Hard Real-Time Java

Profiles and Schedulability Analysis

Bøgholm, Thomas

AALBORG UNIVERSITY

Hard Real-Time Java: Profiles and Schedulability Analysis

Thomas Bøgholm

ABSTRACT

This thesis presents a model based approach to the program analysis of real-time Java systems and the development of Java profiles supporting such analyses. The goal of this work is to enable the development of safety-critical embedded systems in high level modern languages; languages which more elegantly capture the complexities of modern embedded systems. This work is twofold and concerns both tool development and the design of a real-time Java profile named *Predictable Java*. Throughout this project the schedulability analysis tool SARTS is developed, and then improved.

The thesis is based on six primary papers: The first paper presents the SARTS tool, a schedulability analysis tool, which among other things consists of a translator from Java bytecode to timed automata. The second paper presents design and implementations of the *Predictable Java* profile, and a constructive criticism on the upcoming Safety Critical Java(SCJ) specification. In the fourth paper, the complete reconstruction of SCJ and its predecessor the Real-Time Specification for Java(RTSJ), and the resulting four simple profiles, is presented. The third paper presents an improvement of SARTS is developed which includes the analysis of Java finalizers, made possible by the earlier work on predictable Java. In the fifth paper, a language for specifying abstract system specifications is designed, along with a techniques for checking implementation relations and schedulability for the full system; a step towards compositional schedulability analysis. The last paper describes the current state and recent work on real-time Java.

DANSK SAMMENFATNING

Denne afhandling præsenterer en modelbaseret tilgang til programanalyse af real-tids systemer og udviklingen af Java profiler, der støtter op om sådanne analyser. Målet med dette arbejde er at gøre det muligt at udvikle sikkerhedskritiske systemer i moderne højniveausprog; sprog der mere elegant kan håndtere kompleksiteten af moderne indlejrede systemer. Dette arbejde er todelt og omhandler både udviklingen af værktøjer og et design af en real-tids Java profil kaldt *Predictable Java*. Igennem hele projektet bliver analyseværktøjet SARTS udviklet og forbedret.

Denne afhandling består af 6 primære artikler: Den første artikel præsenterer SARTS værktøjet, et skeduleringsværktøj der blandt andet består af oversætter fra Java byte-code til tidsautomater. Den næste artikel præsenterer design og implementationer af profilen *Predictable Java*, som er et modspil til den kommende Safety Critical Java(SCJ) specifikation. I en senere artikel, bliver en komplet rekonstruktion af SCJ og dens forløber Real-Time Specification for Java(RTSJ), og fire simple profiler som er resultatet af denne rekonstruktion, præsenteret. En artikel præsenterer en forbedring af SARTS, som inkluderer analyse af Java finalizers, muliggjort af det tidligere arbejde med Predictable Java. I en anden artikel bliver et sprog til abstrakte systemspecifikationer designet, sammen med teknikker til at tjekke implementeringsrelationer og skedulerbarhed af det fulde system; et skridt imod kompositionel skedulerbarhedsanalyse. En sidste artikel præsenterer nuværende og tidligere arbejde med realtids Java.

Contents

Chapter 1

Introduction

Embedded systems play an important role in most of the technology surrounding us today. They are for example the invisible computer systems in the traction control system of a modern car, the control system of a power plant, the control systems of the water pumps and the power metres in our houses. They also play a major role in small devices, such as hearing aids and pace makers.

Embedded computers and software differ greatly from the personal computer and office software we know and normally perceive as *computer systems*. Embedded computer systems are themselves a part of a larger system, often referred to as an *intelligent system* or *cyber physical system*. Cyber physical systems contain embedded subsystems, and thus depend largely on their correct behaviour. Correct operation is especially important since most embedded systems are mass produced, and have little or no possibility for software updates or other maintenance after being deployed [86].

This puts great emphasis on software dependability because embedded systems are often mass produced, they often carry out vital tasks inside heavy machinery, medical equipment etc., and because embedded software often cannot be updated as bugs are being detected. Therefore, even small errors can be costly or endanger human lives; there are several historical examples of how even subtle errors can cause disastrous situations:

Therac 25, 1985-1987 The *Therac 25* radiation device, where a race-condition, a subtle error triggered by people typing too fast, resulted in the death and severe injuries of several people. A detailed investigation of the software related errors is presented in [96].

Ariane 5 Flight 501, 1996 The maiden flight of the Ariane 5 rocket, where the rocket exploded 39 seconds after lunch, delaying the project by one year, costing approximately $400 million. This explosion was caused by an error in the system: an exception raised during the conversion from

64bit floating point to a 16bit signed integer; [92] presents an analysis of the Ariane 5 accident.

Car recalls Car companies are some of the major consumers of embedded systems technology; it is estimated that up to 20% of the systems produced are used by the car industry. Therefore it is not surprising that faulty systems occasionally make it to the market, affecting a great number of people, and such problems have in fact resulted in both problems and product recalls. Some examples from the media include [84]:

- Toyota recalls 75,000 Prius hybrid cars in 2005. The engine would shut down due to a software error [111],
- in 2003 There was an example of a computer crash in a BMW. Unfortunately, the finance minister of Thailand was inside the car when it happened, and a guard was forced to break a window to let him escape, since the windows, the doors, and air conditioning did not work [151],
- in 2002, BMW recalls the 745i. The fuel pump would stop working if the fuel tank was less than one third full,
- in 2001, 52,000 Jeeps were recalled due to software error.

Furthermore, a large recall of up to 6.5 million vehicles and the discontinuation of eight models by Toyota in 2010 due to *uncontrolled acceleration* problems are suspected to be related to embedded system failure, although Toyota is claiming it to be a problem with the door mat catching the pedals[3].

These are of course some of the most published cases of malfunctioning systems. They are used here to illustrate how subtle bugs can manifest themselves in a larger system. More common, however, are noncritical crashes of network switches, mobile phones, dish washers etc., errors often solved by restarting the device and most often resulting in only some inconvenience for users the involved.

However, being able to reduce the amount of errors in embedded systems will lead to higher quality products and lower recall rates, and will therefore give a company a competitive advantage and increased revenue.

Real-time embedded systems A subgroup of embedded systems are also subject to *real-time requirements*. Real-time systems are systems where not only functional correctness, i.e. correct computation, is important, but also the time needed for the computation. In real-time systems, the *timeliness* of the computation is just as important; if the result of a computation is not available within a set amount of time, it may result in a system failure.

In the example of a modern car, the traction control system has a real-time requirement in that when measurements of a wheel indicate a loss of road grip, an action must be performed immediately to prevent the car from sliding. It is not enough to calculate how to prevent the car from sliding, variables must be calculated and the result ready for processing, such that an action can be taken before the car actually slides.

Therefore, with the increased integration of embedded systems in our society and the demand for increased functionality, the complexity of these systems increases accordingly. This increase in complexity, and the fear for the aforementioned failures, calls for new techniques and tools for embedded systems development, as the use of traditional development techniques and tools for the now very complicated systems are becoming increasingly challenging.

This is partly caused by the programming languages in use in systems development today. The languages such as C and assembly, which are often used for todays systems, do not support the capturing and structuring of the complexities in modern systems, as well as modern programming languages.

Java is an example of a very popular modern language, taught in todays computer science classes, and used in a variety of large scale projects. Java is an object oriented language, giving an advantage over purely imperative languages like C when programs grow large, and was designed as an improvement over the existing popular languages. It is especially inspired by C/C++ syntax, but differs in in some important aspects; it was designed to result in *simpler programs*:

- A large difference is the allocation and handling of memory. In Java, programs will never run into problems caused by dangling pointers; pointers to deallocated memory. Allocation and deallocation is automatically handled, usually by a garbage collector, making dangling pointers impossible.

- Java is concurrency aware and have the notion of synchronized methods and regions. This is a large improvement over C/C++, where concurrency is achieved only through libraries, of which many implementations with varying semantics across platforms exists, hampering portability.

- Java is single inheritance, resulting in a simpler inheritance semantics; However, in Java, classes are allowed to implement multiple *interfaces*, mimicing multiple inheritance.

- Another advantage is that, because of these differences along with the Java standard library, productivity in Java is increased, fewer bugs are introduced, and it is easier to debug; [117] presents a study comparing the two languages based on bugs and productivity.

Another reason to consider Java is that this language is becoming popular in universities, and is to many students their first and primary programming language. This results in a situation with a shortage of C-programmers and an abundance of Java programmers, and with the increased interest in embedded systems development, finding qualified programmers in the languages used today may be prove difficult; in [106], the authors presents a view on this discussion.

1.1 Java for Embedded Systems

Java is a popular language for large scale systems, however, many of the features that make Java a language suitable for large projects are also features that make it unsuitable for embedded systems development. Embedded systems are often special purpose devices, and their interfaces are not as well defined and polished as the devices we use on personal computers. Programming such systems often require direct memory access in order to communicate with actuators and sensors. This is one of the reasons why low-level languages like C are used for such systems, as their low-level nature easily provides access to the underlying hardware.

In Java there is no direct access to the underlying hardware, as Java is built for a virtual machine, acting as an extra layer between the code and the hardware executing the program. This requires a virtual machine implementation to run on small devices, requiring additional memory, space and processing power.

The abstract layer exposed to the programmer also makes it difficult to access hardware and memory directly in standard Java, however, techniques such as *hardware objects* [133] have been suggested to allow hardware access in a structured fashion.

This abstract layer also makes reasoning about execution time very difficult, as multiple actions in multiple layers are required for each Java instruction. The difficulties in execution time reasoning are further complicated by the garbage collection based memory management of Java. Garbage collection makes reasoning about execution time very difficult, since the garbage collector can, in principle, run at any given time interrupting the program for an unspecified amount of time, i.e. the time it takes to search for dead objects and releasing the memory. To the contrary, in low-level languages, the execution time is more directly visible, as the program is translated into instructions executed directly by the hardware; another reason why low-level languages are still used extensively. In recent years, however, much research has been put into real-time enabling the high-level programming language Java, starting with Real-Time Specification for Java (RTSJ) [35, 37], and later focusing on safety-critical and embedded systems in Safety Critical Java (SCJ) and related research [137, 130, 76, 91].

There are three main areas which needs work for improvement for Java to become more suitable for safety critical embedded systems:

Language The development of a *Java Profile*; a tweaking of the Java language in a way, such that programs written in this profile will be suitable for program analysis,

Tools Analysis tools for validating programs written in the defined profile. These tools will be used to check overall properties properties such as system schedulability, but also that programs are within the boundaries set by the profile.

Platform The development of a predictable execution platform with minimal footprint, making it suitable for small devices, and predictable memory allocation,

This thesis will focus on the two first points, **Language** and **Tools**, while utilizing recent work on the last point, **Platform**.

Chapter 2

Technical Background

This chapter presents the technical background for the papers presented later in this thesis. The first section gives a short introduction to real-time systems, followed by the definition of a real-time model as a way of describing real-time systems. Section 2.3 presents real-time systems development in Java, first the real-time specification for Java, followed by a summary of the upcoming safety critical Java specification. Section 2.4 gives an overview of real-time scheduling techniques, then a section on real-time systems analysis in section 2.5, followed by a section on real-time model-checking techniques and tools in section 2.6. This chapter concludes by discussing the present and future state of real-time systems analysis.

2.1 Real-time Systems

This chapter gives an overview of real-time systems design and defines the terminology used in the rest of this thesis. First the types of real-time systems are shortly introduced, then a task model used to reason about such systems is presented along with real-time scheduling principles.

There are three types of real-time systems, differing in the importance of timeliness and the consequences of a missed deadline.

Soft real-time systems are systems where the timeliness is least important, and where a missed deadline is merely an inconvenience, such as a user waiting for a response to the action he just performed. This could be the interface on a mobile phone.

Firm real-time systems is a subclass of soft real-time systems, where deadline misses are tolerated, and the result of a delayed calculation provides no value to the system, and is therefore discarded. An example of this is a media streaming system. If a frame is delayed a few seconds, it will be dropped and not displayed.

Hard real-time systems are the strictest of the three, where a deadline miss is considered an error. In a hard real-time system such an error may be disastrous, e.g. a missile defence system, missing an incoming attack.

Often, systems are compositions of subsystems in some of these categories. It is important the hard real-time system components do not suffer from interference from none or soft real-time components [45]. Depending on whether the class of system developed, different techniques apply, especially in choosing scheduling algorithm and analysis techniques. In this thesis, we are only concerned with **hard real-time systems**.

2.2 Real-time Systems Model

This section defines a model used to describe real-time systems. This model is used in the design phase of the development and is the input for some verification techniques. In this model, a real-time system is expressed as a group of tasks, with task properties, such as deadline, and execution time. Tasks are released, i.e. their execution is started, according to a release pattern, which is either periodic or aperiodic. This release pattern continue until system shutdown or possibly forever. A special case of the aperiodic release pattern is introduced, as an aperiodic pattern with a minimum inter-arrival time constraint, known as the sporadic release pattern. This constraint makes reasoning about aperiodic tasks in a hard real-time environment possible, as the release rate is bounded by the added constraint. It would otherwise be hard to include a guarantee that all tasks would meet their deadline, since a short time burst of aperiodic task releases may delay all the tasks of a lower priority.

The basic parameters of tasks in this model are:

- D, the deadline relative to the task release,
- T, period, in case of a periodic task, or the minimum inter-arrival time in case of sporadic tasks,
- C, the execution cost of the task, or Worst Case Execution Time (WCET) of the task, and
- B, the worst case blocking time experienced by a task.

The two first parameters, D and T are part of the specification of the tasks, and come from the problem domain. For example, in a missile defense system, there is a hard deadline on the task calculating how to shoot down an incoming missile, which may be calculated based on assumptions on the distance a missile can be detected, and the time before impact. Such assumptions will also be the base of the choices of periods or minimal inter-arrival time.

The execution time C is the result of a worst case execution time analysis, a conservative estimate of the maximum time needed to execute the task, in isolation without any interruption from other tasks; a good overview of techniques and tools for WCET analysis is provided in [167].

B, the worst case blocking time is a conservative estimate of the maximum blocking a task may experience, in there case of shared resources between tasks. This is an upper bound, and it is not certain this blocking will be experienced at all. A task is blocked if it, according to the scheduling strategy, is the task with the highest priority, however, its execution is prevented by a lower priority task holding a resource.

Before presenting scheduling principles and related algorithms, a notation for presenting the execution of real-time systems in this model is presented in the next section.

2.2.1 Time Line Notation

To be able to reason about tasks and their execution patterns, we define a graphical notation of task executions. This graphical representation of task release and execution patterns will be used throughout this thesis.

This time line notation, inspired by [45], is depicted in Figure 2.1.

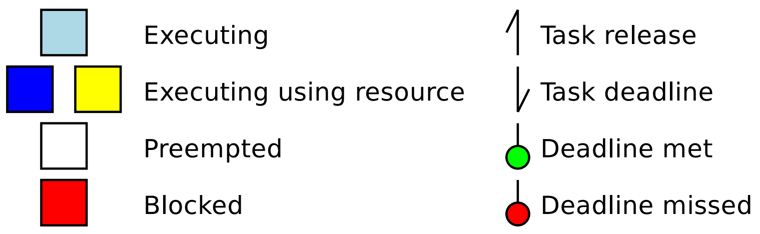


Figure 2.1: Time line notation

In this terminology, *when a task is*:

- Executing: the task is exclusively using the processor.
- Executing using resource: the task is exclusively using the processor holding the resource represented by the colour.
- Preempted: the task is interrupted by higher priority task.
- Blocked: the task is unable to run because a lower task holds a resource and has elevated priority.

The time points can be of the following types:

- Task release: the task is started, meaning that it is either executing, executing using resource, preempted, or blocked.
- Task deadline: the task must have finished execution by this time.
- Deadline met: the task finishes its executing before the deadline.
- Deadline missed: the task failed to finish its execution before the deadline.

Consider the example in Figure 2.2. In this example there are two periodic tasks, Task 1 and Task 2, with deadlines 4 and 17, and periods 9 and 18 respectively. Task 1 has the highest priority, starts by executing for two time units with exclusive rights to the resource marked as the yellow resource, and then executes for one time unit, finishing before reaching its deadline. Task 2, the lower priority task, starts out preempted for three time units, while Task 1 is executing; then executes for five time units, before entering a critical section using the yellow resource for three time units, blocking Task 1 for two time units since Task 1 is trying to enter a critical section using the resource. At 11, Task 2 is preempted by Task 1, which executes as in last release, but misses its deadline at 13.

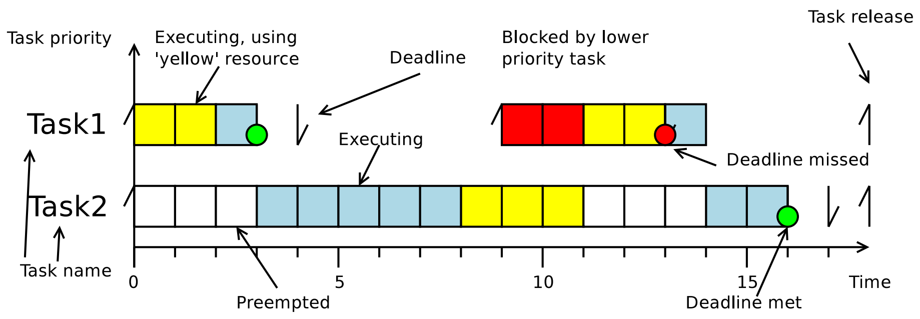


Figure 2.2: Time line example

2.3 Real-time Programming Languages

Embedded systems are today mainly programmed in the C programming language, or in a variant designed especially for embedded systems. This is because C, while being more abstract than assembly languages, allows low-level access to the underlying hardware and it allows the production of fast programs with low memory overhead. Its semantics are, however, influenced by the hardware on which it is executed and there is no language support for concurrency [124]; these are just two of the many reasons why C is not the best language choice for critical systems.

Java is a modern language suitable for complex systems development, however, it is not very suitable for real-time embedded systems, especially due to lack of predictability in execution time. In recent years, however, Java has received much attention from the real-time community, and is being suggested as a new alternative to the traditionally used languages.

The Real-Time Specification for Java (RTSJ) was the first standardisation project working with standardising a real-time version of Java. RTSJ was proposed in 1998 as JSR-1[1], and the initial specification was accepted by Java Community Process (JCP) in 2000. Since then, a number of bugs and inconsistencies were discovered, and six years after the initial version, version 1.02 was released June 2006. The requirement for compatibility with the first version by the JCP, resulted in a new proposal for RTSJ 1.1 was suggested in 2006, as a new specification under the name JSR-282 [53], allowed to break compatibility with the original specification. The newest version of this was released in 2009 [54].

The RTSJ received a great deal of attention in the real-time community, and it became apparent that the RTSJ was far too dynamic and expressive for high integrity systems [129, 122, 137, 91].

This lead to the work on smaller real-time profiles based on RTSJ, restricted with a smaller set of available, such that programs written are easier to analyse using static analysis techniques and thereby more likely to be certified. The work on such profiles relate to what has been previously done for Ada to enable the development of certifiable safety-critical systems. This work has resulted in Ada profiles such as SPARK [48, 18] and the Ada Ravenscar profile [56].

2.3.1 The Real-time Specification for Java

The real-time specification for Java was proposed to allow real-time programming in Java, as standard Java lacks the properties needed for a real-time language. Predictability in particular was an issue, however, it is an attractive language for real-time and embedded systems programming. Java is an attractive language because it by design offers safety, object orientation, language level concurrency, and of course the write once, run everywhere philosophy of Java; features that all are attractive for such systems.

Standard Java, however, is not designed with real-time or critical systems in mind. In fact, some particularities, for example thread behaviour, are intentionally specified weakly [36], when designed by James Gosling in 1995. This was done in order to achieve high portability, which is one of the main design criteria for Java.

RTSJ strengthens the Java specification through refined semantics and a number of classes. The design philosophy behind RTSJ is based on cri-

[1]`http://goo.gl/vkRLm` (`http://jcp.org/aboutJava/communityprocess/mrel/jsr001/index2.html`)

teria such as backwards compatibility, predictability, standard Java syntax, etc. However, in some cases these criteria are conflicting; an example is backwards compatibility which suffers in order to provide predictability, e.g. standard Java threads have been changed to allow the design of a predictable extension. The RTSJ extensions are residing in the `javax.realtime` package, and provides improvements in six important areas [36, 163]:

1. **Real-time threads** Real-time threads are introduced in the RTSJ due to the semantics of standard Java threads [36] being weakly specified, intentionally, to ease potability to other systems. Two new thread classes are introduced: `RealtimeThread` and `NoHeapRealtimeThread`. These, and their relationship to standard Java is shown in Figure 2.3. The latter of the two, `NoHeapRealtimeThread`, is not allowed to access heap memory, and can thus be allowed to safely preempt the garbage collector, as the heap state will always be consistent.

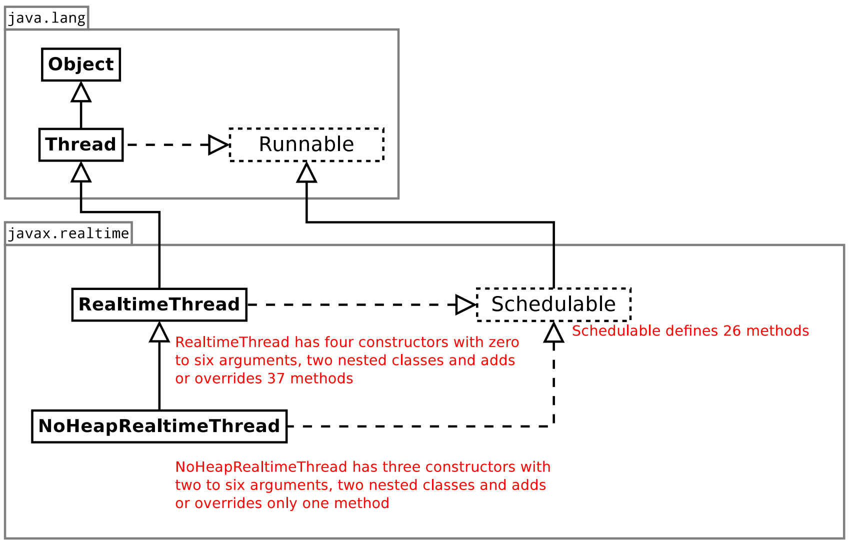

java.lang
Object
Thread
Runnable
javax.realtime
RealtimeThread
Schedulable
Schedulable defines 26 methods
RealtimeThread has four constructors with zero to six arguments, two nested classes and adds or overrides 37 methods
NoHeapRealtimeThread
NoHeapRealtimeThread has three constructors with two to six arguments, two nested classes and adds or overrides only one method

Figure 2.3: RealtimeThread classes in RTSJ

The `RealtimeThread` extends `java.lang.Thread` further by adding `ReleaseParameters` and `MemoryParameters` to allow implementation of periodic threads and other common thread types. The real-time threads implements the newly introduced `javax.realtime.Schedulable` interface, and defines 37 new methods and 4 constructors implementing RTSJ thread related capability, such as a refined thread group concept in the case of scoped memory, overrun handlers, and asynchronous interruption.

2. **Scheduling** RTSJ adds a number of classes in the area of scheduling in order to provide real-time facilities to Java. An overview of the

classes defined related to scheduling can be seen in Figure 2.4. The `Schedulable` interface, extending the `java.lang.Runnable` interface, is introduced to generalize the concept of schedulable objects making RTSJ independent of the thread concept, and making it possible to support any type of concept other than threads, e.g. tasks. It is worth noting that this interface is quite elaborate, defining 23 methods to be implemented.

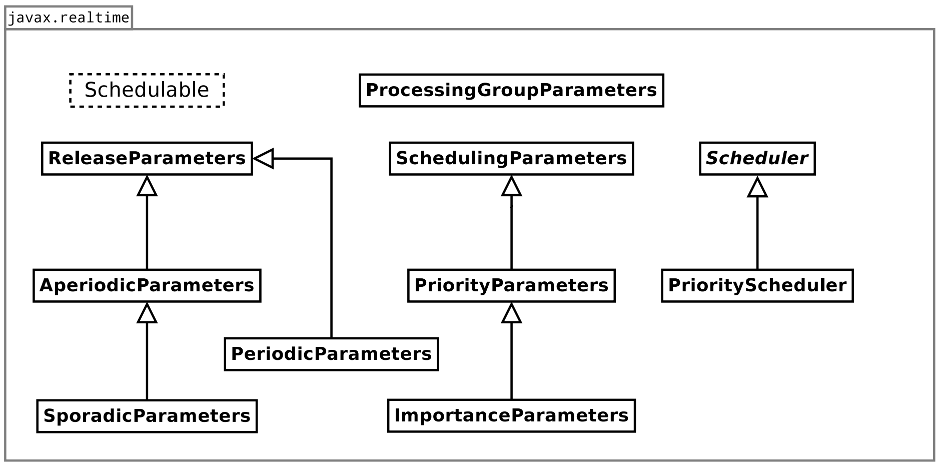


Figure 2.4: Scheduling related classes in RTSJ

Apart from implementing the `Schedulable` interface, a schedulable entity must have information on release criteria and other scheduling requirements. This information is contained in the various `parameters` class hierarchies of Figure 2.4:

`ReleaseParameters` which describes the release-pattern of a schedulable object, with the default implementations: `AperiodicParameters`, `SporadicParameters`, and `PeriodicParameters`, parameters for each type of real-time thread or task; parameters such as sporadic parameters hold information about minimal inter-arrival time, and an exception will be thrown in case of violation of this property;

`SchedulingParameters`, which contains information used by the scheduler; default implementations are `PriorityParameters`, determining the priority of a schedulable object and used when execution order is determined by a single integer, and the subclass `ImportanceParameters`, which adds an importance parameter, an extra priority parameter used in the case when two tasks share the same priority level.

Scheduling in the RTSJ is intended to be as flexible as possible. Therefore it should be possible to add new scheduling algorithms; The specification [36] even mentions dynamic loading of scheduling policy modules, in future versions of the specification. It is the intention to allow new scheduler implementations through extension of the abstract `Scheduler` class, however, RTSJ does not make this possible as an RTSJ user, but this should be part of a specific RTSJ implementation.

A minimum RTSJ implementation must provide at least one scheduler implementation, the `PriorityScheduler`. This is a fixed priority preemptive scheduler with 28 unique priority levels. Of the current RTSJ implementations from Sun (Java RTS), Aicas (JamaicaVM), Timesys, and IBM (WebSphere Real-time), the `PriorityScheduler` is the only scheduler implementation available.

3. **Memory management** There are several memory related classes in RTSJ, and they are purposed to allow different types of memory allocations in different parts of a program, with the main types being immortal memory, heap memory, and scoped memory; an overview of the memory management classes is shown in Figure 2.5. The classes

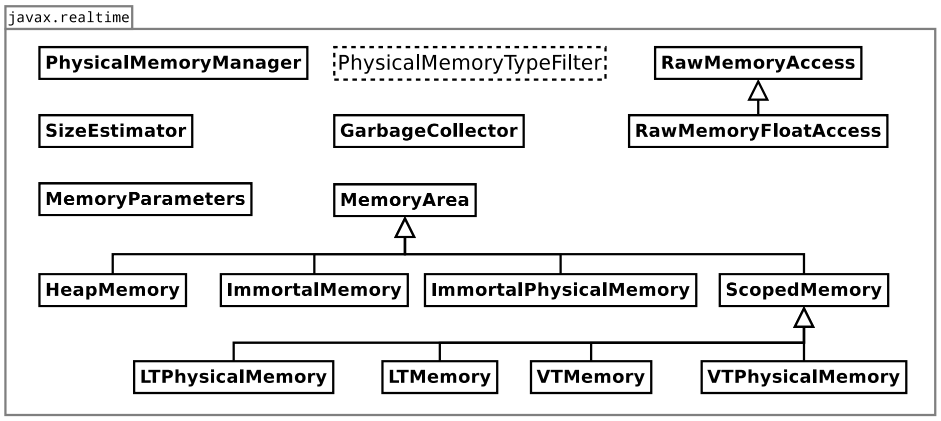


Figure 2.5: Memory related classes

of the `MemoryArea` hierarchy provides the functionality to allocate objects, each with different semantics. The `MemoryArea` provides methods for allocating objects inside the area, querying consumed memory in the particular area, getting the total size of the area, and different methods used to execute logic using the memory as allocation context. That is, objects allocated using `new` in the logic executed in the area, including libraries used, are allocated inside the memory area and are deallocated when the area is closed.

ImmortalMemory Objects allocated in this area will never be deallocated or garbage collected, and hence objects in immortal memory may be referenced from every thread and schedulable object in the program.

ImmortalPhysicalMemory This is a separate class with the same properties as `ImmortalMemory`, however, it uses a specified physical address space.

HeapMemory The `HeapMemory` class allows logic non-heap allocation contexts execute and allocate in heap memory, using the method `executeInArea(Runnable logic)`.

ScopedMemory This is a type of memory area differing most from ordinary heap memory known from standard Java. This hierarchy provides an areas with limited life time, freed when no schedulable object has access to objects inside. The scoped memory areas are advantageous from a performance and predictability point of view in the sense that no garbage collection is needed, and no deallocation of objects is performed. The scope is simply torn down as the data within a stack frame. However, it is a very challenging, error prone, and unusual programming model for ordinary Java programmers [119, 78, 118, 38]; techniques and patterns have been suggested in [118, 38], to ease the use of scoped memory areas. The main issue with scoped memory areas is that they introduce the possibility of dangling references; this is not allowed in Java, and therefore run-time checks are needed to ensure that no dangling references exist. These runtime checks are performed at every memory operation and exceptions are thrown at illegal access. An overview of memory areas and references between areas is shown in Figure 2.6; only references to objects of an area with longer lifetime are allowed.

There are four types of scoped memory areas specified by RTSJ, these are:

- Linear time scoped memory: `LTMemory` and its physical memory variant, `LTPhysicalMemory` which specify the use of a given physical address space range. These areas guarantee allocation in linear time in the size of the allocated object.
- Variable time scoped memory: `VTMemory` and its physical variant `VTPhysicalMemory`; these areas do not specify bounds on allocation time.

Additional memory classes for raw memory access are provided, giving direct access to a fixed sequence of bytes in physical memory. Access set/get methods are provided for most primitive types, including float types in the specialized `RawMemoryFloatAccess` class. Objects cannot be allocated this way, as this would be unsafe [36] because the memory it is not managed.

Some classes not directly associated with allocation are provided:

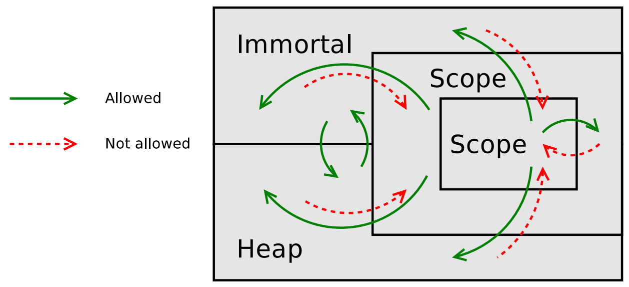


Figure 2.6: Rules for references in the RTSJ memory hierarchy

- `SizeEstimator`: provides size estimates of objects.
- `GarbageCollecter`: provides information about garbage collector overhead and temporal behaviour, and allows limited capabilities to alter the garbage collection algorithm.
- `PhysicalMemoryManager`: responsible for physical memory access, and should not be implemented by the RTSJ user, but rather equipment manufacturers or the like. An additional interface related to this is `PhysicalMemoryTypeFilter`, describing memory characteristics of devices, and is to be used only by the physical memory manager.

4. **Time** There are five time related classes in RTSJ, depicted in Figure 2.7. One `Clock` class used to access time values, and three time representations generalized in the `HighResolutionTime` class:

   High resolution time The class `HighResolutionTime` represents time with nanosecond granularity. This class is used as frequent as possible throughout the API and its subclasses will be used appropriately according to the time they represent. The direct subclasses are `RelativeTime` and `AbsoluteTime`. These time-representations has an associated clock-object which determines how the time is interpreted e.g. in timers.

   Rational time The class `RationalTime`, is represents intervals divided into subintervals, and is supposed to represent period in periodic events, but allow for some jitter. The use of this type of time representation has been subject to much confusion and as of RTSJ 1.0.1 this class is marked as deprecated.

5. **Asynchrony and timers** RTSJ provides the ability to bind logic to internal and external events, and the ability to perform asynchronous transfer of control. The class `AsyncEventHandler` contains logic executed when events occurs, and have much the same parameters as

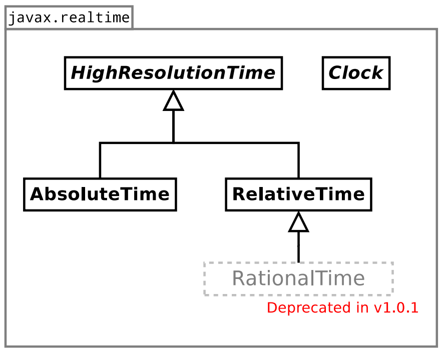


Figure 2.7: Time related classes

`RealtimeThread` classes, such as associated memory area, release parameters etc., and it also implements the `Schedulable` interface. When events occur the handler is dynamically bound to a real-time thread, though a specialized version, `BoundAsyncEventHandler` has this execution context preallocated.

Asynchronous event handlers are fired by asynchronous events, represented by the `AsyncEvent`. The `AsyncEvent` class can be used directly to fire an event handler, or the `Timer` subclasses, which are the `PeriodicTimer` or the `OneShotTimer`.

RTSJ introduces the concept of asynchronous transfer of control, which will immediately take control from a running schedulable object. This is achieved using special exceptions thrown inside a thread, by running thread.interrupt(), terminating the currently executing method, and not continue execution, even after handling the event. As this could cause an inconsistent state, threads will only be interrupted, i.e. accept the transfer of control, if it is currently executing method declaring that it throws the `AsynchronouslyInterruptedException`. There are some special semantics used in these types of exceptions. After catching the exception, the program must manually stop the propagation of the exception using `AsynchronouslyInterruptedException.clear()`.

6. **Standard Java** Backwards compatibility to standard Java programs is one of the guiding principles behind the design of RTSJ, however, some modifications to standard Java classes are required in order to do a proper implementation. The need for modifications is due to the introduction of Real-time threads in RTSJ which inherit from the Java thread class, `java.lang.Thread`, and override the set/getPriority methods. The most important difference affecting standard Java is the requirement that priorities set by `Thread.setPriority` must not violate the synchronization protocol in use: priority ceiling or priority inheritance, controlled by the classes `PriorityCeilingEmulation`

and `PriorityInheritance`. It is interesting to note that in the family of `RealtimeThread` descendants, `setPriority` will throw IllegalArgumentException in case the set priority is out of the currently allowed range[2]. Additionally, `setPriotiy` may throw `ClassCastException` if the thread is a real-time thread and its `SchedulingParameters` object is not an instance of `PriorityParameters`. An overview of these changes can be found in the Standard Java Classes section of the RTSJ 1.0.2 specification[37].

Java thread groups, `ThreadGroup`, are changed to adhere to the memory management of RTSJ. Changes are needed because a root ThreadGroup object resides in heap- or immortal memory, and from that ThreadGroup objects are created in a tree like manner. From this follows:

- A thread group cannot be created in scoped memory, since references to scoped memory are not allowed from outside the scope.
- No thread created in a scope can be part of a thread-group, and thus holds a `null` thread group reference.
- `NoHeapRealtimeThread` can not be a member of a thread group.

This has some interesting consequences for threads and thread groups; the interested reader is referred to the Standard Java Classes section of the RTSJ 1.0.2 specification. It seems such changes may seriously affect the backwards compatibility principle, however, this effect is to be expected and the changes somewhat justified. One could argue, however, that such changes are dangerous, as there will be programmers who are firm in Java, but new to RTSJ – and after all, an argument often used in the discussion about Java for real-time systems, is the large number of Java developers.

Apart from these changes, classes for handling monitor control i.e. either priority inheritance or priority ceiling, other general system settings, and tools such as wait-free queues etc. are also part of the RTSJ. Also, there are 18 new exceptions which have been added to RTSJ.

2.3.2 Ravenscar Java

The real-time specification for Java has been criticised for being too large and dynamic for high integrity systems, and work in the direction of defining analysable and safer subsets of the RTSJ have since been done. Inspired by the Ravenscar profile for Ada [56], which has become the *de facto* standard in the high integrity system domain [91], the Ravenscar Java profile [91] is

[2]A set of threads in RTSJ may have different schedulers, so the allowed priority-range may vary for each thread in the system.

a Java profile designed for high-integrity systems. It alleviates some of the problems of the RTSJ, and is an attempt to design a Java profile resulting in more reliable systems, following a set of software guidelines used by the U.S. Nuclear Regulatory Commission [1], and provides predictability in the areas of memory utilization, timing, and control/data flow.

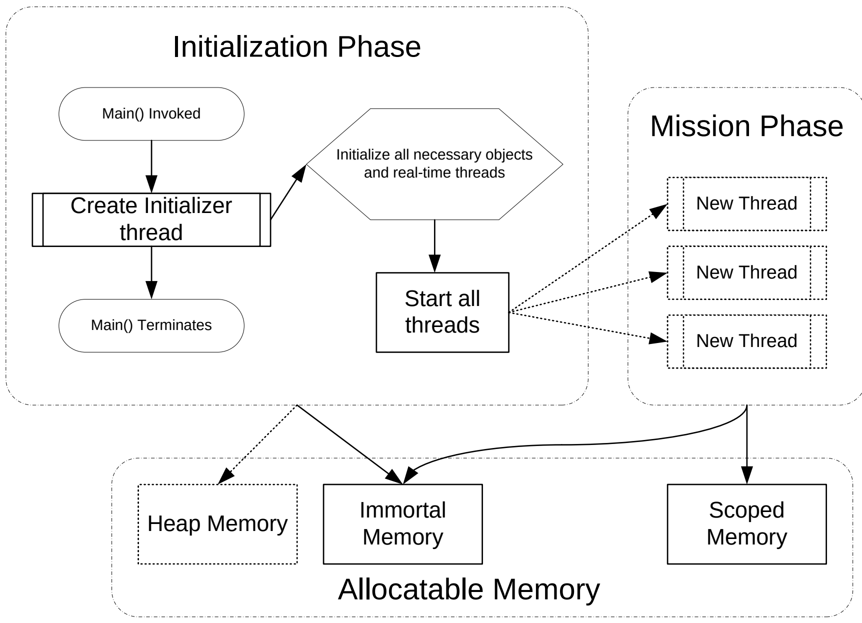


Figure 2.8: Execution phases in Ravenscar-Java [91]

The computation model offered by the Ravenscar profile is divided into two phases: the initialization phase and the mission phase, which are depicted in Figure 2.8. The initialization does not have real-time requirements, and is responsible for performing initialization of data and threads, where the mission phase, is the critical phase of the program subject to real-time requirements, where the system will be spending most of the time. In a mission, there are sporadic or periodic threads subject to fixed priority preemptive scheduling.

To simplify the memory model, the use of garbage collector in Ravenscar is forbidden, and only the linear time memory and immortal memory is offered, as defined by RTSJ. Furthermore, memory areas cannot be shared between tasks and cannot be nested.

To simplify the program structure and flow, the profile:

- Requires that all classes defined in the program must have constructors performing initialization of all class-fields.

- Forbids the use of asynchronous transfer of control.

- Puts restrictions to the use of continue and break.
- Disallows the use of the wait and notify methods.
- Requires that all for-loop constraints are statically defined.

Issues The Ravenscar Java profile builds on the body of knowledge gained in the high-integrity Ada community, and certainly defines a more analysable subset of RTSJ, suitable for high-integrity systems. The approach, however, builds on the RTSJ and through class inheritance inherits some of the complexities instead of hiding these through composition. Moreover, some of the enhancements may result in more complex programs, as a mixture of requirements, such as deadlines and periods and parameters for execution time cost. These are pointed out in [137]. Additionally [144] points out inconsistencies and weaknesses in the semantics, and questions the notion of deadline missed handlers, and furthermore suggests the *RAVENSCARlet*, which is a concept analogous to the MIDlet of Java Micro Edition (Java ME).

2.3.3 The Safety-Critical Java Specification

This section provides an overview of the Safety Critical Java Specification, based an early draft from August 2008[3], as some of the work in this thesis is based on this version. Later, some differences between this and the latest draft, version 0.79, May 16 2011, are highlighted.

The Safety Critical Java specification is a Java profile proposal, aimed at development of safety critical embedded applications certifiable under the DO-178B/ED-12B[125, 126] specification[4], level A.

DO-178B Specification

The DO-178B specification is a specification used in the certification process in the aviation industry. It contains five levels rating from A to E, where the levels A, B, C are critical levels, and the levels D, E, are noncritical levels. The levels are informally described in [103] as:

Level A Errors at this level are *catastrophic* and will prevent continued safe flight and landing.

Level B Errors at level B are *hazardous* potentially causing fatal injuries to a small number of the aircrafts occupants.

Level C At this level, errors are a *major failure* condition, resulting in discomfort and possibly injuries to aircraft occupants.

[3]Safety Critical Specification for Java, version 0.5, August 2008

[4]ED-12B is the European version of DO-178B.

Level D A non-critical level where errors are *minor* and do not reduce aircraft safety or functionality.

Level E At this level, errors have *no* effect on aircraft operational capability.

The three critical levels, A, B, C, are focused upon in SCJ.

SCJ Introduction

SCJ, while being based on RTSJ, puts additional constraints on the Java language. SCJ inherits functionality from RTSJ, but constrains some of the RTSJ complexities through the use of annotations, such as the `SCJAllowed` annotation, which, for example, specifies the level e.g. a method is allowed. In SCJ the following areas are changed:

Concurrency and memory Many new restrictions are put on allocation, heap usage and concurrency to simplify program analysis. These restrictions are common in the safety critical community [103, p. 6]. These restrictions are based on RTSJ (version 1.0.2) and Java version 5. A safety-critical application will be able to execute correctly on an RTSJ compliant platform with Safety Critical Java libraries.

Classes The new classes provided by SCJ will be implementable using RTSJ, however, they will be hiding much of the complexities from RTSJ.

Annotations SCJ comes with annotations, used to restrict memory management and concurrency to aid analysis. They are used to document programs with programmer made assumptions.

The levels A,B,C in the DO-178B specification corresponds to three levels defined by SCJ, but denoted level 0,1,2 respectively.

SCJ Overview

To facilitate certification, SCJ describes a more restrictive programming model than RTSJ, especially with regards to concurrency and memory usage. At its top level, SCJ introduces missions and safelets, both concepts are new and not part of RTSJ.

Safelets The logic of an SCJ program is implemented in terms of missions, as a single mission or as a sequence of missions. As the entry point to an application, SCJ uses safelets, a concept very similar to that of midlets in J2ME. This is different from RTSJ, which uses an entry method, `main`, as do standard Java applications.

The `Safelet` interface defines two methods, the `getLevel` method, which returns the current level of the application, and the `getSequencer` method, which returns an implementation of the abstract `MissionSequencer` class.

The mission sequencer will start, shutdown and switch between missions, and is itself a `BoundAsyncEventHandler`. This, among other things, means the initial allocation context can be non-heap memory; heap-memory would be used when executing the main method.

Missions The mission concept is introduced as a balance between flexibility and restrictions in order to provide limited startup, shutdown, and mission restart. In SCJ, a mission is simply a container for handlers.

During during the life of a mission it enters different modes, as depicted in Figure 2.9.

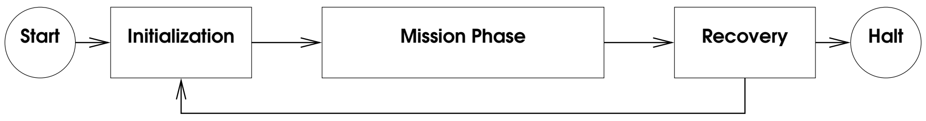


Figure 2.9: SCJ mission modes [104, p. 30]

Missions start out in an initialisation mode, where object allocation is allowed in the immortal memory and the mission memory. Objects created in mission memory will be deallocated when the mission ends, upon exiting the mission. After initialization mode, missions move to mission mode, this is the actual system operation mode where they will spend most their time. In this mode no allocation in immortal memory or mission memory is allowed. From this mode, mission can be shut down. At this point, it is uncertain if there is a shutdown mode to do cleanup, however, it is possible to restart the mission and reinitialise the mission [104, p. 9][103, p. 10].

These complexity levels are nested with concurrent missions allowed SCJ level 2, the highest most liberal level, and a single thread of execution at level 0, being the lowest most restrictive level.

Execution during a mission is performed by schedulable objects as defined in the RTSJ, see Section 2.3.1. Schedulable objects have associated private memories that are not shared with other schedulable objects. The number of schedulable objects running concurrently and of which type (aperiodic, periodic, no heap real-time threads) are allowed, is decided by the SCJ compliance level. These levels are described as:

SCJ level 0 The most conservative level is SCJ level 0, and it consists of only one single thread of execution. This results in a cyclic scheduling model as described in Section 2.4.1; this model is shown in Figure 2.10. At this level, a mission consists of only periodic event handlers, and as there is only one thread of execution, no synchronization will be necessary. However, developers are encouraged to use such synchronization mechanisms, as this will ensure correctness if a is executed on a platform complying to level level 1 or higher. The operations `wait` and `notify` are not allowed, and only periodic events are allowed, i.e. no threads nor aperiodic events; schedulable objects have private memories used for allocation which are cleared at the end of a run.

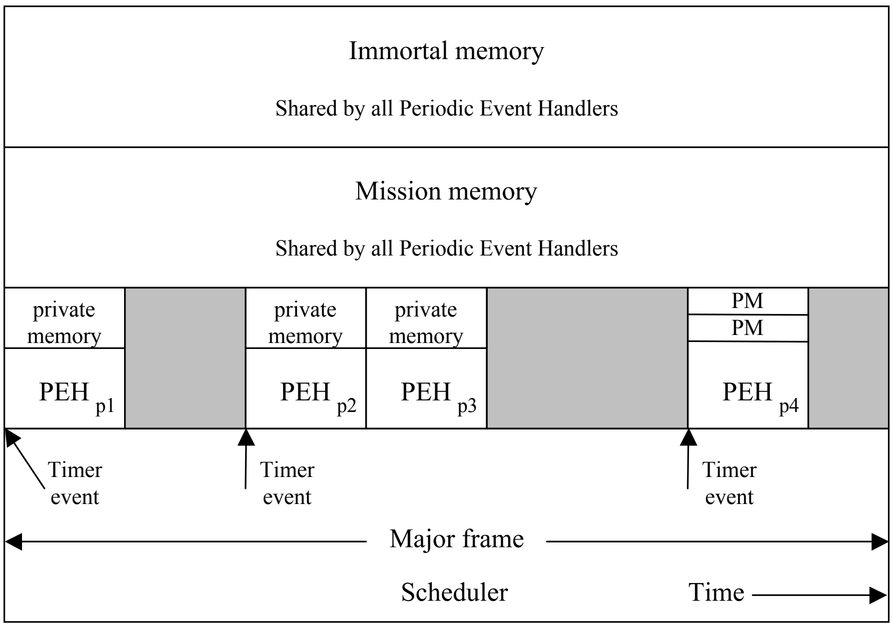


Figure 2.10: SCJ level 0 [104, p. 11]

SCJ level 1 At level 1 schedulable objects are executed concurrently using a fixed priority preemptive scheduling mechanism, as described in Section 2.4.2; this is shown in Figure 2.11. This level allows both aperiodic and periodic tasks, however, threads at this level are disallowed. Schedulable objects have private memories, cleared at the end of their execution, and which cannot be shared; `wait` and `notify` are not allowed.

SCJ level 2 Level 2 is the level allowing the most complexity. It allows most schedulable objects, aperiodic, periodic, and no-heap real-time threads, and multiple nested missions. Each mission has its own mission memory. At this level, the methods `wait` and `notify` are available. The execution model provided at level 2 is shown in Figure 2.12.

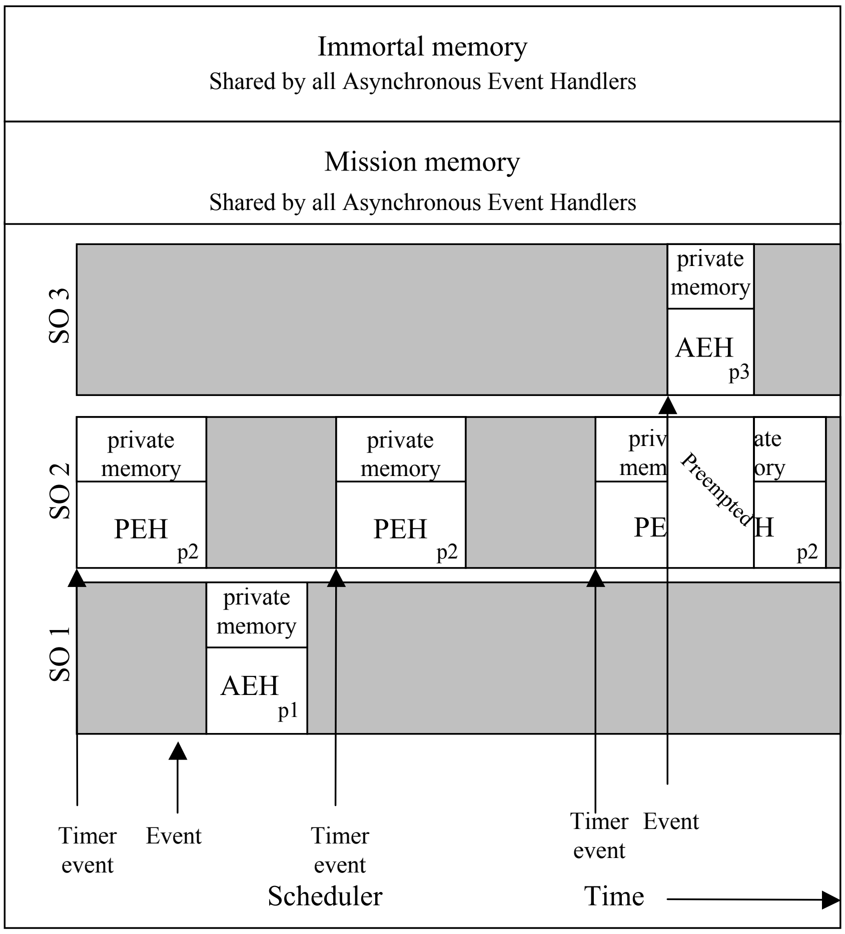


Figure 2.11: SCJ level 1 [104, p. 13]

SCJ does not contain sporadic event handlers, as this would require additional checks on the arrival of such events and error handling in case they arrive too often. This is deemed too complex to handle for critical applications, even at level 2. SCJ also do only provide immortal, mission and scoped memory; the use of heap is therefore disallowed, and thus there is only the concept of `NoHeapRealtimeThread` in the specification. Schedulable objects such as `RealtimeThread`, from RTSJ, and `java.lang.Thread` are therefore not allowed in SCJ applications. It is important to notice that many of the details of RTSJ are disallowed, such as `RealtimeThread` and `Thread`. This is done by using annotations, and by annotating all allowed classes and methods with details on at which level they are allowed.

The hierarchy of mission related classes and the schedulable objects allowed in SCJ and their relation to RTSJ is shown in Figure 2.13. The mission

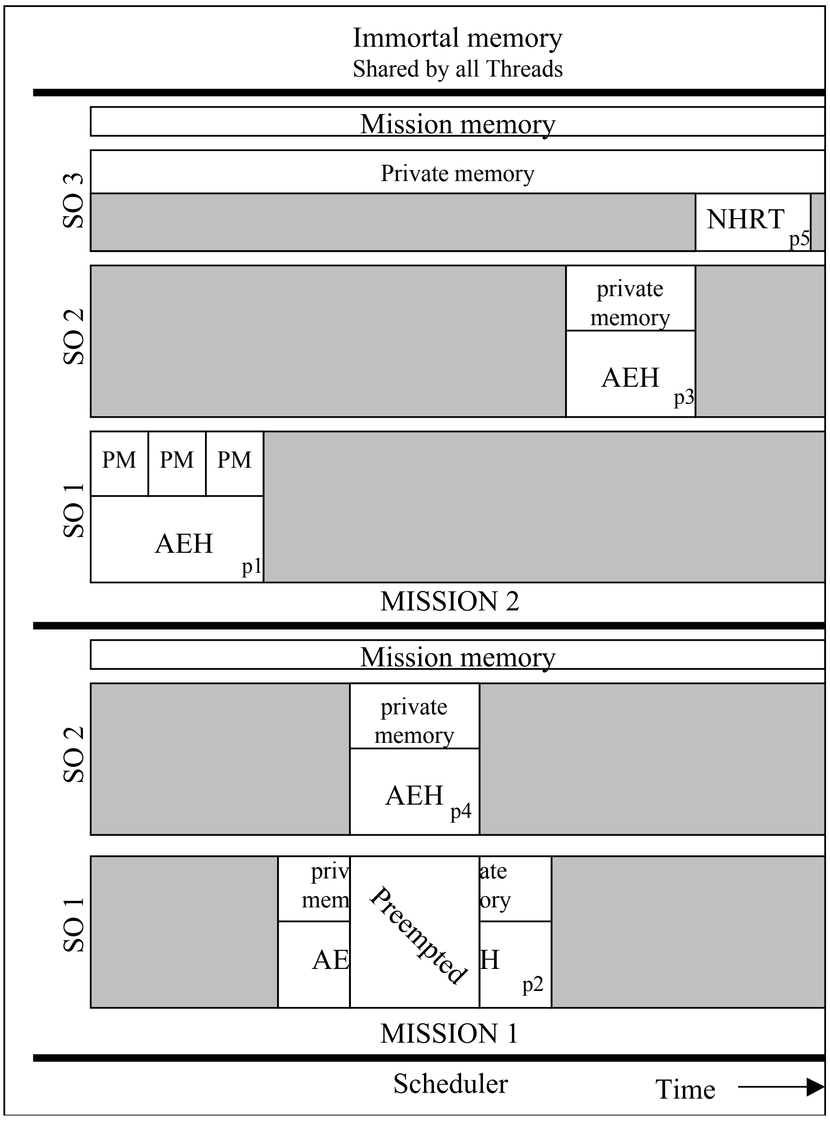


Figure 2.12: SCJ level 2 [104, p. 15]

concept in SCJ is handled by multiple classes:

MissionDescriptor A mission is simply a grouping of schedulable objects and a mission memory, contained by the `MissionDescriptor` class[5]. This class defines methods such as `initialize` and `cleanup`. The ini-

[5]The SCJ draft specification additionally mentions a Mission class, either to be extended by MissionDescriptor, or as a separate class - seems, however, that MissionDescriptor and Mission are at this point the same class

tialize method will prepare handlers and memory in the mission.

MissionManager The mission manager is responsible for starting and stopping missions, and it defines methods such as `start`, `requestTermination`, and the cleanup method `performShutdown`.

MissionSequencer The mission sequencer is describing how missions are executed, and in which order. This class is actually an implementation of an asynchronous event handler which will perform the actual mission initializations. The specific implementation of `MissionSequencer` is defined in an implementation of the `Safelet.getSequencer` method.

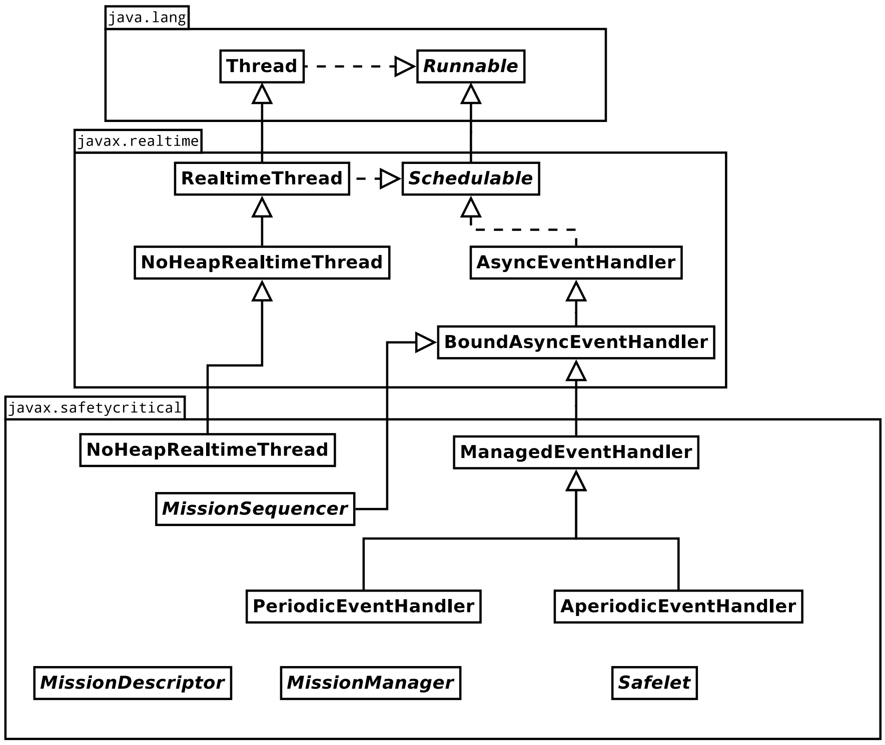


Figure 2.13: The class hierarchy related missions and schedulable objects

SCJ Memory Management

Two new memory classes are introduced in SCJ, however, the memory related classes and their rules result in a simpler hierarchy and model. These are `PrivateMemory`, for use in event handlers and `MissionMemory` used by missions. These memory classes are based on RTSJ`LTMemory`. Each handler is executed using its own `PrivateMemory`, which is an instance of `ScopedMemory`, which cannot be shared with other handlers. Handlers are allowed to create new nested scopes, however, these scopes cannot be shared with other handlers. The `MissionMemory` is shared between handlers, however, no allocation in this memory is allowed during the mission phase; all objects are allocated during mission initialisation.

2.4 Real-time Scheduling

The concept of a real-time scheduling is to execute the tasks of a real-time system in an order, such that no task misses its deadline. There are different aspects to consider when choosing a scheduling mechanism. Perhaps most important, it must be possible to perform a feasibility analysis of the system to ensure correct behaviour of the system under a given scheduling approach. This is an important field, discussed in Section 2.5. It is also important to consider the type of system to be scheduled, as scheduling approaches differ substantially in terms of overhead and the number of preemptions and scheduling complexity. However, the behaviour of the system is greatly affected by the choice of scheduling algorithm. Some algorithms perform predictably under stress, and some do not, and also the computational overhead must be taken into account.

There are three main categories of real-time scheduling:

- Off-line scheduling
- Fixed-priority scheduling
- Dynamic scheduling

2.4.1 Off-line Scheduling

Off-line scheduling is an approach to scheduling where a static schedule, i.e. the interleaving of task invocations, of the system is designed such that no deadline is missed [17, 45], and a single program, called the cyclic executive, is constructed.

The full static schedule is actually repeating a finite schedule of the tasks to be executed, since the full schedule is likely to be infinite. The schedule is divided into a major cycle which is repeated an infinite number of times. The major cycle determines the actions to be performed during a fixed period of time. The length of this major cycle is the least common multiple of the periods of all tasks in the system, as this is the time from where the past execution will be repeated forever.

This major cycle is further divided into smaller minor schedules of the code to perform the actions of the tasks, called the minor cycle. The size of a minor cycle should be a common divisor for all the periods in order to accommodate all task releases.

Advantages of the cyclic executive approach include:

- The proof of schedulability is done by constructing the cyclic executive and thus no other schedulability tests are needed.
- A low overhead, as there is no need for preemption and no need for ensuring integrity using for example mutexes, as only one thread is running the whole program.

Disadvantages include:

- This approach is not very suited for systems containing tasks of aperiodic nature, as room in multiple minor cycles must be allocated to handle these, which may lead to a substantial amount of idle CPU time in the system.
- The construction of a cyclic executive with larger tasks can be somewhat complicated, as any sizable tasks will have to be split up to fit into the minor cycles.

2.4.2 Fixed Priority Scheduling

Fixed priority preemptive scheduling is a more flexible approach when compared to that of cyclic executives. In this approach, tasks are assigned an offline calculated fixed priority, which it keeps throughout the lifetime of the program.

This approach has an advantage when applied to more dynamic systems where tasks have very different execution times, as there is no need to split up tasks to fit into different minor cycles, as in the cyclic executive. The application will itself also be more flexible, and minor changes in requirements will not affect the implementation as much as in the statically defined cycle; for the interested reader, an in depth comparison of the cyclic executive and priority scheduling is presented in [102].

Priorities are usually assigned according to task deadlines, where shorter deadlines equal higher priorities; this is called Deadline Monotonic Scheduling(DMS) [95], which is a successor to the earlier more restrictive approach of basing priorities on task periods, known as Rate Monotonic Scheduling(RMS) [100].

DMS is optimal among static scheduling approaches given the simple model presented in Section 2.2. The priority based scheduling approach will in many cases also prove helpful in debugging errors caused by missed deadlines, as the task execution patterns are predictable. Deadlines will be missed by tasks with a lower priority, and thus a higher deadline.

2.4.3 Dynamic Scheduling

In the dynamic scheduling approach, priorities are assigned dynamically to the running tasks during the execution of the system according to some importance measure.

An example of a dynamic scheduling algorithm is the Earliest Deadline First (EDF) algorithm [100], where the importance is based on the time left before a task meets its deadline. This algorithm has the advantages of being able to achieve higher processor utilization than the fixed priority approach, and being able to handle more dynamic systems than the cyclic executive. This approach is also optimal among all the scheduling approaches on single processor systems, meaning that if a set of tasks characterised by execution-time, deadline and arrival pattern is schedulable by any algorithm, it will be schedulable by EDF.

EDF holds advantageous properties, however, it is a complex algorithm to implement, especially in limited systems. It has a high overhead, and in the case of an overloaded system, it will not behave predictably, as is the case when priorities are assigned statically.

2.5 Real-time Systems Analysis

The correctness of a real-time system is heavily dependent on the timely reaction to external events, i.e. it will execute an event handler while the result of the computation is still useful. This means that the system, at any given point, must have enough available resources, computing power or simply time, to handle all occurring events before their respective deadlines. This property is known as schedulability; a system is schedulable if all events are handled before their respective deadlines.

Because real-time systems are often critical systems, and are often embedded systems produced in large quantities, the schedulability property must be proven as a part of the development, before the product is shipped. This proof is achieved through schedulability analysis, which is discussed in the rest of this chapter.

2.5.1 Traditional Schedulability Analysis

The established approach to schedulability analysis, known as traditional schedulability analysis, is based on the work by Liu and Layland in 1973 [100]. In the original work, Liu and Layland present a task model based on cost, period, and deadline, in which tasks are scheduled by either the static scheduling algorithm Rate Monotonic Scheduling (RMS), where periods are mapped to priorities, or the dynamic scheduling algorithm EDF, where the currently running tasks with earliest deadline will have highest priority. Additionally, four restrictions are placed on the system limiting expressiveness:

1. A system consists of only periodic tasks.
2. All tasks in the system has a deadline equal to their period.
3. All tasks are independent.

4. All tasks have a constant computation time.

Additionally, some implicit assumptions are made in the original work, and clarified in [15]:

- No process may voluntarily suspend itself.
- All processes are released as soon as they arrive.
- All overheads are ignored; they are assumed to be 0.

For this model the authors present a schedulability analysis based on processor utilisation, the utilisation test. This amounts to calculating the processor utilisation (eq. 2.1), of a given system, and then this is compared to an upper bound on processor utilisation (eq. 2.2). If the processor utilisation is lower than the upper bound, the system in question is guaranteed schedulable, a sufficient, but not necessary test [45], meaning systems passing this test are guaranteed schedulable, and a system failing the test may still be schedulable.

Utilisation for a set of m tasks is given by the following equation:

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i} \tag{2.1}$$

where C_i is the cost and T_i is the period of $task_i$.

The upper bound for utilisation is dependent only on the number of tasks, and it is given by the following equation:

$$m(2^{\frac{1}{m}} - 1) \tag{2.2}$$

where m is the number of tasks.

m	utilisation
1	100%
2	82.8%
3	77.9%
4	75.6%
5	74.3%
10	71.7%

Table 2.1: Processor utilisation bounds

Examples of processor utilisation bounds are given in Table 2.1, and as the number of tasks in the system goes to infinity, the utilisation bound settles at 69.3%:

$$\lim_{m \to \infty} m(2^{\frac{1}{m}} - 1) = 0.693$$

This means the amount of wasted processing time in most systems will be around 30%, which is a relatively high number for limited devices.

2.5.2 Response Time Analysis

A substantial amount of research has been invested in real-time systems analysis since the utilisation based tests were developed. The major disadvantage to using utilisation as a schedulability test is the coarseness of the analysis and the assumptions and restrictions put on the system expressiveness. The response time analysis [14] is a further development of the technique and allows analysis of more expressive systems. More specifically, the basic response time analysis relaxes some important constraints and allows:

- Sporadic tasks, with minimum inter-arrival times.
- Dependencies between tasks through the notion of blocking time for each task, i.e. the time a task can be blocked by a lower priority task through exclusive access to resources.
- Release jitter, the time between release and actual execution of a task.

In the response time analysis, the response time is calculated for each task, and the system is schedulable if the response time for a task is less than its deadline. Sporadic tasks are essentially included as periodic tasks with their period set to the minimum inter-arrival time. The blocking time is calculated based on priority inversion avoidance protocols, priority inheritance or priority ceiling.

We use the following notation:

- T denotes period or minimum inter-arrival time.
- D is the deadline relative to the task release.
- B is the time a task is blocked by lower priority tasks.
- C is WCET.
- $hp(i)$ is the set of tasks of higher priority than task i.

Synchronisation Protocols

In a system with interdependent tasks, tasks can experience blocking caused by lower priority tasks holding a shared resource, which is also required by a higher priority task. This blocking time is affecting the response time of the task, and must therefore be bounded.

These analyses assume the use of synchronisation protocols to avoid unbounded priority inversion. The two protocols considered here are the priority inheritance protocol and the priority ceiling protocol [140]. The latter of the two has two slightly different versions, Immediate Priority Ceiling Protocol (IPCP) and Original Priority Ceiling Protocol (OPCP).

The priority inheritance protocol only avoids unbounded priority inversion, where the priority ceiling protocol also avoids deadlocks and the worst case blocking for the two ceiling protocols is at the most one critical section.

Priority Inheritance In priority inheritance, if a lower priority task is blocking a higher priority task, it will inherit the priority of the blocked task.

After the priority boosted task finishes executing the critical section causing the block, it will be reassigned its original priority. This protocol will prevent unbounded priority inversion, however, it will not prevent deadlock situations.

Priority Ceiling The Priority Ceiling Protocol (PCP) ensures freedom from deadlocks, predictably bounds blocking times, and it will result in less context switches than Priority Inheritance Protocol (PIP). To each resource in the system, a priority equal to the maximum priority of all the threads accessing the resource is assigned, and each task has a static priority.

- OPCP then works as follows:
  - The priority of a task after locking a resource:
    - remains, if no higher priority tasks are blocked because of acquired resource,
    - is raised to the maximum of inherited priority and the static priority, if it is blocking a higher priority task.
  - A resource can only be acquired by a task if it has a priority higher than any locked resource not locked by the task itself.
- In IPCP, a task priority is immediately raised to the maximum of the locked resources.

In IPCP, even tasks without critical sections may be blocked, whereas in OPCP, these tasks will only be indirectly blocked by blocked higher priority tasks. Additionally, the difference lies in the number of context switches and hence the number of locking operations, where IPCP will perform less locking operations and cause less switches than OPCP.

2.5.3 Analysing Response Time

For a set of tasks, indexed $i = 1..n$, with static priority, P_i, the response time analysis of $task_i$, R_i is given by the equation:

$$R_i = C_i + B_i + I_i \tag{2.3}$$

where:

- C_i is the execution time of $task_i$.
- B_i is the maximum blocking time of $task_i$. This is dependent on the protocol used for priority inversion avoidance.
  - The maximum blocking for $task_i$ using PIP is given by the equation:
    $$B_i = \sum_{r \in R} usage(i,r) C_r \tag{2.4}$$
  - The maximum blocking time for $task_i$ using PCP is given by:
    $$B_i = \max_{r \in R} usage(i,r) WCET_r \tag{2.5}$$

  where R is a set of resources, C_r is the WCET of the critical sections involving resource r, and $usage(i,r)$ evaluates to 1 if resource r is used by a task with lower priority than $task_i$ and used by a task with priority higher than or equal to the priority of $task_i$; or 0 otherwise.
- I_i is the maximum interruption time of $task_i$ by tasks of higher priority, given by the equation:
  $$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.6}$$
  where $hp(i)$ is the set of tasks of higher priority than task i.

This gives the following recursion:

$$R_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{2.7}$$

for which a fix-point solution, $R_i^{n+1} = R_i^n$ with $R_i^0 = C_i$, is guaranteed if the utilisation is less than 1 [14].

2.6 Model Checking

Model checking is another interesting approach to program analysis. Models can be built as a part of the real-time system specification, and some tools, such as TIMES [11], provide code synthesis from models, in order to gain some correspondence between the model and the code. Other tools use binary code as input and perform analysis on a model generated directly from this [50], again to ensure tight correspondence between the analysis and the actual program. Similarly, functional correctness is checked in tools such as Bandera [49] by generating models from the code. Java pathfinder [40] is a similar tool which works by symbolically executing the byte-code.

Model checking is also an interesting method for solving the schedulability problem [90, 59, 60], which is more flexible than traditional approaches, and it is able to take into account more of the problem specific details, such as the environment and task interaction. The general approach taken is to build an abstract specification of the system on which the schedulability property is verified, it is then implemented after a positive verdict, or, by building an abstract model of an existing implementation to verify that it is actually correct with regards to schedulability.

2.6.1 Timed Automata

In this section an overview of Timed Automata is presented, based on [16, 9].A Timed Automaton is a finite directed graph, a set of clocks, and a set of constraints on these clocks. The nodes of the graph we call locations and the edges are called switches.

For a set of clocks C, a clock valuation v, is a mapping $C \rightarrow \mathbb{R}_+$, assigning a nonnegative real value to each clock in C; $v[X := 0]$ denotes the valuation v where zero is assigned to the clocks in $X \subseteq C$. For $d \in \mathbb{R}_+$, $v + d$ denotes the valuation where each clock $c \in C$ maps to $v(c) + d$.

Locations and switches each have an associated set of clock constraint, called invariant for locations and guard for switches.

Definition 1 (Clock constraint)**.** Let x be an integer and $c \in C$ be a clock, then a clock constraint is defined by the grammar:

$$\phi := c \leq x | c < x | c = x | c > x | c \geq x | \phi \wedge \phi$$

Time elapses in locations, and only while the invariant holds, while switches are instantaneous. Any switch can reset a set of clocks.

Definition 2 (Timed Automata)**.** [9]
A timed automaton is a tuple $\langle L, l_0, F, \Sigma, C, E, I \rangle$ where:

- L is a finite set of locations.
- $l_0 \in L$ is the initial location.
- $F \subseteq L$ is a set of accept locations.
- Σ is a finite set of actions.
- C is a finite set of clocks.
- $E \subseteq L \times \Phi(C) \times \Sigma \times 2^C \times L$ is the set of switches, where $\langle l, g, a, r, l' \rangle \in E$ is the switch between locations, from l to l', guarded by the constraint g, a is an action, and r is the set of clocks to be reset.
- $I : L \rightarrow \Phi(C)$ assigns a set of invariants to each location.

We use $l \xrightarrow{g,a,r} l'$ to denote $\langle l, g, a, r, l' \rangle \in E$.

Definition 3 (Semantics of Timed Automata)**.** A state in a timed automata is a pair $(l, v) \in L \times \mathbb{R}^C$ where $\Phi(l)$ holds for the clock valuation v of C. The initial state is (l_0, v) where $\forall c \in C : v(c) = 0$, and a transition is either a delay transition or a location transition:

- Delay transition: for a state (l, v) and a delay d, $(l, v) \xrightarrow{d} (l, v + d)$, if $\forall d' : 0 \leq d' \leq d$, $v + d' \in I(l)$.
- Location transition: $(l, v) \xrightarrow{a} (l', v)$, if there exists $l \xrightarrow{g,a,r} l'$ $s.t.v \in g, v' = [r := 0], v' \vdash I(l')$

2.6.2 Timed Automata in Practice

Uppaal is a tool for modeling and verification of real-time systems using a variation of timed automata as the underlying formal foundation. A system is expressed as a network of timed automata along with variable, function, and other declarations, and requirements are expressed as Timed Computational Tree Logic (TCTL) formulae [16]. This section gives a short introduction to the modeling language of Uppaal, based on [24].

Uppaal works on an extended definition of timed automata. This extended definition includes bounded variables along with an imperative C inspired language for manipulating these variables. Elements are also added to make the modelling of commonly used concepts easier, such as urgent locations. Additionally elements from the imperative language can be used in guards, for example, thus increasing flexibility. Clocks are defined as usual, however, the action concept varies slightly, as this is used for synchronization.

An automaton in Uppaal is then defined as in Definition 4 inspired by [158, 24].

Definition 4 (Uppaal Timed Automata)**.**
A timed automaton in Uppaal is a tuple $\langle L, l_0, \Sigma, V, F, C, E, I \rangle$ where

- L is a finite set of locations.
- $l_0 \in L$ is the initial location.
- Σ is a finite set of actions, co-actions, and the internal action τ.
- V is a finite set of variables.
- F is a finite set of function declarations.
- C is a finite set of clocks.
- $E \subseteq L \times \Phi(C) \times \Psi(V, F) \times \Sigma \times 2^C \times 2^V \times L$ is the set of switches, where:

- $\langle l, g, a, r, u, l' \rangle$ is the switch from location l to location l' where:
    - g is a guard: $g = (g^c, g^v) \in \Phi(C) \times \Psi(V, F)$, a conjunction of clock-constraints, g^c, and variable-constraints, g^v.
    - a is an action.
    - r is the set of clocks to be reset.
    - u is the set of variable updates.

- $I : L \to \Phi(C) \times \Psi(V, F)$ assigns a set of invariants to each location.

Where the set of clock constraints $\Phi(C)$ is defined as in Definition 1 (except x is now an expression evaluating to an integer), and $\Psi(V)$ is defined by:

$$\psi := i \oplus k \mid \psi \wedge \psi \mid \psi \vee \psi$$

where $\oplus \in \{<, \leq, =, \geq, >\}$ and i, k both evaluate to integers.

Actions and co-actions are denoted ! and ? respectively. They are now used for synchronization and are referred to as channels. A state now includes variables: $(l, v, x) \in L \times R^C \times Z$, with $Z \subseteq \mathbb{Z}^V$ as variables are bounded.

A model, or Network of Timed Automata (NTA), in Uppaal consists of a set of timed automata, executed in parallel, communicating through synchronization and shared variables. The definition of an NTA is a natural extension to a vector of timed automata: $\langle L_i, l_i^0, \Sigma, V, F, C, E_i, I_i \rangle$ where the set of locations, initial location, edges, and invariants are vectors, and a state is now the triple $(\bar{l}, v, x) \in (L_1 \times \cdots \times L_i) \times R^C \times Z$ with $Z \subseteq \mathbb{Z}^V$.

The initial location is $\overline{l_0} = (l_1^0, \ldots, l_n^0)$, and the transition system is defined by the following transition relations:

- Delay transition: for a state $(\bar{l}, v, x)$ and a delay d, $(\bar{l}, v, x) \xrightarrow{d} (\bar{l}, v + d, x)$ if $\forall d' : 0 \leq d' \leq d, (v + d', x) \in I(\bar{l})$.

- The internal (τ) transition: $(\bar{l}, v, x) \xrightarrow{a} (\bar{l}[l'_i/l_i], v', x')$ is defined if there exists $l_i \xrightarrow{g,\tau,r,u} l'_i$ $s.t.$ $v \in g$, $v' = [r := 0], x' = eval(x, u), and\ v' \in I(\bar{l}[l'_i/l_i])$.

- The synchronous transition: $(\bar{l}, v, x) \xrightarrow{a} (\bar{l}[l'_i/l_i, l'_j/l_j], v', x')$ is defined if there exists a switch $l_i \xrightarrow{g_i,c?,r_i,u_i} l'_i$ and $l_j \xrightarrow{g_j,c!,r_j,u_j} l'_j$ $s.t.$ $v \in (g_i \wedge g_j)$, $v' = [r := 0], x' = eval(x, u), and\ v' \in I(\bar{l}[l'_i/l_i, l'_j/l_j])$.

where *eval(u,x)* is a mapping to an updated variable environment, after evaluating updates in u, on the variables in x. Multiple statements are evaluated sequentially.

The rest of this section will informally cover additional Uppaal features, used later in this thesis.

The bounded variables in Uppaal are accompanied by a C like language for manipulating them, including many of the features known from C, such as records, custom type definitions, functions, and arithmetic operations. These operations can be fired during switches from one location to another, on decorated edges.

Templates In UPPAAL , automata are created from automata templates. Templates are a parameterized form of automata, analogous to classes and concrete objects in Java.

Template declarations have the same form as function declarations, which closely follow C++ syntax, although in UPPAAL the content of a template is more than just a block of code.

Templates are represented graphically, and are essentially a graphical representation of the above definitions, along with a code section, containing local declarations and function definitions to be used in the model. At the NTA level, there is also declaration section, for globally shared declarations.

System Declaration The language for system declarations is different from the aforementioned C like declaration language. Templates are instantiated by UPPAAL in the `system` section, however, parameter binding must be done separately. Given a template declared as: `Template(int i)`[6] A new name for the template with a parameter binding can then be created:

```
P1 = Template(1);
system P1;
```

This will result in the new name: P1, with i bound to 1. However, `system Template(1);` is not a legal system declaration, as parameter bindings must be done in advance.

This may seem strange at first, however, please consider the situation where the parameter is an ID:

```
// An instantiation of a template
// with a single integer argument:
// Template(int id) // pseudo code
Process1 = Template(1);
Process2 = Template(2);
system Process1, Process2;
```

This is a legal, but very rigid and somewhat inflexible way of achieving the desired result. If later, the `id` span more than just 1..2, we would have to add template instantiations for each id. As these ids are used more than one place, this requires careful modification throughout the model. UPPAAL allows omitting the binding of some or more formal parameters, which is a useful feature in such common situations. To leave the formal parameters of the template unbound, we write:

[6]In UPPAAL 4.0 this is done graphically

```
system Template;
```

However, Uppaal will create an instance for every possible combination of the parameters. In this situation the result is one instance for every value of the type `int`. This is solved by defining an ID type, in this example, `id_t`:

```
typedef int[1,2] id_t; // the integer type from [1 to 2]
```

and using this as the type of the parameter to our template. Uppaal will in this case create an instance for the two values of `id_t`;

Binding is actually just a renaming, and partial binding is allowed. This is very useful in situations where multiple instances are created sharing some of the same values:

```
// Template2(int a, int b)
Partial(const int x) = Template2(x, 1);
FullyBound() = Partial(1);
system FullyBound;
```

In the system line, priorities can be set for instantiated processes by defining an order using `<`. System declarations use the following grammar[7]:

```
System ::= 'system' ID ((',' | '<') ID)* ';'
Process ::= ID '(' Arguments ')'
Instantiation ::= ID [ '(' Parameters ')' ] '=' ID '(' Arguments ')' ';'
```

Locations and Edges Apart from invariants, Uppaal defines different types of locations:

- Urgent location: In an urgent location, time in the model is not allowed to pass. This is basically syntactic sugar for an invariant constraint of the form $c \leq 0$, where c is a clock, with statements resetting c on all incoming edges of this location.
- Committed location: In a committed location:
  - time is stopped, as in urgent locations.
  - the entire state is committed.
  - a transition from the committed state must involve a switch from a committed location[8].
- Uppaal supports the stopping of individual clocks. This is done in locations as a special invariant, setting the rate of the clock. Uppaal

[7]Directly copied from the manual of Uppaal version 1.1.9

[8]In the case more than one location is committed, enabled switches will be selected non-deterministically

however only supports the rates zero and one. Given a clock c, the rate syntax is: $c' == i$ for $i \in 0, 1$; this expression always evaluates to true.

In addition to guards, edges in UPPAAL can have:

- Update statements, which allow the update of variables or function calls; function side effects are allowed.
- Synchronization with other automata using channel expressions of the form `[channel] [!?]`, where `channel` is an expression evaluating to channel; $c!$ is initiating synchronization (action), and $c?$ is accepting, or waiting for, synchronization (co-action), on channel c;
  - a special kind of channel, the broadcast channel, allows multiple receivers of a single initiating synchronization; this is a one-to-many synchronization.
  - channels can also be declared `urgent`, preventing delay transitions in the source location, if the switch is enabled.
- Select statements are a shorthand notation for nondeterministically selecting the value of a variable from a set of values. As an example, consider the select statement: `i : int[0,3]`. Here, the variable i nondeterministically is bound to a value in the range of 0 to 3 inclusive. This variable can then be used in the guard, the synchronization and the update expressions of this switch only.

Example of an NTA Consider the automata in Figure 2.15 and Figure 2.14. The first automaton represent a simple model of a coffee machine, giving the choice of `Coffee` or `Tea` upon the insertion of a coin, and the second a model of a working scientist. The initial location of the coffee machine is

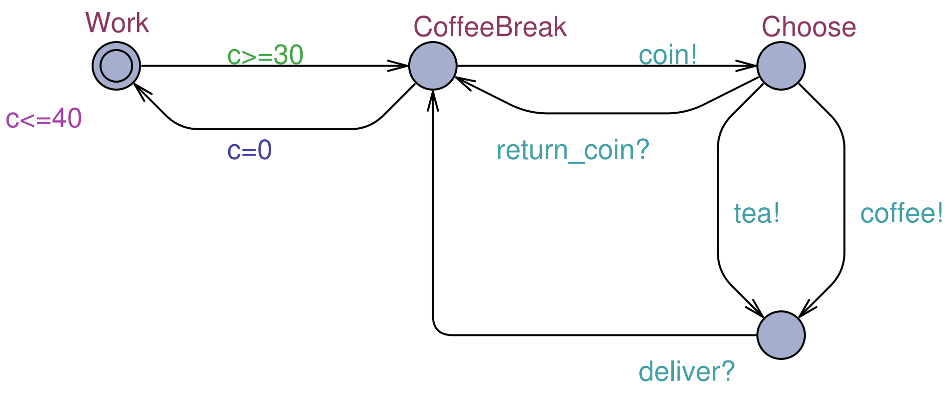


Figure 2.14: Model of a Scientist

labeled `Ready`, and `Work` for the scientist. In the abstract model, a scientist can do two things: work, in the `Work` location, or have a coffee break, by moving to the `CoffeeBreak` location.

There are limits on the periods of time the scientist can work, and this is in the model represented by an invariant on the `Work` location. The clock `c` is introduced, and an invariant, `c <= 40`, associated with the location `Work` states: "While in this location, the clock c must be less than or equal to 40 time units". Additionally, a guard on the transition to coffee break requires `c` to be larger than or equal to 30 time units. This requires the scientist to work between 30 and 40 time units before having a coffee break.

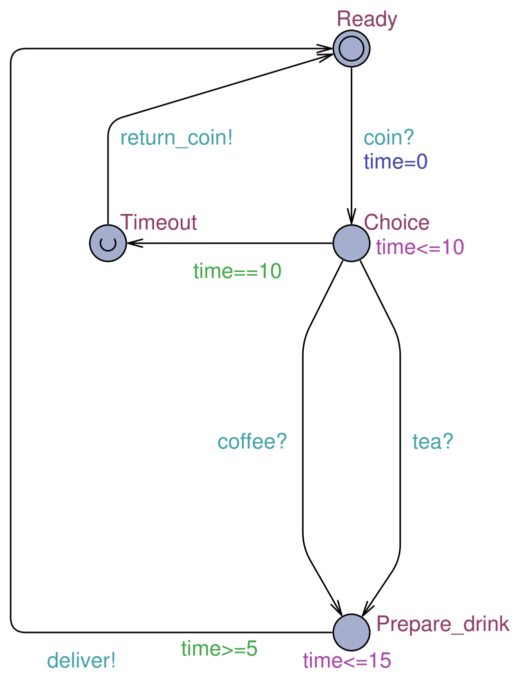


Figure 2.15: Model of a Coffee Machine

In the `CoffeeBreak` location, interaction with the coffee machine is an option. The only transition from the `CoffeeBreak` location is a synchronizing transition, symbolizing a coin input. When taking this transition, both automata will switch location, from the location pair $(CoffeeBreak, Ready)$ to $(Choose, Choice)$ atomically. Notice the coffee machine has a built in coin timeout, stating that if no choice is made within ten time units, the coin will be returned, and the coffee machine will move back to the initial `Ready` state. The scientist can now make a choice between tea and coffee, both synchronising states. When a choice is made nondeterministically, the coffee machine will move to the `Prepare_drink` state, where it will stay in five to 15 time units, before synchronizing on the `deliver` channel, moving back to its initial state. The scientist can now choose to go back to work for at least 30 time units, or stay indefinitely in the `CoffeeBreak` location.

2.7 Platform

It is important the platform is both real-time enabled and provides analyzable execution of programs.

Many real-time platforms exist, such as the RTSJ implementations from Sun (Java RTS), Aicas (JamaicaVM), Timesys, and IBM (WebSphere Realtime). These platforms, while focusing on predictable performance, are focusing on RTSJ, and thus they offer little analysability to the extent needed for safety-critical systems.

Small footprint implementations of virtual machines exist, such as Jelatine [4], Keso [155], and the open source JamVM[9]. However, these virtual machine implementations focus on size and performance of the virtual machines, and do not directly have analysability and predictability in focus.

Work has been done in this area in order to provide predictable execution, the Java Optimized Processor (JOP) [130, 131, 132, 136] and recently the Hardware near Virtual Machine (HVM) [88, 105, 87] are steps in the direction of predictable execution platforms.

The JOP is a Java processor implemented directly in Field Programmable Gate Array (FPGA) hardware. This is intended to make Java execution time predictable at the instruction level. This processor does not, as most processors do today, optimise by adding cashes and long pipelines. This is because, in general, such techniques are improving the average execution time only, while making the worst case situations very costly. In a real-time setting the worst case situation has become the interesting situation, as this is where important deadlines may be missed; this means average case optimisations are superfluous and may actually make the analysis more difficult.

The JOP processor, however, does have predictable optimisations such as an analysable method cache [132]. The JOP is a great step in the direction of time predictable Java programs; the JOP project includes the WCET Analyzer (WCA) tool [134], designed for timing analysis of programs running on JOP.

Another approach to real-time embedded Java is the HVM. The HVM is a Hardware near Java Virtual Machine implementation, targeting general purpose hardware, instead of replacing them. It is a very small Virtual Machine (VM) implementation, targeting tiny devices such as the ATMega 8bit platform with 8KB RAM and 256KB flash [88]. Here, the focus is the size of the execution platform and the ability to analyse. However, analyses

[9]`http://jamvm.sourceforge.net/`

of programs running on such a platform do need to take into account caches, branch prediction, and pipelines [50, 105].

Chapter 3

Summary and Contributions

This section summarises the two main areas of interest, languages and analysis, and highlights the issues considered in this thesis. A short summary of the papers comprising this thesis is presented where relevant. The papers are:

A: Model-based schedulability analysis of safety critical hard real-time Java programs
Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Published and presented at the 6th International Workshop on Java Technologies for Real-time and Embedded Systems, 2008 (JTRES'08) [33].

B: A predictable Java profile: rationale and implementations
Thomas Bøgholm, René R. Hansen, Anders P. Ravn, Bent Thomsen, and Hans Søndergaard. Published and presented at the 7th International Workshop on Java Technologies for Real-time and Embedded Systems, 2009 (JTRES'09) [29].

C: Schedulability analysis for Java finalizers
Thomas Bøgholm, René R. Hansen, Anders P. Ravn, Bent Thomsen, and Hans Søndergaard. Published and presented at the 8th International Workshop on Java Technologies for Real-time and Embedded Systems, 2010 (JTRES'10) [30].

D: Refactoring Real-Time Java profiles
Hans Søndergaard, Bent Thomsen, Anders P. Ravn, René R. Hansen, and Thomas Bøgholm. Published and presented at the 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'11) [145].

E: Schedulability Analysis Abstractions for Safety Critical Java
Thomas Bøgholm, Bent Thomsen, Alan Mycroft, and Kim G. Larsen. Nominated for the 'best paper' award. Published and presented at the 15th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'12)[34].

F: Towards harnessing theories through tool support for hard real-time Java programming
Thomas Bøgholm, Christian Frost, René R. Hansen, Casper Svenning Jensen, Kasper Søe Luckow, Anders P. Ravn, Hans Søndergaard, Bent Thomsen. Published in Innovations in Systems and Software Engineering [46].

This section highlights some of the issues in the tools, methods, and techniques used today, leading to specific improvements proposed in this thesis.

3.1 Languages

In common application development, the shift from simple imperative languages like C to modern programming languages has been started, and is well established in the industry. Modern object oriented languages such as Java and C# are being taught at universities, along with modern software development techniques such as Object-Oriented Analysis and Design (OOAD) [108].

The embedded software industry is, however, more conservative in their choice of languages, and most often uses C, assembly languages, and to some extent C++. As pointed out in Section 2.3, these languages do not elegantly support the complexities of modern embedded systems. Where C uses libraries for features such as concurrency, and even common operations such as memory allocation, languages like Java have built in support for such features and specify their semantics.

It is well established knowledge that standard Java is not suitable for many applications outside the domain of enterprise applications and the like, and this includes embedded systems in general, and certainly safety-critical systems. This has given rise to specifications for Java in these other domains, such as Java ME for mobile phones, Java Card for smart cards, RTSJ and the ongoing specification of SCJ, started in 2006.

After the first specification for real-time systems was released as RTSJ, it became evident it was too large and dynamic for many applications with hard deadlines and with safety critical aspects. This lead to a number of profiles [91, 137, 122, 29], and safety critical systems are now officially targeted in the upcoming specification for safety critical Java, SCJ.

The goals of SCJ are to define a simple but expressive subset of the Java language and RTSJ, to support the development of safety critical systems capable of certification under DO-178B/ED-12B. Although most of the SCJ approach serves its purpose, there are two important language related issues, where it could be significantly improved.

3.1.1 Language Issues

Class Hierarchy SCJ is built using RTSJ by means of inheritance for limitation. This means unwanted functionality from RTSJ pollutes the SCJ API. This functionality is seemingly available, but disallowed in the specification; this results in unwanted clutter when using the specification and it may confuse developers of safety critical systems.

Mission Concept Missions in the specification are designed as mere containers of schedulable objects. This results in additional infrastructure for handling missions, adding to the hierarchy and methods, and complicating the specification even further.

3.1.2 Contributions

These language related issues lead to the work of Predictable Java (PJ), described in Paper B and the refactoring of SCJ and RTSJ presented in Paper D.

Paper B This paper presents a new profile proposal, Predictable Java (PJ), which is used as constructive criticism on the upcoming SCJ profile. This paper presents:

- The ideal profile designed using the principle inheritance for specialization, with PJ as a parent to RTSJ, instead of the reverse.
- An implementation directly on an open source JVM using Xenomai Linux.
- An implementation of PJ compliant with RTSJ, as it is recognized RTSJ is too important to ignore.
- A redesign of the Mission concept, with missions made first class handlers with initialization, termination and transition semantics.

This paper was presented at JTRES 2009.

Paper D In this paper it is argued there is a need for refactoring of SCJ and RTSJ. This new simplified structure results in a clean core set of interfaces and profiles for each SCJ level. The paper presents:

- The definition of a common core subset of the SCJ levels and RTSJ. This results in a common set of interfaces, named `rtcore`.
- Four profiles, corresponding to the three SCJ levels and RTSJ, which can be implemented separately.

This paper was presented at ISORC 2011.

3.2 Schedulability Analysis

Traditional schedulability analysis operates on very simple abstract models of real-time systems.

Starting from the pioneering work by Liu and Layland in the early seventies [100], since then, substantial research has been performed in extending the simple models and utilization based tests, to lesser restricted models including synchronization, sporadic tasks and release jitter.

The advantage of these methods are the fact they are easy to compute and easy to understand, and under the right restrictions they do provide tight results, and sometimes an exact schedulability verdict. However, the traditional approaches are somewhat limited in what behaviour can be expressed, while still keeping the computation and precision advantage of using these methods. For example there seems to be no general way of extending the traditional approaches with the ability to express interdependence between tasks; as an example a periodic task firing either *sporadic task A* or *sporadic task B*.

The shortcomings of traditional approaches are addressed in the TIMES tool [11]. Using TIMES, a developer can design a system specification, using a graphical user interface, and specify tasks and their relationships. TIMES allow sporadic, periodic and controlled tasks, dependencies between tasks, and synchronisation. Using the UPPAAL tool, the specification can be verified, and code can be generated from the verified model.

3.2.1 Tool Issues

Traditional approaches to schedulability analysis are limited to analysing a simple schedule, and given a set of tasks and their execution times, it is relatively easy to compute a verdict. The shortcomings to this approach and the limited model on which it operates, are addressed by tools, such as TIMES.

However, in a tool like TIMES, there is a distance between the execution time of the actual program and the analysed model. When using TIMES, one must ensure that the system implementing the specification actually has the same properties as the analysed model. Maintaining, and even ensuring this correspondence, is not trivial.

3.2.2 Contributions

Paper A This paper presents an approach inspired by the TIMES approach, in a tool called SARTS. The focus of this tool is schedulability analysis, however, with additional focus on the correspondence between code and model. The contributions of this work include:

- The design and implementation of a prototype tool capable of performing schedulability analysis of real-time Java programs.
- Schedulability analysis is performed on a timed automata model of the system, a model generated from compiled Java code to achieve the closest possible correspondence.
- A case study example successfully verified using this tool is presented.

This paper was presented at JTRES 2008.

Paper C This paper presents improvements to the SARTS tool, utilising the properties of the profile presented in Paper B, relaxing restrictions and increasing the reliability of the profile; these results can also be applied to SCJ. The contributions of this work includes:

- The argument that Java finalizers can be made predictable, and could in the end result in safer programs.
- An addition to traditional schedulability analysis including consideration of finalizers under certain conditions.
- An extension of the SARTS tool to include finalizers in the schedulability analysis.

This paper was presented at JTRES 2010.

Paper E This paper presents a branch of the SARTS tool, building upon abstract system specifications to achieve a compositional approach to schedulability analysis, while maintaining correspondence between specification and implementations. The contributions of this work include:

- The timed specification language, TRSL, for specifying task timing and synchronization behavior.
- The translation of TRSL to timed automata to verify language inclusion and hence the specification/implementation relation; it is argued code translates directly into TRSL, and hence the model/code correspondence is maintained.
- The translation of TRSL to timed automata to verify schedulability and related properties.

This paper was presented at ISORC 2012[1].

3.2.3 Perspective

Paper F gives an overview of the current state of real-time Java and related tools, and identifies further work.

[1]Nominated for best paper award.

Chapter 4

Future Work

Traditional methods rely on WCET tools for computing the cost of each task in the system. WCET analysis is also an area of substantial interest in the real-time community, with a myriad of mature techniques and tools [167]. Modern WCET analysis techniques include information about the hardware executing the code, and attempts are made to improve the analysis. This includes including the behaviour of the platform specific details, such as caches and pipelines, which are part of most modern hardware today. It is not obvious, however, how to benefit from the information about caches and pipelines in the analysis of a priority based preemptive system. Since the state of the cache and pipeline may be invalidated by every preemption during execution. Including every possible preemption in the analysis, would certainly affect the results. To be able to benefit from the state of caches and pipelines, the preemption task interaction should be considered, and not only a possible worst case execution time. A future direction would be combine SARTS with a WCET analysis tool such as TetaJ [65], since the analysis of TetaJ is based on a detailed model of hardware with more features than the JOP processor used in SARTS. The first steps in this tool combination have already been initiated.

In the current approach, code must be annotated with loop bounds in order to determine the number of loop iterations for a given loop. Such static bounds provided by the programmer is considered unsafe, and are in other perspectives problematic, e.g. when considering modularity in development as it is difficult and often impossible to put static bounds inside program libraries. Deductive reasoning is based on logic statements about the program written as program annotations, which is then proven to be correct using theorem proving techniques. Promising work in deductive methods has resulted in mature tools [22], and work has been done on finding provably correct loop bounds [82], using the Java modelling language[93]. A future direction is to add symbolic bounds and deductive techniques to the model to reduce the uncertainty of the analysis result, and provide a more accurate analysis.

This would allow verified TRSL-like specifications of real-time libraries.

Chapter 5

Papers

The papers on which this thesis is based, are presented in their full version in the following pages. The papers have been subject to reformatting to fit the layout of this thesis.

Paper A:

Model-based schedulability analysis of safety critical hard real-time Java programs

Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen,
Bent Thomsen, and Kim G. Larsen

Department of Computer Science
Aalborg University, Denmark

Abstract

In this paper, we present a novel approach to schedulability analysis of Safety Critical Hard Real-Time Java programs. The approach is based on a translation of programs, written in the Safety Critical Java profile introduced in [137] for the Java Optimized Processor [130], to timed automata models verifiable by the Uppaal model checker [25]. Schedulability analysis is reduced to a simple reachability question, checking for deadlock freedom. Model-based schedulability analysis has been developed by Amnell et al. [11], but has so far only been applied to high level specifications, not actual implementations in a programming language. Experiments show that model-based schedulability analysis can result in a more accurate analysis than possible with traditional approaches, thus systems deemed non-schedulable by traditional approaches may in fact be schedulable, as detected by our analysis.

Our approach has been implemented in a tool, named SARTS, successfully used to verify the schedulability of a real-time sorting machine consisting of two periodic and two sporadic tasks. SARTS has also been applied on a number of smaller examples to investigate properties of our approach.

1 Introduction

Traditional schedulability analysis are based on the *critical instant* and assume maximum interference and blocking; an approach which often results in a very pessimistic analysis. Due to this pessimistic nature, a new approach is desirable.

Several modeling tools exist, where the general idea is to model the system, and verify that certain properties hold. Some tools also allow the developer to check whether deadlines are missed, based on a scheduling strategy and a WCET for each task; other tools must be used to estimate this WCET. A tight correspondence between the model used in these tools and the actual implementation is required, in order to rely on the guarantees given. One such tool is the TIMES tool [11] which builds on timed automata models of systems and generates C code.

However, in many circumstances a high-level model of a hard real-time system from which code can be generated does not exist. Instead the code of the system has to be analyzed to give verifiable guarantees. This paper focuses on improving and automating the schedulability analysis of such systems, where the implementation language is Java.

Several real-time profiles for Java exists: the Real-Time Specification for Java (RTSJ) [35, 37], the Ravenscar-Java Profile [91], which is based on the Ravenscar profile for Ada [56], and the Safety Critical Java (SCJ) profile [137]. Furthermore, there is currently a huge standardization effort underway by academia and industry to provide a standard Safety Critical Java profile under the Java Community Process which has issued the Java Specification Request 302 (JSR-302).

RTSJ is a general, somewhat complex, real-time framework with many dynamic features. Often these dynamic features inhibit static analysis and dynamic checks have to be performed, e.g. checks for budget overruns and missed deadlines, with associated miss handlers. The Ravenscar-Java profile and the current direction of the expert group for the JSR-302, both define extended subsets of RTSJ, which remove many of the dynamic features of RTSJ, making them more suitable for static analysis. SCJ also removes many dynamic features and many parameters from RTSJ to ensure implementations are verifiable such that they can be deployed in high integrity systems. SCJ presents a programming model similar to the midlet model of J2ME MIDP for mobile phones. In SCJ release parameters of schedulable entities (periodic and sporadic threads) are time and not priority. The implementation uses a priority based preemptive scheduler which maps the time requirements according to the deadline-monotonic order. This relieves the programmer of the error prone assignment of priorities.

The approach developed in this paper, is the translation of an existing implementation of a hard real-time system written in the SCJ profile [137] for the Java Optimized Processor (JOP) [130], to an abstract time preserving model, on which the Uppaal model-checker [25] can be used to verify that deadline misses never occur. The schedulability analysis considers blocking, interference, context switches, and event interactions between tasks. This improves the accuracy of the analysis, while ensuring a tight correspondence between the model and the actual implementation.

The contributions of this paper, is the tool SARTS. SARTS performs a fully automatic translation of real-time Java applications into Uppaal models, on which schedulability analysis is performed using the theoretical foundations of Fersman and Yi [59, 60, 90]. It is shown how this approach can result in a more accurate result than possible with traditional approaches.

2 Related Work

A traditional approach to schedulability analysis, involves WCET calculation of tasks, and combining these with formulae, e.g. utilization test or response time analysis [45]. WCA [134] is a tool developed for JOP [130], supporting WCET calculation for a single method of a real-time Java program. The result is intended to be used in conjunction with the afore mentioned formulae.

Several modeling tools for Java already exists, such as Bandera [49] which translates Java source code to an intermediate representation, on which slicing and abstraction is performed. This intermediate representation is translated to abstract models, on which safety properties of the implementation can be verified. However, Bandera has no notion of time, which is critical in real-time systems, and is therefore not suitable for schedulability analysis.

The TIMES tool [11] already supports a schedulability analysis of a real-time system, but the focus is on high-level models of systems. However, TIMES supports generation of source code from the model, and is therefore an approach, opposite from that of SARTS. Furthermore, TIMES puts several restrictions on what computation is actually possible by periodic and sporadic tasks and TIMES includes no context switch or scheduler cost in the schedulability analysis, which may be significant in some systems.

Polychrony is another interesting tool, which allows translation of Java to its input language SIGNAL targeted hard real-time systems [149]. However, it is unclear how Polychrony handles WCET, and therefore how it can be utilized as a schedulability tool.

Java PathFinder [40] is a model checker to verify properties of executable Java bytecode programs. Java PathFinder is a Java Virtual Machine (JVM) that systematically explore all potential execution paths of a program to find violations of properties like deadlocks or unhandled exceptions. Java PathFinder has been used to verify properties of RTSJ programs, basically by implementing (a subset of) RTSJ on top of the Java PathFinder JVM using discrete event simulation as a basis for modeling time. Real-time threads are modeled in Java PathFinder as ordinary Java threads, constrained to run one at a time, modeling resource contention, such as scheduling, through discrete event programming [99].

3 SARTS

SARTS automatically translates real-time Java systems into UPPAAL models. The Java system must be implemented in SCJ; a safety critical hard real-time profile for Java [137] implemented and documented in [31]. SCJ supports periodic and sporadic tasks, and uses a fixed priority scheduler. Furthermore, a priority inversion control mechanism is included. In SCJ priority ceiling emulation is the only available protocol. Release parameters of schedulable entities (periodic and sporadic threads) in SCJ are time and not priority. An implementation that uses a priority based preemptive scheduler maps the time requirements according to the deadline-monotonic order. As SCJ does not allow dynamic creation of threads during mission phase this mapping can be done on the transition from the initialization to the mission phase. This relieves the programmer of the error prone assignment of priorities. SCJ does not have budget parameters, as WCET and schedulability analysis is supposed to be performed to guarantee that no budget overruns or deadline misses will ever happen, thus eliminating the need for miss handlers.

SARTS translates the Java application to SARTS Intermediate Representation (SIR), on which analyses and transformations are performed. SIR represents an abstraction of the actual Java bytecode via a class graph, where each class contains a set of methods represented as control flow graphs. SIR is extracted from a Java class file using the BCEL library [21].

In the current implementation, WCET calculation and simple collapsing is performed. SIR is translated to a UPPAAL model. For a description of UPPAAL see [24].

The following sections describe the principles of the translation to UPPAAL. The models are created to simulate the execution of the system on JOP. The scheduler, preemption, and interrupt mechanisms are modeled directly as the actual implementations on JOP.

3.1 The Scheduler

The purpose of the scheduler is to schedule the thread with the highest priority, according to a deadline monotonic priority assignment. The scheduler is depicted in Figure 1.

Initially the broadcast channel `GO!` is synchronized to ensure all threads are in their correct state. The scheduler simulates execution, by waiting for `wcet` time. If any schedulable thread exists, the highest priority thread is selected, by setting the corresponding index in the `running` array to 1. If no

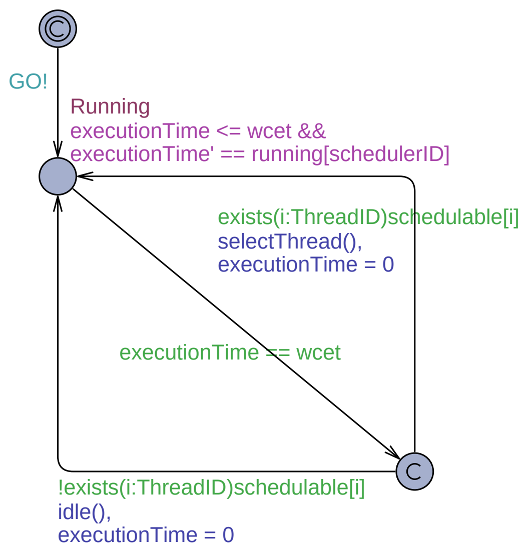


Figure 1: Scheduler

threads are schedulable all indices in `running` are set to 0. This is handled by the two functions `selectThread()` and `idle()` respectively. The values in the array, `running`, are used in stopwatch expressions to determine which thread is executing, modeling preemption.

3.2 Periodic Thread

For each periodic and sporadic thread in the Java program, a base model is added. This model must be supplied with parameters to determine its ID, period, deadline, and offset corresponding to the actual Java implementation of the thread. The base model for the periodic thread is depicted in Figure 2.

Initially the thread waits if an offset is specified. If the thread has a higher priority than the currently executing thread, preemption occurs and the scheduler is started, by calling `runScheduler()`. In the actual Java implementation, it is not the thread's responsibility to notify the scheduler, but the scheduler's responsibility to schedule interrupts at the correct time. However, this implementation is not suitable in UPPAAL, since it would make the model unnecessarily complicated. The implementation of `runScheduler()` is shown in Listing 1.

If the system is in a synchronized region an interrupt is queued. Once the synchronized region is left the scheduler is started if scheduled interrupts exist. Otherwise all threads are stopped and the scheduler is started.

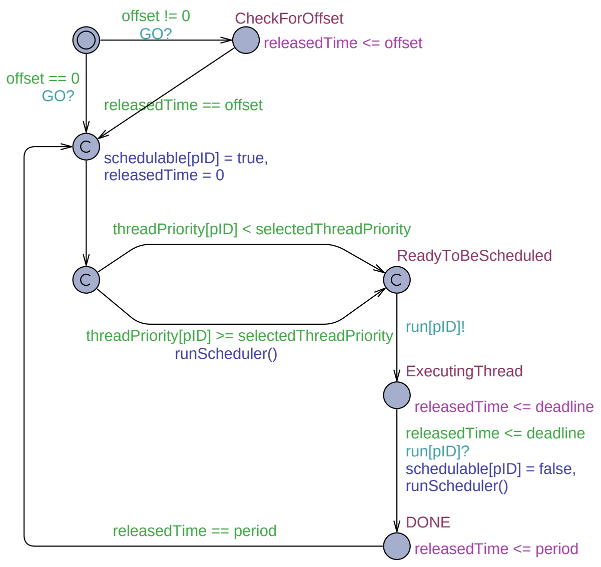


Figure 2: PeriodicThread base model

```
void runScheduler(){
  int i;
  if (synchronized){
    interruptWaiting = true;
  } else {
    for (i = 0; i <= totalThreads; i++){
      running[i] = 0;
    }
    running[schedulerID] = 1;
  }
}
```

Listing 1: Implementation of runScheduler in Uppaal

To start the run logic for the thread, synchronization is performed on the correct channel in the `run` channel array. This synchronizes with the template containing the run logic for the thread. The template waits in `ExecutingThread` for a synchronization on the same channel, indicating the thread is done with its run logic. An example of a run method is explained in Section 3.5. It is ensured that the thread has completed before its deadline, otherwise a deadlock occurs, and the system is not schedulable. The scheduler model is invoked to determine which thread to schedule next. The same procedure continues for each period of the periodic thread.

3.3 Sporadic Thread

The sporadic model is similar to the periodic model, except it must be invoked by synchronizing on the correct channel in the `fire` array. This synchronization occurs when a thread chooses to fire the given thread. The base model is depicted in Figure 3. This model must be supplied with parameters for its ID, minimum inter-arrival time, and deadline. When the run logic is done, the template waits in `DONE`. Once the minimum inter-arrival time has passed since the last release, the `firable` array is set to true for the specific thread, and it is ready to be fired again.

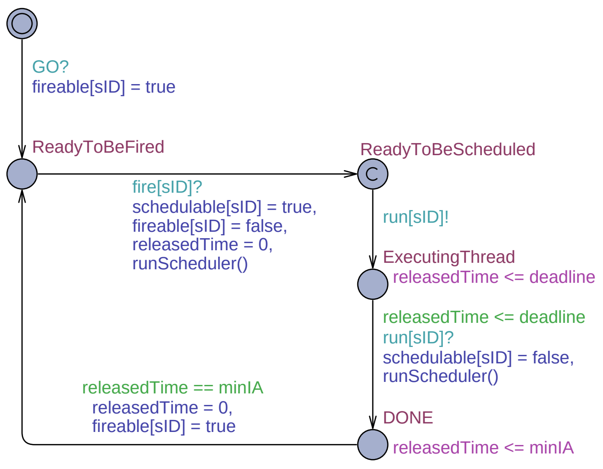


Figure 3: SporadicThread base model

3.4 Basic Blocks

As an abstraction to the actual Java bytecode, the concept of basic blocks is introduced. A basic block contains a list of the Java bytecode instructions it represents and the cost of executing these along with extra information, e.g. loop bound in the case of a loop basic block.

SIR consists of basic blocks, which are translated to a corresponding part of the UPPAAL model.

- **SimpleBasicBlock:** This is a sequence of bytecode instructions with exactly one predecessor and exactly one successor.
- **MethodCallingBasicBlock:** This represents a method invocation. It contains a set of possible methods, which can be invoked.
- **SporadicInvokeBasicBlock:** This is a special case of a method invoke, where a sporadic event is fired.
- **IfBasicBlock:** Represents an if branch, and therefore contains two outgoing edges.
- **LoopBasicBlock:** Represents any kind of a loop. It contains an estimated upper iteration bound for the actual loop, specified by the developer of the real-time system.

- **MonitorEnter- and MonitorExitBasicBlock:** Represent start and end of synchronized regions. Synchronized regions are handled by setting the variable `synchronized`, see Listing 1, to `true`; disabling interrupts.

- **EmptyBasicBlock:** These blocks do not represent actual instructions, and are added for convenience reasons, e.g. one is added in the beginning and the end of a method.

A simple basic block modeled in UPPAAL is depicted in Figure 4. An invariant is added to ensure the model stays in this state for as long as the WCET of the represented bytecode, denoted by `instX`. Whether the given thread is executing is denoted by `executionTime' == running[tID]`, a stopwatch expression. Preemption is done by setting `running[tID]` to `0`; stopping the clock, `executionTime`. The execution time is reset on the outgoing edge to reuse the clock in the next basic block.

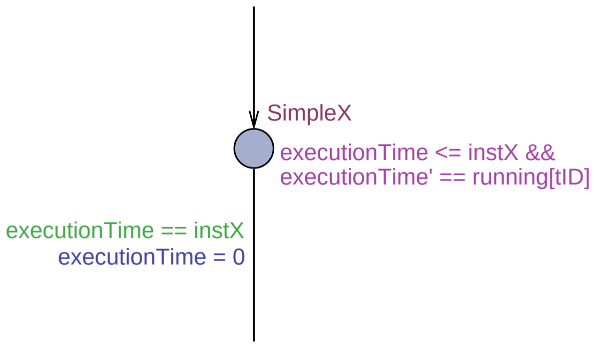


Figure 4: Simple basic block

Each basic block, uses the same notation to represent the correct amount of time spent in a state. The other types of basic blocks are implemented in a similar way, by adding the control flow, e.g. a branching basic block, and special variable updates, e.g. a monitor enter or exit.

3.5 Example

As a small example of how Java code is translated to a UPPAAL model, a simple periodic run method is shown in Listing 2. The translated UPPAAL model is depicted in Figure 5. This model is slightly modified from the translated model to reduce the size, and focus on the essential aspects. All invariants, guards, and updates have been removed, as they all follow the pattern of the simple basic block depicted in Figure 4, i.e. the model waits in each state for the correct amount of time, corresponding to the represented bytecode instructions.

```
protected boolean run() {
  if (condition){
    //then statements
  } else {
    //else statements
  }
  return true;
}
```

Listing 2: Run method example

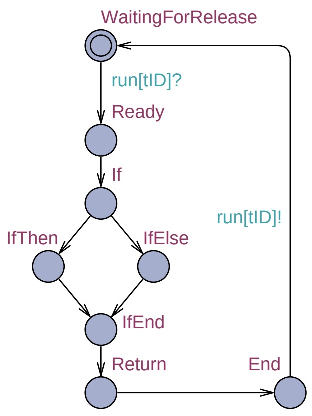


Figure 5: Translated Uppaal model

Each time the periodic thread is released, the template enters the `Ready` state. However, it does not start executing until the thread is selected by the scheduler. It enters the `If` state and performs a nondeterministic choice between the two branches, and returns from the method, by synchronizing on `run[tID]`. The corresponding periodic template, Figure 2, enters the `DONE` state, and waits for the period to elapse, before the periodic thread is released again.

3.6 Method Invoke

When a method invocation is performed, it corresponds to switching to another template in Uppaal. This is modeled as depicted in Figure 6. A model is created for each method in the system. Each of these has an array of channels, one for each thread. This is done to enable different threads to invoke the same method. A method invoke consists of synchronization on the correct channel, transferring control to the invoked method, and waiting for a synchronization on the same channel, meaning the method has returned.

In Figure 6, `methodName1` to `methodNameN` denote the uncertainty of method invokes due to dynamic dispatching.

Using this design Uppaal will nondeterministically consider all possible method candidates for this call.

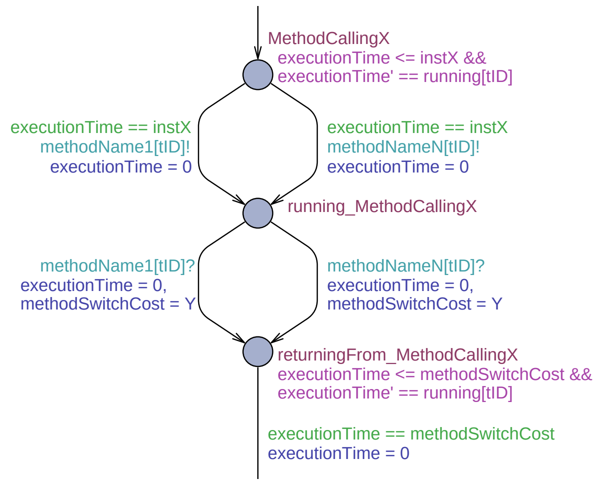


Figure 6: Invoke of a standard method

JOP has a method cache to improve performance, however, the method cache complicates WCET analysis and in the current implementation of SARTS method caches are always assumed to miss, thus making a conservative approximation. We expect the rather conservative approximations that both dynamic dispatch and cache miss assumption introduce, can be reduced significantly by combining traditional control flow and call graph analysis within the SARTS model generation module.

4 Case Study

A case study of a real-time system has been implemented in SCJ. It was originally designed and implemented in [32]. The system is a sorting machine called RTSM, depicted in Figure 7. The machine is built in LEGO, using motors and sensors controlled and monitored by JOP[1].

RTSM is a prototypical example of a hard real-time system, representative of many real-life real-time control systems. It includes periodic and sporadic tasks, blocking regions, and dependencies between tasks. These are interesting properties of a system, when performing schedulability analysis.

The implementation contains two periodic and two sporadic tasks, where the sporadic tasks are two instances of the same class. The cyclomatic complexity of each tasks in the system is shown in Table 1, indicating the complexity of the system being verified. The system consists of 17 different methods, and contains more than 300 lines of code. The generated model contains 20 templates, one for each method and the three base models. The instantiated system contains 65 instances of templates, and a total of about

[1]A video of the machine in action is available at `http://sarts.boegholm.dk/`

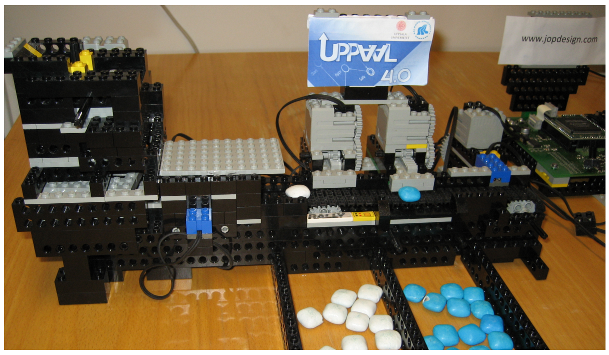


Figure 7: Real-Time Sorting Machine

Task	Cyclomatic Complexity
Periodic 1	9
Periodic 2	17
Sporadic	7

Table 1: Cyclomatic complexity of tasks

700 template locations.

Periodic 1 reads the input from the sensors, to determine whether an object has passed by, and of which color. Each time an object is detected, the time of detection is added to a bounded buffer. *Periodic 2* reads this buffer, and fires a sporadic event, when the object must be pushed off the conveyor belt. The two sporadic tasks push off the objects depending on their color.

```
protected boolean run(){
  if (state == IDLE){
    motor.setMotorPercentage(
             Motor.STATE_FORWARD,
             false, 100);
    state = FORWARD;
  } else if (state == FORWARD){
    motor.setMotorPercentage(
             Motor.STATE_BACKWARD,
             false, 100);
    state = BACKWARD;
  } else if (state == BACKWARD){
    motor.setMotorPercentage(
             Motor.STATE_BRAKE,
             false, 100);
    state = IDLE;
  }
  return true;
}
```

Listing 3: Code example from RTSM

The code in Listing 3 contains the run logic for the sporadic thread pushing objects off the conveyor belt. This sporadic thread is fired three times for each object detected on the conveyor belt, keeping an internal state for each operation, *forward*, *backward* and *brake*. Note that the method invoke, `motor.setMotorPercentage` has a cyclomatic complexity of 2. A simple version of RTSM, called RTSM$_{\text{SIMPLE}}$, has been developed for performing experiments. RTSM$_{\text{SIMPLE}}$ has a lower complexity, which allows for faster verification. The difference between RTSM and RTSM$_{\text{SIMPLE}}$ is that the periodic thread, *Periodic 2*, can only fire the sporadic threads at two code points instead of six in RTSM.

5 Experiments

This section presents experiments conducted to evaluate the implementation of SARTS. In [31] it is shown that SARTS is comparable to WCA [134] in terms of WCET analysis accuracy.

The experiments presented here show how the model-based schedulability analysis is able to exploit the control flow of the analyzed system, in order to achieve a more accurate analysis result, followed by experiments illustrating the scaling properties of the generated models.

5.1 Conditional Sporadic Events

For this experiment, the example system consists of one periodic thread and two sporadic threads. The logic of the run method for the periodic thread is shown in Listing 4, in which the periodic task `Experiment1` fires either event 1 or event 2, but never both in the same period. The period and minimum inter-arrival times are set to 4 microseconds, and the sporadic tasks have the same WCET.

```
public class Experiment1 extends PeriodicThread {
  public boolean run() {
    if(b)
      RealtimeSystem.fire(1);
    else
      RealtimeSystem.fire(2);
    return true;
  }
}
```

Listing 4: Conditional sporadic invoke

The WCET for the periodic run method is 161 clock cycles and 64 clock cycles for the sporadic run method. The period calculated into clock cycles is 240, and the calculation of the processor utilization is performed as follows:

$$\left(\frac{161}{240}\right) + \left(\frac{64}{240}\right) + \left(\frac{64}{240}\right) = 1.20$$

Following traditional schedulability analysis approaches, this system will be deemed not schedulable since processor utilization is greater than 1 [45]. Running SARTS on this system will correctly show it as being schedulable, since the model checker can deduct that the two sporadic events will never be fired at the same time. A time-line for the execution of the system can be seen in Figure 8.

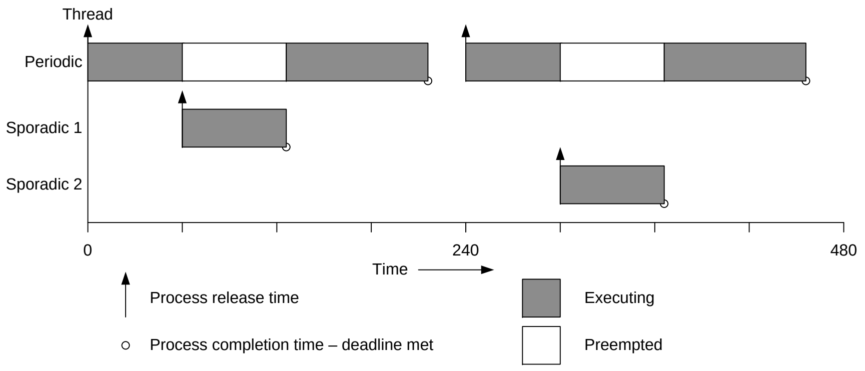


Figure 8: Time-line for conditional sporadic invoke

5.2 Scalability

Several experiments have been conducted to illustrate the scalability of SARTS. The experiments consider only the time used to verify the system in Uppaal, since the translation time is insignificant. The example system being verified is RTSM$_{\text{SIMPLE}}$ compiled using two different Java compilers generating slightly different code. This small change in the generated bytecode is, enough to make a measurable difference in verification time.

The execution of the verification for the two generated systems is shown in Table 2. These results indicate that even small variation in the generated bytecode, can lead to huge variations in the verification time of the generated model.

Compiler	**Verification time**	**Result**
Javac	14m 29s	Satisfied
Eclipse	1m 55s	Satisfied

Table 2: Verification time of RTSM$_{\text{SIMPLE}}$

The cause of the difference in verification time is illustrated in Figure 9 and 10. These two models are semantically equivalent, disregarding execution time. The Javac version, Figure 9, has a single return statement and a jump to this statement from the other branch, the Eclipse version has two return statements.

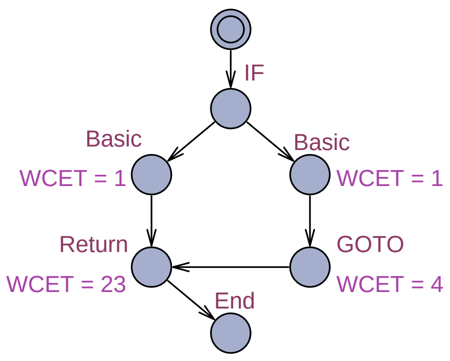


Figure 9: Javac compiled model

In the Eclipse version, Figure 10, both branches have the same execution time. The state of the model, when the template enters the `End` location, is therefore independent of the previous branch, since the choice of branch becomes insignificant and thus no additional traces are considered by the model checker.

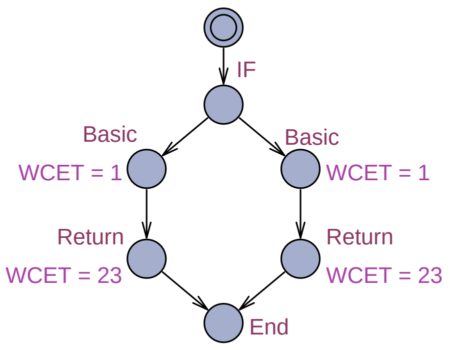


Figure 10: Eclipse compiled model

The result of this experiment, is that small factors in the system and hence the model generated, have significant impact on verification time.

However, it is possible to reduce the time needed to verify the systems, using the options available in UPPAAL. Additional tests have therefore been conducted. Table 3 is the same experiments where a depth first search instead of breath first search is used, and aggressive state space reduction is enabled.

UPPAAL also supports a convex-hull approximation option, reducing verification time at the cost of an over approximate answer. If UPPAAL using convex hull determines a safety property to be satisfied, then it is also satisfied without the approximation. The result of this experiment is shown in Table 4.

In addition to verifying $\text{RTSM}_{\text{SIMPLE}}$, the full version of RTSM has also

Compiler	Verification time	Result
Javac	4m 23s	Satisfied
Eclipse	51s	Satisfied

Table 3: Verification time of RTSM$_{\text{Simple}}$ using depth first search and aggressive state space reduction

Compiler	Verification time	Result
Javac	16s	Satisfied
Eclipse	9s	Satisfied

Table 4: Verification time of RTSM$_{\text{Simple}}$ using convex-hull approximation

been verified, requiring substantially more verification time than RTSM$_{\text{Simple}}$ due to the increased complexity. The verification times using different optimizations can be seen in Table 5. In all cases, the system is deemed schedulable.

Settings	Compiler	Verification time
Standard	Javac	27h 15m 26s
Standard	Eclipse	5h 42m 10s
Aggressive	Javac	6h 30m 01s
Aggressive	Eclipse	1h 28m 29s
Convex Hull	Javac	52s
Convex Hull	Eclipse	37s

Table 5: Verification time of Full RTSM

The experiments show that even small changes in the analyzed program, the compiled code, or the Uppaal template can significantly increase the verification time. How the different parameters interact is still an open research question.

6 Improvements

In the current implementation all branches and loops from the Java code are present in the Uppaal model. This is done to maintain the control flow of the actual application, however, it will be possible to collapse branches if this change is made to the analyzed system as well, to keep consistency. Collapsing branches applies to branches where the contained instructions do not affect the overall system, such as firing a sporadic task. This could be done by calculating the worst case path through the branch and creating a single basic block with WCET equal to that worst case. This way the state space could be significantly reduced, since fever traces are explored by the model checker. The code being analyzed must be changed correspondingly to correctly reflect this change in the model. This is due to interleavings, where a blocking region can be moved past a point where it would have prevented a higher priority thread from executing.

In order to illustrate this problem, a small system consisting of two threads is analyzed. The actual time-line for the execution is depicted in Figure 11. The thread *b* includes a blocking region larger than the deadline of the higher priority thread *a*, resulting in a deadline miss. Note, this is due the implementation of synchronized regions on JOP, which implements the priority ceiling protocol with all locks assigned the highest priority.

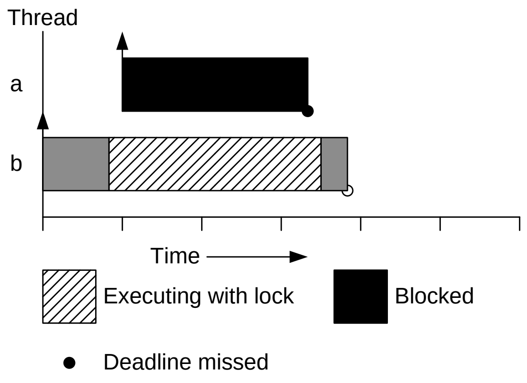


Figure 11: Blocking example, actual execution

The actual implementation of the system is therefore not schedulable. However, a pessimistic WCET in the verified model might postpone the blocking region, moving it past the release of task *a*, deeming the system schedulable, depicted in Figure 12. It is therefore necessary to consider all paths with different execution time, in order to rely on the result.

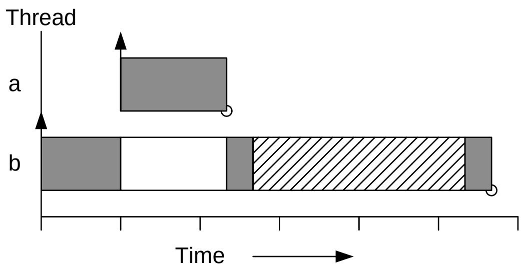


Figure 12: Blocking example, generated model

A way to circumvent this problem is to perform changes in both the model and the program itself, i.e. the Java class file, by padding the cheapest branch, adding execution time, a technique well know in secure applications to prevent timing attacks.

As an example, consider a simple branching if-statement. The cheapest in terms of execution time, of the two branches, could be padded with `nop` instructions with the execution time of 1 clock cycle, such that the branch is execution time symmetric. Since selecting either branch is of no significance to the execution time, this branch can be collapsed into one block.

This technique will add execution time to the system, but the WCET is preserved and the additional time is therefore not a problem for the overall system, since average execution time is not important.

7 Conclusion

In this paper we have presented a novel model-based schedulability analysis of Java based safety critical hard real-time systems. The approach has been implemented in the SARTS tool, which automatically translates a real-time system implemented in Java to an abstract time preserving UPPAAL model.

Verification can be performed on this model and schedulability analysis translates into a simple reachability question checking for deadlock freedom. The translation is an abstraction of the Java code, including an analysis of the actual bytecode, in order to determine the WCET. The WCET analysis is based on published bytecode execution time for the FPGA implementation of JOP [131].

The automatic translation from Java to UPPAAL ensures direct correspondence between the actual implementation, and the model being verified. This automatic translation also allows the developer to abstract away from the actual verification process and no knowledge of model checkers is required. In the future we invision SARTS integrated into the Eclipse development environment. Currently the developer has to annotate loop bounds, which is a potential source of errors. However, we believe that this source of errors could be eliminated by integrating into SARTS the loop bounds analysis presented in [82].

Several experiments have been conducted, in order to compare SARTS to existing techniques and tools, and the actual execution on JOP. The results are that SARTS is capable of performing WCET analysis comparable with tools such as WCA. Furthermore, the model-based approach can lead to more accurate results than traditional approaches to schedulability analysis. We believe the more accurate analysis can deem systems runnable on cheaper hardware.

The improved accuracy comes at the cost of verification time, and scalability of the approach is clearly dependent on scalability of the model checking tool. Currently there is an upper bound on the state space the UPPAAL system can handle. Clearly more powerful implementations of UPPAAL, such as the current effort to implement it on 64bit multi-core systems, can analyze more complex systems. Furthermore, the translation from Java to timed automata can perhaps be improved to reduce the complexity of the verified model. As our experiments show, even the small difference in semantically equivalent code generation between javac and the Eclipse compiler, yield a huge difference in the state space.

Currently only the SCJ profile introduced in [137] is supported and the only execution platform supported is the JOP [130]. We believe supporting other Java processors such as the AJ-100 from aJile Systems [6] is straightforward, only requiring published execution times for all Java bytecode instructions. A somewhat more ambitious goal is to support real-time JVMs on mainstream real-time Linux platforms on ARM and Intel Processors as getting time predictable Java bytecode instruction will depend on JVM implementation, operating system, and hardware platforms.

The implementation of the SCJ profile on JOP uses a fixed priority scheduler with deadline monotonic priority assignment. We believe that experimenting with other scheduling policies and priority assignments, such as Earliest Deadline First and Value-Based Scheduling (VBS), should be possible, only requiring a change to the Uppaal template modeling the scheduler.

There is currently huge standardization effort underway by academia and industry to provide a standard Safety Critical Java profile under the Java Community Process which has issued the JSR-302. The SCJ profile shares many commonalities with JSR-302 and we believe that in the future we will be able to analyze JSR-302 compliant Java programs adhering to the upcoming standard, at least to level 1.

A much more challeging task is to apply our approach to programs written in RTSJ with its many dynamic features. However, as our approach shares some commonalities with the approach used to model RTSJ in Java PathFinder in [99], such as modeling threads as coroutines running one at a time, scheduled by resource contention through discrete events, we have some expectations that this might work with Uppaal.
A web based version of the SARTS tools has been developed, and is available at `http://sarts.boegholm.dk/`.

8 Acknowledgment

We would like to thank Anders P. Ravn, Martin Schoeberl, Hans Søndergaard, and Stephan Korsholm for feedback and interesting discussions, and Alexandre David for feedback on Uppaal technicalities.

Paper B:

A predictable Java profile: rationale and implementations

Thomas Bøgholm[1], René R. Hansen[1], Anders P. Ravn[1], Bent Thomsen[1], and Hans Søndergaard[2]

[1]*Department of Computer Science*
Aalborg University, Denmark

[2]*VIA University College*
Horsens, Denmark

Abstract

A Java profile suitable for development of high integrity embedded systems is presented. It is based on event handlers which are grouped in missions and equipped with respectively private handler memory and shared mission memory. This is a result of our previous work on developing a Java profile, and is directly inspired by interactions with the Open Group on their on-going work on a safety critical Java profile (JSR-302). The main contribution is an arrangement of the class hierarchy such that the proposal is a *generalization* of Real-Time Specification for Java (RTSJ). A further contribution is to integrate the mission concept as a handler, such that mission memory becomes a handler private memory and such that mission initialization and finalization are scheduled activities. Two implementations are presented: one directly on an open source JVM using Xenomai and another, based on delegation, on an RTSJ platform.

1 Introduction

The Real-Time Specification for Java (RTSJ) [36] was a major breakthrough for Java as a language for programming embedded software, but soon after it had emerged, the discussion began as to whether it was too large or too dynamic to really support high integrity applications. This led to proposals for smaller profiles with a rationale presented in [122] and an implementation in the Ravenscar Java profile [91]; the focus was on embedded systems and on making programs amiable to analysis with state-of-the-art techniques. These formed a starting point for our own work with a predictable Java profile [144, 137] which considered using integrated analysis tools instead of programmer supplied parameters to provide predictable programs. More recently the Open Group has formed a committee to develop a profile for Safety-Critical Java [152, 12], and they have outlined their approach in a recent paper [76]. During the work in the committee we have had access to intermediate drafts which we have commented. This paper is a summary and consolidation of points where we find that the current draft may improve.

We have found the Open Group's approach refreshing, because they solve a major issue by settling for handlers as the schedulable entities in a real-time system. RTSJ supports both a handler paradigm and a thread paradigm, but the latter is hampered by inheritance from Java threads, for instance with unwanted asynchronous interrupts and conditional waits. Threads are not really suitable as logical processes. The handler concept is much closer in spirit to a logical process. In this, as in many other details, we agree with the SCJ draft.

The points where we would see further advances, and thus the contributions of this paper are:

1. An arrangement of the class hierarchy such that the proposal is a *generalization* of RTSJ, as we believe it should be, because it has far fewer details and options. The SCJ draft uses *specialization.*

2. The SCJ draft organizes handlers in missions. We propose to make missions first-class handlers, such that initialization, termination and transition between missions may be given a precise semantics. Furthermore it makes mission memory equal to handler memory.

In Section 2 we elaborate on constructs that make missions first-class schedulable entities which gives a structured replacement for the Scheduling Groups of RTSJ and the Thread Groups of Java. These concepts form the core of our proposal for a Predictable Java (PJ) profile[2].

It is evident that for practical reasons any viable profile must be compliant with RTSJ, but we observed that defining a profile by *specializing* or subclassing RTSJ leads to much clutter. That this must be so is rather clear

[2]We have avoided to name the profile SCJ in order to avoid confusion, but if the main ideas are taken up by SCJ, PJ has served its purpose.

if one considers which profile really extends the other: RTSJ is a very flexible and detailed theory, whereas what we are searching for is an abstraction or *generalization* of it. Thus in an ideal world, RTSJ would be a specialization or refinement of a smaller profile. Doing it the other way around is like deriving the class of natural numbers from a class of rational numbers: sign inversion would need to be disallowed, division would become a partially defined operator, subtraction likewise, etc. Yet, the world is not ideal, so in a first step, we define a profile by manually abstracting from RTSJ classes and interfaces for relevant entities. Then a few specific classes are introduced to deal with handlers and missions. This gives a very compact and orthogonal organization of a package for the profile which is explained in Section 3. For example, the RTSJ-class `AsyncEventHandler` has 7 constructors and 32 methods. In PJ, we need of those only 1 constructor and 6 methods in the superclass `ManagedEventHandler` - a considerable reduction. Another benefit compared with the *specialization* approach is that we avoid annotations like `@SCJAllowed` annotations to prohibit unwanted methods.

An application that uses the PJ package may compile and, given that semantics is preserved, run under RTSJ with an adapter layer that through delegation disallows some RTSJ methods and give default values for some parameters. This is demonstrated in Section 4. Furthermore we outline a more direct implementation with a modified JamVM [85] on top of Xenomai [68] and Linux.

Finally, Section 5 investigates what kind of static constraints are needed to make the profile truly predictable and what tool support would be feasible for checking the constraints; and in Section 6 we conclude.

2 Key Concepts

In this section we introduce the key concepts in the Predictable Java profile: handlers and mission and the supporting resource concepts of memory and schedulers. The interplay between resources, handlers and missions determine predictability of applications.

2.1 Handlers

An application programmer must have the means to define temporal scopes [45] (period, deadline, execution time budget) and the specific algorithm for handling a periodic or aperiodic event triggered computation when developing real-time applications. This is succinctly encoded as a periodic or aperiodic event handler.

A periodic event handler in an application is a specialization of the `PeriodicEventHandler` of the profile, see Section 3.

```
class Periodic extends PeriodicEventHandler
{
  protected Periodic(PriorityParameters priority,
                     PeriodicParameters pp,
                     Scheduler scheduler,
                     MemoryArea memory)
  { super(priority, pp, scheduler, memory); }

  public void handleEvent() {
    // the logic to be executed every period
  }
}
```

It defines the temporal scope in the parameter `pp`. The `priority` parameter is for use by a `scheduler` and the handler has a `memory` to be used during execution of the algorithm. These parameters are concerns of the system programmer that assembles the application; the programmer of individual tasks focuses on giving an algorithm that specializes the `handleEvent` method.

The semantics is a periodic execution of the algorithm with the given period within its deadline which both are specified in the value `pp` of the `PeriodicParameters` object. The semantics is conditional on the algorithm completing within its execution time budget, included in `pp`, without declaring more temporary objects than can be accommodated in the `memory`, and refraining from declaring non-temporary objects. These semantic conditions can be checked by conservative approximation using abstract interpretation. This is discussed further in Section 5.

The aperiodic event handler is very similar:

```
class Aperiodic extends AperiodicEventHandler
{
  protected Aperiodic(PriorityParameters priority,
                      AperiodicParameters ap,
                      Scheduler scheduler,
                      MemoryArea memory)
  { super(priority, ap, scheduler, memory); }

  public void handleEvent() {
    // the logic to be executed when an event occurs
  }
}
```

It defines the deadline and cost, in the parameter `ap`. The remaining parameters are analogous to the ones for a periodic handler. The logic is given by specialization of the `handleEvent` method.

The semantics is an activation and execution of the algorithm within its deadline when an event is pending on an `Event` object to which the handler is attached. As for periodic handlers, the semantics is conditional on the algorithm completing within its execution time budget, without declaring too many temporary objects and refraining from declaring non-temporary objects.

A further condition is that event occurrences are sufficiently separated. A safe interpretation is that they are separated at least by the specified deadline. However, this may be unduly pessimistic, for instance with a shutdown handler, thus we have considered adding a minimal interarrival time to the parameters, as in RTSJ `SporadicParameters`. The troublesome point is whether this is statically checkable. For external events, this is clearly not possible; they must be assured by assumptions about the environment. However, for internal events, model checking based tools like TIMES [11] are able to combine analysis of event passing with schedulability analysis, again assuming some conservative approximation of the actual algorithm.

A bolder interpretation of aperiodic events is that they are just indistinguishable ticks, counting them is sufficient, and with a `long` counter, there should be space for even the most lively interrupt generators. It is then up to the application to handle and reset the counter. Such liberal semantics could be considered.

2.2 Missions

The functionality of an application is made up of handlers; but they have to be executed according to a feasible schedule implemented by a scheduler. Handlers use private memories for temporary objects, but they may use more permanent shared objects protected through mutual exclusion mechanisms as specified by the Java `synchronized` method qualifier. These are placed in the memory of the mission. The embedded software systems programmer designs the architecture in terms of missions that encapsulates a set of handlers.

A mission is essentially a set of tasks that collaborate on providing a desired functionality. When application functionality changes over time - mode transitions - there are multiple missions. In very simple cases there is only one mission, which may need an initialization and termination. In more complex cases, missions may compose sequentially, conditionally or even in parallel, from which there is but a small step to having statically nested missions.

Already initialization and termination indicates that a mission is more than a simple container for a set of handlers. It is itself a *handler* for termination or in some cases an initialization event. Thus we collect the responsibilities of a mission in a handler class that contains handlers.

The alternative, the `Mission` as a simple container for handlers, we found, would further complicate the profile: a new class hierarchy is introduced along with special mission memory, and even new types of handlers for mission start, stop, initialize etc. might be introduced, to handle missions, and possibly more. This does not contribute to the framework since what is needed to express the mission concept already exists in handlers: private handler memory, used as mission memory, the handler concept allows functionality to be expressed, such as start, stop etc.

The code below gives the termination handler aspect of the mission in the `handleEvent` logic. Initialization is done in the constructor of a mission, where the container aspect is the vector of `eventHandlers`. Individual handlers belonging to the mission are added by the `addToMission` method. Note that handlers can be added only, and only during initialization, thus a mission contains a static and finite set of handlers. One could consider replacing the dynamic vector with a simple array since the length is known at initialization time.

```
public class Mission extends AperiodicEventHandler
{
  Vector<ManagedEventHandler> eventHandlers;

  protected Mission(PriorityParameters priority,
                    AperiodicParameters ap,
                    Scheduler scheduler,
                    MemoryArea memory)
  {
    super(priority, ap, scheduler, memory);
    eventHandlers = new Vector<ManagedEventHandler>();
  }

  public void addToMission(ManagedEventHandler eh)
  { eventHandlers.add(eh); }

  ...

  public void handleEvent() {
    // the logic to be executed to terminate a mission
  }
}
```

The elided part denoted by the ellipsis above are methods that interact with the scheduler.

```
  public Vector<ManagedEventHandler> getEventHandlers() {
    return eventHandlers;
  }

  public boolean add() {
    return getScheduler().add(this);
  }

  public boolean remove() {
    return getScheduler().remove(this);
  }
```

A mission is submitted to its scheduler by `add`. The scheduler knows that the calling handler is a `Mission` and contains a list of handlers. They can be accessed through `getEventHandlers`, and it is then up to the scheduler to schedule the set if it is feasible. The handlers of a mission are assumed to be started from a common start time.

Correspondingly, as a step in termination, the set may be removed from the scheduler through `remove`. There are several possibilities for a termination semantics. These are discussed below under schedulers in subsection 2.4. The method `getScheduler` is defined in the superclass, `ManagedEventHandler`.

2.2.1 Mission examples

A basic mission contains periodic handlers without any termination. It uses a cyclic scheduler declared in `main` and has a Linear Time `LTMemory` at its disposal. Since it never terminates and therefore does not need to be scheduled as a handler, it does not need any priority or release parameters.

```
public class Basic extends Mission
{
  protected Basic(PriorityParameters priority,
                  AperiodicParameters ap,
                  Scheduler scheduler,
                  MemoryArea memoryArea)
  {
    ... // initialization
  }

  public static void main (String[] args) {
    new Basic(null, null, new CyclicScheduler(),
              new LTMemory(10*1024));
  }
}
```

The interesting part is the initialization of the periodic handlers (the ellipsis above):

```
   RelativeTime C = new RelativeTime(4,0);
   RelativeTime D = new RelativeTime(20,0);
   RelativeTime T = new RelativeTime(20,0);

   PeriodicParameters pp = new PeriodicParameters(C,D,T);

   addToMission(new Periodic(null,pp, getScheduler(),
                               new LTMemory(1024)));

   addToMission(new Periodic(null,pp, getScheduler(),
                               new LTMemory(1024)));

   add(); // mission to its scheduler
```

The priority parameters are not needed for a cyclic scheduler and are therefore left as null; but it sets up the release-parameters for the periodic handlers. The scheduler is the cyclic scheduler, and both periodic handlers get a private Linear Time memory. The concrete numbers in the parameters are arbitrary.

Next, we modify the example to include a termination event that can be triggered by a periodic handler. Since the mission is an aperiodic event handler, we use a `PriorityScheduler`. Furthermore a static `event` is used to signal a termination request from the application handlers. This event is handled by the `handleEvent` of the mission. Note that the mission as a handler is included in its handler set.

```
 public class Extended extends Mission
 {
   static AperiodicEvent event;

   protected Extended(PriorityParameters priority,
                      AperiodicParameters ap,
                      Scheduler scheduler,
                      MemoryArea memoryArea)
   { // set up periodic or other aperiodic handlers
     // with prioritites etc.
     ...
     event = new AperiodicEvent(this);
     add(); // start mission
   }

   public void handleEvent() {
     remove(); // from scheduler
     ...       // clean up etc.
   }

   public static void main (String[] args) {
     new Extended(new PriorityParameters(10),
                  null,new PriorityScheduler(),
                  new LTMemory(10*1024));
   }
}
```

With missions, it is possible to build sequential and concurrent missions, assuming that the selected scheduler is able to handle it. An outline of a sequential composition of three sub-missions is:

```
public class ThreeSequentialMissions extends Mission
{
  private Mission[] mission;
  private int active = 0;

  static AperiodicEvent event;
  public ThreeSequentialMissions(...) {
    mission = new Mission[3];
    // set up the three missions
    mission[0] = new Mission(...);
    // add handlers for mission 0
    // including the mission termination
    ...
    mission[1] = new Mission();
    ...
    mission[2] = new Mission();
    ...

    // start the first mission
    mission[active].add();

    event = new AperiodicEvent(this);
  }

  public void handleEvent() {
     mission[active].remove();
     active = (active + 1) % mission.length;
     mission[active].add();
  }
  ...
}
```

The outer mission removes and terminates the inner missions one at a time and starts the next one. Note that the handlers are initialized once. When the next mission is started, any shared objects will have the state in which they were left by the previous mission. If a reinitialization has to take place, the local missions must define a termination handler that takes care of reinitialization.

Analogously, missions may include sub-missions to be executed concurrently. That is, if the scheduler can accept it.

2.3 Memory

When the runtime system starts an application like `Basic`, an object of this class and objects for its arguments are created in what is usually called the heap (but here without GC). The lifetime of those objects is the lifetime of the application.

Since `Basic` extends `Mission` which is an aperiodic event handler this has as consequences:

- Immortal memory in the sense of RTSJ, which contains objects that has to live for the duration of the mission, is not needed, because the scoped memory belonging to the mission, here `Basic`, takes over this role.

- Just one kind of scoped memory is necessary, because there is no semantic difference between mission memory and private memory for a handler.
- Scoped memories are private for their handlers. In particular, a mission's memory contains objects shared by the set of event handlers of the mission.

This simplifies the memory hierarchy. Yet, in order to be compliant with RTSJ we have retained the following classes in a hierarchy. The abstract class

```
public abstract class MemoryArea
{
  // dummy implementations, exceptions not considered
  protected MemoryArea(long size) {}
  public void enter(Runnable logic) {}
  public static MemoryArea getMemoryArea(Object object){return null;}
  public Object newArray(Class type, int number)  { return null; }
  public Object newInstance(Class type)   { return null; }


  public Object newInstance(Constructor constructor,Object[] args){
    return null;
  }
}
```

the intermediate abstraction

```
public abstract class ScopedMemory extends MemoryArea
{
  public ScopedMemory(long size) {
    super(size);
  }
}
```

and the concrete

```
public class LTMemory extends ScopedMemory
{
  public LTMemory (long size) {
    super(size);
  }
}
```

Concrete implementations of `MemoryArea` are shown in Section 4.

2.4 Schedulers

A scheduler must support the mission. Therefore the constructor to a `Mission` has a parameter of type `Scheduler`, see Section 2.2.

The concrete schedulers specialize an abstract profile class:

```
public abstract class Scheduler
{
  private static Scheduler sc;

  protected abstract boolean add(Mission mission);
  protected abstract boolean remove(Mission mission);

  public static Scheduler getDefaultScheduler() {
       return sc;
  }
  protected static void
    setDefaultScheduler(Scheduler scheduler) {
       sc = scheduler;
    }
}
```

We have removed the feasibility checking of RTSJ, because it aims at systems where handlers are added and removed dynamically. Please note that even with nested missions, the overall structure is known at compile time.

New, is the concept of adding and removing missions as a whole. For a simple scheduler that accepts a single mission at a time, the most interesting semantics is termination. We would suggest a mode change semantics, where termination takes place only at points where no handlers are released [143]; but other mode change semantics are certainly possible.

An open point is whether missions are structured counterparts of a RTSJ `ProcessingGroup`. This requires further investigation [165].

Two concrete schedulers are shown in Section 4: a cyclic executive scheduler for a mission with solely periodic event handlers and a fixed-priority preemptive scheduler for missions with both periodic and aperiodic event handlers. These are the schedulers needed to run the examples of Section 2.2

2.5 Synchronization

Objects, shared between handlers, are placed in the mission's memory. In a simple profile, mutual exclusion locking is done by synchronization of the methods of the shared object. Synchronization at block level is not considered because of its added complexity, see the discussion in [45].

Priority inversion can be avoided, either by priority inheritance or priority ceiling emulation. The open source implementation in Section 4 uses priority inheritance because priority ceiling emulation is not implemented in Xenomai.

2.6 IO

Interfacing to devices is an important aspect of real-time applications, and we suggest a solution with Device Objects and Interrupt Handlers; this is discussed in detail in [138, 89].

3 Building the hierarchy

The Predictable Java profile we propose organizes the concepts discussed above in a hierarchy which is a generalization of RTSJ. Here it is useful to recall what inheritance may be used for. Inheritance is a way of forming new classes, allowing subclasses to inherit commonly used state and behaviour from a superclass. This can be interpreted in different ways. Budd [43] lists seven forms of inheritance, of which the most interesting for our purpose are:

Inheritance for specialization: Here a new class is a specialized class of the parent class. The new class satisfies the specification of the parent class. Thus, the new class is a subtype of the parent class. This corresponds to a refinement in a formal semantics setting or a conservative extension in logics.

Inheritance for limitation: (often called implementation inheritance), here the new class redefines the behaviour of the parent class. The new class does not satisfy the specification of the parent class. This means that in the subclass only some of the behaviour from the parent class is allowed in the subclass. This can be done in different ways: a) overriding the unwanted methods in the subclass and let them throw an `IllegalMethod` exception; b) using Java annotations. This does not fit well with formal semantics or logics.

It is evident that since a predictable Java profile is smaller than RTSJ, it cannot be derived by specialization, and for semantic reasons, we are not happy to pursue a limitation approach.

3.1 The ideal profile

The Predictable Java profile is based on RTSJ. It is a restriction of RTSJ in line with Ravenscar Java [91] and Safety Critical Java [76], but those two profiles are both defined from RTSJ by "inheritance for limitation" which results in much clutter.

To get a clean and compact PJ profile, "inheritance for specialization" was used. By this, RTSJ was regarded as a specialization of PJ. This means that the behaviour of a PJ-class is exactly those methods from the corresponding RTSJ-class that are necessary, and no more. Thus instead of using

```
RTSJ - class
{all the methods in RTSJ-class}
  |
  + -- PJ - class
```

```
       {with limitation of methods
        inherited from RTSJ, which
        are not part of PJ-class}
```

we have used:

```
PJ - class
{exactly those methods from RTSJ-
 class necessary to define PJ-class}
  |
  +--RTSJ - class
     {all the methods in RTSJ-class}
```

The resulting classes are shown in Figure 13. A few PJ specific classes are introduced: the `ManagedEventHandler` hierarchy, including the `Mission` subclass, and two subclasses to `AsyncEvent`.

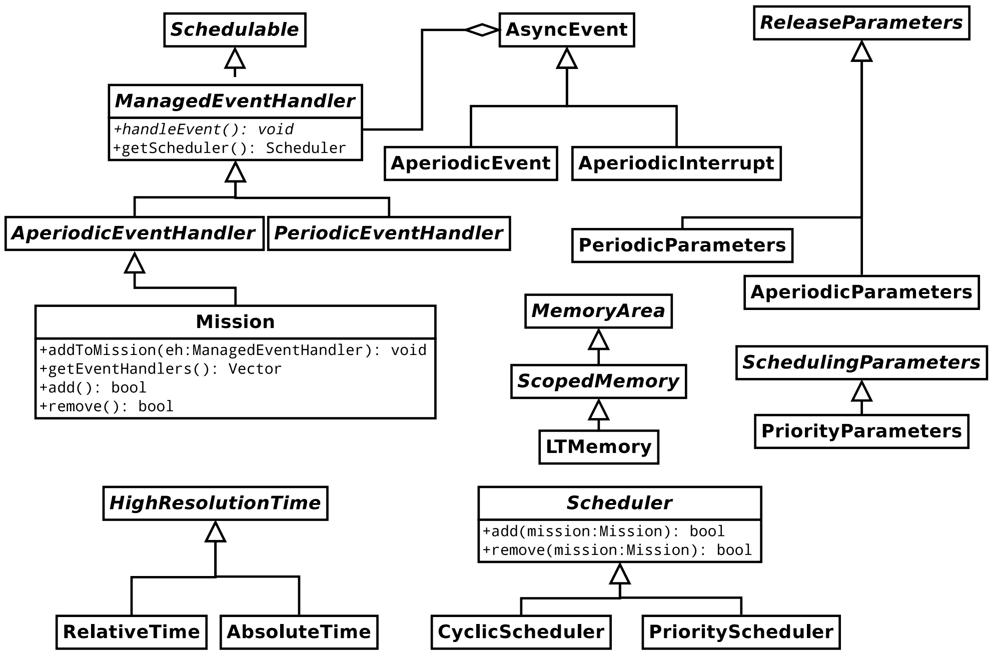


Figure 13: PJ class diagram

For the `Scheduler` we have been forced to start afresh, because the current RTSJ does not recognize missions. We could have removed the mission specific methods and seen them as subclasses of a very abstract scheduler to be fully consistent with our goal. The `Scheduler` in the PJ would then be a specialization with further concrete specializations `CyclicScheduler` and `PriorityScheduler`. They might even be placed outside the profile.

3.2 RTSJ compliance: delegation

The ideal PJ profile described above requires some work-arounds. Because RTSJ already exists, it cannot be defined as subclasses to PJ, but an application that uses the PJ package has to compile and, given that semantics is preserved, run under RTSJ with an adapter layer that essentially disallows some RTSJ methods and gives default values to some parameters. Hereby, the RTSJ compliance requirement is satisfied. An implementation of a concrete adapter layer is shown in the next sections.

4 Implementations

We present two open source implementations of the profile: one native implementation and one RTSJ compliant implementation. Source code for both implementations is available, as explained in Appendix 7.

4.1 Native implementation

In this part we outline how the profile is implemented on an ARM controller using a modified JamVM on top of Xenomai and Linux. JamVM is both extremely small (∼200K) and optimized, and conforms to the JVM specification version 2 [85]. Xenomai is a real-time extension to the Linux operating system. The Native Xenomai API has different services for real-time tasks and task scheduling, synchronization support including mutexes etc. [168].

4.1.1 Schedulers and handlers

In a basic implementation with a `CyclicScheduler`, we have only one periodic Xenomai `RT_TASK` that implements the well-known cyclic executive model [17]. When the single mission that is allowed for a cyclic scheduler is given the list of handlers, it creates a cyclic executive table. In this table the logic for the periodic handlers are set up, in line with the model, and the period of the periodic Xenomai task is calculated as the greatest common divisor of the periods of the periodic handlers (minor cycle).

Furthermore, the `CyclicScheduler` maintains a list of pointers to the private memories of handlers.

When a `PriorityScheduler` is used, for instance when both periodic and aperiodic event handlers are in play, each handler is bound to a Xenomai RT task, has a memory area allocated by the operating system, and a list of locks belonging to synchronized objects in the memory area. This information is gathered in a vector for the set of handlers in the mission, where each element is defined by:

```
typedef  struct handler_info {
  ..
  MEM_AREA *memArea;  // private mem
  RT_MUTEX *rt_locks; // Xenomai mutex
  RT_TASK   task;     // Xenomai task
  ..
} HANDLER_INFO;
```

A periodic handler uses Xenomai TASK services, for instance `rt_task_set_periodic` and `rt_task_wait_period` . An example is periodic invocation of a handler that is implemented as:

```
for (;;) {
  JNI-callback-to-PeriodicEventHandler-run
  rt_task_wait_period(NULL);
}
```

Similarly, Xenomai `EVENT` and `INTERRUPT` services are used to implement events or hardware generated interrupts.

The implementation does not support nested missions or, at the moment, mission termination.

4.1.2 Memory management

The garbage collector in JamVM is switched off so that the heap is used as immortal memory.

For the implementation of a `MemoryArea`, the standard `C malloc` is used to allocate a `MEM_AREA`, which is a new struct we have defined in JamVM. It is similar to the existing heap memory structure in JamVM.

Thus, the essential part of the Java implementation of memory area becomes:

```
public abstract class MemoryArea
{
  long memSize;
  int memID; // a reference to the MEM_AREA

  public void enter (Runnable logic) {
    Native.enterNativeMemArea (memID);
    logic.run();
    Native.leaveNativeMemArea (memID);
  }
  ...
}
```

When new objects are created by `logic.run`, they are placed in the memory area belonging to the handler. This memory area is reset by `Native.leaveNativeMemArea`.

4.1.3 Synchronization

Shared objects are represented by classes with synchronized methods. Synchronized blocks are not allowed which means that the Java bytecodes `monitorenter` and monitorexit which implement a Java synchronized block [98], are not rewritten in JamVM.

In the virtual machine synchronized methods are marked with the flag ACC_SYNCHRONIZED. This means that in JamVM we only need to replace the local objectLock/objectUnlock functions with two new functions called rt_object_lock and rt_object_unlock.

Those two new functions uses Xenomais MUTEX services with methods like rt_mutex_aquire and rt_mutex_release on elements in the list rt_locks.

Xenomai's MUTEX services enforce priority inheritance. Priority ceiling emulation is not enforced yet. It is a weakness with Xenomai.

4.2 The RTSJ adapter layer

This section describes the RTSJ implementation of the profile on top of the timesys 1.0.2 RTSJ implementation [159]. The technique used is encapsulation by delegating responsibility to appropriate RTSJ classes.

4.2.1 Schedulers and handlers

The cyclic scheduler schedules a single mission, containing periodic event handlers only, thus the add method expects a Mission object satisfying this constraint. From this mission, periodic handlers are retrieved and a cyclic executive table is built: an array where each entry contains a list of handlers for the given minor cycle. Execution is performed by a single periodic thread, with a period of *minor cycle*, implemented using the NoHeapRealtimeThread from RTSJ. Each period the handleEvent methods are executed on behalf of each handler in the current minor cycle, in the context of the private memory associated with each handler. The periodic thread will loop through the executive table forever. Thus the remove method has no effect and returns false.

The priority scheduler is implemented using the RTSJ priority scheduler. For each event handler an appropriate schedulable entity is created upon instantiation of the handler. It has a simple logic responsible for executing the event handler logic in the context of its private memory. For periodic handlers this is done using a periodic thread, NoHeapRealtimeThread, and for aperiodic handlers, the RTSJ AsyncEventHandler is used. When a mission is added to the priority scheduler, using the add method, priorities of the RTSJ schedulables are set using deadline monotonic priority assignment, and all periodic threads are started.

Periodic event handlers internally contains a simple implementation of an RTSJ NoHeapRealtimeThread as illustrated in the PeriodicEventHandler constructor:

```
protected PeriodicEventHandler(...) {
  super(priority, pp, scheduler, memoryArea);
  ...
  final PeriodicEventHandler pevh = this;
  rtsj_thread = new NoHeapRealtimeThread(
    null, rtsj_releaseParameters, null,
    ImmortalMemory.instance(), null,
    new Runnable()
      {
        PeriodicEventHandler _pevh = pevh;
        public void run() {
          for(;;){
            _pevh.memoryArea.enter(_pevh);
            NoHeapRealtimeThread.waitForNextPeriod();
          }
        }
      } // logic
  );
}
```

The run method of the `PeriodicEventHandler` class invokes the logic of the event handler, i.e. the `handleEvent` method. The `rtsj_releaseParameters`, supplied to the thread constructor contains the RTSJ equivalent of the PJ periodic parameters. The `NoHeapRealtimeThread` is executed in the context of immortal memory since it requires no additional memory during execution.

Similarly, the aperiodic event handler is implemented using the RTSJ `AsyncEventHandler`:

```
protected AperiodicEventHandler(...) {
  super(priority, ap, scheduler, memoryArea);
  ...
  final AperiodicEventHandler aevh = this;
  rtsj_asyncEventHandler = new AsyncEventHandler(
      null, null, null,
      ImmortalMemory.instance(), null,
      true, // nonheap
      new Runnable()
      {
        AperiodicEventHandler _aevh = aevh;
        public void run() {
          _aevh.memoryArea.enter(_aevh);
        }
      } // logic
    );
}
```

Here, the `rtsj_asyncEventHandler` is a package visible field of `AperiodicEventHandler`, and hence is accessible to the class `AperiodicEvent`. The `fire` method of `AperiodicEvent` then delegates the responsibility of fire to an instance of the RTSJ `AsyncEvent` class associated with `rtsj_-asyncEventHandler`.

The `remove` method has not yet been implemented. We consider using the mode change semantics of [143] and implementations as suggested in [123].

4.2.2 Memory management

The memory area classes are implemented by delegation using their corresponding RTSJ classes. Event handlers has their own scoped memories in which their logic can create temporary objects. Since all handlers must be referenceable by the RTSJ scheduler, the handler objects (Mission, PeriodicEventHandler, and AperiodicEventHandler) are allocated in immortal memory. This violates the hierarchical memory structure, where the handlers of a mission shares the mission memory. Instead immortal memory is used, a temporary solution which in the future could be reworked using techniques inspired by those presented in [7].

5 Profile compliance

The most important tasks in checking that a program is in compliance with the profile as described above are: ensuring schedulability, verifying that (temporary) memory consumption is within bounds, and checking that no non-temporary objects are allocated. For these purposes Worst Case Execution Time (WCET) and Worst Case Memory Consumption (WCMC) analyses are needed and may be combined with model checkers, such as Uppaal [25], to perform a full schedulability analysis.

Additionally, a compliance check may also need to enforce syntactic and structural requirements and constraints demanded by the profile, e.g., that all loops are explicitly bounded and that the program is not recursive. Standard analyses, such as control flow, data flow, and information flow analyses combined with simple syntactic checks are sufficient for this.

The design of our Java profile is intended to facilitate comprehensive tool support for most or all aspects of the profile. In particular, we believe that abstract interpretation, and similar static analysis techniques, combined with model checking can be leveraged to provide automated analysis and verification of important properties such as resource usage and profile compliance. Going beyond resource and compliance checking, static analysis techniques have also been applied with great success in the area of compile time verification of safety and security properties, e.g., prevention of race conditions and deadlocks, guaranteed secure information flow, and bug hunting.

Below we discuss in more detail the relevant analyses and how they may be applied here.

5.1 Resource usage

A prime example of an analysis in this category is the WCET analysis [167, 58, 154, 166] mentioned above. A WCET analysis can statically compute a sound over-approximation of the worst possible execution time behaviour of a program. This is needed, among other things, to determine if a given program can safely be scheduled. Dually, a *best case execution time* [167] computes a conservative under-approximation of the execution time of a given program. This may, in certain cases, be used to give a (conservative) lower bound on the execution time for aperiodic event generators and thus on the minimal interarrival time between aperiodic events.

In [33] a more direct approach to schedulability analysis is taken: here it is shown how to automatically extract a timed-automata based model of a Java bytecode program. The extracted model is then model-checked, using Uppaal, to determine directly if it is schedulable.

Other analyses in this category include analyses to determine worst case memory (WCMC) usage [69, 47] and maximum stack depth, both of which are instrumental for ensuring that a given program can be executed within the bounds set by the platform.

5.2 Safety and security

By computing standard control flow, data flow, and information flow analyses a number of safety and security properties can be guaranteed at compile time. Including the absence of certain bugs, e.g., null pointer exceptions [80], initialisation failures [81], secure information flow [74, 73, 67, 20], and avoiding race conditions [2, 61].

Combining a best/worst case execution time analysis with analyses that extract abstract timing models from a program, e.g., in the form of timed automata [33], it may be possible to give a compile time proof that a given program cannot possibly end in a deadlock state nor a livelock situation.

6 Conclusion

We have presented Predictable Java (PJ), a Java profile suitable for development of high integrity real-time systems. PJ shows that it is beneficial to define a specialized profile as a generalization of RTSJ. This is in contrast to other proposals, such as Ravenscar Java [91] and Safety-Critical Java [152, 12] which are (extended) subsets defined as limitations of RTSJ using subclassing. PJ uses only the handler paradigm, having periodic and aperiodic handlers. Those handlers are grouped in missions which are first-class objects as the `Mission` class is a subclass of the `AperiodicEventHandler` class.

Each handler has a private memory for allocation of local objects during execution of its `handleEvent` method. This private memory is reset every time this method completes. Because a mission is an event handler there is no semantic difference between mission memory and handler memory. This also means that the memory belonging to a mission is reset when a mission comes to end.

In order to ensure that it is a valid profile, it has two prototype implementations on different platforms; An RTSJ platform, showing that PJ is compliant with RTSJ, and a "native" platform.

7 Source code

The implementations mentioned in this paper are open source. The source code for both implementations, and further instructions, is available at `http://pj.boegholm.dk`. Please refer to the individual `README` files for further instructions. The RTSJ layer implementation includes two examples, `CyclicExample` and `PriorityExample`. These examples are using the two scheduling mechanisms together with a few handlers. The reference implementation of RTSJ used is the timesys 1.0.2, available at: `http://timesys.com/java`. A `Makefile` exists for running and compiling the examples, the latter using an `ant` build configuration. Instructions are found in the `README` file. The implementation using `JamVM`, `Xenomai`, and `Linux`, contains a modified version of the JamVM virtual machine, JNI functions, and the PJ implementation, with examples. Instructions on how to compile and run the examples are found in the `README` file.

Paper C:

Schedulability analysis for Java finalizers

Thomas Bøgholm[1], René R. Hansen[1], Anders P. Ravn[1],
Bent Thomsen[1], and Hans Søndergaard[2]

[1] *Department of Computer Science*
Aalborg University, Denmark

[2] *VIA University College*
Horsens, Denmark

Abstract

Java *finalizers* perform clean-up and finalisation of objects at garbage collection time. In real-time Java profiles the use of finalizers is either discouraged (RTSJ, Ravenscar Java) or even disallowed (JSR-302), mainly because of the unpredictability of finalizers and in particular their impact on the schedulability analysis. In this paper we show that a controlled scoped memory model results in a structured and predictable execution of finalizers, more reminiscent of C++ destructors than Java finalizers. Furthermore, we incorporate finalizers into a (conservative) schedulability analysis for Predictable Java programs. Finally, we extend the SARTS tool for automated schedulability analysis of Java bytecode programs to handle finalizers in a fully automated way.

1 Introduction

In Java, *finalizers* provide application programmers with an opportunity to perform clean-up and finalisation at garbage collection time. Specifically, just before the garbage collector releases the memory used for an object it calls the `finalize()` method on that object. The memory used by the object is then released in the *next* garbage collection pass. While finalizers superficially look like destructors, as known from C++, it is important to stress that finalizers are *not* destructors. In particular, since Java objects are not necessarily garbage collected at all, the finalizer may never be called for a given object. This unpredictability combined with potentially poor performance has made the use of finalizers in standard Java somewhat controversial and programmers are often advised not to use finalizers. However, as argued by Boehm in [28]: while finalizers are rarely needed in Java, there are a number of important cases where they are not merely convenient, but of critical importance. Examples include legacy libraries that wrap C code with use of `malloc` and `free`, interaction with hardware [139], as well as robust and reliable cleanup of temporary files and buffers. See [28] for detailed examples.

One argument against finalizers, often found in the Java literature and in blogs, is that their effect can be achieved through disciplined use of the try statement that include a finally block, as in

```
MyClass x = new MyClass();
try {
  ... // use x
} finally {
  x.destroy();
}
```

However, for objects where x.destroy() may only be called once, this requires extreme care be taken by the programmer in terms of where the reference to x goes, especially if x could become shared between tasks. The pattern also introduces a bit of boiler plate code that could get tangled up when several objects of this kind are needed. Last, but not least, the use of this pattern gives a rather unnatural programming style when a Java object structure is needed to mirror a legacy C++ object structure. In such cases it is much more natural to include the call to the C++ object's destructor in the Java objects finalizer. Furthermore, using a language construct, such as finalizers (with the semantics discussed in this paper) is much safer, as a language construct facilitates compiler and analysis tools support.

In real-time Java profiles the opposition to finalizers is even stronger: the use of finalizers is either actively discouraged, e.g., RTSJ and Ravenscar Java, or even disallowed, e.g., JSR-302. This is mainly because of the unpredictability of finalizers but also due to their potential impact on schedulability and, not least, schedulability analysis of the tasks comprising a real-time system. Yet, since real-time applications written in Java use managed memory allocation, it is very useful to have a destructor mechanism. Therefore, we investigate an automated schedulability analysis of programs that use finalizers in a setting with a disciplined use of scoped memory, for instance as we propose in Predictable Java [29].

The predictable execution of finalizers makes it possible to incorporate finalizers into a conservative schedulability analysis for Predictable Java programs (Section 3). Finally, we extend the SARTS tool for automated schedulability analysis of Java bytecode programs to handle finalizers in Predictable Java (bytecode) in a fully automated way (Section 4).

2 Predictable Java and Finalizers

The idea underlying the *Predictable Java* (PJ) profile [29] is to have a profile that facilitates the use of static analysis tools to provide safety, security, compliance, and performance guarantees. It is thus related to and inspired by RTSJ specializations such as the Ravenscar Java profile [91] and the Safety-Critical Java profile [76].

2.1 Key Concepts

The key concepts in the Predictable Java profile are: *handlers* , which are application tasks, and *missions* , supported by the resource concepts of memory and schedulers, shown in Figure 14. A mission basically contains a memory area along with a set of handlers. A mission is itself a (specialized) handler, which allows for nesting of missions. Each handler is either periodic or aperiodic, contains some logic to be run at each release, and has a private memory area used during execution. The Java concept of a thread is not part of the profile.

The interplay between resources, handlers and missions determine the predictability of applications. We focus here on memory resources in particular, because finalizers are linked to objects that occupy memory.

An application programmer defines *periodic* or *aperiodic* event handlers. A periodic event handler is a specialization of the `PeriodicEventHandler` class:

```
class Periodic
  extends PeriodicEventHandler {
  protected Periodic(
      PriorityParameters priority,
      PeriodicParameters pp,
      Scheduler          scheduler,
      MemoryArea         memory)
  {
    super(priority, pp, scheduler, memory);
  }

  public void handleEvent() {
    // the logic to be executed every period
  }
}
```

The parameter `pp` defines the temporal scope. The `priority` parameter is for use by a `scheduler` and the handler has a private `memory` to be used during execution of the algorithm. The exact values for these parameters are the concern of the system programmer that assembles the application; the programmer of individual tasks focuses on the algorithm that specializes the `handleEvent` method.

The underlying semantics imply the periodic execution of this algorithm within its deadline. The period and deadline of the handler are specified in the `pp` parameter. The semantics are conditional on the algorithm completing within its execution time budget, included in `pp`, without declaring more temporary objects than can be accommodated in the `memory`, and refraining from declaring non-temporary objects. In order to check these assumptions, loops must be statically bounded and recursion disallowed.

Aperiodic event handlers are defined in an analogous way and we omit the details for brevity.

Missions The functionality of an application is made up of handlers; the handlers must be executed according to a (feasible) schedule implemented by a scheduler. Handlers are collected into *missions* that are themselves handlers of termination and, in some cases, initialization events. Thus we collect

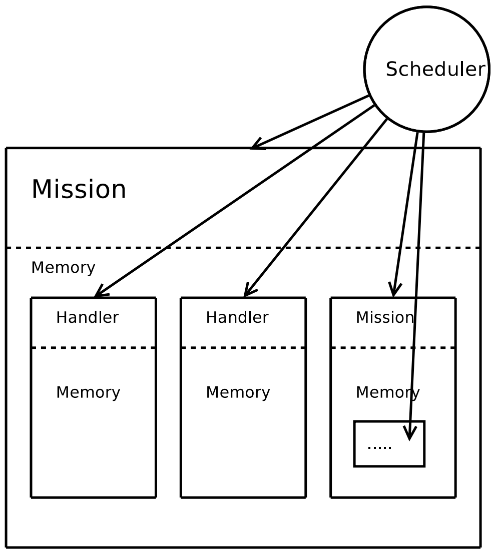


Figure 14: Predictable Java overview

the responsibilities of a mission in a handler class that contains handlers.

Handlers use private memories for temporary objects, but they may use more permanent shared objects protected through mutual exclusion mechanisms as specified by the Java `synchronized` method qualifier. These are placed in the memory of a mission that encapsulates a set of handlers. This does not introduce major complexities; the private handler memory is now the shared mission memory, and the handler concept allows functionality to be expressed, such as start, stop, etc.

The code below gives the termination handler aspect of the mission in the `handleEvent` logic. Initialization is done in the constructor of a mission, where the container aspect is the vector of `eventHandlers`. Individual handlers belonging to the mission are added by the `addToMission` method. Note that handlers can be added only, and only during initialization, thus a mission contains a static and finite set of handlers.

```
public class Mission
  extends AperiodicEventHandler {
  Vector<ManagedEventHandler> eventHandlers;
  protected Mission(
      PriorityParameters priority,
      AperiodicParameters ap,
      Scheduler scheduler,
      MemoryArea memory) {
    super(priority, ap, scheduler, memory);

    ... // set up handlers

  }
  public void addToMission (
      ManagedEventHandler eh
      )
  {
    eventHandlers.add(eh);
  }
  ...
  public void handleEvent() {
    ... // logic
  }
}
```

The omitted part above are methods that interact with the scheduler: `add()`, `remove()`, and `getEventHandlers()`. A mission is submitted to its scheduler by `add`. The scheduler knows that the calling handler is a `Mission` and contains a list of handlers. They can be accessed through `getEventHandlers`, and it is then up to the scheduler to schedule the set of handlers if feasible. The handlers of a mission are assumed to be scheduled using their release parameters from a common initial time point. Correspondingly, as a step in termination, the set may be removed from the scheduler through `remove`. The method `getScheduler` is defined in the superclass, `ManagedEventHandler`. We shall not enter into a further discussion of schedulers.

Memory When the runtime system starts an application, the outermost mission is initialized with its mission memory. During initialization objects are allocated in the mission memory, and thus they can be released again, including finalization, if and when the mission terminates.

Handlers that are allocated in a particular mission get a private memory that is allocated as an object in the mission memory. Objects declared during initialization of a handler are placed in its private memory. More interesting in this context are the objects allocated during a `handleEvent` invocation. These are placed in the same private memory, but are released and finalized when the invocation completes. The important point is that objects are released and finalized from the handler memories as a sequel to `handleEvent` invocations; the basis for our treatment of finalizers in the following.

2.2 Implementation

A restricted version of the Predictable Java profile has been implemented on an ARM controller using a modified JamVM on top of Xenomai and Linux [29]. JamVM is both extremely small (∼200K) and optimized, and conforms to the JVM specification version 2 [85]. Xenomai is a real-time extension to the Linux operating system. The Native Xenomai API has different services for real-time tasks and task scheduling, synchronization support including mutexes, etc. [168].

The garbage collector in JamVM is switched off and the heap is used as immortal memory. When new objects are created, they are placed in the memory area belonging to the creating handler and added to a list. Just before the memory area is reset, leaving `handleEvent`, the list of objects is visited, and each of their finalizers are executed.

3 Schedulability Analysis with Finalizers

The traditional response time analysis for tasks scheduled using a Fixed Priority Preemptive Scheduler may easily be extended to take finalizers into account. In the equation

$$R_i = C_i + \sum_{j \in hp(i)} \lceil R_j / T_j \rceil C_j \tag{1}$$

all that is needed is to include in the execution time C_i for task i the WCET for each finalizer for objects allocated by that task in its private memory.

$$C_i = WCET_i + \sum_{k \in WCObj(i)} WCETfinalizer_k \tag{2}$$

Here $WCET_i$ is the ordinary worst case execution time (WCET), $WCObj(i)$ is the set of objects allocated in worst case by task i in its private memory, and $WCETfinalizer_k$ is the WCET of the finalizer for object k.

Basically, this formula states that a system is fixed priority pre-emptive schedulable if there is enough time left before the deadline to execute finalizers for all objects created by the given task during an invocation of its logic. This simple extension is made possible by linking the scoped memory tightly to an invocation of the logic of a specific task. In a more liberal RTSJ-setting, where scoped memories can be shared by several tasks across a hierarchy, it is not possible to assign finalization costs to a specific task. One could consider a solution, where all involved tasks should be prepared to cover the cost. However, this would give a very pessimistic, although real, schedulability analysis, which in most cases would make systems unschedulable.

4 Automated Schedulability Analysis with SARTS

SARTS [33] is a tool for schedulability analysis for real-time Java programs. In this section we show how the schedulability analysis performed by SARTS can be extended to include finalizers in the PJ profile: A short introduction to SARTS is given, essentially a summary of [33], followed by the finalizer extension.

SARTS works by analysing bytecode of compiled Java programs, and assume predictable execution time for byte-code instructions, as is e.g. the case for code executed by the predictable Java processor JOP[130]. From the Java bytecode and meta-data such as loop-bounds information, a network of timed automata corresponding to the components of the Java program, is generated; each method in the Java program corresponds to one parametrised template for a timed automaton. This results in a timed automata model of the full system, which can then be verified using the model checker Uppaal [25].

Three additional predefined components are added to the final model.

- An automaton corresponding to the scheduler strategy for the underlying platform.
- Two types of templates representing periodic and aperiodic tasks, essentially transferring control to the correct logic.

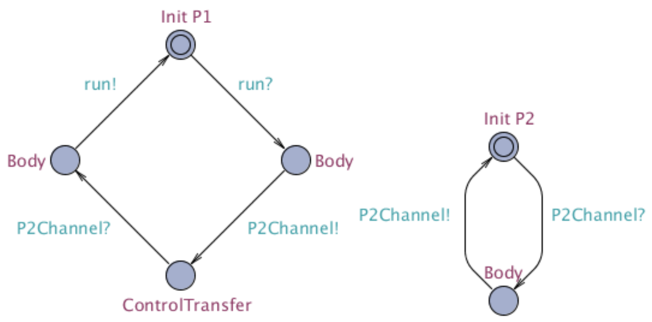
The small controller automata for each task glue the application specific logics in the task to the scheduling machinery. Task identifiers are used to synchronize with individual automata embodying the logics.

4.1 Model generation

From the bytecode, a control-flow graph is generated in the form of a timed automaton, where each bytecode represents a pattern, reflecting the timing semantics.

Each method and task in the analysed system results in a timed automaton, and through synchronization, control is transferred between automata, used to model method invocation. Waiting for syncronization on channel `chan` is written as `chan?` and initiating synchronization on channel `chan` is written as `chan!`. A very general pattern for generated automata is shown in Figure 15. In this figure, Figure 15(a) exemplifies a `run` method, and Figure 15(b) an arbitrary method:

- Initially, 15(a) waits for synchronization on channel label `run?`; a signal on channel `run` will move P1 to the location `Body`. `Body` represents a sequence of bytecode patterns.
- In body, the individual patterns forces time to elapse, simulating execution time.
- From body, a synchronous step in both automata transfers P1 to location `ControlTransfer`, and P2 to `Body`, simulating control-transfer, since P1 is waiting for synchronization.
- Similarly, time elapses while P2 is in location `Body`, before synchronizing with P1.
- P1 is now in Body, time elapses, before P1 synchronizes on channel label `run!`



(a) Automaton P1 (b) Automaton P2

Figure 15: General method pattern

This is the general principle in the models generated by SARTS. In the task automaton initializing the `run` synchronization, a *guard* will cause the entire model to *deadlock* if too much time elapses before re-synchronization. This means that if no deadlock state exists in the generated model, no deadline miss exists in the original program [59, 33].

In Figure 15, the `Body` location represents sequences of smaller patterns; a pattern for each bytecode instruction. Two example patterns are illustrated in Figure 16.

The simple basic block represents an instruction with one entry point and one exit point, illustrated in Figure 16(a). The circle represents a location, modelling one or in some cases a sequence, of bytecode instructions, labelled *Simple*, in which time may pass. Associated to the location *Simple*, is a two-line expression, placed to the right of the location: The first line represents a simple invariant stating that the clock, `time`, must be less than or equal to the required execution time for this bytecode instruction; represented by the constant `inst35`. The second line is a *stopwatch expression* stating that time may pass in this automaton iff the value of `running[tID]`, where `tID` is the *task identifier*, is 1; a technique for modelling pre-emption. The outgoing transition is associated with one guard expression, `time == inst35`, which must be true for the transition to be enabled, i.e. the time elapsed in the location `Simple` must be exactly the execution time of the corresponding instruction, represented by `inst35`. Additionally, taking the outgoing transition will set the clock-variable `time` to zero. Note, that in general, any outgoing transition sets the clock-variable to zero.

Figure 16(b) illustrates the pattern for a branching instruction, an instruction with multiple exit points. Following the principle from Figure 16(a), allowing for preemption and forcing time to elapse in the location. This pattern has two exit transitions, with the same guard and update, chosen non-deterministically.

Figure 17 illustrates the pattern for a method invoke instruction. The channel synchronization, in this case is symbolised by `MethodName1` and `MethodNameN` illustrating the case where more than one method may be considered. In the case of *pure virtual method invoke*, all possible method candidates will be considered by branching and non-deterministic synchronization, transferring control as illustrated in Figure 15.

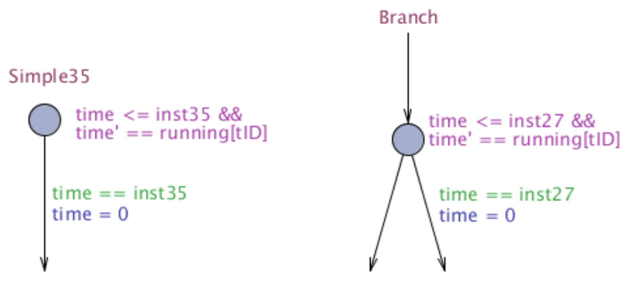
On return, the invoked method automaton will have a transition with the same synchronizing label, waiting for synchronization. This means, that the automaton initializing the synchronization will wait until the receiver re-synchronizes.

Memory is considered since the size of the method may cause difference in execution time. This is modelled by the `methodSwitchCost = CX` on the transition between `running_MethodCallingX` and `returningFrom_MethodCallingX`. This will cause the model to delay for the correct, method dependent, amount of time.

More advanced, patterns exists for other bytecode instructions in the case of loops, synchronization, etc. though these will not be discussed further.

4.2 Finalizers

SARTS has been extended in order to handle finalizers in the schedualbility analysis. The extension of SARTS to perform schedulability analysis of PJ programs implementing finalizers requires: Computing the set of classes



(a) Basic (b) Branching

Figure 16: Simple bytecode automata-patterns

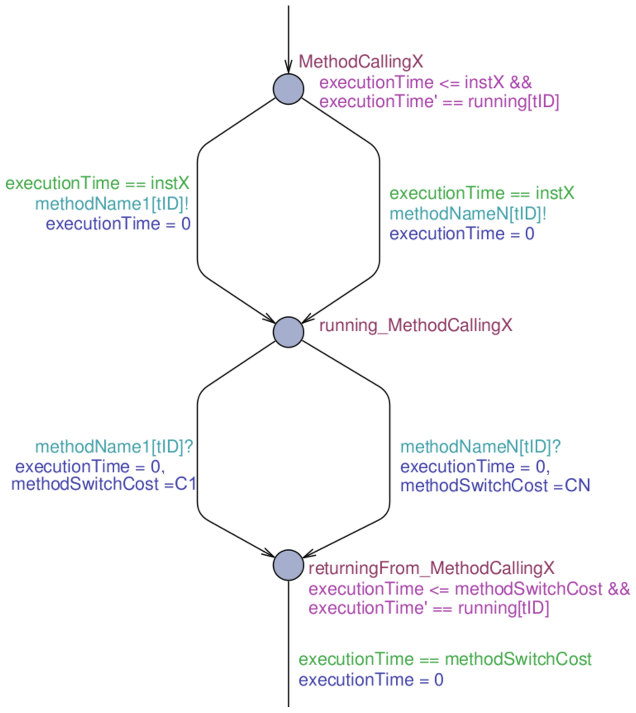


Figure 17: Method invoke

possibly instantiated during execution and identifying finalizers, additional handling of the *new* bytecode instruction, and modelling the execution of finalizers associated with objects created in the scope of an event handler.

Computing the set of classes possibly instantiated during execution is done by analysing the `new` instructions in the program bytecode, counting each class instantiation; the concrete type is statically known. Using this information, a table for registering object creations is added to the generated model. This table maps, for each state, task identifier and class to the number of instances created, $TASKID \times CLASSID \rightarrow N$, and thus the number of finalizers to be considered for each class in the end of an event handler. In UPPAAL this is expressed as an integer array: `int finalizeCount[ThreadID][ClassID];`, where `ThreadID` and `ClassID` are integer types ranging from zero to the number of tasks and classes respectively.

The block pattern introduced for `new` instructions is illustrated in Figure 18. The difference from the *basic block* pattern in Figure 16(a) is an additional update, incrementing the number of finalizers for this particular object, the statement: `finalizeCount[tID][0]++`. Here, `finalizeCount` is the table containing the number of finalizers to be executed for each $(TASKID, CLASSID)$ pair, the `tID` constant refers to the task executing the `new` bytecode instruction, and `[0]` is the index of the concrete class being instantiated; a statically known value.

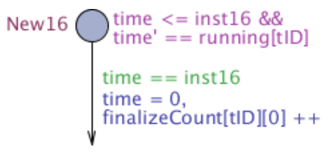


Figure 18: New instruction

This counts, on all from new outgoing transitions, the instance count of each class, for each task in the system. In the end of a `run` method, a small pattern is added such that required finalizers will be considered before the task is completed.

As an example we show a simple example of a periodic event handler, followed by a very simple class containing a finalizer:

```
class Periodic
  extends PeriodicEventHandler{
  protected Periodic(
      PriorityParameters priority,
      PeriodicParameters pp,
      Scheduler scheduler,
      MemoryArea memory)
  {
    super(priority, pp, scheduler, memory);
  }

  public void handleEvent() {
    //logic to be executed every period
    new Obj();
    return;
  }
}

class Obj{
  public void finalize(){
    // clean-up code
  }
}
```

The `handleEvent` method of the *event handler* creates an object of type `Obj` and then returns. `Obj` contains a `finalize` method, and need to be finalized at the end of the period. The generated automaton for the `handleEvent` method is illustrated in Figure 19.

- Waiting in the *initial state*, following the example automaton from Figure 15(a), synchronization is performed on the `run?` transition; the task template performs this synchronisation.
- A pattern for the `new` instruction is inserted, similar to that of Fig-

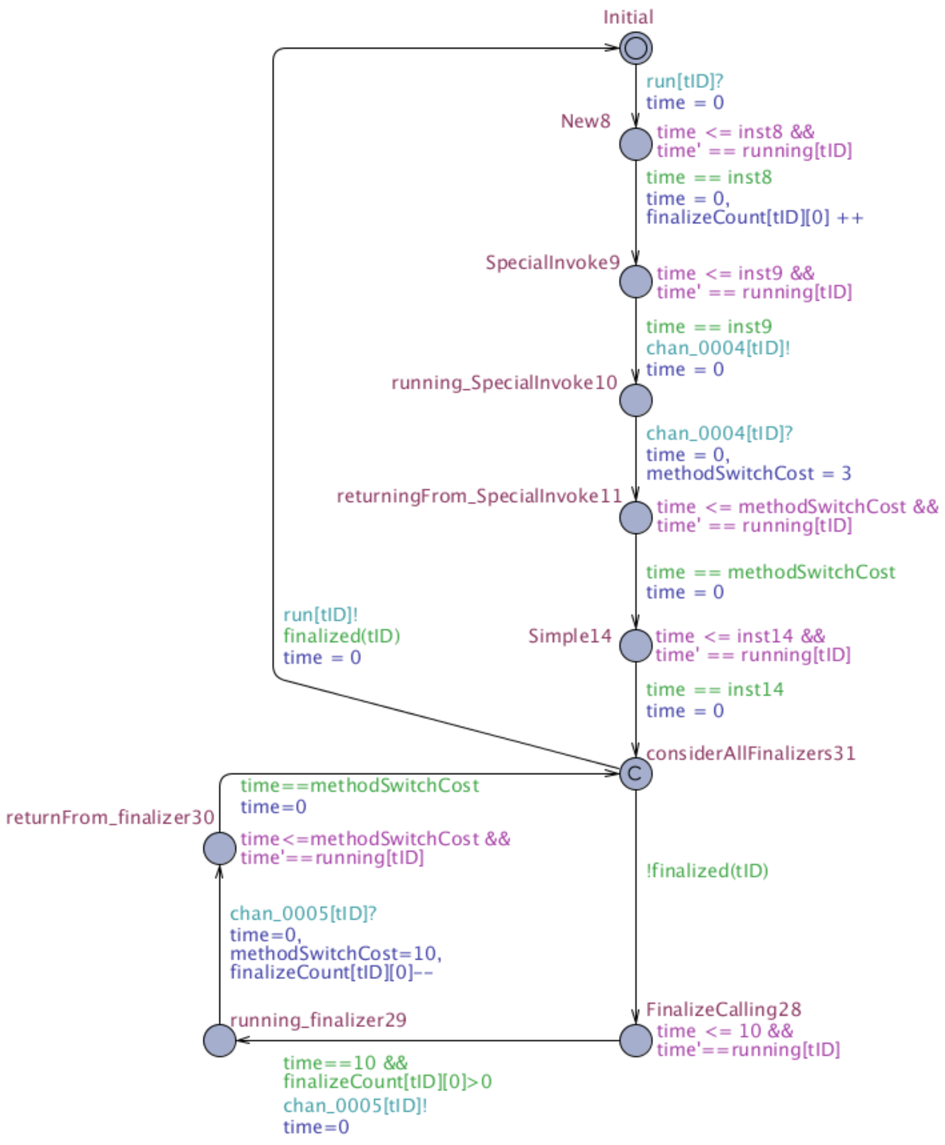


Figure 19: handleEvent automaton

ure 18. This would be followed by a *basic block pattern*, Figure 16(a), in case of reference assignment to a variable, omitted in the example for sake of clarity.

- Then control is transferred to another automaton, modelling the constructor method, using the pattern for *method invoke*, similar to that of Figure 17.
- The locations `considerAllFinalizers`, `returnFrom_finalizer`, `FinalizeCalling`, and `running_finalizer` model the finalization of objects instantiated during execution of the entire event handler. Here, the `finalized(TaskID)` function is an auxiliary function, returning *true* only when all finalizers have been considered.
  - If finalizers exists, the automaton transitions to `FinalizeCalling`, where time elapses.
  - Non-deterministically, a finalizer which has a count larger than zero is chosen, and control is transferred as in the *method invoke* pattern.
  - When control is returned, invoke-count for the invoked finalizer is decremented.
  - Method invoke is performed as normally, and afterwards the location `considerAllFinalizers` is re-entered.
  - These four steps are cycled until no more finalize invokes are pending.
- The last transition returns control to the task control automaton.

The resulting analysis is fully automatic and in terms of accuracy is similar what is presented in [33], where it is claimed that tighter results are achieved, by being able to consider e.g. branching and blocking.

5 Conclusion

In real-time applications that are pure Java, we would not expect to see finalizers used in objects allocated during execution of the handler logic of ordinary periodic and aperiodic tasks. However, when such a task is a stub for a call to a legacy C or C++ handler, a finalizer may be useful, because it is needed for freeing allocations done by the legacy code, and clean-up of temporary files and buffers. For missions finalizers may be directly useful, for instance to empty buffer objects or ensure that logs are complete before the mission terminates. The use of finalizers is acceptable in Predictable Java, for real-time Java programs, since finalizers are more like destructors, as known from C++. We have shown how classic response time analysis can be extended with finalizers, in a conservative manner, and we have presented the tool SARTS, a tool for fully automatic schedulability analysis, and shown how SARTS is extended to handle finalizers in Predictable Java.

The results in this paper apply to programs written in the Predictable Java Profile as the profile ensures unique “ownership” and lifetime to objects through the handlers which created them in their private scoped memory. In RTSJ this is not possible in general as code may enter and leave scoped memories at arbitrary points, thus no single handler assumes ownership. However, one could employ a PJ programming style when using RTSJ and thus the results in this paper would be generally useful. In some circumstances, it would be possible to automatically check that the PJ style of programming has been applied. One hypothesis is that our PJ compliance checker could be extended to become a PJ programming style checker that could work on RTSJ — an idea we will pursue in the near future. We also think that our results are applicable to the upcoming SCJ standard, as SCJ features a private memory area for each handler.

Acknowledgement

The authors would like to thank the reviewers for their comments that help improve the manuscript.

Paper E:

Schedulability Analysis Abstractions for Safety Critical Java

Thomas Bøgholm[1], Bent Thomsen[1], Alan Mycroft[2], and Kim G. Larsen[1]

[1]*Department of Computer Science*
Aalborg University, Denmark

[2]*Computer Laboratory, Cambridge University*
Cambridge, United Kingdom

Abstract

We present a compositional approach to schedulability analysis of safety-critical Java programs. We introduce a specification language in order to write abstract behavioural specifications regarding task execution-time and use of resources. Schedulabilty is checked on a model composed of the abstract specifications, possibly before any implementation, and as the specifications are implemented, these implementations can be checked individually. This means that library routines potentially can be separately checked and reused, and individual tasks can be verified according to their specifications without performing the full-system-analysis.

1 Introduction

In recent years Java has been put forward as a programming language for development of embedded real-time systems. Java in its original form has several characteristics which make development of real-time systems very difficult and Java also lack the notion of a deadline and high-resolution real-time clocks. Several profiles such as *The Real-time Specification for Java* [36], *Ravenscar-Java* [91], *Predictable Java* [29], and *Safety Critical Java* [76], all put forward restrictions and extensions to Java that combined make them suitable for development of real-time systems. Especially the latter three are also intended to make programs easier to analyse and thus make the process of certifying systems easier. An important part of any real-time system development is WCET and schedulability analysis to ensure that a system upholds its deadlines and in fact can run on a given hardware platform. WCET and schedulability analysis can be a time consuming part of the development process, however, in the past few years model based WCET [50, 109, 135, 65] and schedulability [11, 33] analysis has established itself as a viable approach that can automate the process.

However, model based WCET and schedulability analysis is still only capable of handling relatively small systems, typically consisting of a few handlers with relatively simple logic, due to a phenomenon called state-space explosion. The more complex a system is the more states are generated by the model, and although model checking has undergone a tremendous development with lots of improvements, there are still limits to the number of states that can be handled. Furthermore, current model based approaches all make a full program analysis, i.e. include all code paths, even library code in the model. Thus model based analysis currently is not compositional.

In this paper we investigate compositional schedulability analysis. The basic idea is to incorporate abstract descriptions of methods underlying temporal and locking behaviour in method interfaces, allowing (library) routines to be checked separately and independently of the systems they will be used in. The interface behavioural descriptions are usually simple and thus generates models that have far fewer states than the models generated from their byte-code. The system can also be used to reverse engineer a specification from legacy code, i.e. a specification, albeit a very concrete one, can be generated from the Java byte-code and may then later be used as basis for a more abstract description suitable as a component in larger system being specified and analysed.

2 Related work

Traditional schedulability analysis works on a relatively simple model of the system which places many restrictions on the system being analysed e.g.

tasks in the system cannot be dependent on other tasks. These analyses are based on the worst case scenario, and have a somewhat narrow view of the system, based on what is known as the *critical instant*. This makes these methods very pessimistic due to a limited amount of detail considered in the analysis. The standard approach to schedulability analysis is *response time analysis* [14], which is later used for comparison of the proposed approach.

There has been several proposal for hierarchical and compositional schedulability analysis of real-time systems, a good overview can be found in [57]. Some studies have developed interface theory for incremental analysis of component based real-time systems [77]. Both directions build upon and extending traditional schedulability analysis.

Our work builds upon and extends SARTS [33], a system for automatic schedulability analysis of Java programs written in the SCJ profile [76] for the *Java Optimized Processor* (JOP) [107]. Thus the component model we use is the one defined by the class mechanism of the Java language, which is a traditional object oriented component model.

SARTS uses the Uppaal real-time model checker [26] for schedulability analysis based on ideas from TIMES [11] where a model of the scheduler is run alongside models of the real-time tasks of a system under analysis.

We are inspired by the ECDAR system [51], a tool implementing the timed interface theory [52]. The ECDAR tool is designed to check incrementally refinement between specifications using Uppaal Tiga [23], as we do.

Our approach is also inspired by early work on annotating types in functional concurrent programs with behaviors [113, 156, 114], an approach inspired by type and effect systems [150] , which recently has been put into object oriented context [27] and also inspired work session types [66, 160].

We present a simple specification language, *Time and Resource Specification Language* (TRSL). The main purpose of TRSL is to be expressive enough for our purpose, but not too expressive to exclude automatic analysis. A further goal with TRSL is to have a syntactic representation, which can be included as an annotation or comment in the source of the Java program for the system being specified and analysed. One could clearly have disposed of TRSL and just used a timed automata directly. However, these are difficult to include directly in the source and a general timed automata may be too rich and thus hamper or exclude automatic analysis. Alternatively one could used specifications coming from the *Object Management Group* (OMG) *Unified Modeling Language* (UML) *Modeling and Analysis of Real-Time and Embedded Systems* (MARTE)[3] specifications or specifications written in the ACSR timed process algebra [41] or extracted from TADL Architecture Description Language [110] specifications. In all cases the specification formalisms would be too rich and analysis of the speci-

[3]http://www.omgmarte.org/

fication to ensure that only analysable systems were described, would be necessary.

3 Compositional analysis

In this analysis, we introduce a task-model with *resources* rather than *worst case blocking time*. A correspondence-check between implementations and their specifications is also presented. The idea is to annotate tasks in the system with specifications, and then perform schedulability analysis by combining these. If the system is schedulable, implementations of each task are created, maybe using libraries already having specifications.

This divides the analysis into three major parts:

- A specification language in which we can express execution time and locks,
- Correspondence analysis between program specification and implementation, and
- Schedulability analysis on abstractions written in this language, of tasks in the system.

Specifications are concerned with execution time, and critical sections guarded by locks, and are expressed in TRSL. In this language we are able to express the timed behaviour of a piece of code with different levels of precision in the form of intervals, critical sections, branching, and repetition of sequences. The grammar of this language is presented in Section 3.1.

The schedulability analysis works on the task behavioural specifications, translated into timed automata, based on a modified approach of the method used in SARTS [33]. Each task specification is separately translated into a timed automaton and composed into a network of timed automata, together with a model of a strategy for scheduling, e.g. FPS. Using UPPAAL the model can be queried about schedulability, worst case response time and utilization. If a system is deemed not schedulable, an error trace is generated from UPPAAL. This error trace can be used to construct *time line diagrams* similar to Figure 21; this should aid in identifying and solving the problem.

3.1 Specification language

In this section we introduce the language for specifying task behaviours. Behaviours are expressed as *abstract traces*, in which we are concerned only with task execution time, in the form of intervals, branches, the acquirement and release of locks, etc. This differs from traditional approaches described earlier, where tasks are specified by worst case execution time, and worst case blocking time determined by the locking protocol used. This way of describing tasks allows a more detailed description of the actual task behaviour, with regards to execution time and resource usage, but this also requires a more complex schedulability analysis. It also allows many levels of abstractions, from exact execution times, requiring more verification time, and interval execution times, which may result in faster verification, at the cost of precision. We later use a time line notation inspired by [45] to graphically present task behaviors, depicted in Figure 20.

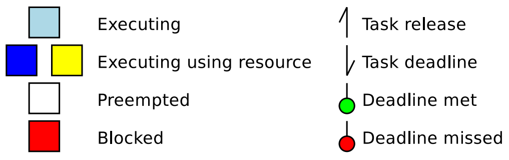


Figure 20: Time line notation

The specification language is given by the following grammar:

$$\begin{array}{lll} Trace & ::= & [Block*] \\ Block & ::= & skip \quad | \\ & & t_s..t_e \quad | \\ & & Trace \quad | \\ & & using(r_1, ..., r_n)Block \quad | \\ & & repeat(n_{min}..n_{max})Block \quad | \\ & & select\{Block_1, ..., Block_n\} \end{array}$$

A *Trace* is a sequence of blocks, and a block can be:

- a skip block, the empty instruction taking no time,
- an interval, $t_s..t_e$, describing the interval between t_s and t_e, we use t as shorthand notation to denote constant times, i.e. the interval where $t = t_s = t_e$,
- a *trace*, the sequence of blocks $[Block_1; ...; Block_n]$,
- a *usage block*, $using(r_1, ..., r_n)Block$, describing a critical section, $Block$, protected by the locks $r_1, ..., r_n$,
- a *repeat block*, $repeat(n_{min}..n_{max})Block$, describing the repetition of a block minimum n_{min} and maximum n_{max} times, or
- a select block, which describes the non-deterministic choice between a set of traces.

We use $Block?$ as shorthand to describe an *optional block*. This is the same as $select\{Block, skip\}$.

3.2 Example

Consider a simple system of:

- two tasks, $Task_1$ and $Task_2$,
- task priorities inversely proportional to task deadline, i.e. lower deadline equals higher priority (deadline monotonic priority assignment),
- locking scheme following the original priority ceiling protocol.[4]

Consider two tasks, $Task_1$ with highest priority, with the real-time requirements:

- $Task_1$: Period: 9, Deadline: 4,
- $Task_2$: Period: 18, Deadline: 17,

and with the following implementation presented in Java code.

```
class Task1 extends PeriodicEventHandler{
  Buffer buf; // shared buffer
  //@ TRSL = [1]
  private int calculate(){..}

  //@ TRSL = [1  ; using(r)[2]  ]
  public void handleEvent(){
    value = calculate(); // wcet: 1
    buf.write(value);    // wcet: 2
} }

class Task2 extends PeriodicEventHandler{
  Buffer buf;             // shared buffer
  //@ TRSL = [5]
  private int calculate(){..}
  //@ TRSL = [2]
  private void prepare(..){..}
   //@ TRSL = [1]
  private void register(..){..}
  //@ TRSL = [1 ; 7? ; using(r)[2] ; 1 ]
  public void handleEvent(){
    if(!ready){            // wcet: 1
      value = calculate(); // wcet: 5
      prepare(value);      // wcet: 2
    }
    input = buf.remove();  // wcet: 2
    register(input);       // wcet: 1
} }
```

[4]though the *immediate priority ceiling protocol* is used in real systems, we show how to analyse the original protocol since this gives slightly more interesting behaviour

WCET analysis for these tasks results in 3 and 11 for the two tasks respectively, both with critical sections of 2 time units. Using the response time analysis described in [45], we get a worst case blocking time of 1 time units for $Task_1$ and zero for $Task_2$, and the calculated response times are 5 for $Task_1$ and 17 for $Task_2$, so using the response time analysis, the system is *not schedulable*.

This is illustrated graphically in Figure 21. The figure depicts time line

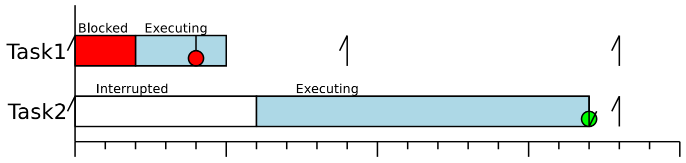


Figure 21: Time line: Response time interpretation of $Task_1$ and $Task_2$

from the *critical instant*, from a classical response time analysis perspective. $Task_1$ is blocked for 2 units and then needs to execute for 3 units, missing its deadline. $Task_2$ is pre-empted for 6 time units, and then needs 11 time units of execution time, before it meets its deadline. In this case, the approximation of blocking time causes the response time analysis to falsely reject the system.

The TRSL specification for $Task_1$ and $Task_2$ expressed using a resource r could look like this:

Specification of $task_1$

```
[ 1 ; using(r)[2] ]
```

Specification of $task_2$

```
[ 1 ; 7? ; using(r)[2] ; 1 ]
```

The specification of the tasks allow the following behaviour, in Figure 22.

T_1 : execute for one time unit, then execute for two time units using resource r,

T_2 : execute for one time unit, then maybe execute for 7 time units, execute for two time units using resource r and execute for one time unit. Task2a shows the case where we do not execute for optional 7 time units, and Task2b the alternative.

This is exactly the behaviour of the implementations of $Task_1$ and $Task_2$[5].

[5]Alternatively, one could use the less precise, but more general specification of $Task_2$: $T_2 = [1..8; using(r)[2]; 1]$ also capturing the implementation.

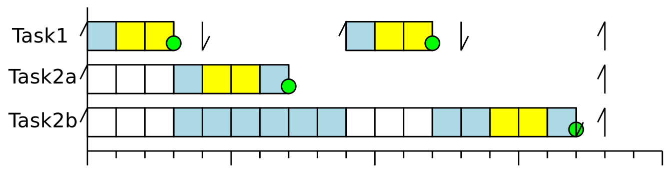


Figure 22: Time line: actual behaviour of $Task_1$ and $Task_2$

These specifications are then translated into *Timed Automata*. The purpose of this translation is to verify schedulability of the abstract system, allowing schedulability analysis early in the development cycle. This translation and the schedulability analysis is described in Section 4.3. A prototype compiler from TRSL-specifications to UPPAAL models for verifying schedulability is available on *sarts.dk*.

Another translation from TRSL to *Timed Automata* is made in order to verify a simulation relation between specifications and implementations. Since this language is able to express the exact timing behaviour of a task, with regards to execution time and synchronization, we are able to provide a translation directly from Java byte-code to TRSL. Note that in the case of variable time byte-code execution times, we can just use intervals or a select over possible traces. This allows us, for two TRSL-texts, $implementation$ and $specification$, to verify the language inclusion property.

4 Verification

The verification is performed in two steps. After writing abstract task specifications, the schedulability analysis can be performed, ensuring that the system will be schedulable if all task implementations follow their respective specification. The relation between implementation and specification is verified separately for each task, by verifying the language inclusion property $L(implementation) \subseteq L(specification)$.

4.1 Implementation verification

The translation is split into a number of patterns, one for each syntactic element in the TRSL language. This section describes the translation from TRSL to timed automata and the verification of language inclusion.

Interval pattern The interval pattern, $t_s..t_e$, consists of only one location, and is depicted in Figure 23. Location l_0 allows time to pass for the selected interval. The *invariant* $c \leq t_e$ on l_0 ensures that we leave this location at the end of the interval. The guard $c \geq t_s$ ensures that we stay in the location for minimum t_s time.

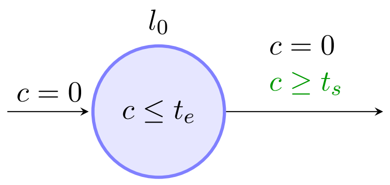


Figure 23: Interval pattern

Trace pattern The *trace pattern* is a sequence of patterns, i.e. blocks, trivially linked together by clock-resetting transitions.

Critical section pattern The *critical section pattern*, $using(r_1, ..., r_n)Block$ is a pattern for modelling critical sections, using a *resource*. This is modeled by surrounding a block-pattern with a pattern simulating the locking operation. In this case, we do not need to specially consider the locking protocol, since all we do is to verify that a task, in isolation, has the behaviour allowed by its specification. This pattern is shown in Figure 24. In this figure, the mutex actions are *lock* and *unlock*.

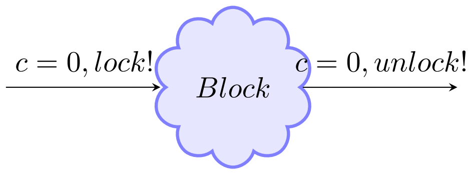


Figure 24: Critical section pattern

Repeat pattern The pattern for repeat, $repeat(n_{min}..n_{max})Block$, is shown in Figure 25. The entrance location is l_0, where time cannot pass, i.e. an *urgent location*, simulating the looping-condition. From this location, two outgoing locations goes to either the body-block of the repeat-pattern, illustrated by the cloud in the figure, or to the next pattern, illustrated by the arrow without destination location. These transitions are guarded by expressions on a private counter-variable, used to make sure that the body-block is iterated for the correct, number of iterations, as stated in the TRSL-text. The transition to the body-pattern makes sure that the body has been reached strictly less than the maximum number of times, n_{max}, by the guard $counter < n_{max}$, and has an update statement incrementing the counter, as well as resetting the clock, c. The transition leaving the repeat pattern is guarded by the lower bound on the number of iterations, $counter \geq n_{min}$, ensuring at least the required number of iterations of the body are taken before exiting the repeat-pattern.

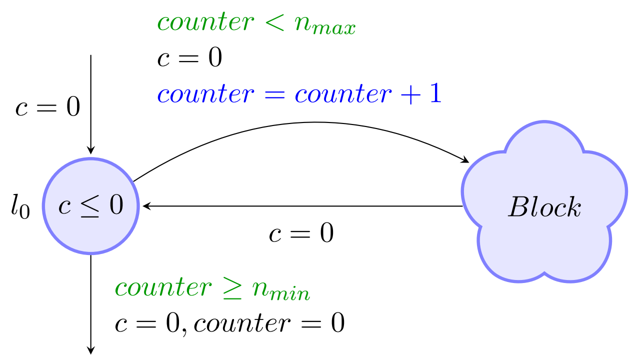


Figure 25: Repeat pattern

Select pattern The pattern for *select*, $select\{Block_1, ..., Block_n\}$, is shown in Figure 26. This very simple pattern will non-deterministically select one of the sub-patterns. The initial location is an urgent location from where unguarded transitions goes to each sub-pattern, and from the sub-pattern to the end location. These urgent locations simplify the model as guards, updates, etc. are placed on only one transition going to the initial location.

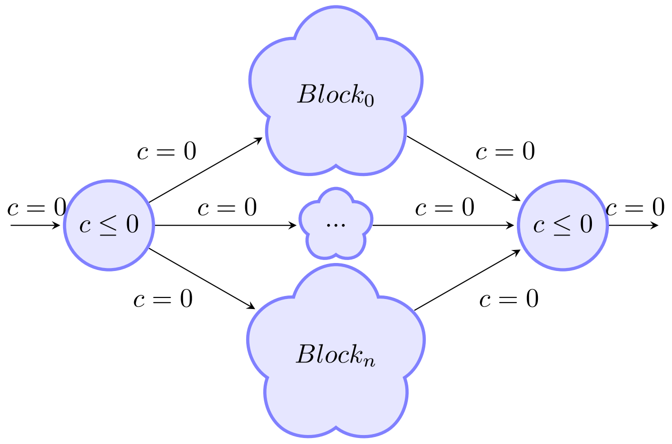


Figure 26: Select pattern

4.2 Verifying implementations

To verify that the implementations satisfy their specifications, their corresponding timed automata are constructed and the *language inclusion property* is verified, i.e. constructing the complement automata and verifying that the intersection is empty:

$$L(TA_{impl}) \bigcap \overline{L(TA_{spec})} = \emptyset$$

This is possible because the generated automata are actually a simple instance of the *Event-Clock Automata*, which are determinisable, and thus closed under *all boolean operations*, where timed automata in general are not closed under *complement* [10]. We use the subclass, event-recording automata. The definition of event-recording automata is similar to that of timed automata, with the restriction that:

- for every symbol $a \in \Sigma$ there is a clock x_a; there are no other clocks in the model,
- when a symbol, $a \in \Sigma$ is encountered, clock x_a is reset.

Since in this translation from specifications and implementations the resulting automaton consist of only one clock, c, which is reset on *every* transition, we immediately have τ on all transitions and x_τ as the only clock, hence, the generated automaton is an *event-recording automaton.* This allows us to verify the language inclusion property.

4.3 Schedulability analysis

As in SARTS [33], schedulability analysis is reduced to checking the model for deadlock freedom, following the work by Fersman and Yi [59, 60, 90], but by using abstract specifications, schedulability model containing greater detail on the use of resources, and less on the individual instructions.

Specifications are translated into timed automata, parallelly composed with a model of a *scheduling strategy*, used by the system, and other glue automata for linking together the generated automata. In our case we use fixed priority preemptive scheduling with the task priorities set according to dead-line monotonic priority assignment, and the original priority ceiling protocol.

While most of the patterns used in the implementation verification from Section 4.1 are directly applicable in the schedulability model, some patterns need extra locations and transitions. For example, patterns modeling execution time must have locations and transitions allowing task pre-emption. We omit details on the scheduler and task control-templates due to their complexity, much of which is similar to the approach taken in [33], with some minor changes. Intuitively the cycle of the scheduler, without considering locking protocol, is as follows:

1. wait for: **a task release** *or* **a finished task**.
2. if a task is running, **preempt the running task**
3. if an eligible task exists, **schedule task with highest priority**; if no eligible tasks exist, **idle**.

A task controller automaton keeps track of parameters such as release-time, deadline, and response time, and signals the scheduler when a task is to be scheduled. The task controller thus acts as a link between the task automaton and the scheduler automaton. This is done through synchronizations in UPPAAL.

Task template For each task in the system, a task template is generated, which is linked to the scheduler model via synchronization. This is illustrated in Figure 27, where the cloud represents the trace of the generated code. In the initial location, *Ready*, the task waits for the task controller to initiate a *run*-signal. From this initial location, the task moves to a pre-empted state, ready to be scheduled. In this preempted state, the clock used to track execution time, c, stopped using the *stopwatch-expression* $c' == 0$. This stops the clock tracking task execution-time, since the task is not running. The task will then be scheduled when it has the highest priority, and yield when it has finished executing, where after the controller template, and hence the scheduler, will be signalled using the run channel used to start the task. This *run*-channel is used only to communicate with this particular task (it is actually an array of channels using the task-id as index). Such a channel exist for each task in the system.

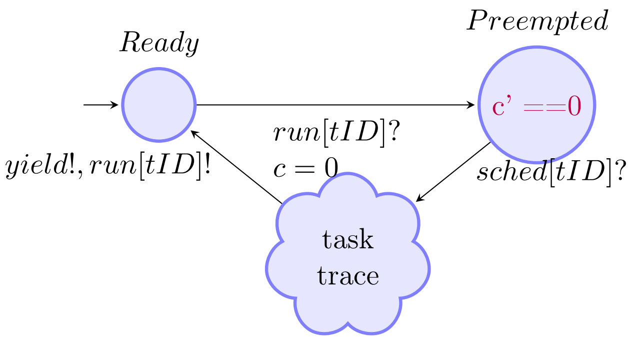


Figure 27: Task template

Interval pattern The interval pattern, $t_s..t_e$, consists mainly of a location as earlier described. Additionally, transitions for task pre-emption and task resumption are needed in order to model pre-emption of the task. The modified pattern is depicted in Figure 28. In this figure, the location *Preempted* is added, in which the execution-time clock is stopped; it is important to note that the response time clock for this task is still running. Transition to this location is enabled when another task is released which means the scheduler must recalculate which task is to be executing, this transition is synchronized with the scheduler using the preempt-channel. The guard $c < t_e$ will prevent preemption of a finished instruction; either the task is done, or preempted

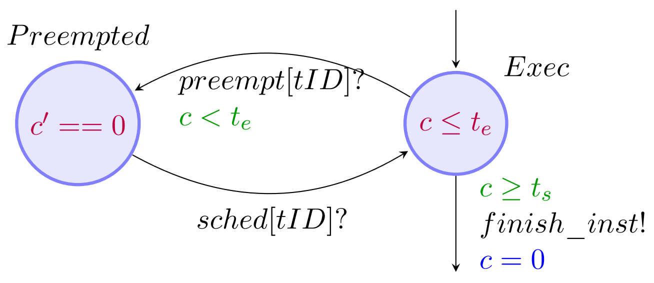


Figure 28: Interval pattern

immediately in the next instruction. The task is resumed when it has highest priority, using channel *sched*.

Critical section pattern To express the blocking of a task we add the location *Blocked* along with the functions:

- *request*, *release*, to handle lock-requests and -releases,
- and the boolean function *acq*, which determines if the request succeeded and the locks were *acq*uired.

If we request a lock which has already been acquired by another task, *acq* evaluate to *False*, enabling the transition to the location *Blocked*, in sync with the scheduler, which will then resume the appropriate task. The scheduler will, when resuming this task using the *sched* channel, reevaluate whether the task can acquire the requested locks. If *acq* evaluates to *True*, the task has successfully acquired the lock, and will resume to the critical section.

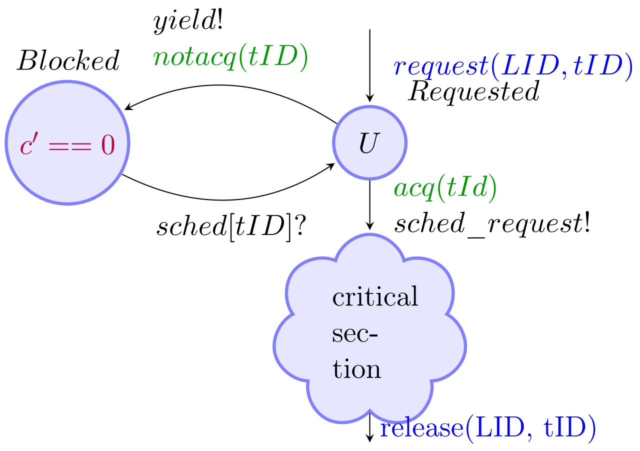


Figure 29: Critical section pattern

Schedulability is verified in the resulting model by verifying the absence of deadlock in the model. Performing schedulability is done on the specifications, which ideally results in a smaller model and generated state-space, which decrease the verification time. The schedulability analysis can also be performed before any implementation is done, and each of the implemented tasks can be verified separately and efficiently according to their specifications. Because of the nature of the models, which pay special attention to resources, the schedulability result is potentially tighter than results from traditional schedulability tests.

5 Conclusion

In this paper we have presented an approach to compositional schedulability analysis of safety-critical Java programs based on annotating tasks with specifications incorporating abstract descriptions of methods underlying temporal and locking behaviour as annotations to method interfaces. We have illustrated the approach on a simple example and tested it on a small prototype system consisting of a few handlers[6]. However, future work clearly include analysis of larger systems and an interesting starting point would be the Canteen library [75] containing Collection classes for Real-time Java.

The approach described in this paper is currently only applicable for SCJ applications running on top of a single core JOP system. JOP has a simple cashing regime, making it relatively easy to model. There is already work in progress on developing multi-core versions of the JOP [136] and more generally it would be interesting to extend our work to software based real-time enabled JVM's running on top of more mainstream (multi-core) processors. For multi-core scheduling we envision exploiting recent results in model based schedulability analysis for multi-core [141] and addressing more mainstream multi-core architectures we envision merging SARTS with ideas from the TetaJ tools, which exploits hardware models made for the Metamoc tool and models of a software implemented JVM, e.g. HVM [87].

The interface behavioural descriptions are usually simple and thus generates models that have far fewer states than the models generated from their Java code. This approach therefore enables analysis of more complex systems and allow library routines to be checked separately and independently of the systems they will be used in. The current specification language for describing temporal and locking behaviour is rather simple in nature and is intended to be simple and close to the descriptions that programmers would write anyway. However, we envision that the specification language could be made richer and in the future be based on modal transition systems and modal I/O automata.

[6]The code for a small sorting machine, modified for the purpose to contain only two periodic tasks. Available on sarts.dk

In our approach, as well as other model based approaches to WCET and schedulability analysis, all loops must be bounded by a fixed number and the analysis rely on potentially unsafe program annotations provided by the programmer. This is a rather conservative approach, especially for nested loops where the bounds for inner loops may depend on the outer loop variable. Extending the interface behavioural descriptions to capture such dependencies would be interesting and it would be possible to use UPPAALs variables to generate models reflecting such dependencies. This approach could furthermore be combined with another approach suggested by Hunt et al. [82] annotating loop bounds using the *Java Modelling Language* (JML) [94], and using the KeY tool [22] to determine the loop bounds symbolically. Some work has already been done on extending JML for WCET analysis [71, 72] and combining deductive methods with our model based approach could potentially eliminate unsafe programmer annotations and provide automated checking with much tighter bounds and thus deem some systems schedulable which today are rejected due to rather conservative loop bounds.

Paper D:

Refactoring Real-Time Java profiles

Hans Søndergaard[1] Bent Thomsen[2], Anders P. Ravn[2], René R. Hansen[2], and Thomas Bøgholm[2],

[1] *VIA University College*
Horsens, Denmark

[2] *Department of Computer Science*
Aalborg University, Denmark

Abstract

Just like other software, Java profiles benefits from refactoring when they have been used and have evolved for some time. This paper presents a refactoring of the Real-Time Specification for Java (RTSJ) and the Safety Critical Java (SCJ) profile (JSR-302). It highlights core concepts and makes it a suitable foundation for the proposed levels of SCJ. The ongoing work of specifying the SCJ profile builds on subclassing of RTSJ. This spurred our interest in a refactoring approach. It starts by extracting the common kernel of the specifications in a core package, which defines interfaces only. It is then possible to refactor SCJ with its three levels and RTSJ in such a way that each profile is in a separate package. This refactoring results in cleaner class hierarchies with no superfluous methods, well defined SCJ levels, elimination of SCJ annotations like `@SCJAllowed`, thus making the profiles easier to comprehend and use for application developers and students.

1 Introduction

Java profiles declare classes and interfaces that embody concepts common for an application area. An application programmer uses the profile to ensure that a concrete implementation conforms to generally accepted construction principles and to get an application that will run on one of several profile implementations. Thus a profile contributes to efficient software development. However, as time passes some features of a profile turn out to be less useful (deprecated) and new features have to be added to cover new technology or there is a desire to specialize the profile to particular areas.

In our work on real-time Java profiles [144, 137, 29], we have observed these phenomena with the Real-Time Specification for Java (RTSJ) which was a first important step in specifying real-time systems in Java [36].

RTSJ was designed to meet the requirements to a Java profile for real-time applications. It extends Java in eight areas: scheduling, memory management, synchronization, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination, physical memory access, and exceptions. However, this profile which in many ways included the "state of the art" as to real-time system development, was (and is) complex to use. Furthermore, RTSJ applications are difficult to analyse, and features like asynchronous transfer of control are hard to use [55].

This has subsequently resulted in different proposals specifying simpler real-time Java profiles, of which Ravenscar-Java [122, 91] and Safety Critical Java (SCJ) [152] are the most prominent.

The SCJ profile is aimed at safety-critical systems which require a safety-critical certification (like DO-178B [125] or ED-12B [126]). Different certification levels in e.g. DO-178B are reflected in SCJ which has three levels: Level 0, 1, and 2, of which Level 0 is the most restrictive. The SCJ profile is based on RTSJ, but with some constraints compared with RTSJ, e.g. the usage of dynamic memory allocation. The programming model is centered on missions, where a mission consists of a set of schedulable objects. At Level 0 only periodic event handlers are permitted; at Level 1, periodic and aperiodic event handlers are permitted; and at Level 2, no-heap real-time threads are permitted too.

Common to those proposals is that they build on RTSJ using *subclassing*. Generally the advantage of using a subclassing approach is first and foremost code and implementation platform reuse. However, as we show in this paper, RTSJ and SCJ with its three levels do not share implementation code.

The disadvantages of using a subclassing approach are among others:

- the class hierarchies become complicated
- the SCJ classes themselves should be simpler and thus have fewer methods, but because they are subclasses of RTSJ classes, they will automatically inherit the public methods from the RTSJ classes

- annotations are therefore introduced and used extensively to tell at which SCJ level a class or a method is allowed
- for application programmers who start real-time programming in Java, the transition to e.g. SCJ becomes unnecessarily complicated
- the same is the case when teaching IT-students real-time programming using the real-time Java profiles.

Therefore, in [29], we concluded that subclassing was not a good way to extend a specification profile, and we suggested that the implementation reuse was ensured by delegation. However, this still left the definitions of the profiles unrelated. Essentially, a class defines an *is-a* relation to superclasses, and subclassing should be a specialization; - in algebraic terms, a conservative extension that preserves all properties including implementations introduced in the superclass. A concept of 'super-classing' or generalization as suggested in [157] could be used to create smaller profiles. However, a super-class or generalization construct does not exist in Java. Furthermore, such a construct is primarily useful for creating a parent class with common or shared behaviour, and as mentioned, RTSJ and SCJ with its three levels do not share implementation code. What RTSJ and SCJ have in common is better described through a *can-do* relation which is implementation independent through the *interface* concept in Java. The purpose of an interface is exactly to abstract from implementations, which also explains why they support multiple inheritance.

The difference is well illustrated by RTSJ, where for instance the concept of time is implemented by abstract and concrete classes. Presumably, here the RTSJ designers have thought in terms of common implementations of the various methods that manipulate time objects. On the other hand, objects that are scheduled are characterized by an interface, because very different objects (handlers, threads) have to do the same operations with different implementations.

In the latest SCJ Draft from July 2010 [153], interfaces are used to some extent, e.g. p.117, Figure 7.2, but without getting a simpler structure, because they are added to the class hierarchy of RTSJ. The same tendency is found in Figure 4 in [164], due to inconsistences in the SCJ Draft. The result is even more complicated interface and class hierarchies than the RTSJ hierarchies, making it more difficult to grasp and use SCJ correctly, especially for newcomers. We return to this example in our refactoring.

With this observation we have been able to find a suitable refactoring of RTSJ such that it supports a clean SCJ definition. The refactoring principle is simple: We start specifying the methods common to SCJ and RTSJ, putting these in a separate real-time core package called `rtcore`. This specification is done entirely by means of Java interfaces. The details of this process are elaborated in Section 2.

This common behaviour in `rtcore`, together with specific methods for SCJ Level0, Level1, Level2, and RTSJ, are next implemented in separate packages which are described in Section 3.

The contributions of this paper are thus: A refactoring of RTSJ and the SCJ levels according to the guidelines above, showing

1. a cleaner structure with clear and well defined levels
2. no changes of the semantics of SCJ and RTSJ
3. no use of `@SCJAllowed` annotations.

Existing applications can thus be compiled with the proposed profiles without any changes, and the resulting code can be run on existing compliant platforms. In the conclusion in Section 4 we give a more detailed assessment of the result.

2 Refactoring

Guidelines for refactoring are found in Fowler's book [64], which also states the constraints on a refactoring: 'When refactoring a software system, it has to be done in such a way that it does not alter the external behavior, yet improves its internal structure'. This is followed in the following refactoring of the SCJ and RTSJ profiles. We observe that SCJ is based on RTSJ and is specified as an extended subset of RTSJ. Furthermore SCJ has three compliance levels, Level0, Level1, and Level2, with the requirements, that any application implemented for a specific level must be able to run correctly on an implementation supporting a higher level.

2.1 Structuring the operations

To get an overview of the operations (methods of classes and interfaces) of all four profiles, SCJ Level0 (SCJ0), SCJ Level1 (SCJ1), SCJ Level2 (SCJ2), and RTSJ, let us look at all the operations of the four profiles. This universe is illustrated by a Venn diagram in Figure 30.

The universe has $2^4 - 1 = 15$ subsets, where

$$rtcore = scj0 \cap scj1 \cap scj2 \cap rtsj$$

This implies that `rtcore` *specifies all the operations which SCJ0, SCJ1, SCJ2, and RTSJ have in common.*

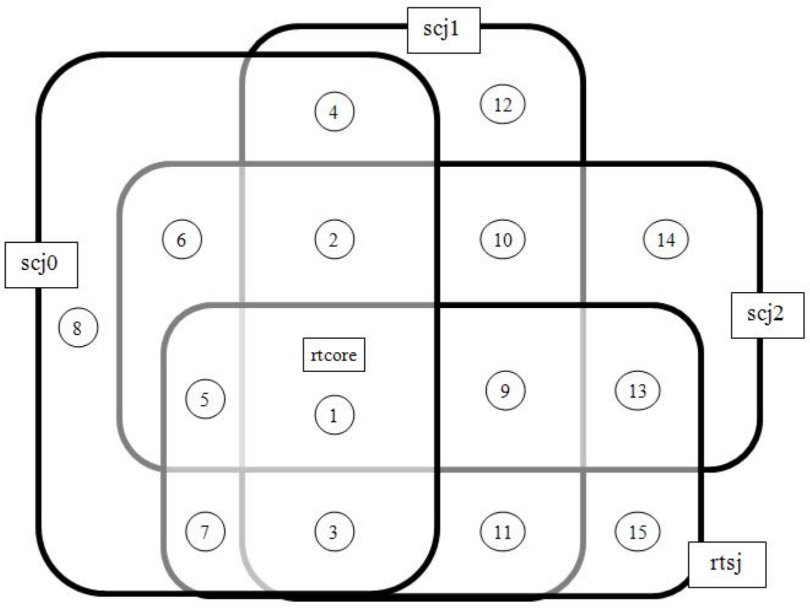


Figure 30: Venn diagram of all the operations of SCJ0, SCJ1, SCJ2, and RTSJ.

More details of the subsets are given in Table 6. The numbers 1-15 correspond to the 15 subsets in Figure 30. As an example,

$$subset1 = scj0 \cap scj1 \cap scj2 \cap rtsj$$

and the crosses (`x`) in all the four profiles mean that they have at least one method in common; e.g. `MemoryArea.enter` is visible in all four profiles and therefore belongs to subset 1. Likewise,

$$subset3 = (scj0 \cap scj1 \cap rtsj) - scj2$$

with zeros (`0`), showing that this subset is empty.

As Table 6 shows, seven of the subsets in Figure 30 are empty, resulting in eight nonempty subsets which are illustrated in Figure 31.

From this analysis follows:

- SCJ1, {1,2,9,10}, is a proper subset of SCJ2, {1,2,9,10,13,14}, but SCJ0, {1,2,8}, is not a subset of SCJ1
- RTSJ, {1,9,13,15}, has operations common to all three SCJ-levels, {1}
- RTSJ has operations common with both SCJ1 and SCJ2, {9}, and also operations common with SCJ2 only, {13}.

This is the basis for the refactoring.

Table 6: Overview of the 15 subsets of operations, including examples

Set	Visibility of methods				Example
	scj0	scj1	scj2	rtsj	
1	x	x	x	x	`MemoryArea.enter`
2	x	x	x		`ManagedEventHandler. cleanup`
3	0	0		0	`- empty`
4	0	0			`- empty`
5	0		0	0	`- empty`
6	0		0		`- empty`
7	0			0	`- empty`
8	x				`Level0Mission. getSchedule`
9		x	x	x	`AsyncEvent.fire`
10		x	x		`AperiodicEventHandler. constructor`
11		0		0	`- empty`
12		0			`- empty`
13			x	x	`NoHeapRealtimeThread. getMemoryArea`
14			x		`ManagedThread.start`
15				x	`SporadicParameters. setMinimumInterarrival`

2.2 Refactoring principles

When refactoring RTSJ and SCJ the following principles have been followed:

- Operations common to RTSJ and SCJ0-SCJ2 are specified in a separate package called `rtcore`
- `rtcore` is specified entirely as Java interfaces
- The four profiles (SCJ0, SCJ1, SCJ2, and RTSJ) have their separate packages
- The four profiles share operations in line with the illustration in Figure 31
- None of the four profiles need to share implementation code.

This means that e.g. the SCJ0 profile will consist of (compare to Figure 31)

- the operations specified in `rtcore`, {1}

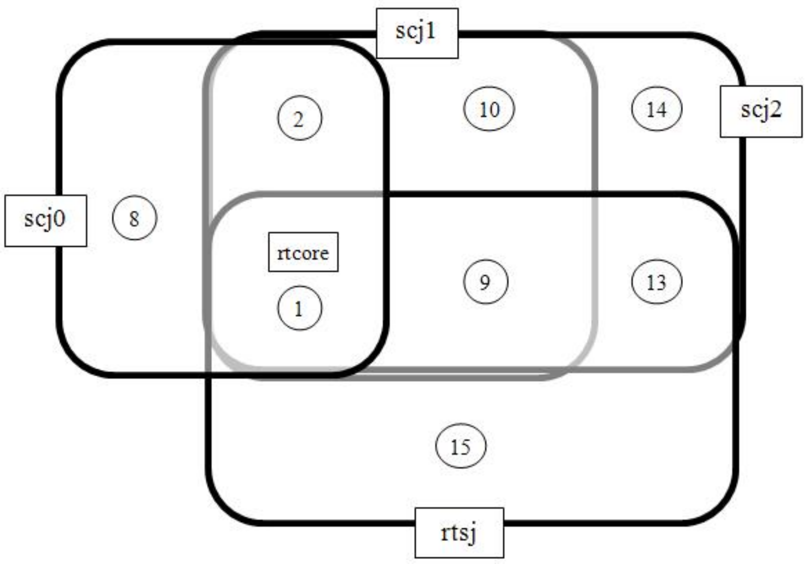


Figure 31: The eight not-empty subsets of operations in SCJ0-SCJ2 and RTSJ.

- some SCJ operations common to SCJ1 and SCJ2, {2}
- some SCJ0 specific operations, {8}.

The same is the case for the other SCJ levels, so that RTSJ consists of

- the operations specified in `rtcore`, {1}
- some operations common to SCJ1 and SCJ2, {9}
- some operations common to SCJ2 only, {13}
- some RTSJ specific operations, {15}.

2.3 Example

The hierarchy in Figure 32 illustrates the ideas in the refactoring through an abstract example.

The common method `m` is specified in the interface `rtcore.IA`. SCJ1 and SCJ2 have one more method in common, called `m12`, and RTSJ has a method called `mrtsj`. All the `A` classes implement the methods specified in the interface `IA` which is the interface `rtcore.IA` or an extension of this interface.

The following test program can be executed, no matter which of the four packages is imported:

```
import scj0.*;  // or scj1, or scj2, or rtsj
```

```
public class TestA
{
  public static void main (String[] args) {
    A a = new A();
    a.m();
  }
}
```

And the following test example works only using package `scj1` or `scj2`:

```
import scj1.*;  // or scj2

public class TestA12
{
  public static void main (String[] args) {
    A a = new A();
    a.m();
    a.m12();
  }
}
```

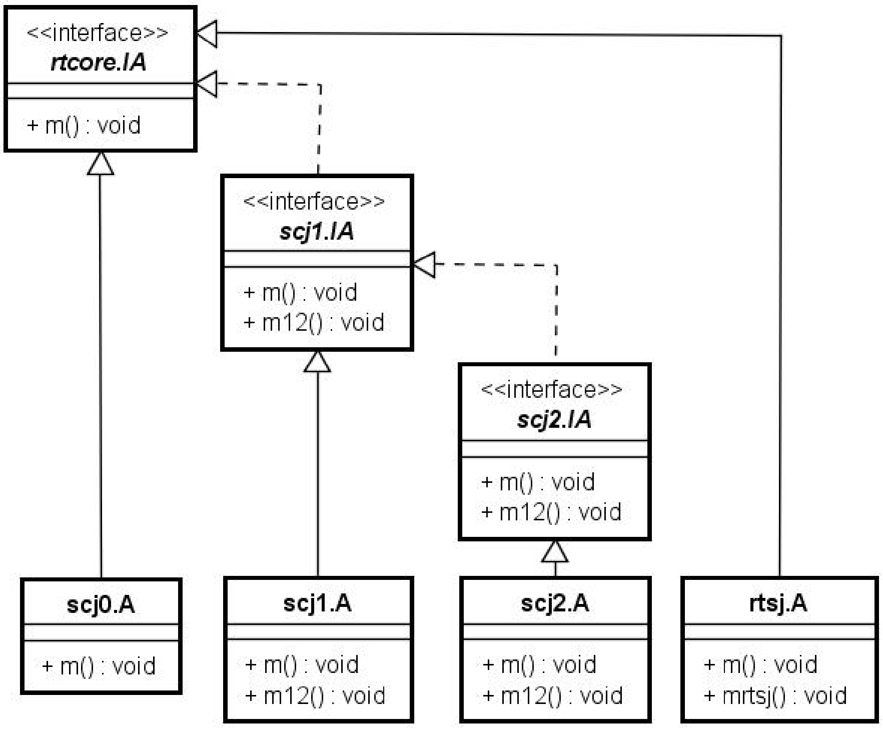


Figure 32: Example showing the refactoring principle.

3 The refactoring outcome

The refactored specification of SCJ0-SCJ2 and RTSJ consists of the five packages shown in Figure 33.

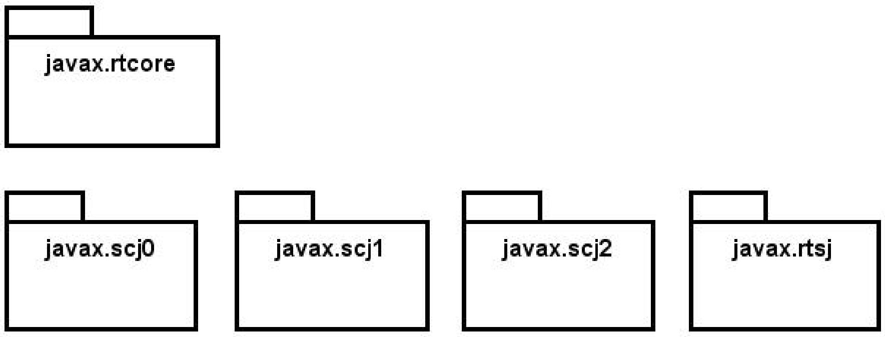


Figure 33: The rtcore, SCJ0-SCJ2, and RTSJ packages.

The content of those packages are described in the subsections below.

3.1 Package javax.rtcore

The `rtcore` package plays a central role. Here is the specification of the common operations of the four profiles. The package consists of 12 interfaces with a total of 39 methods. The names from the corresponding RTSJ classes are left unchanged. We have only prefixed an `I` (the capital letter I) for the interfaces in `rtcore`, - except `interface Schedulable` which already is an interface in RTSJ (as the only interface).

The 12 interfaces are:

- `IScheduler, Schedulable, IReleaseParameters, IPriorityParameters, IPeriodicParameters`
- `IAsyncEventHandler`
- `IMemoryArea, IPortal`
- `IHighResolutionTime, IAbsoluteTime, IRelativeTime`
- `IClock.`

The specification of the methods in the interfaces are unchanged compared to RTSJ and SCJ.

As an example, the `ReleaseParameters` class occurs in both RTSJ and SCJ. In RTSJ, the class has ten methods; in SCJ Level0: no methods; in SCJ Level1 and Level2: three methods. Therefore the interface `IReleaseParameters` will be an empty interface in `rtcore`:

```
public interface IReleaseParameters
{
  // empty
}
```

Similarly, the `AsyncEventHandler` class has two methods in common with both RTSJ and SCJ, resulting in the following interface in `rtcore`:

```
public interface IAsyncEventHandler
  extends Schedulable
{
  public void handleAsyncEvent();
  public void run ();
}
```

3.2 The four profiles

Now, let us look at the implementation of the four profiles.

As mentioned in Subsection 2.2:

- the four profiles have their separate packages,
- they share operations through interfaces,
- but do not share implementation code.

This implies that the class hierarchies for the four profiles are completely separated, each having their separate package, cf. Figure 33. In that way, only the necessary methods and classes are included in a specific profile.

Let us, as an example, look at the implementation of the `IReleaseParameters` interface described above. A class diagram is shown in Figure 34.

First, the implementation of `IReleaseParameters` in SCJ0:

```
package javax.scj0;

import javax.rtcore.IReleaseParameters;

public abstract class ReleaseParameters
  implements IReleaseParameters, Cloneable
{
  protected ReleaseParameters() {..}

  public Object clone() {..}
}
```

Next, let us look at SCJ1 with two new methods specified:

```
package javax.scj1;

import javax.rtcore.IRelativeTime;
import javax.rtcore.IAsyncEventHandler;
```

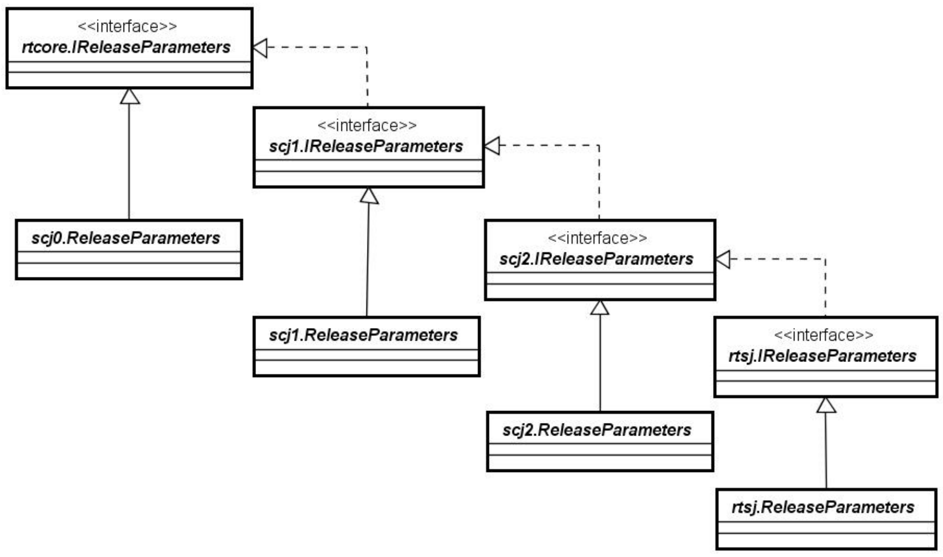


Figure 34: class `ReleaseParameters` in SCJ0-SCJ2 and RTSJ.

```
public interface IReleaseParameters extends
  javax.rtcore.IReleaseParameters
{
  public IRelativeTime getDeadline();
  public IAsyncEventHandler
           getDeadlineMissHandler();
}

package javax.scj1;

import javax.rtcore.IRelativeTime;
import javax.rtcore.IAsyncEventHandler;

public abstract class ReleaseParameters
  implements IReleaseParameters
{
  protected ReleaseParameters(
    IRelativeTime deadline,
    IAsyncEventHandler missHandler) {..}

  public Object clone() {..}

  public IRelativeTime getDeadline() {..}

  public IAsyncEventHandler
    getDeadlineMissHandler() {..}
}
```

SCJ2 is like SCJ1 with no new methods. But RTSJ has seven more methods:

```
package javax.rtsj;

import javax.rtcore.IAsyncEventHandler;

public interface IReleaseParameters
  extends javax.scj2.IReleaseParameters
{
  public IAsyncEventHandler
    getCostOverrunHandler();
  //six more methods are specified in RTSJ
  ..
}

package javax.rtsj;

import javax.rtcore.IAsyncEventHandler;
import javax.rtcore.IRelativeTime;

public abstract class ReleaseParameters
  implements IReleaseParameters,Cloneable
{
  // two constructors

  // a total of 10 methods:
  public IAsyncEventHandler
    getCostOverrunHandler() {..}
  ..
}
```

The implementation of the `AsyncEventHandler` looks similarly, see class diagram Figure 35.

Here, the `AsyncEventHandler` has the same methods in SCJ0-SCJ2, which are all specified in `rtcore.IAsyncEventHandler`, but in RTSJ another 21 methods are specified in the `rtsj.Schedulable` interface, so that all together the
`rtsj.AsyncEventHandler` class has 32 methods and 7 constructors:

```
package javax.rtsj;

import javax.rtcore.IAsyncEventHandler;

public class AsyncEventHandler
  implements IAsyncEventHandler, Schedulable
{
  // seven constructors

  public void handleAsyncEvent () {..}
  public void run () {..}

  // thirty more methods:
  ..
}
```

The remaining classes in the profiles are implemented in the same way, see link in footnote [7] for the complete source code.

3.3 Example

With a much simplified structure of SCJ and with the specification of the common behaviour of the SCJ and RTSJ profiles in the package `rtcore` using interfaces, it is now possible to simplify some of the class hierarchies in SCJ. As an example of this simplification, let us look at the event handling hierarchy from the SCJ Draft [153], and the revised SCJ event handling hierarchy

[7] `http://www.it-engineering.dk/HSO/index.html`

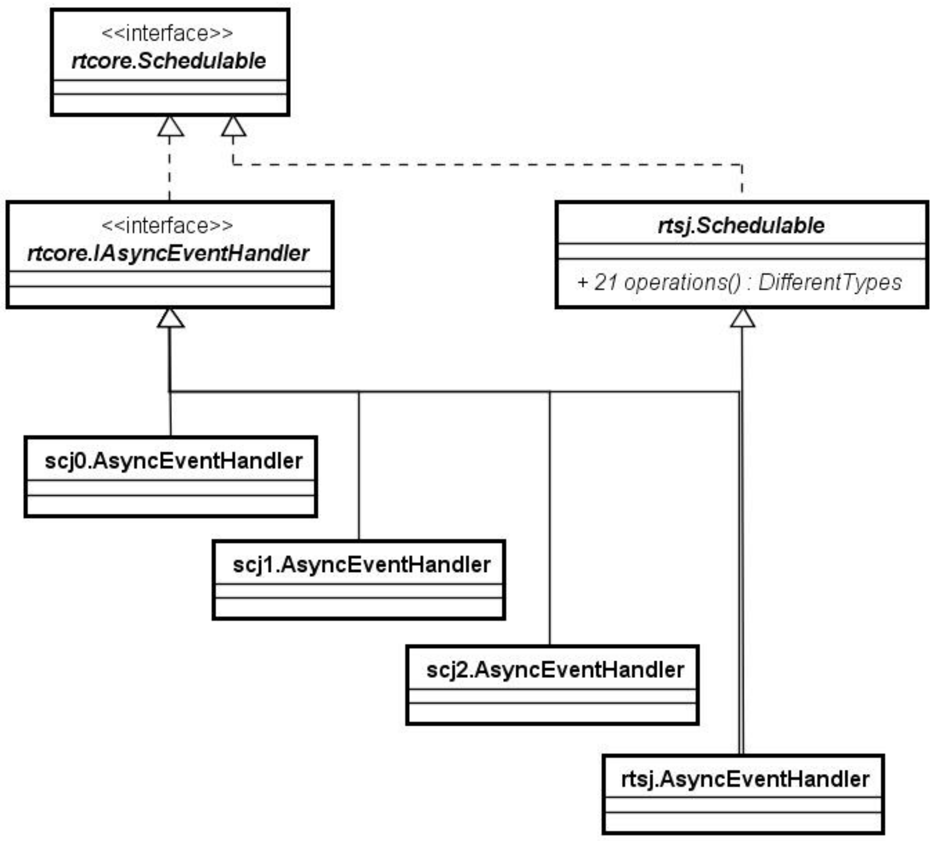


Figure 35: class `AsyncEventHandler` in SCJ0-SCJ2 and RTSJ.

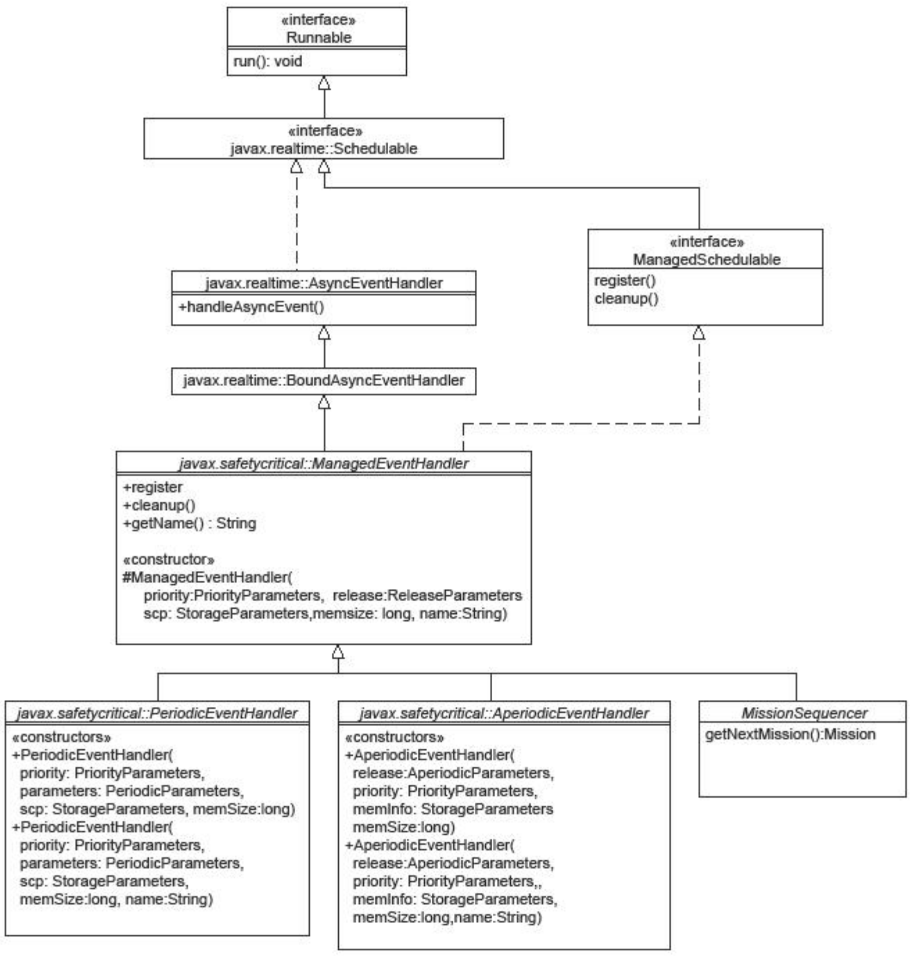
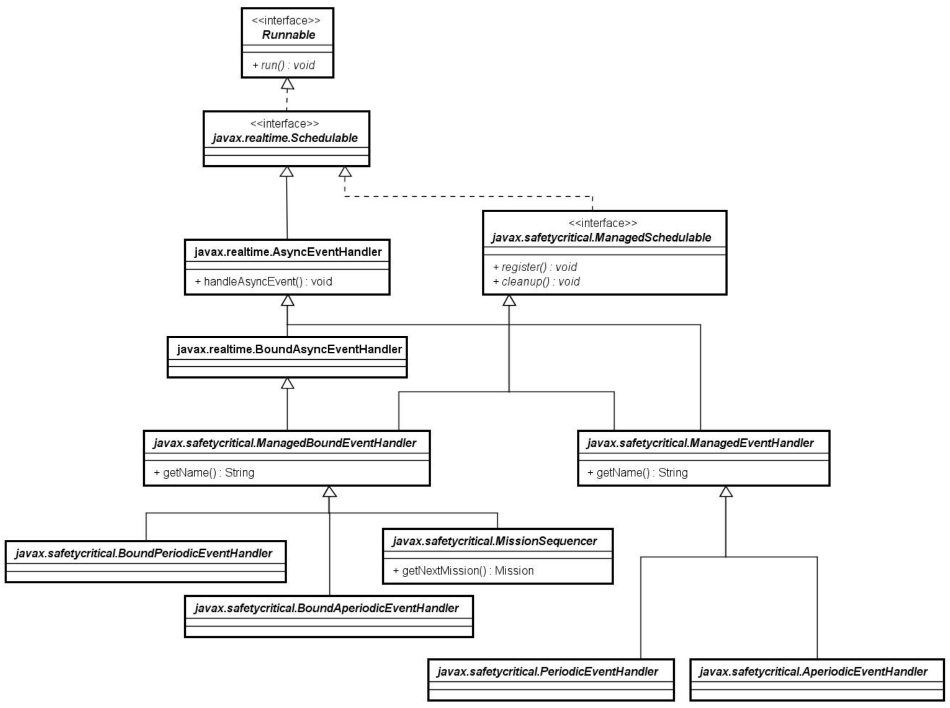

«interface»
Runnable
run(): void
«interface»
javax.realtime::Schedulable
«interface»
ManagedSchedulable
register()
cleanup()
javax.realtime::AsyncEventHandler
+handleAsyncEvent()
javax.realtime::BoundAsyncEventHandler
javax.safetycritical::ManagedEventHandler
+register
+cleanup()
+getName() : String
«constructor»
#ManagedEventHandler(
priority:PriorityParameters, release:ReleaseParameters
scp: StorageParameters,memsize: long, name:String)
javax.safetycritical::PeriodicEventHandler
«constructors»
+PeriodicEventHandler(
priority: PriorityParameters,
parameters: PeriodicParameters,
scp: StorageParameters, memSize:long)
+PeriodicEventHandler(
priority: PriorityParameters,
parameters: PeriodicParameters,
scp: StorageParameters,
memSize:long, name:String)
javax.safetycritical::AperiodicEventHandler
«constructors»
+AperiodicEventHandler(
release:AperiodicParameters,
priority: PriorityParameters,
memInfo: StorageParameters
memSize:long)
+AperiodicEventHandler(
release:AperiodicParameters,
priority: PriorityParameters,,
memInfo: StorageParameters,
memSize:long,name:String)
MissionSequencer
getNextMission():Mission

Figure 36: SCJ Handler Hierarchy, from [153].

from [164], both mentioned in the Introduction and shown in Figure 36 and Figure 37.

This event handling hierarchy is a consequence of using implementation inheritance. By this, the `AsyncEventHandler` class with 32 methods is inherited through the `BoundAsyncEventHandler` class to the event handler classes in SCJ , see Figure 36. This results in some inconsistences in SCJ because a bound asynchronous event handler is permanently bound to a dedicated real-time thread which is self-suspending, and in SCJ Level0 and Level1 the handlers cannot self-suspend. Wellings [164] has solved this inconsistency at the price of an even more complicated class hierarchy, see Figure 37.

Instead of retaining the event handling class hierarchy described above, simpler and more comprehensible hierarchies can be constructed by following



<<interface>>
Runnable
+ run() : void
<<interface>>
javax.realtime.Schedulable
javax.realtime.AsyncEventHandler
+ handleAsyncEvent() : void
<<interface>>
javax.safetycritical.ManagedSchedulable
+ register() : void
+ cleanup() : void
javax.realtime.BoundAsyncEventHandler
javax.safetycritical.ManagedBoundEventHandler
+ getName() : String
javax.safetycritical.ManagedEventHandler
+ getName() : String
javax.safetycritical.BoundPeriodicEventHandler
javax.safetycritical.MissionSequencer
+ getNextMission() : Mission
javax.safetycritical.BoundAperiodicEventHandler
javax.safetycritical.PeriodicEventHandler
javax.safetycritical.AperiodicEventHandler

Figure 37: Revised SCJ Handler Hierarchy, from [164].

the ideas of "inheritance for specialization". This is the idea illustrated by the class diagram in Figure 38.

The problems in SCJ with self-suspending / not self-suspending event handlers, observed and solved by Wellings in [164], are solved in a much cleaner manner as follows:

- SCJ Level0: The periodic eventhandler is implemented according to its specification which tells that the handler is not self-suspending
- SCJ Level1: Similarly at this level for both periodic and aperiodic event handlers
- SCJ Level2: The handlers can self-suspend and are implemented in accordance with this specification.

This emerges clearly from Figure 38, in contrast to the class diagram in Figure 37.

The following example illustrates how a simple SCJ Level0 mission could be written. As described in the Introduction, one of the extensions in SCJ compared to RTSJ is the *mission* concept, where a SCJ application consists of one or more missions, and a mission consists of a bounded set of `ManagedEventHandler` objects. In SCJ Level0, only periodic event handlers are allowed, cf. Figure 38.

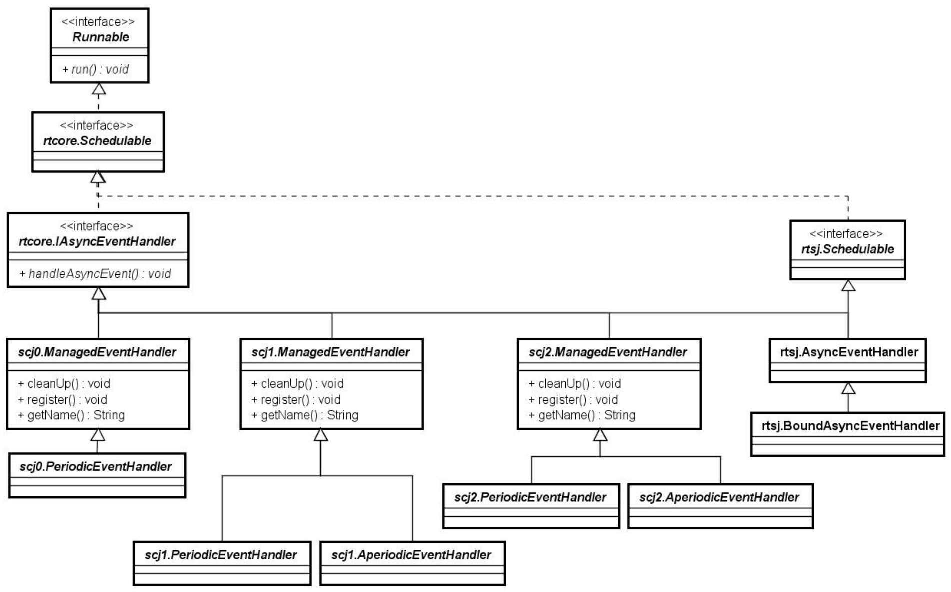


Figure 38: Refactored Event Handler Hierarchy for SCJ and RTSJ.

```
import javax.scj0.*; // or scj1, or scj2

class AMission extends Mission
{
  ManagedEventHandler[] eventHandlers;

  public AMission () {
    eventHandlers = new ManagedEventHandler[1];
  }

  public void initialize () {
    eventHandlers[0] = new Periodic (
      new RelativeTime (0,0), new RelativeTime (500,0)
    );
    eventHandlers[0].register();
  }

  // ...

}

class Periodic extends PeriodicEventHandler
{
  static final int priority = 13;
  static final int nativeStackSize = 1000, javaStackSize   = 1000;
  static final long storeSize = 10000, memSize   = 10000;
  Periodic (RelativeTime start,
            RelativeTime period) {
    super (
      new PriorityParameters(priority),
      new PeriodicParameters(start, period),
      new StorageParameters (storeSize,
            nativeStackSize, javaStackSize),
            memSize);
  }

  public void handleAsyncEvent() {
    // the logic to be executed every period
  }
}
```

This example is implemented to the SCJ Level0, but can be executed at Level1 or Level2, only by changing the `import` statement. Notice, that no `@SCJAllowed` annotations are necessary.

In a similar way, a mission for Level1 can be implemented containing both periodic and aperiodic event handlers. Such a mission can also be executed at Level2, but not at Level0.

4 Conclusion

The presented refactoring of the Real-Time Specification for Java (RTSJ) highlights core concepts and is a suitable foundation for the proposed levels of the Safety Critical Java (SCJ) profile (JSR-302). The specification of the common behaviour of the SCJ and RTSJ profiles in the package `rtcore` using interfaces, and the subsequent implementation of the profiles, show that it is possible to refactor the profiles without changing the semantics of them.

This separation in packages has some direct consequences:

- each profile has only the necessary methods and classes
- the extensive use of `@SCJAllowed` annotations in SCJ becomes unnecessary
- the individual profile is easier to comprehend.

Another consequence of the refactoring process is that it shows that it is now *possible to simplify some of the class hierarchies in SCJ*, e.g. the event handling hierarchy from the SCJ Draft [153] and the revised SCJ event handling hierarchy from [164], is even cleaner in our approach.

This simplification is in line with our earlier work [29] where we advocated for using "inheritance for specialization" instead of using "inheritance for limitation" (also called implementation inheritance). Furthermore, the advantages of clearer and well defined levels should not be underestimated when using the profiles. Especially for beginners and in teaching real-time programming using SCJ, this is of great importance.

Further work includes a full refactoring of implementations. Our plan is here to work with the implementation of Timesys [159] with which we have previous experience as basis for a delegation based implementation of a specialized profile [29]. This will furthermore give us some idea about the potential for sharing of implementations. Although we have renounced on code sharing in favour of clear structure by using interfaces instead of abstract classes as specifications, we are definitely not against implementing common elements through inheritance in a concrete implementation.

5 Acknowledgement

We are grateful for the instructive comments by the reviewers which have helped us clarify several points in this paper.

Paper F:

Towards harnessing theories through tool support for hard real-time Java programming

Thomas Bøgholm[1], Christian Frost[1], René R. Hansen[1], Casper Svenning Jensen[1], Kasper Søe Luckow[1], Anders P. Ravn[1], Hans Søndergaard[2], and Bent Thomsen[1]

[1] *Department of Computer Science*
Aalborg University, Denmark

[2] *VIA University College*
Horsens, Denmark

Abstract

We present a rationale for a selection of tools that assist developers of hard real-time applications to verify that programs conform to a Java real-time profile and that platform-specific resource constraints are satisfied. These tools are specialized instances of more generic static analysis and model checking frameworks. The concepts are illustrated by a case study, and the strengths and the limitations of the tools are discussed.

1 Introduction

For systems that have to meet strict timing or safety requirements it is often argued [39] that code should be automatically synthesised from high level models that facilitate formal verification of critical timing and safety properties. By systematic and careful code synthesis, it is possible to retain all or most of the important properties of the high-level model in the generated code. In this way formally verified, highly robust and reliable software can be obtained at a much lower cost than similar software developed in a more traditional way. However, it is likely to still take a while before the goal of fully automated code synthesis is reached. That raises the question of what researchers can do in the meantime to improve quality and efficiency of the development process? In [101] it was hypothesised that the incremental move away, in the embedded systems community, from the C programming language and real-time kernels, towards more structured languages with good tool support at all levels, would improve the development process.

In this article we report on recent work, enhancing the suitability of Java for developing embedded real-time systems. We have chosen to work with Java because it comes with several defined and documented profiles for real-time programming and because the profiles are supported by platforms that have been demonstrated to work. The profiles and platforms are discussed in Section 2. However, the profiles cannot in themselves ensure that applications perform predictably. There are many issues, some examples are: ensuring that only allowed features and constructs are used, checking that platform resources (memory and processor time) meet the demands of the executing program, and providing interfaces to special purpose hardware. Therefore we have engaged in harnessing theories as well as implementing and adapting tools to assist in verifying properties dictated by the chosen profile and conformance to platform limitations. The tools and their specialisation(s) are introduced in Section 3. In Section 4 we look at two case studies. The first case study looks at tool use. It is the mine pump example well known from the literature. As explained in connection with the case study, the experiments have been encouraging. Then a larger industrial case study is discussed.

We will then, in Section 5, discuss related work, and finally in Section 6, comment on limitations of current tools and the need for tool integration and specialization. This outlines what kind of theories and tools we expect to see harnessed in a truly supportive Real-Time Java development environment.

2 Java and Real-Time Systems

Since its appearance in 1995, Java has spread tremendously as a software development language; it is used to program all kinds of software from servers to smart cards, and it is now the first (and often the only) language for young programmers joining the industry. Especially the Internet propelled Java into mainstream computing, because there was a need for a language that was portable and truly object-oriented, eliminating the error-prone programming of memory allocation and pointer manipulation.

Java features a clean object-oriented model-based on single inheritance with the notion of interfaces to facilitate a safe, albeit limited, form of multiple inheritance. Java presents a relatively clean type system based on a limited set of primitive types and an unlimited set of constructed types, called reference types, all belonging to a type hierarchy with the type `Object` at the top. Java achieves portability via the Java Virtual Machine (JVM) which implements a managed heap, where all objects are allocated and where objects are subjected to garbage collection when they are no longer in use by the program. Java has a wide variety of control features such as sequencing, selection statements and loops. Java also features a clean exception model and the notion checked exceptions, i.e. exceptions are part of the interface of methods and will be checked by the type checker, except for a small number of unchecked exceptions.

Java was one of the first mainstream programming languages to have a platform independent concurrency model-based on a thread model. A thread object has a designated run method that is executed when the thread's start method is called. Threads can collaborate based on a shared memory model, and Java features lock based concurrency control built into every object created by a Java program. Locks are not acquired explicitly, only implicitly via synchronized methods and synchronization blocks. Threads can be suspended waiting on a lock and may be woken up by notify signals issued by another thread holding the given lock. Java has a soft real-time sleep method that suspends a thread for a designated duration.

Originally Java was developed as a programming language for embedded systems, although several of its features make it less suited for predictable, real-time embedded systems: The virtual machine, that gave portability, was considered inefficient both in terms of time and space. Furthermore, the automatic garbage collection and dynamic class loading made it impossible to analyse and predict execution time and memory consumption. Thus several variants, so called profiles, have been proposed to eliminate the features deemed unsuitable for hard-real time embedded system programming. We review three of the profiles in the following subsections.

2.1 RTSJ Profile

The Real-Time Specification for Java (RTSJ) [127] has been specified in order to rectify a number of issues preventing the adoption of the Java programming language for real-time systems development. The RTSJ 1.0 specification is formally defined in the JSR 1 [147] and is currently being revised to RTSJ 1.1 in JSR 282 [116].

RTSJ considers both soft and hard real-time systems. The specification introduces a number of concepts related to real-time systems for the use of programmers; it changes parts of the Java semantics which are problematic for real-time systems; and it provides facilities allowing the programmer to avoid certain elements of Java. These additions and changes can be divided into eight categories: schedulable objects, memory management, real-time threads, asynchronous event handing and timers, asynchronous transfer of control (ATC), physical and raw memory access, time values and clocks, and resource sharing and synchronisation.

Most notably, the concept of *schedulable objects* has been introduced. Schedulable objects are supported by a number of classes allowing the programmer to express real-time concepts such as temporal-scopes, deadlines, release patterns, priorities, and cost. The existing Java thread model is extended with schedulable real-time threads. Furthermore, asynchronous event handlers are schedulable objects, allowing them to express the same computations as real-time threads.

Another notable addition is a new memory model, containing two new types of memory in addition to the existing heap memory. The purpose of the new memory types is to avoid allocation in the heap memory, and thus avoid having a garbage collector to deallocate memory from the heap. The reason for this change is the difficulty in implementing a time predictable garbage collector. The two new memory types are *immortal memory* and *scoped memory*.

Immortal memory allows memory to be allocated only and is meant for persistent objects needed throughout the lifetime of the program.

Scoped memory allows both allocation and deallocation, similar to heap memory, but it only supports deallocation of its entire memory area. That is, the memory area is deallocated as soon as no schedulable objects of the system use it. This results in a memory area supporting time predictable allocation and deallocation.

The RTSJ maintains support for garbage collection and heap memory for applications where incremental soft real-time garbage collection is considered a reasonable solution. However, after the first version of RTSJ appeared in 2000, the focus on real-time garbage collectors have grown. Now different real-time garbage collection algorithms exist [42], and several RTSJ implementations using these algorithms are available today, such as Java RTS from Oracle/Sun Microsystems [146], WebSphere Real Time from IBM [83], and JamaicaVM from Aicas [5].

2.2 SCJ Profile

Since the purpose of the RTSJ is to allow a large range of real-time applications to be developed, it is broad and imposes few limitations on how to structure the application. As an example, both a thread and event handler paradigm is supported. RTSJ is too liberal to effectively support programming of high-integrity applications, therefore effort has been put into defining a profile targeted at safety-critical systems development. The first such profile was the Ravenscar-Java profile [91] which inspired work on the Predictable Java (PJ) profile [29] and the Safety Critical Java (SCJ) profile, developed under JSR-302 [115]. The SCJ standardisation is still ongoing, and the specification is therefore only available as a draft as of now.

Safety-critical applications are often subject to a rigorous certification process, e.g. dictated by a legal statute. Therefore, the SCJ profile is specifically designed to be amenable to such processes.

The SCJ takes into account the presence of the RTSJ. Specifically, the SCJ is a specialisation of the RTSJ where unwanted functionality is avoided using explicit annotations.

Two major improvements of the SCJ with respect to the RTSJ are the introduction of missions and compliance levels which both contribute to a simpler programming model. The two improvements are described in the following.

Missions

An SCJ compliant application consists of one or more missions, which in turn consist of periodic and aperiodic event handlers and `NoHeapRealtimeThread` objects. Missions go through three different phases during their life-time, see Figure 39.

Initialisation: Objects, real-time threads, and memory areas needed throughout the mission's lifetime are created and initialised. The phase is not considered time-critical, meaning that no real-time constraints are guaranteed.

Execution: When the mission is in this phase, the operations are time-critical. Objects created as part of the initialisation phase can optionally be used and modified if they are mutable.

Termination: When all handlers and threads have completed, the mission enters its termination state. Here, a clean-up can be made, and afterwards the mission can either terminate entirely or it can re-initialise by returning to the initialisation phase.

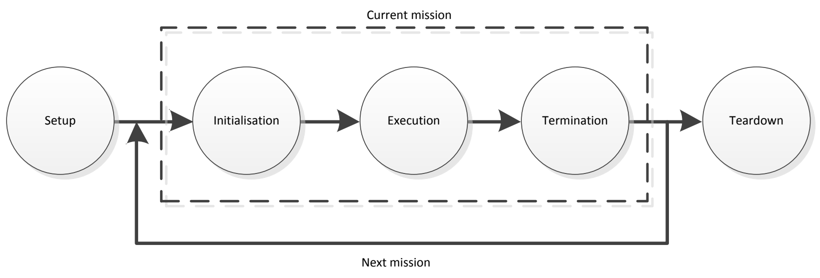


Figure 39: The phases involved in a real-time application's lifetime.

Compliance Levels

The application area for safety-critical systems is wide. That is, applications range from complex multi-threaded to simple single-thread applications. Therefore, since the cost of a certification process is highly dependent on the application's complexity, it is desirable to restrict the complexity, thereby easing the certification process. To accommodate this, the SCJ profile defines three compliance levels:

Level 0 applications consist a sequence of missions, where each mission must follow the programming model needed for a Cyclic Executive Scheduled (CES) application. Each mission can therefore be thought of as a set of periodic tasks placed in a fixed schedule on a time-line. The schedule must either be constructed offline or by an initialization tool.

Level 1 applications consist of a sequence of missions. However, each mission at this level uses Fixed-Priority Preemptive Scheduling (FPS), where handlers are scheduled for execution based on a predefined priority. Handlers are either periodic event handlers or aperiodic event handlers. Due to preemption, access to shared objects must be synchronized using a ceiling protocol.

Level 2 allows applications to use multiple concurrently executing missions. Besides allowing sequential transition between missions, level 2 supports nested missions. Also this level allows the use of `NoHeapRealtimeThread` objects in missions, which are real-time threads that do not access the heap memory, and do not use the Java methods `notify()` and `wait()`.

2.3 PJ Profile

Due to the still ongoing effort of standardising the SCJ profile, we have developed a Predictable Java (PJ) profile [29] suggesting potential simplifications.

The primary contributions of the PJ profile are to redefine the inheritance relationship to the RTSJ and to redefine the programming model of missions.

As previously mentioned, the SCJ profile assumes the presence of the RTSJ from which it inherits. The PJ profile recognises that the notion of *inheritance* has different interpretations depending on its application. The SCJ profile is an instance of the interpretation when inheritance is used for limitation, that is, the SCJ, being a relatively concrete and simple Java profile, inherits from the broader, more flexible RTSJ specification. Effectively, this means that the specifications of subclasses do not comply with the specifications of their respective parent classes. In this interpretation, unwanted functionality from a parent class is in SCJ excluded by annotations. Specifically, the `@SCJAllowed` annotation is used for specifying allowed functionality from parent RTSJ classes. This has the undesirable property of relying on external tool support which examines the source code files to determine whether or not the application conforms to the profile.

PJ uses inheritance for specialisation, that is, the specifications of the subclasses satisfy the specifications of the parent classes. Specifically, the PJ profile has simpler class hierarchies than SCJ which is extended from the much broader and flexible RTSJ specification through inheritance.

SCJ organises schedulable objects into one or more missions depending on whether the system undergoes mode transitions during its life-time. The SCJ regards a *mission* as a simple container of schedulable objects. The PJ profile recognises that missions may in fact be more than simple containers and PJ proposes that missions are handlers. This implies that missions may be nested and sequenced. It also implies that the initialisation and termination phases are part of a mission and thus needs to adhere to strict timing constraints. Besides being a more precise representation of the mission concept, missions as first-class handlers also introduce a variety of simplifications to the PJ. Initially, since missions are handlers like the schedulable objects comprising the system, a new class hierarchy is not necessary.

3 Tool Support for Real-Time Profiles

While the profiles and platforms discussed in the previous section are important for developing real-time applications in Java, they do not by themselves provide any guarantees that the application under development will actually perform predictably, e.g., not exceed the time bounds specified and not consume more memory than available on the platform.

In this section we review the kinds of tools that can provide a programmer with such guarantees about an application under development: ensuring that the application is compliant with the chosen profile and that it does not (attempt to) consume more resources than specified and available on the platform. Today such tools are stand-alone or only partly integrated. How-

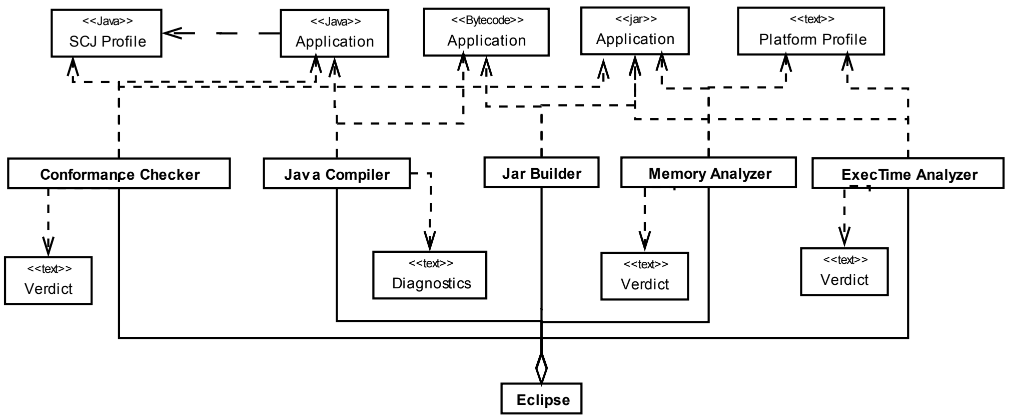


Figure 40: Envisioned tool support: A workbench for analysing Real-Time Java programs.

ever, we believe that such tools will only become truly useful when integrated in the software development environment use by mainstream programmers. The kind of tool we envision is shown in Figure 40. For each tool kind we discuss a few available tools and/or tools currently under development.

3.1 Conformance Checking

For an application programmer working with the different real-time profiles for Java, one of the most fundamental tools is a *conformance checker* that verifies that an application is indeed conforming to the specified profile. This includes checking that only allowed language constructs, classes, and libraries are used in the application. Also, it may enforce specific coding styles, absence of particularly problematic or dangerous code patterns, as well as ensuring that the profile's real-time facilities are used consistently.

We have implemented a prototype tool, the RT-Java tool, that can verify that only *white-listed*, i.e., specifically allowed, library classes and methods are used in a given application. The checker works at the bytecode level and is implemented using the WALA framework [162].

Furthermore, inspired by Ravenscar ADA [44], both SCJ and PJ forbid the use of recursively defined methods. The absence of recursion is easily checked by ensuring that the call graph of an application is acyclic. If this turns out to be too imprecise, i.e., results in too many false positives, the call graph can quite easily be made more precise, e.g., by making the analysis flow- or context-sensitive, at the cost of making it more expensive in terms of both time and memory consumption.

3.2 Exception Analysis

An uncaught exception is highly undesirable in most applications. In an embedded real-time system it may have catastrophic consequences. This is especially so in Java due to the Java semantics of terminating threads with uncaught exceptions without any notification. By performing *exception analysis*, a tool can automatically verify that an application will not give rise to any uncaught exceptions.

We are not aware of any stand-alone tools that perform exception analysis as described above. However, exception analysis is an integrated part of many of the analyses included in the WALA framework [162].

Correct exception handling is of course only one of many properties to be analysed to ensure correct functional behaviour of a real-time Java application. There are many tools that will help a developer ensuring correct functional behaviour of a (real-time) Java application, many of which are based on deductive reasoning such as ESC/Java [62] and the KeY system [22]. A comprehensive overview is beyond the scope of this paper as our main concern is harnessing theories through tools in support of establishing correct behaviour with respect to non-functional requirements such as memory and time.

3.3 Memory Analysis

The *scoped memory* model, employed by several of the real-time profiles discussed in the previous section, gives programmers a high degree of flexibility and control over memory allocation and, in particular, release of memory that is no longer needed. This control and flexibility is achieved by organising the physical memory into scoped memories that are dynamically allocated and deallocated in a structured way, according to the lifetime of the scopes. Thus, in the simplest case, scoped memories are allocated following a stack discipline and, indeed, this is the only allocation ordering allowed by the PJ and SCJ profiles. Furthermore, the PJ profile does not allow dynamic creation of new memory scopes while the SCJ profile does. In contrast to both the SCJ and PJ profiles, the RTSJ profile permits more complex allocation hierarchies for scoped memories.

With control of memory allocation and deallocation left in the hands of the programmer also comes a risk that is not present in garbage collected systems: namely the possibility of creating *dangling references* when deallocating a scoped memory containing an object that is referenced in another scoped memory that has not yet been deallocated. To avoid dangling references, the underlying structure of the scoped memories is used: it is expressly forbidden for a reference in a memory scope with a longer lifetime (as determined by its place on the *scope stack*) to point to an object in a memory scope with a shorter lifetime. Thus, in the SCJ profile and PJ profiles, references may not point to objects in scoped memories that are closer to the top, i.e., scopes that are younger. In the RTSJ profile the situation is more complicated since the scoped memories do not follow a strict stack discipline but allows for the more general structure of a *cactus stack*. Thereby the RTSJ potentially makes it even harder for programmers to understand the runtime memory structure of a program, and they must therefore be even more careful to avoid creating references from a memory scope with a longer lifetime to an object in a memory scope of a shorter lifetime.

RTSJ checks this dynamically at run-time, raising an exception in case of violation of the safety property. However, this solution is first of all expensive as the safety property has to be checked for each assignment and secondly it only removes the danger of dangling pointers at the cost of introducing an exception, which is difficult to handle for the programmer.

To help programmers ensure that all references are pointing in the "right direction", tool support is essential. By performing *memory analysis* a tool can track the memory scope hierarchy at various program points and verify that no references violate the rules and thus potentially result in a dangling reference. Using standard techniques from static analysis, e.g., adding flow- and/or context-sensitivity, it is possible to make the memory analysis more precise on a case by case basis and thereby find an acceptable tradeoff between speed and precision of the analysis for a given project.

Using the WALA framework [162], we have implemented a prototype tool that uses a context-dependent *points-to* analysis, using scoped memories as calling contexts, to determine if a PJ application can potentially violate the rules for scoped memory (as defined by the PJ profile). Due to the straightforward and relatively simple definition and use of scoped memories in the PJ profile, there is no need for programmer annotations or interaction.

In addition to ensuring that scoped memory is used correctly, it is necessary to ensure that the application does not exceed available memory by undertaking a Worst Case Memory Consumption analysis. Such analyses have not received the much attention yet, but one approach is described in [121].

3.4 Worst-Case Execution Time Analysis

In order to analyse the schedulability of the set of tasks comprised by an application, it is necessary to determine the *worst-case execution time* of each of the tasks. This can be done either by static analysis, called *WCET analysis*, or by comprehensive simulation. While simulation has the advantage of being relatively easy to set up and perform, it may give rise to *unsound* results, i.e., results that are overly optimistic and underestimate the true WCET of a task. In non-safety critical and/or soft real-time applications this may be sufficient, but for systems with hard real-time deadlines, potentially performing safety critical tasks, it is essential to have sound WCET estimates for every task in the system.

For analysis based WCET estimates, the inherent difficulty of performing precise program analysis is often evidenced by imprecise and overly pessimistic WCET analysis results. The lack of precision in WCET analysis is often exacerbated by some of the advanced features present in modern hardware architectures, especially caching and pipelining, that have major impact on the actual running time of any given task. One way of overcoming the challenge presented by modern hardware, is for the WCET analysis to make explicit (abstract) models of the underlying hardware and take the relevant features into account.

WCET Analyzer

WCET Analyzer (WCA) [135, 79] is a static code analysis tool for conducting WCET analysis of Java bytecode executed on the Java Optimized Processor (JOP). JOP [128] is a hardware implementation of the Java Virtual Machine which emphasises real-time properties. Among others, JOP facilitates known execution times of each Java bytecode. The relative simplicity and predictability of the JOP architecture [132] and, in particular, the use of a method cache instead of more general cache disciplines, makes it significantly easier to perform precise WCET analysis. In the following we describe the *WCET Analyzer* tool for JOP.

WCA employs two distinct strategies for WCET analysis; one is the Implicit Path Enumeration Technique (IPET) [97] and the other models the real-time application using timed automata in the verification tool Uppaal [25]. The rationale behind supporting two different strategies is that the two represent a trade off between estimation time and precision. In WCA, the IPET strategy yields WCET estimates relatively fast, while the model-based strategy results in more precise estimates at the cost of a relatively long verification time. The precise WCET estimate is a consequence of the model representing the detailed behaviour of the system, especially the cache model.

Common to both WCET estimation strategies is the control-flow graph (CFG) of the application which is constructed by consulting the Java class files using the Byte Code Engineering Library [21]. For the IPET strategy, WCA transforms the CFG into an integer linear programming problem which is solved using the linear programming solver *lp_ solve* [70] resulting in a WCET estimate. In the model-based strategy, the CFG is directly transformed into timed automata models for Uppaal. Currently, WCET estimates using the model-based strategy are computed by making an initial guess of WCET (which can be based on the estimate derived using IPET). Afterwards, Uppaal verifies whether the timed automata are verifiable within the guessed time and, afterwards, the estimate is gradually refined using a binary search tactic.

For unbounded loops, WCA introduces comment-based annotations of source code which make explicit the iteration count of the particular loop. Alternatively, WCA provides the option of using data-flow analysis for extracting these. Obviously not all bounds can be extracted as part of static code analysis and in such cases the programmer needs to insert annotations. Furthermore, WCA performs receiver-type analysis to increase the precision of the WCETs in case of dynamic method dispatch.

Besides printing the resulting WCET estimate to standard output, WCA conveniently generates a detailed HTML report containing a visual representation of the CFG and timings of individual methods including their cache misses.

3.5 Schedulability Analysis

In order to ensure the correct functioning of an embedded real-time system, it is essential that all the tasks in the system meet their deadlines. In other words, the system must be *schedulable*. The schedulability of a system can be verified using techniques such as utilisation test, response time analysis, and model checking.

Below we describe the *TIMES* and *SARTS* tools tool for schedulability analysis.

TIMES is a model-based schedulability analysis tool [11]. That is, all provided information is transformed into timed automata on which the model checker Uppaal [25] is used. Schedulability is verified by checking if a location where a task misses its deadline is reachable in the model. The advantage of using TIMES, is that it allows a wide range of details of the system to be taken into account in the schedulability analysis. Among others, TIMES allow the programmer to specify if shared resources are used, and when they are locked and unlocked.

Tasks can be of one of three types, namely: sporadic, periodic, or controlled, where the releases of the sporadic and periodic are handled by TIMES, according to their release parameters. The release of the controlled tasks are controlled by release patterns modelled as timed automata. This is another detail that potentially increases the accuracy of the schedulability analysis, since it provides the means of describing the release of a task more precisely. To illustrate how release patterns can be modelled consider Figure 41. As shown, the pattern models a periodic release of *task* with a period of 60 time units.

TIMES models tasks in a real-time system by specifying the following three constraints:

Timing Constraints consist of a task's relative deadline and its WCET. Furthermore, it is possible to specify the type of the task to be one of

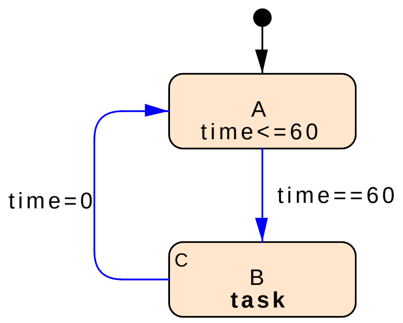


Figure 41: TIMES release pattern, illustrating how the release of a periodic task can be modelled.

the three supported. If the task is periodic or sporadic, its period or minimal inter-arrival time must be specified, respectively.

Precedence Constraints are used if the releases of tasks are dependent on each other, TIMES allows a precedence graph to be specified.

Resource Constraints take into account shared resources: These can be specified in terms of when they are locked and unlocked. The syntax for the constraint is $S_i(P_i, V_i)$ where S_i denotes the name of the resource, P_i denotes the accumulated execution time needed for the task to reach the critical section, and V_i denotes the accumulated execution time needed to exit the critical section.

When all the tasks of a system have been modelled, schedulability can be analysed. Basically, the result of this analysis is a verdict indicating whether deadlines are missed or not. However, if wanted more detailed information is available such as the Worst Case Response Times (WCRTs) for each of the tasks. Furthermore, TIMES allows the programmer to graphically follow the scheduling as a Gannt chart, as depicted in Figure 42. This representation is especially convenient for debugging since it shows precisely what goes wrong and where.

SARTS combines model-based WCET analysis with schedulability analysis [33]. Given a real-time system written in Java, SARTS translates each task in the system into a timed automaton, based on the Java bytecode.

Each timed automaton represents the control-flow of a task at the bytecode level, with timings for each instruction in the bytecode retrieved using information about the platform on which the code is executed, e.g. JOP. The model takes into account instructions that vary in time, for example, the overhead associated with a method call, based on the size of the method, is added to the model. For methods this overhead is known at compile-time, but other variations are modelled as a non-deterministic choice in the model.

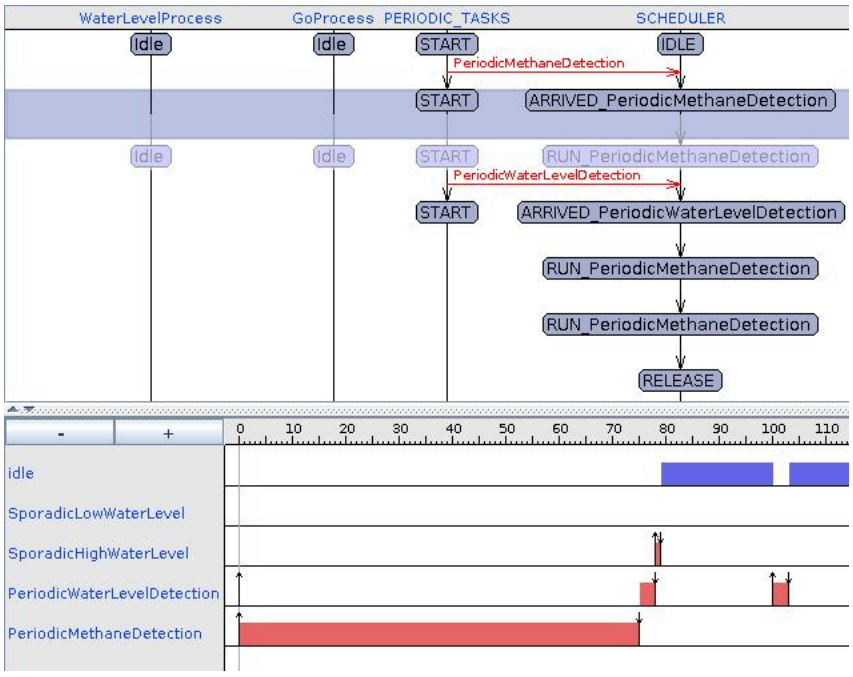


Figure 42: Example of a simulation run in TIMES.

For example, virtual method calls result in a non-deterministic choice among methods.

Task specific information, such as period, offset, and deadline, is also retrieved and used as parameters to a model of the scheduling strategy, i.e. fixed priority pre-emptive scheduling with priority assignments according to deadlines. For the schedulability analysis, the task models, along with a model of the scheduler, are composed in parallel. Using Uppaal the schedulability is verified by checking for deadlock freedom.

This approach is interesting because it considers interactions between tasks at a finer grained level than traditional analyses using plain WCET and worst case blocking. This is possible because the model includes enough information to possibly rule out interference between tasks, due to synchronized methods, where traditional approaches pessimistically include the worst case execution time of the critical section of any other task performing this synchronization. Another advantage of this approach is the tight correspondence between code and abstract model: it is certain that the system is actually an implementation of the abstract model being checked, and it does not rely on the developer having knowledge of timed automata.

4 Case Studies

This section introduces two case studies. The first case study is a smallish case study, based on the text-book example of a mine pump, structured for ease of analysis to test and evaluate profiles and tools. The second case study is a larger industrial case study based on a re-implementation of a C++ based control system for the FTIR infrared analysis technology for analysis of liquid samples developed by FOSS [63].

4.1 The Mine Pump

The first case study of a hard real-time system implemented in Java is based on the text-book example of a mine pump [45]. The purpose of the mine pump is to monitor a number of environmental properties in a mine to safely remove excess water using a water pump.

To focus on the essential functionality, a reduced version has been implemented. The reduced version centralises the various types of real-time tasks while omitting functionality that would only add to the size of the system. It consists of two environmental properties being monitored: the water level in the mine and the methane level. When the water level rises to a predetermined level, the water pump is started, and when the water level drops to another predetermined level, the water pump is stopped. The water pump must not run if the methane levels exceed safe levels. These functionalities have temporal requirements stating the reaction times of the system required for safe operation such as timely stopping the water pump whenever a critical level of methane is reached.

The actual prototype consists of two parts: the physical plant and the control software. Lego is used to construct the physical plant together with Lego NXT sensors and actuators connected to a JOP board. The control software comprises two periodic and two sporadic real-time tasks written in Java. The periodic tasks are responsible for monitoring the methane and water levels. The sporadic tasks are released whenever either the low or the high level has been reached.

An objective of this case study is to compare the SCJ and PJ real-time Java profiles. Objective evaluation criteria for ranking the profiles is a difficult undertaking and, hence, the following will solely present the different approaches for expressing fundamental concepts. Specifically, the following will show how the periodic task for monitoring the methane level is created.

Listing 5 shows the periodic event handler adhering to the SCJ profile.

```
PeriodicMethaneDetection methaneDetection =
     new PeriodicMethaneDetection(
       new PriorityParameters(METHANE_DETECTION_PRIORITY),
       new PeriodicParameters(
              new RelativeTime(0, 0),
                  new RelativeTime(PERIODIC_GAS_PERIOD, 0)),
       new StorageParameters(
                 SCOPED_MEMORY_BACKING_STORE_SIZE,
                 NATIVE_STACK_SIZE,
                 JAVA_STACK_SIZE),
       methaneSensor,
       waterpumpActuator);

methaneDetection.register();
```

Listing 5: An SCJ handler for methane level.

An SCJ periodic event handler has a number of parameters: since the SCJ profile level 1 uses an FPS scheduler, evidently a priority must be specified. Furthermore, a release parameter specifies the start time, the relative initial time for the first release of the handler, and a further relative time gives the period. An instance of `StorageParameters` expresses memory-related constraints for the handler. The objects `methaneSensor` and `waterpumpActuator` are interfaces to a sensor and an actuator. The sensor observes the current methane level and the actuator starts and stops the water pump. When a handler instance has been created, it is set for being scheduled when the `register()` method is invoked.

```
addToMission(new PeriodicMethaneDetection(
    new PriorityParameters(GAS_PRIORITY),
    new PeriodicParameters(new RelativeTime(0,0),
               new RelativeTime(GAS_PERIOD, 0)),
    Scheduler.getDefaultScheduler(),
    new LTMemory(MEMORY_SIZE),
    methaneSensor,
    waterPumpActuator));
```

Listing 6: The methane handler in the PJ profile.

The instantiation of a periodic handler in the PJ profile is similar to that of SCJ, and is shown in Listing 6. The only noticeable difference is the absence of `StorageParameters`, where PJ only requires a memory area with a given size. Further, the handler must be given the used scheduler as argument.

```
public void handleEvent() {
  waterpumpActuator.emergencyStop(
    methaneSensor.isCriticalMethaneLevelReached()
  );
}
```

Listing 7: Detecting the methane level.

Listing 7 shows the event handling method of the periodic event handler `PeriodicMethaneDetection`, implemented in the PJ profile. It is similar to the one in the SCJ profile.

To ensure that the control software adheres to its temporal requirements, schedulability has been analyzed with TIMES. Evidently, this analysis relies on the provision of WCET estimates for which WCA has been used.

WCA allows for a wide variety of configuration options including the Java processor used and architectural properties. It is of course of utmost importance that these configuration options are correctly set to reflect the actual system used. TIMES, on the other hand, is platform-agnostic and only relies on the scheduling algorithm used, temporal properties of the real-time tasks, and their release-patterns. To make the schedulability analysis more precise, the release patterns of the real-time threads have been modelled. Of particular interest is that the sporadic threads have been modelled to reflect that their release in this system cannot occur concurrently.

Since a shared resource is present in the control software, namely the water pump, TIMES requires WCET estimates before, after, and during the acquisition of the resource. WCA allows for easily conducting this process due to the provision of command-line options that lets the user specify the method of interest. Subsequently, the HTML reports generated by WCA can be consulted for extracting the needed information for addressing the presence of a shared resource.

By using WCA and TIMES together, the control software has successfully been verified to satisfy the temporal requirements, thus the system is schedulable on the JOP.

4.2 The FTIR Wine Scan Analyser

The FTIR (Fourier Transform Infrared) is an infrared analysis technology used by FOSS [63] for analysis of liquid samples such as milk and wine. Figure 43 shows the newest wine scan analysis instrument.

Figure 43: Wine Scan from FOSS.

The embedded FTIR system is normally implemented in C++, but FOSS was interested in finding out if a Real-Time Java solution was able to meet the timing constraints of the thirteen tasks defined in the FTIR specification, Table 7.

The most interesting is the Acquirement task. It makes the scanning of the sample. A complete measurement of a sample is composed of 32 scans

Table 7: Tasks in FTIR. T is the period, D the deadline.

Task	**T** (ms)	**D** (ms)
Acquirement	0.333	0.2
Temperature regulation (x4)	1000	100
Temperature reading (x4)	200	200
Monitoring	333	333
Output communication	333	333
Input Communication	500	500
Watchdog	8000	8000

where each scan has 3200 measurements. The results are kept in a 32 x 3200 matrix of type short. The period T of the task is 0.333 millisecs with a deadline D = 0.2 millisecs.

The implementation of the FTIR system is composed of two main parts:

1. The temperature controller module, which monitors and regulates a thermobox, - an isolated box in which the interferometer is placed. The temperatures are observed at four places in the thermobox and are regulated to remain within defined bounds.

2. The interferometer module makes measurements on the sample and creates an interferogram when the measurements have finished.

An overview of the acquirement states is shown in Figure 44, cf. Figure 3.2 in [19]. A Real-Time Java solution implemented in the Ravenscar-Java profile [91] running on an aJ-100 Java processor [144, 6] was described in details in [19] and concludes that the implementation meets the temporal correctness criteria.

In the Ravenscar-Java profile implementation, the two phases in the profile, the initialization phase and the mission phase, are utilized to implement the two main parts of the FTIR system.

A later re-implementation of the FTIR system to Safety Critical Java (SCJ) follows the same structure, but here the mission concept is utilized and the two phases described above are implemented as two missions: InitMission and AcquireMission. Because InitMission only monitors and regulates the thermobox, it includes temperature tasks only, whereas the AcquireMission includes all the thirteen tasks.

Furthermore, the ScopedMemory concept in SCJ is utilized to create objects which are local to a mission. An example is the large matrix for the measurement results. This matrix is used only in the AcquireMission by two tasks and is therefore created in the scoped memory of AcquireMission. In the Ravenscar-Java solution it had to be created in immortal memory. On the other hand, the temperature buffers which store the actual temperatures of

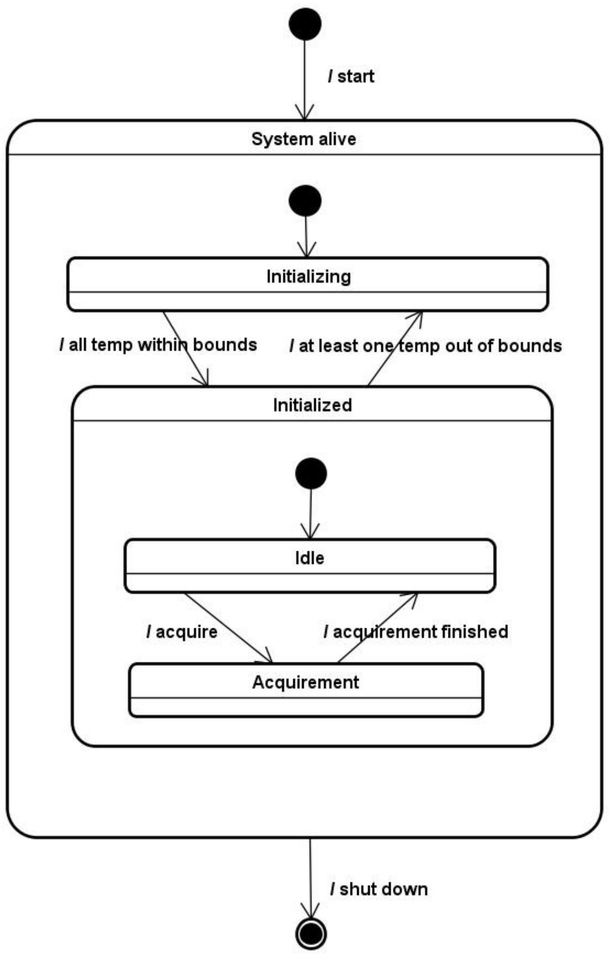


Figure 44: FTIR System States.

the thermobox, are in the SCJ solution created in immortal memory because the values in those buffers are used and updated in both missions, i.e. no change compared to the Ravenscar-Java solution.

Both the Ravenscar and the SCJ profiles are suitable for a Real-Time Java implementation of the logic of the FTIR specification. However, the SCJ profile gives some local options through the mission and the scoped memory concepts which are not present in the Ravenscar-Java profile.

A check of the FTIR code using the RT-Java tool (Section 3.1) shows e.g. that the Ravenscar-Java version uses block synchronization twice, but no cycles are found in the call graph.

A full scale industrial FTIR implementation would include connections to devices like temperature feelers, thermostates, and interferometer. The Ravenscar-Java solution, implemented on the aJ-100 Java processor, could be extended to a full FTIR solution using the JStik board from Systronix [148]. The SCJ implementation has been tested on an implementation running on top of Sun's Java RTS Linux version where these interfaces are left out. A port is underway to the HVM (Hardware near Virtual Machine) [87] which is a software virtual machine targeted at a variety of embedded platforms.

5 Related Work

Our tools should be seen in a larger context of other tools that can support the development of real-time Java programs. The following focuses on tools from a recent larger European project and commercially available tools.

The HIDOORS [161] project proposed an integrated development environment for Java embedded real-time systems. Its ground principle is that the environment must cover the full life-cycle of real-time systems development, meaning that it provides functionality ranging from a timing predictable JVM to a WCET tool. The environment uses the JamaicaVM from aicas, which is a JVM updated to provide time predictable behaviour. That is, JamaicaVM is extended with real-time garbage collection and supports the RTSJ specification. For the WCET analysis, HIDOORS suggests that the underlying hardware is modelled, such that caching and pipelining can be accounted for in the analysis. As part of this, the PAG [8] tool is used for data flow analyses.

The company aicas [5] has commerical tools for real-time Java systems development. Their JamaicaVM is supported by: The Jamaica Builder which is capable of building a single executable containing the Java application and determines the memory necessary to execute it, the VeriFlux tool which conducts static analysis in order to detect various errors and possible deadlocks in the application, and finally the Thread Monitor tool which allows simulation of the behaviour of the application in order to fine-tune applications.

Atego [13] provides a wide variety of tools and development environments for supporting safety-critical systems development targeting engineering sectors such as aerospace, defense, and the automotive industry. Among others, Atego offers different flavours of Aonix Perc which is a package containing virtual machine technology and accompanying tool chain for a variety of targets. One of these flavours is the Aonix Perc Raven package that focuses on a small and fast SCJ-compatible JVM that is amenable to cerification under stringent standards such as DO-178B Level A. Other flavours include Aonix Perc Ultra which is a Java Standard Edition (JSE) compatible JVM with toolchain.

Besides focusing on virtual machine technology, the offered products of Atego also comprise Artisan Studio which is Atego's modelling tool suite. The entire suite contains support for OMG: UPDM, SysML, and UML in a single toolset. The aim of Artisan Studio is to support development by offering different features such as visualisation to provide overviews of complex areas of embedded real-time software. When a model has been established, Artisan Studio provides functionality for automatically generating documentation and for testing the model for correctness and completeness with respect to defined requirements. Finally, an interesting feature is automated synchronisation of the design with the application code such that traceability is maintained. These industrial tools offer a much smoother integration than our experimental tools. However, as far as we can see, they do not use static analysis to the full extent which can be supported by WALA, and they do not use model checking for detailed analyses. Thus there is a potential for improving them based on our work.

6 Discussion and Further Work

The key hypothesis underlying our work is that Java is a promising candidate for a structured language which is well suited to develop hard real-time safety critical embedded software.

Java is by itself far too general to assist programmers in developing such applications, therefore specialized profiles have been developed as outlined above. Essentially the profiles define constructs that control utilization of platform resources like execution time and memory space. Furthermore they support development of programs that implement total functions without uncaught exceptions.

Development of truly predictable software cannot rely solely on trusting programmer specified resource constraints and believing that the implemented programs take care of all exceptional cases. The development process must include verification and validation. It is here that we have explored the potential for harnessing theories from a wide range of subject fields, such as static analysis and model checking, in tools that support validation.

Some of the tools are stand-alone tools, whereas other tools integrate more than one analysis, and yet other tools are already available as plug-ins for the Eclipse integrated development environment. Integration is extremely important, because to be really useful to ordinary programmers it is important that the tools are integrated well in the workbench that the programmer needs for developing, testing and managing code.

The case studies reported here gives some indication that the individual tools are by now so mature that they are useful. Yet, most tools for WCET analysis rely on programmer annotations for loop bounds. This is clearly not safe, and tools for checking correspondence between code and annotations are needed. In [120] the idea of dividing such annotation into two categories: trusted respectively verified annotations, is presented. Some tools like WCA offer (a bit of) automation that can implement the idea of verified annotations, but in general the only known safe approach is by theorem proving e.g.

using the Java Modeling Language and semiautomatic theorem provers like the KeY system [22]. This may also provide a bridge between tools for ensuring correct timing behaviour of embedded systems and the more general properties of ensuring functional correctness of the code.

In further work, especially in the Certifiable Java for Embedded Systems (cj4es) project[8], we intend to focus more specifically on the SCJ-profile and ensure that the tools cooperate with an Eclipse development environment as envisioned in Figure 40. The motto is specialization of the general theories to achieve thorough and yet efficient analyses of real applications.

A further significant challenge is to document the verdicts of the tools so they can feed into a certification of application systems. Extensive verification and validation of the tools themselves is probably out of the question, because they are too complex and under constant development.

A possible solution may be to develop simpler validators which take the verdicts and supporting information, for instance traces or reduced CFGs and checks validity of the verdicts, much in the spirit of proof carrying code [112]. For many of the complex tools a validator will be significantly simpler, recalling that although NP indicates complex searches for solutions using involved heuristics, it also means that solutions can be checked in polynomial time.

7 Acknowledgement

These ideas were first presented at the TTSS'10 workshop in Shanghai. They have been presented at the 6th Working Group Meeting in COST Action IC0701: Formal Verification of Object-Oriented Software, Aalborg, January 26-28, 2011. Comments and remarks from the audiences are gratefully acknowledged.

[8]`http://cj4es.imm.dtu.dk/`

Chapter 6

Bibliography

[1] Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems. Website, online: 27 June 2012, 1996. **shortened url:** `http://goo.gl/X0p8i` **original url:** `http://pbadupws.nrc.gov/docs/ML0634/ML063470583.pdf`.

[2] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[3] ABC News. Runaway Toyotas: The Investigation Continues. Website, online: 25 June 2012, Feb 2010. **shortened url:** `http://goo.gl/KtxDK` **original url:** `http://abcnews.go.com/Blotter/RunawayToyotas/runaway-toyotas-abc-news-investigation-continues/story?id=9810123#.T-i7bjlQqcF`.

[4] G. Agosta, S. C. Reghizzi, and G. Svelto. Jelatine: a virtual machine for small embedded systems. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, pages 170–177, New York, NY, USA, 2006. ACM.

[5] aicas — Realtime Java Technology. Website. Last accessed 8 April 2011.

[6] aJile Systems. Website. Last accessed 21 October 2011.

[7] M. Alrahmawy and A. Wellings. Design Patterns for Supporting RTSJ Component Models. In *Proc. of the 7th international workshop on Java technologies for real-time and embedded systems (JTRES'09)*. ACM Press, 2009.

[8] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *In Proceedings of the International Symposium on Static Analysis, (SAS'95)*, pages 33–50. Springer, 1995.

[9] R. Alur. Timed Automata. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 688–688. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48683-6_3.

[10] R. Alur, L. Fix, and T. Henzinger. A determinizable class of timed automata. In D. Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 1994.

[11] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. *Lecture Notes in Computer Science*, 2791:60–72, 2004.

[12] Aonix. Aonix Research and Development - Safety Critical Java Specification Initiative. `http://research.aonix.com/jsc/index.html`, 6 2009.

[13] Atego - Home - Atego. Website. Last accessed 8 April 2011.

[14] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 9 1993.

[15] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8:173–198, 1995. 10.1007/BF01094342.

[16] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[17] T. P. Baker and A. Shaw. The Cyclic Executive Model and Ada. *The Journal of Real-Time Systems*, 1:7–25, 1989. 10.1007/BF02341919.

[18] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[19] T. Baron, P. Jean, and G. Mercier. Design and implementation of a real-time embedded application. Master thesis, Aalborg University, Jan. 2007.

[20] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2007.

[21] BCEL. *Apache Software Foundation.* `http://jakarta.apache.org/bcel/`, 2006.

[22] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, Berlin, Heidelberg, 2007.

[23] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and L. Didier. UPPAAL-Tiga: time for playing games! In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 121–125, Berlin, Heidelberg, 2007. Springer-Verlag.

[24] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[25] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — A Tool Suite for Automatic Verification of Real-time Systems. *Hybrid Systems III*, pages 232–243, 1996.

[26] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a tool suite for automatic verification of real-time systems. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0020949.

[27] G. Bierman and M. Parkinson. Effects and effect inference for a core Java calculus. *Electronic Notes in Theoretical Computer Science*, 82(8):82–107, 2003. WOOD2003, Workshop on Object Oriented Developments (Satellite Event of ETAPS 2003).

[28] H.-J. Boehm. Destructors, finalizers, and synchronization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–272, New York, NY, USA, 2003. ACM.

[29] T. Bøgholm, R. R. Hansen, A. P. Ravn, B. Thomsen, and H. Søndergaard. A Predictable Java Profile: Rationale and Implementations. In M. T. Higuera-Toledano and M. Schoeberl, editors, *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2009, Madrid, Spain, September 23-25, 2009*, JTRES '09, pages 150–159, New York, NY, USA, 2009. ACM.

[30] T. Bøgholm, R. R. Hansen, A. P. Ravn, B. Thomsen, and H. Søndergaard. Schedulability analysis for java finalizers. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 1–7, New York, NY, USA, 2010. ACM.

[31] T. Bøgholm, H. Kragh-Hansen, and P. Olsen. Model-Based Schedulability Analysis of Real-Time Systems. Master's thesis, Aalborg University, 2008.

[32] T. Bøgholm, H. Kragh-Hansen, and P. Olsen. Real-Time Java. Technical report, Aalborg University, 2008.

[33] T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen. Model-based schedulability analysis of safety critical hard real-time Java programs. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, JTRES '08, pages 106–114, New York, NY, USA, 2008. ACM.

[34] T. Bøgholm, B. Thomsen, K. G. Larsen, and A. Mycroft. Schedulability analysis abstractions for safety critical java. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, pages 71–78, 2012.

[35] G. Bollella and J. Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, jun 2000.

[36] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java.* Java Series. Addison-Wesley, 2000.

[37] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. F. D. Hardin, M. Turnbull, R. Belliardi, D. Holmes, and A. Wellings. JSR001 Real-time Specification for Java, Version 1.0.2. Website, online: 25 June 2012, 2006. **shortened url:** `http://goo.gl/vA8K1` **original url:** `http://www.rtsj.org/specjavadoc/book_index.html`.

[38] A. Borg and A. Wellings. Towards an Understanding of the Expressive Power of the RTSJ Scoped Memory Model. In R. Meersman, Z. Tari, and A. Corsaro, editors, *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, volume 3292 of *Lecture Notes in Computer Science*, pages 315–332. Springer Berlin / Heidelberg, 2004.

[39] B. Bouyssounouse and J. Sifakis, editors. *Embedded Systems Design*, volume 3436 of *LNCS*. Springer Verlag, 2005. The ARTIST Roadmap for Research and Development.

[40] G. Brat and S. Park. Java PathFinder - Second Generation of a Java Model Checker. In *Proceedings of the Workshop on Advances in Verification*, 2000.

[41] P. Brémond-Grégoire, I. Lee, and R. Gerber. ACSR: An algebra of communicating shared resources with dense time and priorities. In E. Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-57208-2_29.

[42] E. J. Bruno and G. Bollella. *Real-Time Java Programming with Java RTS*. Prentice Hall, 2009.

[43] T. Budd. *Understanding Object-Oriented Programming with Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[44] A. Burns. The ravenscar profile. *Ada Lett.*, XIX:49–52, December 1999.

[45] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison Wesley Longmain, Boston, MA, USA, 4th edition, 2009.

[46] T. Bøgholm, C. Frost, R. Hansen, C. Jensen, K. Luckow, A. Ravn, H. Søndergaard, and B. Thomsen. Towards harnessing theories through tool support for hard real-time java programming. *Innovations in Systems and Software Engineering*, pages 1–12. 10.1007/s11334-012-0185-4.

[47] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In *Procöf 13th International Symposium on Formal Methods (FM'05)*, number 3582 in Lecture Notes in Computer Science, pages 91–106. Springer-Verlag, 2005.

[48] B. Carré and J. Garnsworthy. SPARK – an annotated Ada subset for safety-critical programming. In *Proceedings of the conference on TRI-ADA '90*, TRI-Ada '90, pages 392–402, New York, NY, USA, 1990. ACM.

[49] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[50] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *WCET*, pages 113–123, 2010.

[51] A. David, K. Larsen, A. Legay, U. Nyman, and A. Wąsowski. ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems. In A. Bouajjani and W.-N. Chin, editors, *Automated Technology for Verification and Analysis*, volume 6252 of *Lecture Notes in Computer Science*, pages 365–370. Springer Berlin / Heidelberg, 2010.

[52] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, HSCC '10, pages 91–100, New York, NY, USA, 2010. ACM.

[53] P. Dibble. The current status of the RTSJ and JSR 282. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, pages 1–1, New York, NY, USA, 2006. ACM.

[54] P. Dibble and A. Wellings. JSR-282 status report. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 179–182, New York, NY, USA, 2009. ACM.

[55] P. C. Dibble. *Real-Time Java Platform Programming.* BookSurge Publishing, second edition, 2008.

[56] B. Dobbing and A. Burns. The Ravenscar tasking profile for high integrity real-time programs. *Ada Lett.*, XVIII(6):1–6, Nov. 1998.

[57] A. Easwaran, I. Lee, I. Shin, and O. Sokolsky. Compositional Schedulability Analysis of Hierarchical Real-Time Systems. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, ISORC '07, pages 274–281, Washington, DC, USA, 2007. IEEE Computer Society.

[58] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New Developments in WCET Analysis. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation. Theory and Practice. Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *LNCS*, pages 12–52. Springer Verlag, 2007.

[59] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theor. Comput. Sci.*, 354(2):301–317, 2006.

[60] E. Fersman and W. Yi. A generic approach to schedulability analysis of real-time tasks. *Nordic J. of Computing*, 11(2):129–147, 2004.

[61] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232. ACM, 2000.

[62] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. *SIGPLAN Not.*, 37:234–245, May 2002.

[63] FOSS. Website. Last accessed 21 October 2011.

[64] M. Fowler. *Refactoring: improving the design of existing code.* Object Technology Series. Addison-Wesley, 1999.

[65] C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen. WCET analysis of Java bytecode featuring common execution environments. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 30–39, New York, NY, USA, 2011. ACM.

[66] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. *SIGPLAN Not.*, 45:299–312, January 2010.

[67] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *Proc. of the International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'05*, volume 3385 of *Lecture Notes in Computer Science*, Paris, France, Jan. 2005. Springer Verlag.

[68] P. Gerum. The Xenomai Project. Implementing a RTOS emulation framework on GNU/Linux. In *Third Real-Time Linux Workshop*, 2001.

[69] P. Giambiagi and G. Schneider. Memory consumption analysis of Java smart Card. In *In Proc. of Conferencia Latinoamericana de Informatica (Latin American Computing Conference), CLEI'05*, Santiago de Cali, Columbia, Oct. 2005.

[70] H. Gourvest, W. Pattton, P. Notebaert, M. Berkelaar, K. Eikland, and J. Dirks. lp_solve reference guide, 2010. Last accessed 8 March 2011.

[71] G. Haddad, F. Hussain, and G. T. Leavens. The design of SafeJML, a specification language for SCJ with support for WCET specification. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 155–163, New York, NY, USA, 2010. ACM.

[72] G. Haddad and G. T. Leavens. Specifying subtypes in SCJ programs. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 40–46, New York, NY, USA, 2011. ACM.

[73] R. R. Hansen. A Hardest Attacker for Leaking References. In D. Schmidt, editor, *Proc. of European Symposium on Programming, ESOP'04*, volume 2986 of *Lecture Notes in Computer Science*, pages 310–324, Barcelona, Spain, Mar./Apr. 2004. Springer Verlag.

[74] R. R. Hansen and C. W. Probst. Non-Interference and Erasure Policies for JavaCard Bytecode. In *Workshop on Issues in the Theory of Security, WITS'06*, pages 174–189, Vienna, Austria, Mar. 2006.

[75] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad. Toward Libraries for Real-Time Java. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:458–462, 2008.

[76] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for Safety-Critical Applications. *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, Mar. 2009.

[77] T. Henzinger and S. Matic. An Interface Algebra for Real-Time Components. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 253–266, april 2006.

[78] M. Higuera-Toledano. Towards an understanding of the behavior of the single parent rule in the RTSJ scoped memory model. In *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 470–479, march 2005.

[79] B. Huber and M. Schoeberl. Comparison of Implicit Path Enumeration and Model Checking Based WCET Analysis. In N. Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany.

[80] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *Proc. of the 10th International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 132–149. Springer-Verlag, 2008.

[81] L. Hubert and D. Pichardie. Soundly Handling Static Fields: Issues, Semantics and Analysis. *Proc. of the 4th International Work-*

shop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09), ENTCS, 2009. To appear.

[82] J. J. Hunt, F. B. Siebert, P. H. Schmitt, and I. Tonin. Provably correct loops bounds for realtime Java programs. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, pages 162–169, New York, NY, USA, 2006. ACM.

[83] IBM WebSphere Real Time. Website. Last accessed 12 April 2011.

[84] J. Ganssle, EE Times. Total Recall. Website, online: 25 June 2012, 2006. **shortened url:** `http://goo.gl/Vc2mM` **original url:** `http://www.eetimes.com/discussion/embedded-pulse/4025634/Total-Recall`.

[85] JamVM. JamVM 1.5.3. `http://jamvm.sourceforge.net/`, 4 2009.

[86] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications (The Springer International Series in Engineering and Computer Science)*. Springer, 1997.

[87] S. Korsholm. HVM (Hardware near Virtual Machine). Last accessed 21 October 2011.

[88] S. Korsholm. Flash memory in embedded Java programs. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 116–124, New York, NY, USA, 2011. ACM.

[89] S. Korsholm, M. Schoeberl, and A. P. Ravn. Interrupt Handlers in Java. In *11th IEEE International Symposium on Oriented Real-Time Distributed Computing (ISORC)*, pages 453–457. IEEE Computer Society Press, 2008.

[90] P. Krcal and W. Yi. Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata. *Lecture Notes in Computer Science*, Volume 2988/2004:236–250, 2004.

[91] J. Kwon, A. Wellings, and S. King. Ravenscar-Java: A High Integrity Profile for Real-time Java. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, JGI '02, pages 131–140, New York, NY, USA, 2002. ACM, ACM.

[92] G. Le Lann. An analysis of the Ariane 5 flight 501 failure-a system engineering perspective. In *Engineering of Computer-Based Systems, 1997. Proceedings., International Conference and Workshop on*, pages 339–346, mar 1997.

[93] G. T. Leavens, A. L. Baker, and C. Ruby. Jml: a java modeling language. In *In Formal Underpinnings of Java Workshop (at OOPSLA'98*, 1998.

[94] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java Modeling Language. In *In Formal Underpinnings of Java Workshop (at OOPSLA'98)*, 1998.

[95] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[96] N. Leveson and C. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, jul 1993.

[97] Y. Li and S. Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. *ACM SIGPLAN Notices*, 30(11):88–98, 1995.

[98] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Prentice Hall, 2 edition, 1999.

[99] G. Lindstrom, P. C. Mehlitz, and W. Visser. Model Checking Real Time Java Using Java PathFinder. In *ATVA*, pages 444–456, 2005.

[100] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20:46–61, January 1973.

[101] Z. Liu, V. Mencl, A. P. Ravn, and L. Yang. Harnessing Theories for Tool Support. In *International Symposium on Leveraging Applications of Formal Methods, ISoLA 2006*, pages 371–382. IEEE, 2006.

[102] C. D. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4:37–53, 1992. 10.1007/BF00365463.

[103] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. Safety Critical Java Technology Specification. The Open Group, Latest version, 0.79, Early Access, 16 May 2011. `http://download.oracle.com/otndocs/jcp/safety_critical-0.78-edr-oth-JSpec`.

[104] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. Safety Critical Java Technology Specification. The Open Group, Version 0.5, Early Access, August 2008.

[105] K. S. Luckow, B. Thomsen, and S. Korsholm. Towards a Real-Time, WCET Analysable JVM Running in 256kB of Flash Memory. In *Proceedings of the 23rd Nordic Workshop on Programming Theory*, NWPT '11, pages 86–88, 2011.

[106] M. Barr, EE Times. Real men program in C. Website, online: 27 June 2012, 2009. **shortened url:** `http://goo.gl/QOdHI` **original url:** `http://www.eetimes.com/electronics-blogs/industrial-control-designline-blog/4027479/Real-men-program-in-C`.

[107] Martin and Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1-2):265–286, 2008.

[108] L. Mathiassen. *Object-oriented Analysis & Design.* Marko Publishing, 2000.

[109] A. Metzner. Why Model Checking Can Improve WCET Analysis. In R. Alur and D. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 298–301. Springer Berlin / Heidelberg, 2004.

[110] M. Mohammad and V. Alagar. TADL - An Architecture Description Language for Trustworthy Component-Based Systems. In *Proceedings of the 2nd European conference on Software Architecture*, ECSA '08, pages 290–297, Berlin, Heidelberg, 2008. Springer-Verlag.

[111] N. Lewis, EE Times. Behind Toyota's Software Recall. Website, online: 25 June 2012, 2005. **shortened url:** `http://goo.gl/FjxIM` **original url:** `http://www.eetimes.com/electronics-news/4182705/Behind-Toyota-s-Software-Recall`.

[112] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[113] F. Nielson and H. Nielson. From CML to process algebras. In E. Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 493–508. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-57208-2_34.

[114] F. Nielson and H. Nielson. Type and Effect Systems. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer Berlin / Heidelberg, 1999.

[115] The Open Group. *Safety Critical Java Technology Specification (JSR-302) - Draft*, 2010.

[116] Oracle. *RTSJ 1.1 Alpha 6, release notes*, 2009.

[117] G. Phipps. Comparing observed bug and productivity rates for Java and C++. *Softw. Pract. Exper.*, 29:345–358, April 1999.

[118] F. Pizlo, J. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: design patterns and semantics. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pages 101–110, may 2004.

[119] F. Pizlo and J. Vitek. An Emprical Evaluation of Memory Management Alternatives for Real-Time Java. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 35–46, dec. 2006.

[120] A. Prantl, J. Knoop, R. Kirner, A. Kadlec, and M. Schordan. FROM TRUSTED ANNOTATIONS TO VERIFIED KNOWLEDGE.

[121] W. Puffitsch, B. Huber, and M. Schoeberl. Worst-case analysis of heap allocations. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part II*, ISoLA'10, pages 464–478, Berlin, Heidelberg, 2010. Springer-Verlag.

[122] P. Puschner and A. Wellings. A Profile for High-Integrity Real-Time Java Programs. In *4th IEEE International Symposium on Oriented Real-Time Distributed Computing (ISORC)*, pages 15–22. IEEE Computer Society, 2001.

[123] J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[124] D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan. UNIX Time-Sharing System: The C Programming Language. *Bell Sys. Tech. J.*, 57(6):1991–2019, 1978.

[125] RTCA. Software considerations in airborne systems and equipment certification. DO-178B, RTCA Inc., Wasington, D.C., 1992.

[126] RTCA & European Organisation for Civil Aviation Equipment. Software considerations in airborne systems and equipment certification. ED-12B, EUROCAE, Paris, 1992.

[127] RTSJ.org. *RTSJ 1.0.2*, 2010. Last accessed 25 February 2011.

[128] M. Schoeberl. JOP: A Java Optimized Processor. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889 of *LNCS*, pages 346–359, Catania, Italy, November 2003. Springer Verlag.

[129] M. Schoeberl. Restrictions of Java for embedded real-time systems. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pages 93–100, may 2004.

[130] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems.* PhD thesis, Vienna University of Technology, 2005.

[131] M. Schoeberl. *JOP Reference Handbook.* `http://jopdesign.com/doc/handbook.pdf`, 2007.

[132] M. Schoeberl. A Java Processor Architecture for Embedded Real-Time Systems. volume 54, pages 265–286, New York, NY, USA, January 2008. Elsevier North-Holland, Inc.

[133] M. Schoeberl, S. Korsholm, T. Kalibera, and A. P. Ravn. A Hardware Abstraction Layer in Java. *ACM Trans. Embed. Comput. Syst.*, 10(4):42:1–42:40, Nov. 2011.

[134] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, pages 202–211, New York, NY, USA, 2006. ACM.

[135] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case Execution Time Analysis for a Java Processor. *Software: Practice and Experience*, 40(6):507–542, 2010.

[136] M. Schoeberl, P. Puschner, and R. Kirner. A Single-Path Chip-Multiprocessor System. In *Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, SEUS '09, pages 47–57, Berlin, Heidelberg, 2009. Springer-Verlag.

[137] M. Schoeberl, H. Søndergaard, B. Thomsen, and A. P. Ravn. A Profile for Safety Critical Java. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 94–101, Washington, DC, USA, May 2007. IEEE Computer Sciety Press.

[138] M. Schoeberl, C. Thalinger, S. Korsholm, and A. P. Ravn. Hardware Objects for Java. In *11th IEEE International Symposium on Oriented Real-Time Distributed Computing (ISORC)*, pages 445–452. IEEE Computer Sciety Press, 2008.

[139] M. Schoeberl, C. Thalinger, S. Korsholm, and A. P. Ravn. Hardware Objects for Java. In *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 445–452, Washington, DC, USA, 2008. IEEE Computer Society.

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

[141] W. Sheng, Y. Gao, L. Xi, and X. Zhou. Schedulability Analysis for MultiCore Global Scheduling with Model Checking. In *Microprocessor Test and Verification (MTV), 2010 11th International Workshop on*, pages 21–26, dec. 2010.

[142] F. Siebert. Jeopard: Java environment for parallel real-time development. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, JTRES '08, pages 87–93, New York, NY, USA, 2008. ACM.

[143] H. Søndergaard, A. P. Ravn, B. Thomsen, and M. Scoeberl. A Practical Approach to Mode Change in Real-Time Systems. Technical Report 08-001, Department of Computer Science, Aalborg University, 2008.

[144] H. Søndergaard, B. Thomsen, and A. P. Ravn. A Ravenscar-Java profile implementation. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 38–47, New York, NY, USA, 2006. ACM.

[145] H. Sondergaard, B. Thomsen, A. P. Ravn, R. R. Hansen, and T. Bogholm. Refactoring real-time java profiles. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, pages 109–116, 2011.

[146] Sun Java Real-Time System. Website. Last accessed 6 April 2011.

[147] Sun Microsystems. *RTSJ 1.0, release notes*, 2006.

[148] JStik Systronix. Website. Last accessed 21 October 2011.

[149] J. TALPIN, E. GAMATI, B. L. DEZ, D. BERNER, and P. L. GUERNIC. Hard real-time implementation of embedded software in JAVA, 2003.

[150] J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *J. Funct. Program.*, 2(3):245–271, 1992.

[151] The Age. Computer glitch traps Thai minister in BMW. Website, online: 25 June 2012, 2003. **shortened url:** `http://goo.gl/1yFLH` **original url:** `http://www.smh.com.au/articles/2003/05/12/1052591731421.html`.

[152] TheOpenGroup. JSR 302: Safety Critical Java Technology. `http://jcp.org/en/jsr/detail?id=302`, 2006.

[153] TheOpenGroup. Safety Critical Specification for Java. Draft Version 0.77, TheOpenGroup, July 2010.

[154] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics. In *Proceedings of the International Performance and Dependability Symposium (IPDS)*, pages 625–632. IEEE Computer Society Press, June 2003.

[155] I. Thomm, M. Stilkerich, C. Wawersich, and W. Schröder-Preikschat. KESO: an open-source multi-JVM for deeply embedded systems. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 109–119, New York, NY, USA, 2010. ACM.

[156] B. Thomsen. Polymorphic Sorts and Types for Concurrent Functional Programs. In *In John Glauert (Ed.): Proceedings of the 6th International Workshop on the Implementation of Functional Languages. University of East Anglia*, 1994.

[157] L. L. Thomsen, B. Thomsen, and K. Nørmark. Computational Abstraction Steps. *Journal of Object Technology*, 9(6):1–23, Nov. 2010.

[158] C. Thrane and U. Sorensen. Slicing for Uppaal. In *Student Paper, 2008 Annual IEEE Conference*, pages 1–5, feb. 2008.

[159] Timesys. RTSJ Reference Implementation (RI) and Technology Compatibility Kit (TCK). `http://www.timesys.com/java/`, 2009.

[160] V. T. Vasconcelos, S. J. Gay, A. Ravara, N. Gesbert, and A. Z. Caldeira. Dynamic interfaces. In *In FOOL'09*, 2009.

[161] J. Ventura, F. Siebert, A. Walter, and J. J. Hunt. HIDOORS-A High Integrity Distributed Deterministic Java Environment. In *IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 113–118, 2002.

[162] T.J.Watson Libraries for Analysis (WALA). Website. Last accessed 1 April 2011.

[163] A. Wellings. *Concurrent and Real-Time Programming in Java.* John Wiley & Sons, Ltd, 2004.

[164] A. Wellings and M. Kim. Asynchronous event handling and Safety Critical Java. In *JTRES 2010: Proceedings of the 8th international workshop on Java technologies for real-time and embedded systems*, pages 53–62, New York, NY, USA, 2010. ACM.

[165] A. J. Wellings and M. S. Kim. Processing group parameters in the real-time specification for Java. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 3–9, New York, NY, USA, 2008. ACM.

[166] R. Wilhelm. Determining Bounds on Execution Times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1,14–23. CRC Press, 2005.

[167] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7:36:1–36:53, May 2008.

[168] Xenomai. Xenomai Native Skin API Reference Manual, 2006.

Chapter 7

Additional Papers

Paper G:

Model-based Analysis of Embedded Java Programs

Thomas Bøgholm, Anders P. Ravn, and Bent Thomsen

Department of Computer Science
Aalborg University, Denmark

Abstract

The research topic presented here is model based analysis of Java programs for embedded applications. The aim is to combine model-checking and program analysis techniques, such that model checking takes care of dynamic features e.g. caching and program analysis provide suitable abstractions of program blocks and control flow. In this paper we present initial results, a case, study and future directions.

7.1 Introduction

In the area of embedded systems, Java can be considered a suitable replacement for traditional programming languages, such as C and assembler, since the complexity of such systems continue to increase. Java provides abstractions, relieving the programmer from worries about the low-level and error prone details, such as memory management e.g. (de)allocation, pointers, etc., and hardware specific details.

The use of Java, however, complicates program analyses, such as the required *schedulability analysis* and analysis of *memory usage*. This is partly due to the layered architecture: Java Virtual Machine (JVM), operating system and underlying hardware, and partly the use of garbage collection for memory management. Additionally, applying such analyses on systems for multi-core architectures still is an open research area [142].

The use of Java for embedded systems thus calls for development in two interdependent areas:

Programming Language Profiles: Since standard Java is unsuitable for dependable systems development because of the dynamic nature and unpredictable execution times, caused by e.g. garbage collection, dynamic class-loading, and unspecified scheduling scheme, a safe subset of the language should be provided. Though much work has been done in this area, room for improvement still exist. Part of this research is focused on developing a predictable Java-profile, which smoothly interacts with analysis tools.

Platform Dependent Analyses: As with programming languages, much work has been done in the development of program analysis techniques, some especially suited for real-time applications. Standard schedulability techniques require the computation of WCET for each task in the system, and provide pessimistic, but safe, answers [45]. Abstract interpretation, static analysis and model-checking techniques provide general approaches to program analyses for ensuring program correctness with regards to certain properties, such as *null-pointers*, *array index errors* etc. The goal in this project is to develop new techniques combining these approaches in developing a single tool aimed at the aforementioned Java-profile, using model-checking, aided by suitable analysis techniques.

7.2 Initial Experiment

In an initial effort in this project [33], a tool named SARTS, Schedulability Analyzer for Real-Time Systems, was developed.

The SARTS tool analyzes Java programs and constructs control flow graphs decorated with timing information in the form of timed automata for the model checker UPPAAL. UPPAAL can then be used to perform schedulability analysis, by verifying the absence of deadlocks in the timed automata model.

For Java programs to be analyzed using SARTS, these programs must be written in the Java profile, SCJ [137], a profile providing a simple analyzable framework for safety-critical embedded Java systems, intended to be executed on the time-predictable Java processor JOP [130]. In this framework, a real-time system is a three phase system, *initialization*, *mission*, and *shutdown*, with appertaining sporadic and periodic tasks. The initialization phase is a non-critical phase setting up the system before execution, a phase not considered in the schedulability analysis. The mission phase is the safety-critical execution phase where periodic and sporadic tasks are released. The shutdown phase performs a controlled shutdown of a running system.

The two task classes, periodic and sporadic task, each contains task specific parameters used in the analysis: for periodic tasks, these are offset, period, deadline, and for sporadic tasks: minimum inter-arrival time and deadline. A task implementation contains implementation of a Java method, *run*, invoked when the task is released.

This framework structure is reflected in the generated UPPAAL model, in which standard templates are created for periodic and sporadic tasks, and a template for the scheduler; a deadline monotonic scheduling strategy using a priority ceiling protocol.

For each method in the Java program, including the task run-methods, a template is generated, representing control flow at the byte-code level, decorated with execution time for each byte-code instruction, information about blocking, preemption and method invocation. These templates act as building blocks in the final UPPAAL model and linked together using synchronization channels. The resulting model follows this predefined pattern:

- One scheduler instance
- Task controllers corresponding each task in the Java program, linked with their corresponding *run* methods
- N method models for each Java method, where N is the total number of tasks in the Java program; this is to allow concurrent execution of each method from different tasks

This results in a single model with $1 + T + (T * M)$ timed automata run in parallel, where T is the number of tasks and M is the number of methods in the Java program. Method invocation is performed through channel synchronization, blocking time is modeled by preventing preemption in the scheduler, preemption is performed using *stopwatches*, and the execution time for each task is tracked using *clocks*. In the case of a task deadline overrun, the failing task will cause a deadlock in the model; hence, the schedulability property of the Java program is verified in Uppaal by verifying the absence of deadlocks.

SARTS has been applied successfully on a case study, a sorting machine built in lego, controlled by the JOP processor and a Java control program consisting of two periodic and two sporadic tasks, amounting to an approximate 300 lines of code. Using *convex hull* approximation, a safe over approximation, this system has been verified in approximately 37 seconds using Uppaal on standard hardware.

An outline of future work includes further development of the approach using static analysis to improve analysis result and performance, the development of further analyses such as memory consumption, cache behavior, multi-core scheduling and finally the integration of these techniques in a single tool and possibly integration in an IDE, e.g. Eclipse, in order to make the techniques available to developers.

Using static analysis techniques to reduce the non-determinism in the model applies not only to the schedulability analysis technique, but could also further improve analyses such as memory consumption. More precise modeling of the platform should include e.g. caching behavior, such that the analysis result become more accurate.

The approach taken in SARTS, i.e. the translation of Java programs to Uppaal models is believed to be applicable to memory consumption analysis, by modeling the memory allocation scheme of the Java profile. Tracking object creation in the model is deemed a promising approach in being able to answer questions about worst case memory consumption. This technique applied to the analysis of schedulability in a multi-core setting is also considered an interesting future development.

Further development of Java profiles includes defining annotations to aid the analyses. As of now, loop bounds must be specified by the developer in the form of simple comments containing a constant iteration count for a each loop. Since supplying correct loop bounds is the responsibility of the developer, this is a potential source of errors. Improvements in this area includes automatic loop bound verification inspired by the techniques presented in [82]. Additionally, more expressive loop bound notations could be introduced, since cases exists where an upper bound cannot be determined accurately local to the loop e.g. a list sort function contain loop bounds which cannot be determined locally, since they depend on the size of the list in question.

Finally, in integrating these techniques in an IDE, it may be possible to visualize the analysis results. In the case schedulability, code locations with heavy execution times may be highlighted, or in the case of loop bound detection, complicated loop expressions may be identified.

7.3 Acknowledgements

We would like to thank Henrik Kragh-Hansen, Kim Guldstrand Larsen and Petur Olsen for their efforts during the initial work of this research project [33].

Paper H:

Formal Modelling and Analysis of Predictable Java

Thomas Bøgholm[1], René R. Hansen[1], Anders P. Ravn[1], Hans Søndergaard[2], and Bent Thomsen[1]

[1] *Department of Computer Science*
Aalborg University, Denmark

[2] *VIA University College*
Horsens, Denmark

This is *not* a peer reviewed publication

The full article can be found in: ERCIM NEWS - Number 81, April 2012
European Research Consortium for Informatics and Mathematics
www.ercim.eu

Special theme: Computational Science / Scientific Computing
Simulation and Modelling for Research and Industry
Pages 52-53
http://ercim-news.ercim.eu/images/stories/EN81/EN81-web.pdf

While embedded systems development is becoming increasingly complex due to demands for greater functionality and a shorter time to market, it is still using low-level, close to hardware, implementation languages. Modern languages like Java handle such complexities more elegantly, but issues with predictability hinder their adaptation in the embedded systems world. We have developed Predictable Java, a Java-based framework for safety-critical embedded systems, along with analysis tools based on formal modelling.

Embedded systems are everywhere, and are increasingly affecting our everyday life. Demands for shorter time to market, greater functionality and lower costs makes it difficult to develop these systems using traditional programming languages. Years ago, the traditional software industry shifted from low-level languages to modern languages with great success, increasing productivity significantly. Standard Java, however, is difficult to deploy and analyse in a real-time system setting. Java is based on a rather complex virtual machine, making analysis hard, and being object oriented, it has a very dynamic behaviour with almost unpredictable memory and time consumption. Additionally, Java is based on garbage collected memory, causing unforeseen interruptions which further complicates analyses.

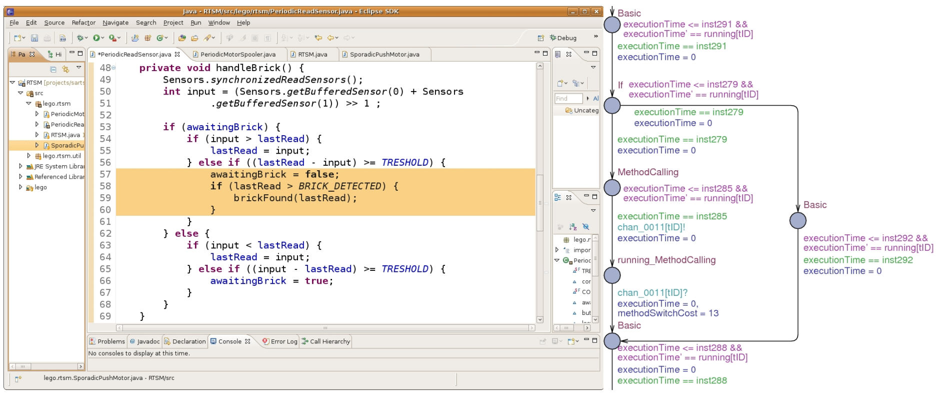
We suggest that modern software engineering practices and languages be used in embedded systems software development by providing the required technology. The framework based on our Predictable Java profile provides the ability to express very complex systems in a simple, understandable and clear language which is recognizable and easily adaptable by most Java programmers.

Our tool SARTS (schedulability analsyser for real-time systems) is a schedulability analysis tool based on a formal model of timed automata, for proving schedulability of systems developed using our framework. This work is funded by DaNES (Danish Network for Intelligent Embedded Systems) and CISS (Center for Embedded Software Systems) at Aalborg University, and is publicly available.

In an attempt to reach the closest correspondence between actual running code and the properties being verified, SARTS works directly on the actual executed bytecode. This allows for a very close correspondence between the running system and the verified properties. For our prototype, we use the time-predictable Java processor, JOP, developed by Martin Schöberl at the Vienna University of Technology. This is an open and well-documented processor designed with predictability in mind.

The bytecode from the Java compiler is converted into a formal model, representing the real system, which is analysed in UPPAAL: a tool for modelling and verification of real-time systems.This approach provides a fully automatic process from code to verdict; developers need no training in formal verification, timed automata or UPPAAL.

Our technique provides a tight analysis by taking into consideration the control flow in the program code. This allows for analysing blocking, pre-



```
private void handleBrick() {
    Sensors.synchronizedReadSensors();
    int input = (Sensors.getBufferedSensor(0) + Sensors
            .getBufferedSensor(1)) >> 1 ;

    if (awaitingBrick) {
        if (input > lastRead) {
            lastRead = input;
        } else if ((lastRead - input) >= TRESHOLD) {
            awaitingBrick = false;
            if (lastRead > BRICK_DETECTED) {
                brickFound(lastRead);
            }
        }
    } else {
        if (input < lastRead) {
            lastRead = input;
        } else if ((input - lastRead) >= TRESHOLD) {
            awaitingBrick = true;
        }
    }
}
```

No consoles to display at this time.

Java - RTSM/src/lego/rtsm/PeriodicReadSensor.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
*PeriodicReadSensor.java
PeriodicMotorSpooler.java
RTSM.java
SporadicPushMotor.java
Problems Javadoc Declaration Console Error Log Call Hierarchy
lego.rtsm.SporadicPushMotor.java - RTSM/src
Basic
executionTime <= inst291 && executionTime' == running[tID]
executionTime == inst291
executionTime = 0
If executionTime <= inst279 && executionTime' == running[tID]
executionTime == inst279
executionTime = 0
executionTime == inst279
executionTime = 0
MethodCalling
executionTime <= inst285 && executionTime' == running[tID]
executionTime == inst285
chan_0011[tID]!
executionTime = 0
running_MethodCalling
chan_0011[tID]?
executionTime = 0,
methodSwitchCost = 13
Basic
executionTime <= inst288 && executionTime' == running[tID]
executionTime = 0
executionTime == inst288
Basic
executionTime <= inst292 && executionTime' == running[tID]
executionTime == inst292
executionTime = 0

Figure 7.1: (left) A small snippet of a Java code verified using SARTS; (right) a simplified UPPAAL model of the highlighted code.

emption and many of the dynamic features available in Java, which are considered hard to analyse, such as finalizers and dynamic method dispatch.

The Predictable Java framework provides a target for developing other analyses. We are designing the language profile as well as the analyses for the language profile. This enables us to fine-tune the developer framework in order to reach a reasonable compromise between restrictions and analyzability, making Java suited for safety-critical embedded systems.

Inspired by the techniques applied in SARTS, we are also developing memory consumption analysis, along with Eclipse IDE plugins that will assist programmers using the profile and profile conformance checker.

Links:
SARTS and Predictable Java: http://pj.cs.aau.dk
DaNES: http://danes.aau.dk
CISS: http://www.ciss.dk
UPPAAL tool: http://uppaal.com

Please contact:
Bent Thomsen
Aalborg University / DANAIM, Denmark
Tel: +45 9940 8897
E-mail: bt@cs.aau.dk