Noname manuscript No.
(will be inserted by the editor)

Harnessing Theories for Tool Support

A Real-Time Java Case

Thomas Bøgholm · Christian Frost · René Rydhof Hansen · Casper Svenning Jensen · Kasper Søe Luckow · Anders P. Ravn · Hans Søndergaard · Bent Thomsen

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.

Keywords Java · real-time systems · safety-critical · static analysis · model checking

1 Introduction

For systems that have to meet strict timing or safety requirements it is often argued [?] 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 [?] 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.

C.Frost, C.S.Jensen, K.S.Luckow,
T.Bøgholm, R.R.Hansen, A.P.Ravn, B.Thomsen
Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, DK-9220, Aalborg East, Denmark.
E-mail: {cfrost09,csje09,klucko09}@student.aau.dk
E-mail: {boegholm,rrh,apr,bt}@cs.aau.dk

H.Søndergaard
VIA University College, Chr. M. Oestergaards Vej 4, DK-8700 Horsens, Denmark. E-mail: hso@viauc.dk

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.

Tool use is illustrated by results from a case study in Section 4. It is the mine pump example well known from the literature. As explained in connection with the case study, the experiments have been encouraging. 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) [?] 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 [?] and is currently being revised to RTSJ 1.1 in JSR 282 [?].

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 [?], and several RTSJ implementations using these algorithms are available today, such as Java RTS from Oracle/Sun Microsystems [?], WebSphere Real Time from IBM [?], and JamaicaVM from Aicas [?].

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 Safety Critical Java (SCJ) profile, developed under JSR-302 [?] is one of these. The 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 1.

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.

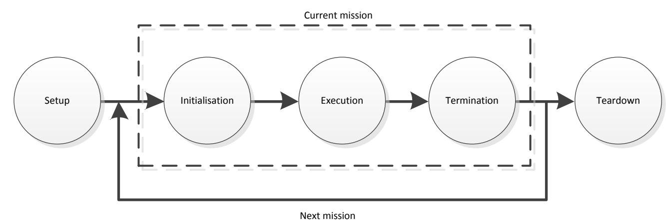

Fig. 1 The phases involved in a real-time application's lifetime.

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 of one mission and must follow the programming model needed for a Cyclic Executive Scheduled (CES) application. The mission can therefore be thought of as a set of periodic tasks placed in a schedule on a time-line. The schedule must either be constructed manually by the programmer or by an offline tool.

Level 1 applications consist of a single mission. However, 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 [**?**] 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 recognises that missions may in fact be more than simple containers due to the inclusion of initialisation and termination phases of these. These phases can themselves be controlled by handlers, thus, the PJ proposes that missions are handlers for the respective phases.

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. A vision for the kind of tool support is shown in Figure 2. 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 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 [**?**].

Furthermore, 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. This is deemd to be sufficiently precise for the often conservative programming style employed in safety-critical systems.

3.2 Exception Analysis

An uncaught exception is highly undesirable in most applications. In an embedded real-time system it may

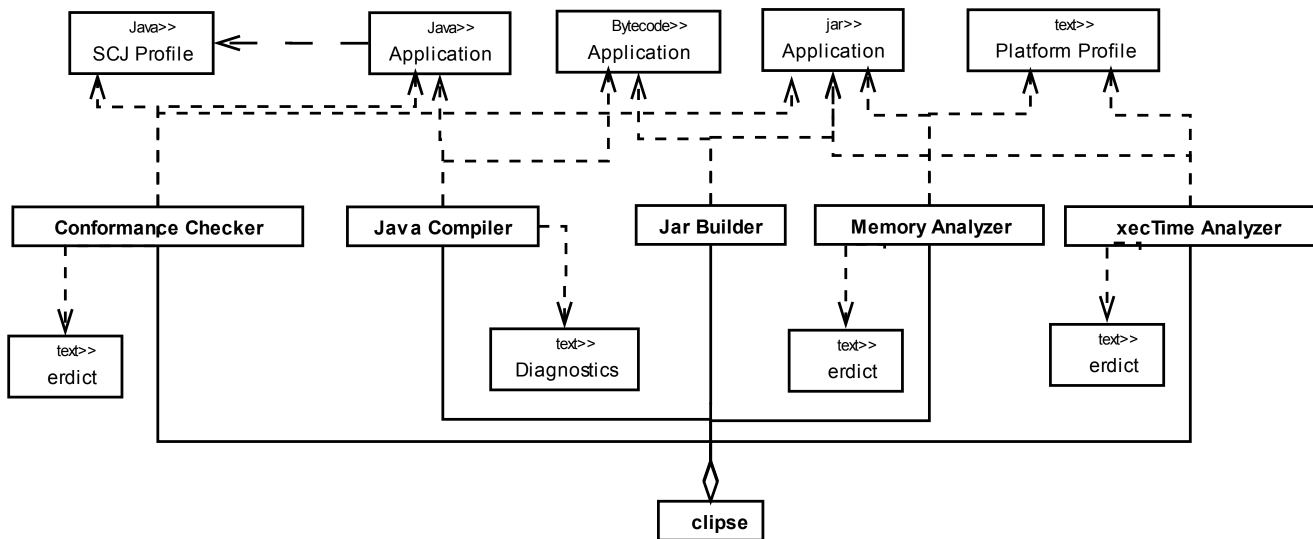

Fig. 2 Envisioned tool support: A workbench for analysing Real-Time Java programs.

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 [**?**].

3.3 Memory Analysis

The *scoped memory* model, employed by several of the real-time profiles discussed in the previous section, affords 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 SCJ profile while 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 the problem of 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, 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*. Thus, 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 [?], 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.

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) [?,?] is a static code analysis tool for conducting WCET analysis of Java bytecode executed on the Java Optimized Processor (JOP) [?]. JOP 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 [?] 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) [?] and another models the real-time application using timed automata in the verification tool Uppaal [?]. 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 process. The precise WCET estimate is a consequence of the model representing the detailed behaviour of the system, especially the cache model, more precisely.

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 [?]. For the IPET strategy, WCA transforms the CFG into an integer linear programming problem which is solved using the linear programming solver *lp_solve* [?] 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, it 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 this 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 all their fundamental temporal requirements, e.g., period and deadline. 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* tool for schedulability analysis.

TIMES

TIMES [?] is a model-based schedulability analysis tool. That is, all provided information is transformed into timed automata on which the model checker Uppaal [?] 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 3. As shown, the

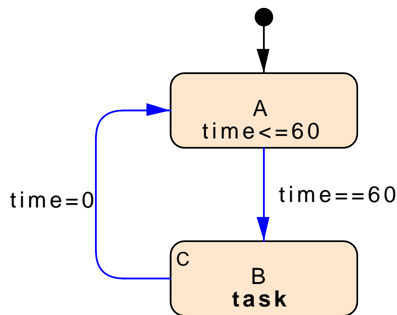

Fig. 3 TIMES release pattern, illustrating how the release of a periodic task can be modelled.

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 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 4. This representation is especially convenient for debugging since it shows precisely what goes wrong and where.

SARTS

SARTS [?] combines model-based WCET analysis and model-based schedulability analysis. 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. 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 preemptive 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.

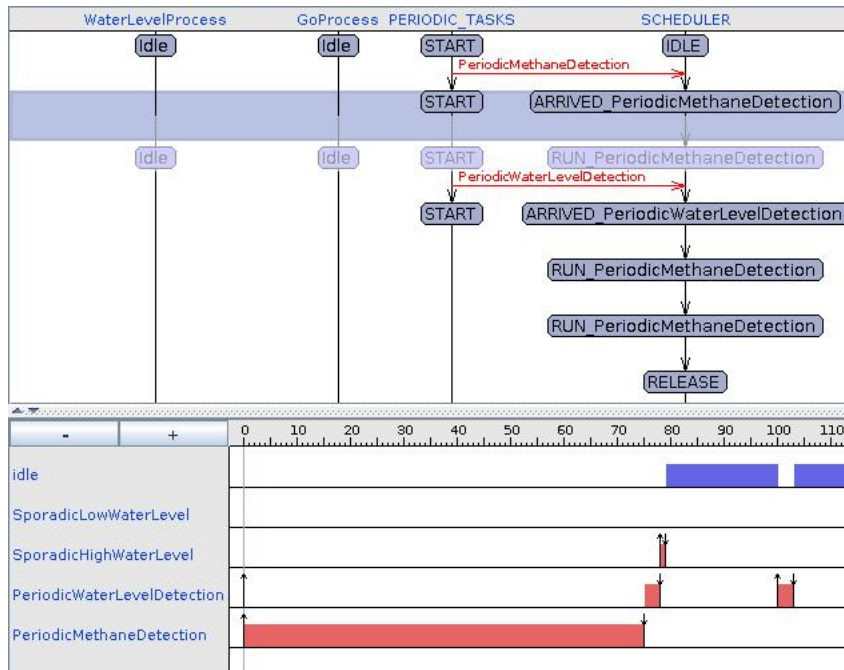

Fig. 4 Example of a simulation run in TIMES.

This approach is interesting because it considers interactions between tasks at a more fine grained level than traditional analyses using plain WCET and worst case blocking. This is 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 to have knowledge of timed automata.

4 Case Study

The case study of a hard real-time system implemented in Java is based on the classical text-book example of a mine pump [?]. 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.

4.1 Comparison of Real-Time Java Profiles

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 1 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 1 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 2 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 2. 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 3 Detecting the methane level.

Listing 3 shows the event handling method of the periodic event handler `PeriodicMethaneDetection`, implemented in the PJ profile. It is similar in the SCJ profile.

4.2 Evaluating Schedulability of Control Software

To ensure that the control software adheres to its temporal requirements, schedulability analysis with TIMES has been conducted. 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.

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 [?] 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. With respect to the JVM, the environment uses JamaicaVM from aicas, one of the partners discussed below, which is updated to provide time predictable behaviour. That is, JamaicaVM is extended with a real-time garbage collector and supports the RTSJ specification. For WCET analysis, HIDOORS suggests that the underlying hardware must be modelled, such that caching and pipelin-

ing can be accounted for in the analysis. As part of this, the PAG [?] tool is used for data flow analyses.

The company aicas [?] 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 [?] 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.

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 study reported here gives some evidence 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. Some tools like WCA offer (a bit of) automation, 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 [?]. 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, we intend to focus more specifically on the SCJ-profile and ensure that the tools cooperate with an Eclipse development environment. 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. 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.

Acknowledgements 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.