s run() method can only be runtime exceptions and must not be checked ones (as extending the set of checked exceptions to be thrown by an overridden method is not possible). With normal heap-using threads (both real-time or standard Java ones), each thread belongs to a thread group. A runtime exception thrown by run() propagates to the thread group and takes the default “uncaught exception” action, which usually means to print the stack trace notifying the outside world about the error.

However, NoHeapRealtimeThreads (and RealtimeThread objects allocated in scoped memory) do not belong to a thread group, which means that the uncaught exception cannot perform the default action to print the stack trace. In the current version of the RTSJ RI, the exception is simply “lost” without any further output notifying the user about the situation.\(^1\)

jiotto overcomes this “inconvenience” of lost information about such error situations by including all code of threads’ run() methods in a try{} block. This is only necessary in the two classes ControlThreadSingleton and ReusableNoHeapRealtimeThread and can thus be maintained easily. (Also, see jiotto.ReusableRealtimeThread for more information on this topic.)

B.3 Scoped Memory

The flexibility of scoped memory classes is quite limited when being used for periodically executed tasks: In order to prevent memory leaks that might be introduced by repeatedly constructing scoped memory objects, it is necessary to create an object with fixed functionality (see section 4.3.2.1). While it is possible to transfer status information of primitive types by setting according attributes of this object from the outside, the functionality itself cannot be changed once the object has been created.

Having to use distinct objects for different functionalities, a well-planned

\(^1\)Also, see http://cio.nist.gov/esd/emaildir/lists/rtj-discuss/msg00620.html on this topic.

The “current version” mentioned here is TJVM (build 1.0fcs-ar[1.0.0(R-110)], native threads). Note that upcoming versions of the RI might include functionality to print the stack trace, even in case the thread does not belong to a thread group.
B.4 Memory Leaks

With the introduction of memory areas that are not garbage-collected, also the danger of memory leaks has been brought to Java. Algorithm 2 on page 23 shows how to use scoped memory objects when working with only non-heap memory – which for example is the case with the jiotto package that uses NoHeapRealtimeThreads with immortal memory as the default memory area.

B.4.1 Strings

It is obvious that the process of objects being repeatedly created with the new operator when allocating from immortal memory introduces continuous consumption of irretrievable memory. Note though that also the usage of strings can be very problematic.

The invocation of System.out.println("anything") consumes memory. If such a print-statement, which is often used for logging functionality, is invoked in a NoHeapRealtimeThread, it consumes either immortal memory or scoped memory, depending on which memory area the code is invoked from. Even System.out.println() with no message at all is a memory leak when being continuously invoked on immortal memory. Executing this code in a loop will eventually lead to a "mem_registry.c::InvokeNewInstance() - Went out of memory" error.

Note that it was also possible to increase the size of the immortal memory from the default 2 MB to something higher: Exporting the environment variable IMMORTAL_MEMORY=10000000 sets the immortal memory size to 10 MB. While this can be used to debug e.g. Jiotto applications prior to being deployed, in practice an embedded system shall of course run without any irretrievable memory being consumed for one-time operations.

In order to overcome this situation, it is necessary to enter a scoped memory object every time a logging message is to be constructed. Note that, due to the nature of RTSJ's scoped memory, the strings themselves can not be changed this way, as a distinct scoped memory object with a fixed functionality has to be created (see section B.3). Nevertheless, status information can be provided by setting primitives-based fields of the scoped memory object prior to invoking the logging.

B.4.2 Primitives

Primitives and object references are placed on the stack, which's memory is "reclaimed" automatically due to its nature once the corresponding scope is
left. This means that primitives and object references declared locally within a
method do not constitute memory leaks.

B.5 Periodicity

With the reference implementation running on a default Linux or on TimeSys’
Linux/GPL kernel, the resolution of time for periodic threads is 10 milliseconds.
This means that, with a period other than an integer multiple of 10, the thread
will not be invoked at the explicit millisecond it should be: Assuming a period
of 55 ms, the time that passes between invocations of waitForNextPeriod()
equals the series 50, 60, 50, 60, and so on. While this comes close to the initial
specification, it doesn’t meet it exactly. Also, note that a period of 10 ms is not
possible either.

Concerning the determinism of the RTSJ’s periodicity: At the beginning of
an application, the period is usually off by about 10 ms. During run-time, under
the constraints mentioned above, the period has been observed to be accurate
most of the time and it is only sometimes off by a few milliseconds.

Note that these measurements are based on the RI in combination with a
vanilla Linux as well as TimeSys’ Linux/GPL kernel and do not qualify or dis-
qualify the RTSJ for applicability to any real-world real-time problems. Quot-
ing Dibble [4]: “Although the reference implementation is excellent for exper-
imentation, it is not designed for commercial use. It does not take the care
with performance or memory use that you’d expect from a commercial product.”
Similar considerations have to be taken into account concerning the underlying
Linux-based platform, as for example the kernel code of Linux 2.4 and its pre-
vious versions is not pre-emptible and is thus possibly not applicable to hard
real-time problems (see chapter A).

B.6 Miscellaneous

Note that the reference implementation uses high priority threads internally,
for example to handle asynchronous events.2

Unfortunately, due to a new threading model in Linux 2.6, the current ver-
ion of the RTSJ reference implementation does not run properly on the new
kernel version.3

Also, note that, by default, CTRL+C does not abort the application: Use
another console and run something like killall −9 tjvm to forcibly exit the
RTSJ application.

2See http://cio.nist.gov/esd/emaildir/lists/rtj-discuss/msg00599.html and

3The “current version” mentioned here is TJVM (build 1.0fcs-ar[1.0.0(R-110)],
native threads).
Appendix C

Object Oriented Programming and Design Patterns

C.1 Object Oriented Programming (OOP)

Usually, when developing applications, the same work has to be performed repeatedly. With functional programming languages such as C, code fragments or whole functions often have to be implemented several times, or are “re-used” through “copy and paste”. Unfortunately, this is error-prone as syntactic or semantic errors might easily be introduced this way. Furthermore, it makes a large project hard to manage, as possibly necessary changes on already existing code have to be deployed manually in several places throughout the code.

Object oriented programming (OOP) languages such as Java offer help here and make the development process easier. Not only is the problem domain and thus the developer’s focus shifted from the platform-centric and implementation-driven area of functional languages to a more real-world oriented scope, which makes it much easier to deal with the actual problem and to find an adequate and often more reliable solution more quickly. Further improvements to the development process include the classical OOP features such as encapsulation, inheritance and polymorphism. These are very powerful characteristics, especially when being used in combination:

While sticking with the original interface, encapsulation makes it possible to fully exchange an objects implementation without anyone noticing anything but, for example, the improved performance of the new version. ¹

Inheritance is a perfect way of saving development time by knowing that

¹Note that Giotto’s idea of separating an application’s functionality and timing from the underlying platform can directly be compared to OOP’s encapsulation: As long as the interface between the working object and the outside world stays the same, the implementation itself can be changed.
an inherited class has the same characteristics (and very often even the same implementation) as the super class does. Using a classical example of the OOP literature: A square and a circle both are shapes and can be implemented by inheriting from a shape class. Changes to this base class are not restricted to shape, but also directly affect all derived classes. This means that both square and circle implicitly inherit the changes too.

Polymorphism is yet another feature easing code creation and maintenance. The idea is to have type-specific implementations of a certain functionality. As an example, a program uses the two objects circle and square, both being a shape. Assume that shape offers a method draw(), which draws a shape object on the screen. It is clearly available in both objects circle and square. Nevertheless, they need a distinct algorithm each. The concept of late binding offers exactly that: While the two implementations of the method draw() can be different and specific to each object’s type, it is still possible for the rest of the system to handle the objects as shapes. Invocation of a to shape up-casted circle or square object's draw() implicitly executes the correct type-specific code. Using the model of treating objects as their most basic class possible not only helps to reduce code size and complexity but also holds advantages such as easing eventual changes: The new shape triangle can be introduced with hardly any changes necessary to already existing code, as it can also be treated as a shape.

C.2 Design Patterns

Design patterns are yet another useful addition to the OOP world. A design pattern offers a well-proven, structured solution to a typical design-oriented or architectural problem. Instead of “re-inventing the wheel”, it is much easier, faster and more reliable to deploy a design pattern if possible.

Gamma et al. [7] present an extensive list of important design patterns; this book constitutes as the reference literature on this topic. Still work in progress but already quite complete and well comprehensible is Eckel [6].

C.2.1 Singleton

The Singleton pattern both ensures that there is only one instance of a given class and offers a mechanism to get access to this object. The idea is to make the class itself responsible for the creation of the object. Outside objects cannot create the singleton object but may only request a reference to it using a class-wide method. If the object already exists, the reference is simply returned, otherwise it is created beforehand. Algorithm 7 on the facing page shows a typical Java based implementation of the Singleton pattern.

With minor modifications such as the addition of a counter, it is also possible to extend this pattern to provide “up to i” objects instead of only one.
### C.2. DESIGN PATTERNS

```java
public final class MySingleton {

    /* Private constructor, needed explicitly! */
    private MySingleton() {
    }

    /* Private reference to the single object */
    private static MySingleton instance = null;

    /* Public method to return a reference */
    public static MySingleton getReference() {
        if (instance == null)
            instance = new MySingleton();
        return instance;
    }
}
```

Algorithm 7: Singleton Pattern in Java

### C.2.2 Factory Method

Metsker [26] defines an operation to implement the *Factory Method* pattern if the following is true:

- The operation creates a new object,
- it returns a type that is an abstract class or an interface
- and it is implemented by several classes.

One further important aspect mentioned in the book is that the initiating part of the program does not know and even does not care which class the object to be returned is of exactly, it only has to be an implementation of a given interface or abstract class. Another design pattern, the *Iterator*, is mentioned to be a classic example for a factory method.
Appendix D

A Jiotto Sample Application

This chapter shows the complete code of the jiotto sample application described in section 5.3.

D.1 The Code

D.1.1 The Main Application

/*
 * Jiotto Sample Application (S7_SampleApplication.java)
 */
import jiotto.*;

public class S7_SampleApplication extends Initialiser {

    /*
     * Initialisation Phase ...
     */
    public void run() {

        /*
         * Jiotto Control Thread
         */
        ControlThreadSingleton ct =
            ControlThreadSingleton.getReference();

        /*
         * P O R T S
         */

        /* Set the overall numbers of different Ports. */
D.1. THE CODE

ct.setNumberOfInPorts(2);
ct.setNumberOfOutPorts(2);

/* Create the ports. */
InPort i0 = (InPort)
   ct.createPort(InPort.class, 0, 1234, "InPort-0");
InPort i1 = (InPort)
   ct.createPort(InPort.class, 1, 5678, "InPort-1");
OutPort o0 = (OutPort)
   ct.createPort(OutPort.class, 0, 1, "OutPort-0");
OutPort o1 = (OutPort)
   ct.createPort(OutPort.class, 1, 8765, "OutPort-1");

/*
 * D R I V E R S
 */

/* Set the overall number of Drivers. */
ct.setNumberOfDrivers(4);

/* Create driver guards. */
TaskDriverGuard taskGuardTrue = new S7D0D1GuardTrue();
ModeDriverGuard guardD2 = new S7D2Guard();
ModeDriverGuard guardD3 = new S7D3Guard();

/* Create drivers and configure transfers. */
TaskDriver d0 =
   (TaskDriver) ct.createDriver(
       TaskDriver.class, 0, taskGuardTrue, 1, "d0");
d0.transfer(o0, i0);

TaskDriver d1 =
   (TaskDriver) ct.createDriver(
       TaskDriver.class, 1, taskGuardTrue, 1, "d1");
d1.transfer(o1, i1);

ModeDriver d2 =
   (ModeDriver) ct.createDriver(
       ModeDriver.class, 2, guardD2, 1, "d2");
d2.transfer(o0, o1);

ModeDriver d3 =
   (ModeDriver) ct.createDriver(
ModeDriver.class, 3, guardD3, 1, "d3");
d3.transfer(o1, o0);

/*
 * TASKS
 */

/* Set the overall number of tasks. */
ct.setNumberOfTasks(2);

/* Create task functions and tasks
 * and add their ports. */
TaskFunction t0Function = new S7T0Function();
Task t0 = ct.createTask(0, t0Function, "t0");
t0.addInPort(i0);
t0.addOutPort(o0);

TaskFunction t1Function = new S7T1Function();
Task t1 = ct.createTask(1, t1Function, "t1");
t1.addInPort(i1);
t1.addOutPort(o1);

/*
 * MODES
 */

/* Set the overall number of modes. */
ct.setNumberOfModes(2);

/* Create modes and set the number of
 * task invocations and mode switches. */
Mode m0 = ct.createMode(0, 400, "m0");
m0.setNumberOfModeSwitches(1);
m0.setNumberOfTaskInvocations(1);

Mode m1 = ct.createMode(1, 400, "m1");
m1.setNumberOfModeSwitches(1);
m1.setNumberOfTaskInvocations(1);

/* Mode Switches */
m0.addModeSwitch(m1, 2, d2);
m1.addModeSwitch(m0, 1, d3);
/* Add task invocations */
m0.addTaskInvocation(t0, 4, d0);
m1.addTaskInvocation(t1, 1, d1);

/* F I N A L S T U F F */

/* Set Start Mode */
ct.setStartMode(m0);

/* Kick off the ControlThreadSingleton. 
* After a few sanity checks and some calculations, 
* the mission phase begins ... */
ct.go();
try {
    ct.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

/* This must not be reached ... */
System.err.println("Ooops: The control thread finished!");
System.exit(1);

public static void main(String[] args) {
    new S7_SampleApplication().start();
}

D.1.2 Task Drivers Guard

import jiotto.*;
class S7D0D1GuardTrue implements TaskDriverGuard {
    public boolean evaluate(
        ModePort[] modePorts, SensePort[] sensePorts) {
        return true;
    }
}

D.1.3 Mode Driver Guards

import jiotto.*;
class S7D2Guard implements ModeDriverGuard {
    public boolean evaluate(
        ModePort[] modePorts, SensePort[] sensePorts) {
        if ((modePorts[0].getValueInt() % 5) == 0) {
            return true;
        } else {
            return false;
        }
    }
}

import jiotto.*;
class S7D3Guard implements ModeDriverGuard {
    public boolean evaluate(
        ModePort[] modePorts, SensePort[] sensePorts) {
        if ((modePorts[1].getValueInt() % 5) == 0) {
            return true;
        } else {
            return false;
        }
    }
}

D.1.4 Task Functions

import jiotto.*;
class S7T0Function implements TaskFunction {
    public void f(
        InPort[] inPorts,
        ModePort[] modePorts,
        PrivatePort[] privPorts) {
        modePorts[0].setValueInt(inPorts[0].getValueInt() + 1);
        System.out.println(
            " t0: New value of o0 = "
            + modePorts[0].getValueInt());
    }
}

import jiotto.*;
class S7T1Function implements TaskFunction {
    public void f(
        InPort[] inPorts,
        ModePort[] modePorts,
D.2. VARIATIONS

```java
PrivatePort[] privPorts) {
    modePorts[1].setValueInt(inPorts[1].getValueInt() - 1);
    System.out.println(
        " t1: New value of o1 = 
        + modePorts[1].getValueInt());
}
```

D.2 Variations

In the implementation as shown above, the user space functionality such as the task functions and driver guard conditions are implemented in distinct named classes. Note that it was also possible to use Java’s anonymous inner classes, resulting in code like the following:

```java
Task t0 = ct.createTask(0, new TaskFunction() {
    public void f(
        InPort[] in,
        ModePort[] out,
        PrivatePort[] priv) {
        out[0].setValueInt(in[0].getValueInt() + 1);
        System.out.println(
            " t0: New value of o0 = 
            + out[0].getValueInt());
    }, "t0");
```

While, on the one hand, this might make the application class more complex, it might also help to avoid having to create many and often small classes on the other hand.
Bibliography


List of Figures

2.1 Models for real-time programming .......................... 7

3.1 Basic Giotto Model ........................................ 11

3.2 Relations between Giotto Components .......................... 14

5.1 Jiotto Class Hierarchy: Ports ............................ 43

5.2 Jiotto Class Hierarchy: Tasks ............................ 45

5.3 Jiotto Class Hierarchy: Drivers ............................ 48

5.4 Jiotto Class Hierarchy: Modes ............................ 50

5.5 Sample Application: Modes $m_0$ and $m_1$ ................. 58

5.6 Sample Application: Mode Switch $m_0$ to $m_1$ ................. 58

5.7 Sample Application: Mode Switch $m_1$ to $m_0$ ................. 59

6.1 Elevator: Modes, Actuator Updates and Task Drivers ......... 66

6.2 Elevator: Mode Switches .................................. 67

6.3 Elevator2: Some of the Changes Compared to Elevator ...... 75
# List of Algorithms

1. Possible memory leak in RTSJ .................................................. 23
2. How to prevent memory leaks in RTSJ ................................. 23
3. Hello Periodic Real-Time World ......................................... 27
4. Initialiser ................................................................. 35
5. Essence of `ReusableNoHeapRealtimeThread` ...................... 52
6. Linux kernel patch to extend the accepted priority range ... 82
7. Singleton Pattern in Java .............................................. 89
Index

ActuatorUpdate, 51
asynchronous event, 25
asynchronous event handler (AEH), 25
asynchronous transfer of control (ATC), 25, 59
control thread (CT), 34, 36, 59
ControlThreadSingleton, 36, 59
Debug, 56
debugging, 55
design decisions, 59
design patterns, 88
determinism, 3, 18, 21, 29, 86
Driver, 47
driver, 10, 11, 47
DriverFactorySingleton, 38
DriverGuard, 49
dummy tasks, 47

E code, 10
E machine, 10, 16, 36
earliest deadline first scheduling (EDF), 5
Elevator, 65
Elevator2, 73
Embedded Machine (E machine), 10
embedded system, 1
encapsulation, 87
exceptions, 55, 84

factories, 54
Factory Method, 54, 89
FIFO scheduling, 5, 28, 44, 80
fixed priority scheduling (FPS), 5, 24
garbage collection (GC), 18, 21, 29, 30
Giotto, 1, 9
Giotto Development Kit (GDK), 16
Giotto Micro Steps (GMS), 13, 39
GiottoViolationException, 55
guard, 12
hard real-time, 3, 9, 29, 80
heap, 21, 26, 29, 84
high-resolution time, 26, 78

immortal memory, 21, 30, 35, 46, 60, 85
inheritance, 87
initialisation phase, 30, 31, 35
Iterator, 89
JamaicaVM, 29
Java, 17
Jiotto, 2, 34, 77
JiottoViolationException, 55

kernel pre-emption, 80
late binding, 88
Linux, 28, 80, 86
logging, 55

memory leak, 22, 85
memory management, 18, 21, 26, 29, 30
mission phase, 30, 31, 36
Mode, 50
mode, 10, 12, 50
MoDECS, 16
ModeFactorySingleton, 38
ModePort, 42
INDEX

ModeSwitch, 51

NoHeapRealtimeThread, 25, 31, 36, 46, 85

object oriented programming (OOP), 87

performance, 17, 41, 78
periodic real-time thread, 24, 46
physical memory, 26
polymorphism, 88
Port, 42
port, 10, 11, 42, 63
Port subclasses, 42
PortFactorySingleton, 37
PortMethods, 44
pre-emption, 4, 5, 80
priority ceiling, 6, 26, 28, 31
priority inheritance, 6, 26, 28
priority inversion, 6, 26, 28, 57

Ravenscar-Java (RJ), 30, 59
raw memory access, 26
real-time operating system (RTOS), 4
real-time program, 1
Real-Time Specification for Java (RTSJ), 20
real-time system, 3
RealtimeThread, 24
ReusableNoHeapRealtimeThread, 51, 60, 84
ReusableThreadStillWorkingException, 54, 61
round-robin scheduling (RR), 5, 80
RTSJ reference implementation (RI), 28, 83

sample application, 57, 90
scheduled programming model, 8, 10
scheduler, 4, 81
scoped memory, 21, 35, 84
SensorsActuatorsFunction, 56, 72
SensorsThread, 56, 72, 74

setPeriod(), 38, 59
setTaskPriority(), 46
simulations, 56, 65, 74
Singleton, 88
soft real-time, 3, 80
stack, 21, 85
synchronisation, 26, 31, 57
synchronous programming model, 7, 10

Task, 46
task, 10, 12, 44, 60
TaskFactorySingleton, 38
TaskFunction, 45
TaskInvocation, 51
time-safe, 8
timed programming model, 8, 10
transfer(), 49
transferGeneric(), 47

wait-free queues, 26
worst-case execution time (WCET), 16, 78

zero-time, 7, 10, 13, 39