Model-Based Schedulability Analysis of Safety Critical Hard Real-Time Java Programs

Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, Kim G. Larsen
Department of Computer Science
Aalborg University, Denmark

{boegholm,cromwell,petur,bt,kgl}@cs.aau.dk

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 [21] for the Java Optimized Processor [18], to timed automata models verifiable by the UPPAAL model checker [23]. Schedulability analysis is reduced to a simple reachability question, checking for deadlock freedom. Model-based schedulability analysis has been developed by Amnell et al. [2], 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.

Categories and Subject Descriptors

I.6.3 [**Simulation and Modeling**]: Applications; D.3.2 [**Language classifications**]: Object-oriented Languages; D.3.4 [**Programming Languages**]: Processors - Runtime environments; D.4.1 [**Operating Systems**]: Process Management - Scheduling, Threads; J.7 [**Computer Applications**]: Computers in Other Systems - Real time

General Terms

Real-time systems, Software verification

Keywords

Real-time Java, Java processor, Schedulability analysis, Model checking

JTRES'08 September 24-26, 2008, Santa Clara, California, USA
Copyright 2008 ACM 978-1-60558-337-2/08/9 ...$5.00.

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 [2] 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) [11], the Ravenscar-Java Profile [16], which is based on the Ravenscar profile for Ada [10], and the Safety Critical Java (SCJ) profile [21]. 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 [21] for the Java Optimized Processor (JOP) [18], to an abstract time preserving model, on which the UPPAAL model-checker [23] 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 [12, 13, 15]. 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 [8]. WCA [20] is a tool developed for JOP [18], 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 [9] 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 [2] 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 [22]. However, it is unclear how Polychrony handles WCET, and therefore how it can be utilized as a schedulability tool.

Java PathFinder [7] 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 [17].

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 [21] implemented and documented in [5]. 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 [3].

In the current implementation, WCET calculation and simple collapsing is performed. SIR is translated to a UPPAAL model. For a description of UPPAAL see [4].

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 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,

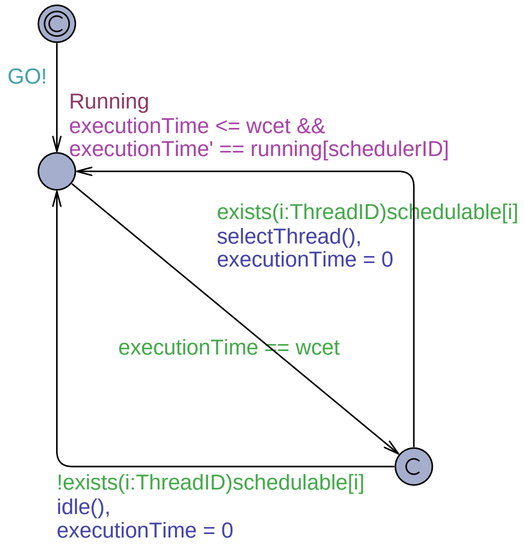

Figure 1: Scheduler

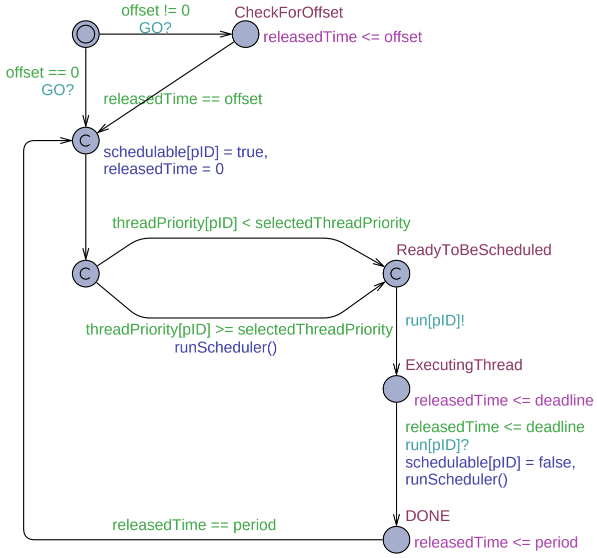

Figure 2: PeriodicThread base model

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.

```
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.

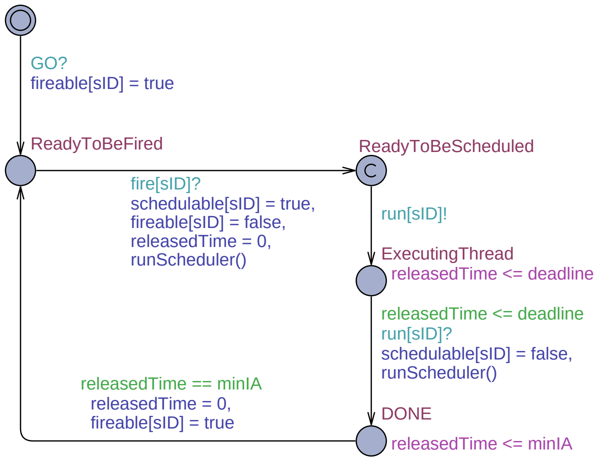

Figure 3: SporadicThread base model

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.

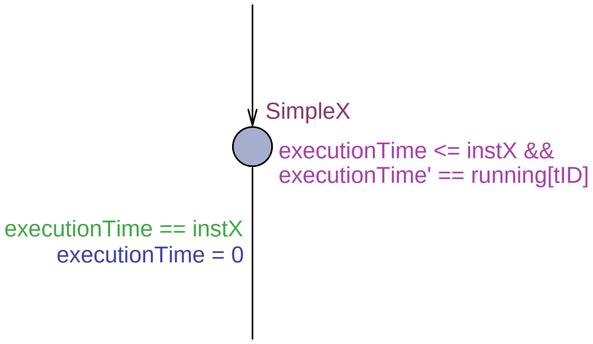

Figure 4: Simple basic block

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.

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

Listing 2: Run method example

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.

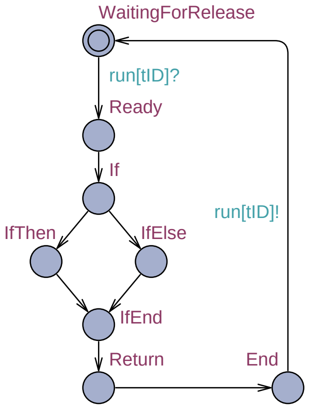

Figure 5: Translated Uppaal model

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.

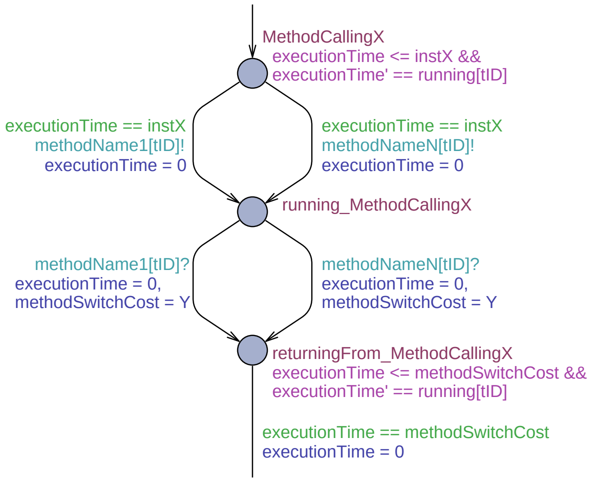

Figure 6: Invoke of a standard method

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 [6]. 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].

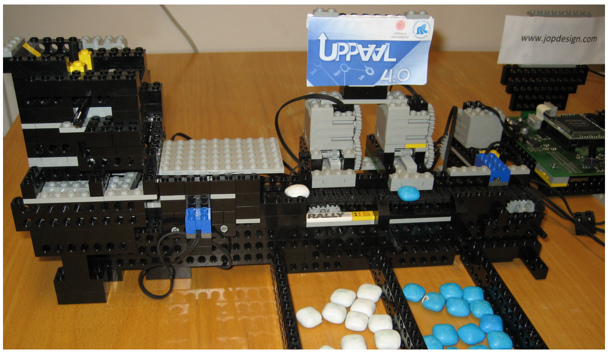

Figure 7: Real-Time Sorting Machine

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.

Task	Cyclomatic Complexity
Periodic 1	9
Periodic 2	17
Sporadic	7

Table 1: Cyclomatic complexity of tasks

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 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.

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.

```
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

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 [5] it is shown that SARTS is comparable to WCA [20] in terms of WCET analysis accuracy.

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

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 [8]. 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.

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 $\text{RTSM}_{\text{Simple}}$ 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 $\text{RTSM}_{\text{Simple}}$

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.

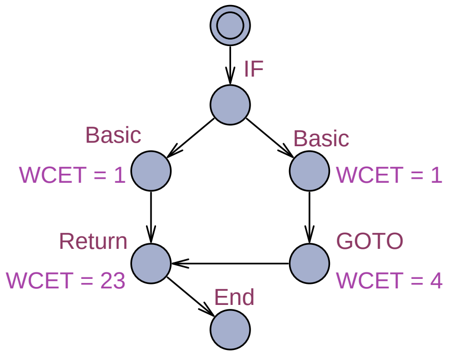

Figure 9: Javac compiled model

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.

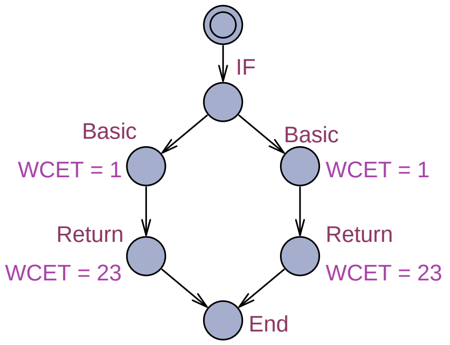

Figure 10: Eclipse compiled model

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.

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

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

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

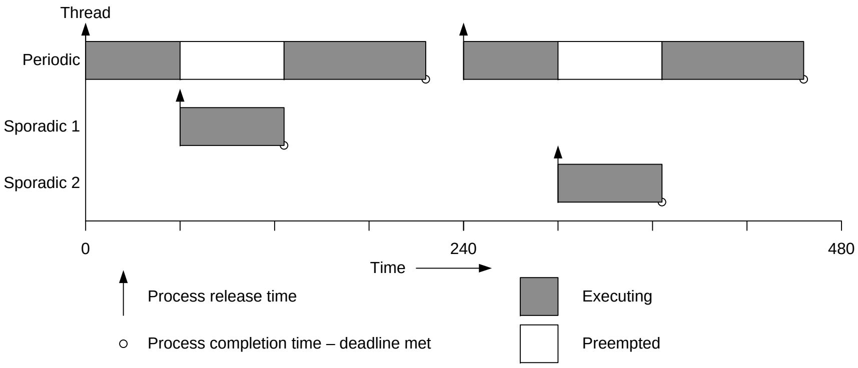

Figure 8: Time-line for conditional sporadic invoke

a safety property to be satisfied, then it is also satisfied without the approximation. The result of this experiment is shown in Table 4.

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

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

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

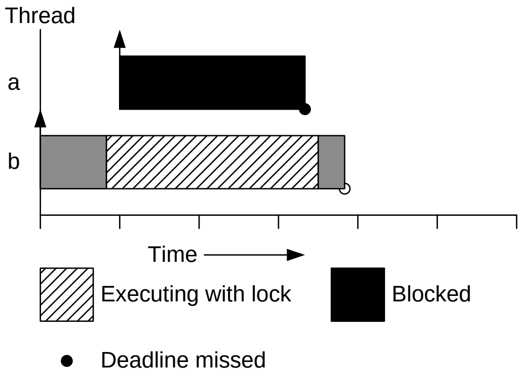

Figure 11: Blocking example, actual execution

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.

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

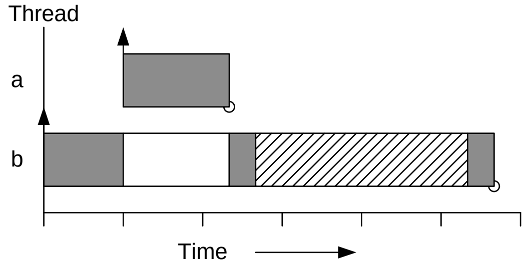

Figure 12: Blocking example, generated model

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 [19].

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 [14].

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 [21] is supported and the only execution platform supported is the JOP [18]. We believe supporting other Java processors such as the AJ-100 from aJile Systems [1] 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 [17], 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.

9. REFERENCES

[1] aJile. *aJile Systems, Inc.* http://www.ajile.com/, 2008.

[2] 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*, 2003.

[3] BCEL. *Apache Software Foundation.* http://jakarta.apache.org/bcel/, 2006.

[4] 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.

[5] T. Bøgholm, H. Kragh-Hansen, and P. Olsen. *Model-Based Schedulability Analysis of Real-Time Systems*. https://services.cs.aau.dk/public/tools/library/details.php?id=1213086732, 2008.

[6] T. Bøgholm, H. Kragh-Hansen, and P. Olsen. *Real-Time Java*. https://services.cs.aau.dk/public/tools/library/details.php?id=1199704154, 2008.

[7] G. Brat and S. Park. Java pathfinder - second generation of a java model checker. In *In Proceedings of the Workshop on Advances in Verification*, 2000.

[8] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley Longmain, 2001.

[9] 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.

[10] B. Dobbing and A. Burns. The ravenscar tasking profile for high integrity real-time programs. In *SIGAda '98: Proceedings of the 1998 annual ACM SIGAda international conference on Ada*, pages 1–6, New York, NY, USA, 1998. ACM.

[11] G. B. et al. *The Real-Time Specification for Java*, 2000.

[12] 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.

[13] E. Fersman and W. Yi. A generic approach to schedulability analysis of real-time tasks. *Nordic J. of Computing*, 11(2):129–147, 2004.

[14] 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*, pages 162–169, New York, NY, USA, 2006. ACM.

[15] 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.

[16] J. Kwon, A. Wellings, and S. King. Ravenscar-java: A high integrity profile for real-time java. In *In Joint ACM Java Grande/ISCOPE Conference*, pages 131–140. ACM Press, 2002.

[17] G. Lindstrom, P. C. Mehlitz, and W. Visser. Model checking real time java using java pathfinder. In *ATVA*, pages 444–456, 2005.

[18] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[19] M. Schoeberl. *JOP Reference Handbook*. http://jopdesign.com/doc/handbook.pdf, 2007.

[20] M. Schoeberl and R. Pedersen. Wcet analysis for a java processor. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 202–211, New York, NY, USA, 2006. ACM.

[21] 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, 2007. IEEE Computer Society.

[22] J. TALPIN, E. GAMATI, B. L. DEZ, D. BERNER, and P. L. GUERNIC. Hard real-time implementation of embedded software in java, 2003.

[23] UPPAAL. *Uppsala University and Aalborg University*. http://www.uppaal.com/, 2006.