TetaSARTS: A Tool for Modular Timing Analysis of Safety Critical Java Systems

Kasper Søe Luckow, Thomas Bøgholm, Bent Thomsen, Kim Guldstrand Larsen
Department of Computer Science, Aalborg University, Denmark
{luckow,boegholm,bt,kgl}@cs.aau.dk

ABSTRACT

We describe the design and the capabilities of the static timing analysis tool TetaSARTS that assists in temporal verification of Safety Critical Java (SCJ) systems. The primary functionality of TetaSARTS is schedulability analysis, which takes into account the scheduling policy and task interactions. TetaSARTS also facilitates analysing processor utilisation and idle time, Worst Case Execution Time, Worst Case Response Time, and Worst Case Blocking Time.

In the analyses, TetaSARTS accounts for the execution environment hosting the analysed system; both hardware implementations of the Java Virtual Machine as well as software implementations hosted on common embedded hardware are supported. Several parameters of the execution environment can be adjusted prior to performing the analyses e.g. the clock frequency of the hardware.

The enabling technology for supporting the analyses and for achieving high flexibility is model checking. In a process resembling the stages of an optimising compiler, TetaSARTS translates the SCJ system into a Network of Timed Automata amenable to model checking using the Uppaal model checker.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking; D.3.2 [**Language Classifications**]: Object-oriented languages; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Verification, Languages, Experimentation

Keywords

Real-time Java, Real-time embedded systems, Schedulability analysis, Model checking, Safety Critical Java, UPPAAL

JTRES'13, October 09 - 11 2013, Karlsruhe, Germany

http://dx.doi.org/10.1145/2512989.2512992.

1. INTRODUCTION

Java is one of the most popular programming languages. With its *write once - run anywhere* approach, it was quickly propelled into mainstream computing. It is, however, well known, that Java in its traditional form is unsuited for development of safety-critical hard real-time embedded systems. The Java community, through JSR 302, has made tremendous progress towards enabling development of this type of systems, and an important step has been taken with the upcoming Safety Critical Java (SCJ) standard [17]. However, for SCJ to establish itself as a viable technology and a competitor to C in the embedded hard real-time market, SCJ must be complemented with tools that support the development, and aid in verifying the correctness of the system. Besides being dependent on logical correctness, real-time systems are characterised by having real-time constraints requiring reasoning on temporal correctness as well. The primary component in verifying the latter is the *schedulability analysis* whose goal is to check that all real-time tasks finish executing before their deadline.

While traditional and generic approaches to schedulability analysis could be used, these also come at the expense of conservatism due to insensitivity to the control-flow, and, in addition, often due to an abstract view of dynamic features such as caches and pipelining. Consequently, these analyses are often pessimistic, and may prevent high processor utilisation of the hosting platform of the real-time system or, even worse, deem the system not schedulable when it actually is. For real-time systems developed in Java, this pessimism can be even higher. This is partly due to the fact that Java is object-oriented, but also due to the fact that Java usually is implemented via a translation to Java Bytecode, which is then either interpreted by a Java Virtual Machine (JVM) or further translated to native code. This level of indirection then complicates formal analysis as both program and JVM implementation have to be taken into account for a given hardware platform; some of this complexity can, however, be reduced by a hardware implementation of the JVM.

In embedded real-time systems, it will clearly be difficult to achieve the *write once - run anywhere* advantage that Java programs enjoy on the desktop and in server side programming, as embedded systems often come with hardware specific needs and always will be dependent on timing constraints of the underlying hardware platform. Thus, timing properties will vary significantly depending on either a hardware implementation of the JVM, such as JOP or Ajile-100 [1], or a commodity hardware platform such as the Atmel AVR ATmega2560 microcontroller equipped with a suit-

able implementation of the JVM, such as the FijiVM [22] or the Hardware near Virtual Machine (HVM) [18].

To accommodate these issues, we put forward a tool called TetaSARTS[1], that will allow the programmer to write the program in a platform independent way, using the SCJ profile, and then analyse whether it can be scheduled on the particular platform. Hence, this enables a *write once - run wherever possible* development approach. The primary purpose of the tool is schedulability analysis of SCJ tasks with a refined system model including the exact release patterns of tasks, interleavings, and resource sharing. Additionally, it features a pluggable platform model allowing analysis of systems running on common execution environments with a software implementation of the JVM and commodity embedded hardware, as well as hardware implementations of the JVM. Due to this system model, safe and less pessimistic schedulability analysis can be conducted, and in addition it facilitates analysing processor utilisation and processor idle time, Worst Case Execution Time (WCET), Worst Case Response Time (WCRT) taking into account pre-emption and task interactions, and Worst Case Blocking Time (WCBT).

The tool employs a model-based approach where the program analysis problem of determining schedulability is viewed as a reachability model checking problem similar to the approach in the TIMES [2] and SARTS [8] tools. TetaSARTS can be viewed as an optimising compiler that produces a model of the system amenable to model checking using Uppaal [5][19], given the program source as input. The model is constructed such that model checking simulates an abstract execution of the real-time tasks, taking into account the exact execution environment and scheduling policy.

This paper presents TetaSARTS in terms of its design and capabilities and evaluates on its applicability as well as the effects of the employed optimisations and is structured as follows: in Section 2, we present related work followed by the design of the TetaSARTS tool in Section 3 and present how the timing model is constructed in Section 4. Section 5 presents the optimisations used for reducing analysis time and memory consumption. Section 6 presents the timing analyses the constructed timing model can be used for, followed by Section 7, in which the analyses are evaluated using representative real-time systems, and present the effect of the optimisations. Section 8, presents the conclusion.

2. RELATED WORK

Traditional methods for schedulability analysis include response time analysis [9]; for each task, the response time is calculated based on WCET and blocking times, and the system is schedulable if the response times for the tasks are less than their respective deadlines. In contrast to our approach, it is difficult to include detailed information about the underlying hardware, such as caching, and in general, response time analysis is based on a coarse, control-flow insensitive system model based on worst-case scenarios that often do not happen in practice.

TetaSARTS is inspired by other tools for conducting timing analyses; XRTJ [16], TetaJ [13], METAMOC [11], TIMES [2], and SARTS [8]. TetaJ is a model-based, WCET analysis tool intended for Java systems written in the emerging SCJ profile [17]. As opposed to XRTJ, which relies on the provision of timing documents (VMTMs [15]) describing the timing properties of the execution environment obtained by e.g. measurements, TetaJ incorporates an explicit notion of the behaviour of the execution environment. TetaJ is further inspired by METAMOC; a WCET analysis tool that uses the Uppaal [5] model checker and analyses C programs. In contrast to the timing model of TetaJ (and METAMOC), the model of TetaSARTS also facilitates the analysis of schedulability, WCRT, processor utilisation and idle time, and WCBT.

The TIMES [2] tool presents a generic, model-based technique for schedulability analysis in which a specification for the real-time system is built as a set of tasks modeling their timing properties such as cost, dependencies, and deadlines. This results in a Network of Timed Automata (NTA) model which is checked using Uppaal [5]. A Timed Automaton (TA) is a finite state machine extended with real-valued clocks, and an NTA is the parallel composition (using the CCS operator) of a number of TAs sharing clocks and actions (for the semantics, see [6]). Contrasting TetaSARTS, TIMES does not perform timing analysis of the code associated with the tasks and can only be used for analysing schedulability.

Inspired by the model-based ideas of TIMES, SARTS [8] is a schedulability analysis tool for Java systems written in a variant of the SCJ profile. From the compiled Java code, SARTS builds an NTA model with a tight correspondence to the analysed code and combines it with a model simulating a Fixed-Priority Pre-emptive Scheduler. The actual analysis is viewed as a reachability model checking problem and the model is exhaustively explored for determining whether there exists a state, such that a task misses its deadline. In contrast to TetaSARTS, it assumes that the Java Bytecode execution times are fixed, and that the analysed system is hosted on the JOP. Furthermore, it does not facilitate analysing other temporal properties e.g. WCET.

Using the NTA model for other timing-related analyses than schedulability, is inspired by the work presented in [21]. In that work, the system model allows for determining the WCRT of tasks, processor utilisation and idle time, and blocking time. The system model is, however, not automatically constructed from the program source thus implying a loose correspondence between analysis and code that does not easily accommodate changes in the code. Furthermore, real-time task parameters, such as WCET and the behavior of the real-time tasks, are manually encoded in the NTA.

Approaching NTA generation from program source as an optimising compiler is inspired by the Bandera [10] tool, which is capable of generating automata descriptions for various model checkers such as PROMELA for the SPIN [14] model checker from Java program source. The resulting models can be used for verification purposes; especially the verification of properties related to concurrent systems.

3. TETASARTS

TetaSARTS is a tool that conducts timing analyses of SCJ tasks taking into account the task release patterns, task interactions using shared resources, and blocking. Currently, it offers schedulability analysis, processor utilisation and idle time analysis, WCRT analysis, WCET analysis, and WCBT analysis to be performed. It was built with modularity and flexibility in mind, and as such, makes it possible to adjust a wide variety of options to make the analyses reflect

[1]The tool can be obtained at `http://people.cs.aau.dk/~luckow/tetasarts/`

the configuration of the analysed system precisely. It accounts for the execution environment hosting the analysed system both making possible to analyse systems featuring a traditional execution environment with a software implementation of the JVM running on top of common embedded hardware, as well as the case where a hardware implementation of the JVM is used. The ease with which the options can be adjusted facilitates various cases of development:

- Since embedded systems are usually produced in large quantities, mitigating the unit price is an imperative. Hence, in case the execution environment has not been decided upon, TetaSARTS enables determining which execution environment configuration suffices for the system to be in accordance with its temporal constraints.
- Using a similar rationale, a desirable trait is to lower the running costs of the deployed system. Reducing the energy consumption of the system can be used for partly achieving this which is influenced by the clock speed of the processor. TetaSARTS supports this since it is parameterised on the processor clock speed, thus allowing to conclude whether a specific setting still ensures schedulability of the system. This has been explored in [20].

TetaSARTS offers support for ARM and AVR which are reused from METAMOC. Hence, support for more hardware platforms follows the development of METAMOC. In addition, TetaSARTS supports the HVM and JOP.

Another usage of TetaSARTS is to apply the supported analyses during system development: the complementation of e.g. WCRT and WCBT analyses can be used for identifying the cause to non-schedulability. It may for instance be possible to identify regions of code that contribute significantly to the WCET of a task.

TetaSARTS adopts model checking for its supported analyses; the Java Bytecode of the system and the JVM implementation are individually transformed to NTAs. These, along with an NTA simulating the hardware, are combined into a single NTA that can be model-checked using Uppaal. The entire translation process is fully automatic, thus facilitating that a tight correspondence can be kept between the model used for analysis and actual code. Generation of the final NTA is reasonably fast (30s-60s) and hence accommodates a rapid development process where timing properties are (re-)evaluated even after small changes to the code.

The transformation process draws many similarities with the stages of an optimising compiler. Figure 1 shows the major components involved.

Initially, the Java Bytecode system is transformed into TetaSARTS Intermediate Representation (TIR) similar to a Control-Flow Graph (CFG). TIR is an appropriate format for conducting various high-level analyses and transformations such as loop identification analysis. This is used for extracting the loop bound annotations whose format is $//@loopbound = [bound]$ and must precede a loop construct. Recursion is currently not supported. The resulting decorated and optimised TIR is then subject to a coarse translation to an NTA, referred to as the *Program NTA*. Subsequently, it is synthesised with a model of the hardware, referred to as the *Hardware NTA*, and behavioural information about the JVM, yielding a single NTA capturing the behavior of the entire system.

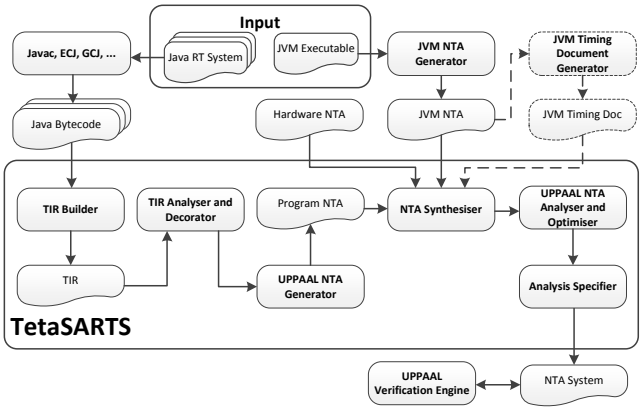


Figure 1: The constituents of the TetaSARTS tool.

JVM information can be synthesised from either of two sources: from a *JVM NTA*, capturing the control-flow behavior of all Java Bytecode implementations including hardware state, thus providing a precise timing model, or from a *JVM Timing Doc* (similar in some respects to VMTMs [16]) capturing the Best Case Execution Time (BCET) and WCET of the Java Bytecode implementations as obtained by the *JVM Timing Document Generator* (a model-based execution time analysis tool). The latter thus provides an abstract and less precise timing model, because simulation of pipeline and cache state between instruction execution and task interleaving is disregarded. However, the abstraction is at the benefit of reduced analysis time, since the precise timing model adds to the state space size due to the inclusion of the JVM NTA. It is beyond the scope to elaborate on the process for constructing the JVM NTA. However, it draws similarities with that of constructing the Program NTA and requires loop bound annotations as well as annotations for identifying the Java Bytecode implementations. The timing model representations are further explored in Section 4.

The synthesised NTA is subject to various analyses and optimisations for reducing the state space size of the model. This serves to reduce model checking time (i.e. the time for conducting the analyses) as well as reducing memory needs. The final step in the transformation process is to apply various adjustments to the system model and generate the Uppaal specifications corresponding to the desired analyses the NTA model should be used for.

4. NTA MODEL

The synthesised NTA is organised according to the major constituents of the analysed system and can architecturally be viewed as a layered model (see Figure 2) whose interactions are realised using dedicated communication channels. In each layer, the NTA simulates the corresponding behavior which in turn is based on the behavior of the layer below. This single level of dependency achieves high flexibility since a layer can be substituted as long as it adheres to the interface, which is just a co-action and a shared variable. This flexibility makes it possible to support virtually all configurations of the Java real-time system in terms of execution environment and scheduling policies. Figure 2 also shows the NTA constituents when a precise timing model and an abstract timing model of the execution environment is used.

The *Scheduler TA* is depicted in Figure 3. `Initial` marks

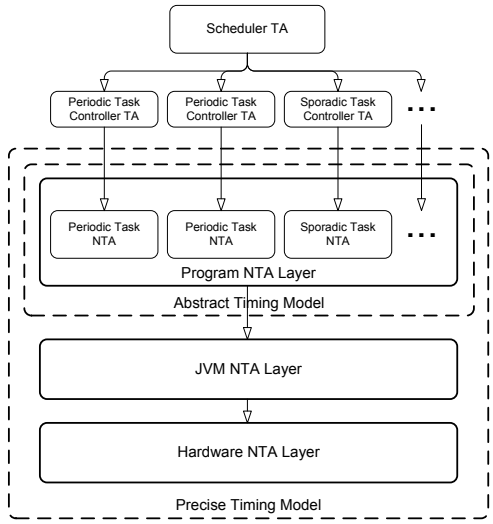


Figure 2: The NTA which is composed of a number of layers.

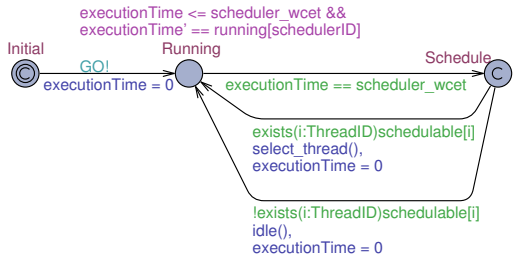


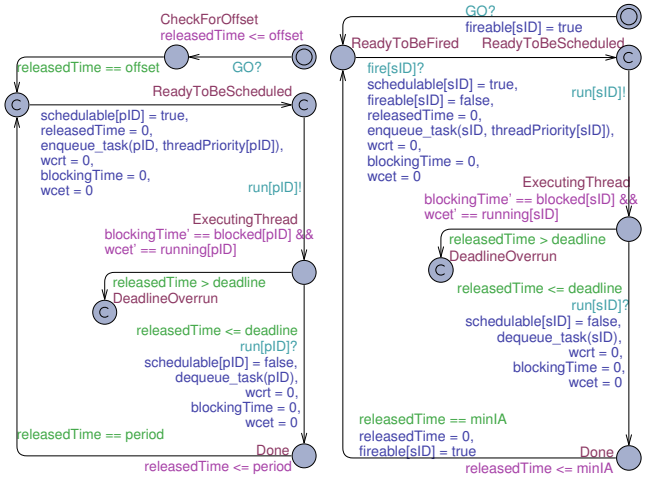
Figure 3: Generic scheduler TA.

the initial location of the TA. From there, it initiates the real-time tasks by broadcasting on the `GO` channel. The Scheduler TA waits in the `Running` location according to its WCET. The wait behavior is implemented using the boolean expression $executionTime == scheduler\_wcet$, known as a *guard* which must evaluate to true for enabling the firing of the edge, and the boolean expression

$$executionTime \leq scheduler\_wcet \;\&\&$$
$$executionTime' == running[schedulerID]$$

known as an *invariant* that must hold as long as the TA is in that location. `executionTime` is a real-valued *clock-variable* that progresses synchronously with the other clock-variables. The term $executionTime' == running[schedulerID]$ is a *stop-watch expression* which, depending on whether the right-hand side evaluates to 1 or 0, starts or stops the clock appearing on the left-hand side, respectively. `running` is an array of schedulable entities indexed by a designated ID (in this case `schedulerID`) returning true (that is, 1) in case that entity is set to run as governed by the scheduler. In the `Schedule` location, the edge corresponding to whether there are schedulable entities at the given instant will be fired. `select_thread()` implements the scheduling policy (for SCJ, fixed priority pre-emptive scheduling).

The NTA is also composed of a number of *Task Controller TAs*, see Figure 4, that are associated with the real-time tasks for controlling and monitoring their state during execution. These form the basis for the supported analyses, see Section 6. They are distinguished by their release pattern; the TA in Figure 4a is associated with periodically released tasks, while the TA in Figure 4b applies for sporadically released tasks. The latter is included because aperiodic tasks, which are part of the current SCJ draft, cannot be accounted for in the analyses due to their irregular release pattern and potential of being released at every time step.



(a) Periodic task controller. (b) Sporadic task controller.

Figure 4: Task controller TA models controlling and monitoring the states of the associated tasks.

The behavior of the two Task Controller TAs is the same except that the Sporadic Task Controller TA captures the firing of an event on the `fire[sID]` channel, while the Periodic Task Controller TA implements periodically releasing the task with time offsets. They both enter the `ReadyToBeScheduled` location whenever the associated task is released and eligible for being scheduled. Instantaneously, the clock, `releasedTime`, is reset and is used for monitoring whether the task completes before its deadline. A time unit is a clock cycle and is obtained by conversion of the time parameter (e.g. the deadline) at the task instantiation in the source code taking into account the clock frequency of the used hardware. When the task is done executing, it synchronises on the `run[pID]` channel (assuming a periodic task identified with `pID`), making the associated Task Controller TA enter the `Done` location. If it does not complete before the deadline, the `DeadlineOverrun` location is entered.

4.1 Program NTA

The Program NTA captures the control-flow behavior of the real-time tasks in the system. Let M_t denote the set of methods involved in the call-graph built from the real-time event-handler of task t such that $t \in Tasks_{sys}$ where $Tasks_{sys} = Tasks_{per} \cup Tasks_{spo}$, that is, the set of real-time tasks disregarding the release pattern. Let $\mathcal{A}_m$ denote the TA capturing the control-flow behavior of method m. The Program NTA can then be defined as $\{\mathcal{A}_m \mid m \in M_t \text{ and } t \in Tasks_{sys}\}$.

For the sake of simplicity, we introduce $L_{\mathcal{A}}$ and $E_{\mathcal{A}} \subseteq L_{\mathcal{A}} \times L_{\mathcal{A}}$ to denote the set of locations and edges of a TA, $\mathcal{A}$, respectively. The actual construction of $\mathcal{A}_m$ for method m is based on the CFG information of TIR and is dependent on whether a precise or abstract timing model of the

execution environment is used. In both cases, if $i_1, i_2, \ldots, i_n$ is a sequence of instructions following a control-flow path of the CFG where i_j for $1 \leq j \leq n$ is a symbolic placeholder for the instruction appearing at position j, we generate corresponding locations $l_1, l_2, \ldots, l_n \in L_{\mathcal{A}_m}$ and edges $\langle l_1, l_2 \rangle$, $\langle l_2, l_3 \rangle, \ldots, \langle l_n, l_{n+1} \rangle \in E_{\mathcal{A}_m}$ where l_{n+1} is a symbolic location where the clock of $\mathcal{A}_m$ cannot progress.

For a precise timing model, the execution of a Java Bytecode instruction is dependent on the actual behavior of the underlying execution environment; the execution time of i is not static. In that case, we generate the excerpt of a TA shown in Figure 5.

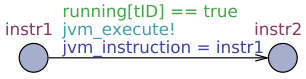


Figure 5: Excerpt of the TA used for simulating a precise execution of a Java Bytecode instruction.

Here, the guard, $running[tID] == true$, ensures that the edge can only be fired if the real-time task with ID `tID` is running according to the scheduling policy. `jvm_execute` is an *urgent channel* meaning that if an action, denoted `!`, is performed, and there exists a corresponding co-action, denoted `?`, then no delay must occur i.e. the synchronisation must be performed urgently. The JVM NTA contains the co-action on this channel, and will consult the variable `jvm_instruction` which will contain the current instruction to simulate.

For an abstract timing model, the execution of each Java Bytecode is provided in terms of a BCET and a WCET. The BCET is included to obtain a safe result such that the execution time, ET_i of instruction i, is $ET_i \in [BCET_i, WCET_i]$. This contrasts with response time analysis which is safe, but considers the conservative interval $ET_i \in [0, WCET_i]$, thus being less precise. In this case, we generate the excerpt of a TA shown in Figure 6.
Here, the guard

$$executionTime \geq BCET\_instr1 \;\&\&$$
$$executionTime \leq WCET\_instr1$$

is used to constrain the firing of the edge to only happen when $executionTime \in [BCET\_instr1, WCET\_instr1]$. $executionTime$ is reset, i.e. set to zero, on all edges. The invariant

$$executionTime \leq WCET\_instr1 \;\&\&$$
$$executionTime' == running[tID]$$

ensures that the edge will be taken when $executionTime \leq WCET\_instr1$. The stop-watch expression, $executionTime' == running[tID]$, is used for capturing pre-emption, i.e. the clock, $executionTime$ is

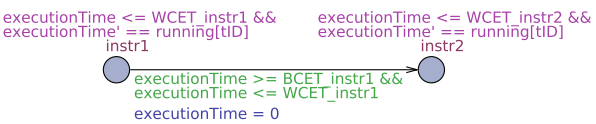


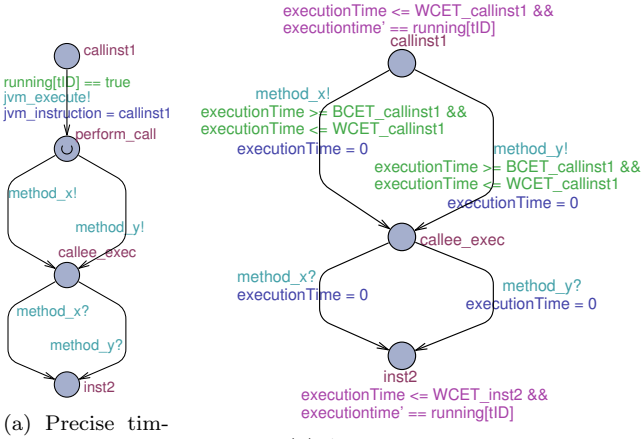
Figure 6: Excerpt of the TA used for simulating an abstract execution of a Java Bytecode instruction.

only allowed to progress when the real-time task identified on tID is running.

Whenever a method calling instruction is encountered, i.e. an instruction belonging to the set

$$CallInst = \{invokevirtual, invokespecial, invokedynamic, invokeinterface, invokestatic\}$$

the TA, $\mathcal{A}_m$ such that $l_i \in L_{\mathcal{A}_m}$ for instruction $i \in CallInst$, will capture the semantics of a method call by synchronising with the TAs capturing the possible callees of method m. Let $callees(i)$ denote the function that for instruction $i \in CallInst$, appearing at a unique program point, returns the set of possible callees. How $callees(i)$ is actually implemented and accounts for dynamic dispatch, is elaborated on in Section 5. For the translation to TA, assume $callees(i)$ returns two possible callees, denoted x and y, and i appears in method m. The excerpt of the resulting TA is shown in Figure 7.



(a) Precise timing model.

(b) Abstract timing model.

Figure 7: Excerpt of TAs used for simulating method invocation.

Here, the actions `method_x!` and `method_y!` initiate the invocations of method x and y, respectively. The TA, $\mathcal{A}_m$ will wait in the `callee_exec` location until the respective method has finished execution. At that point, the callee will make an action on the same channel which will be received in $\mathcal{A}_m$ by using the co-actions `method_x?` or `method_y?` depending on the invoked method.

Synchronised regions (and methods) are also accounted for in the model and unbounded priority inversion is handled by simulating the Immediate Ceiling Priority Protocol (ICPP) as dictated by the SCJ specification.

4.2 JVM NTA

The co-action on the `jvm_execute` channel is handled by the *Interpretation Scheme TA* of the JVM NTA. The interpretation scheme captures the concepts of the interpretation employed by the specific JVM implementation. In Figure 8, an excerpt of the scheme is shown for an iterative interpreter containing a fetch, analyse and execute stage.

Whenever a Java Bytecode instruction is to be simulated from the Program NTA, the Interpretation Scheme will first synchronise with a TA capturing preprocessing functionality (if there is any) and proceed to the `analyse` location. Here it consults the variable `jvm_instruction` which has been set

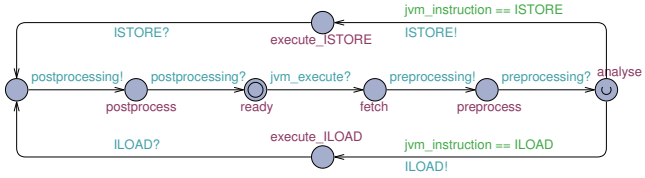


Figure 8: TA capturing the behavior of iteratively interpreting Java Bytecode instructions.

in the Program NTA, and accordingly synchronise with the TA capturing the behavior of the Java Bytecode instruction in a similar way as how method calls were modelled in the Program NTA. When the corresponding TA of the Java Bytecode has finished, a TA capturing possible post-processing is consulted. This could for instance comprise functionality resetting used variables etc.

The TAs capturing the behavior of the Java Bytecode implementations are constructed in a similar way as methods in the Program NTA. From the JVM executable, the CFG (TIR) is reconstructed and the Java Bytecode implementations are identified and respective TAs are generated. If $asm1$ and $asm2$ denote two machine instructions executed sequentially according to the control-flow, the TA excerpt shown in Figure 9 is generated.

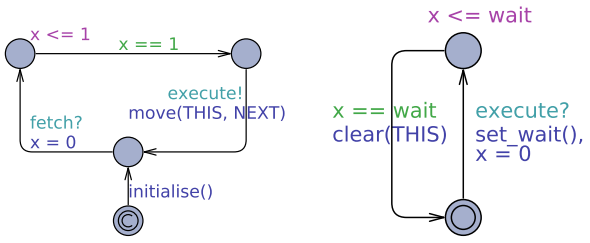
Figure 9: Excerpt of a TA simulating the execution of a machine instruction.

The TetaSARTS distribution contains a component for automatically constructing the JVM NTA given the executable as input. It currently offers support for AVR executables, but this is extendible.

4.3 Hardware NTA

The Hardware NTA captures the behavior of executing the assembly instructions on the hardware platform and contains the co-action of the `asm_execute` channel used in the JVM NTA. The TAs are not constructed by TetaSARTS but are instead provided by the METAMOC project.

In Figure 10, the TA models of the fetch and execute stages of a pipeline are shown.



(a) Pipeline fetch stage. (b) Pipeline execute stage.

Figure 10: Hardware TA models from METAMOC [11].

The interaction between the JVM NTA and the Hardware NTA is similar to how the Program NTA interacts with the JVM NTA. In the NTA shown here, the fetch stage, see Figure 10a, takes constant time of 1 time unit, i.e. one clock cycle, which is modelled using the invariant $x \leq 1$ and the guard, $x == 1$, on the outgoing edge. Subsequently, the execute stage is performed by making an action on the `execute` channel and move the instruction from the fetch to the execute stage using the `move(THIS, NEXT)` function. Next, the execute stage simulates the action of executing the instruction by waiting for a variable amount of time depending on the assembly instruction.

5. OPTIMISATIONS

To make real-sized systems tractable for analysis, a variety of optimisations are employed that mitigate the state space size.

5.1 JVM Specialisation

Many embedded systems do not use floating point arithmetic hence leaving out all the Java Bytecodes that handles doubles and floats. Moreover, many other Java Bytecodes are only rarely used. Due to this, TetaSARTS employs an analysis that conservatively estimates the set of Java Bytecodes the program is actually using. The analysis works by traversing TIR and visit every instruction of the program. The set of distinct Java Bytecode instructions visited is recorded. Using this information, all TAs of Java Bytecode instruction implementations that have not been visited, can safely be removed from the NTA, thus reducing both the size of state space and the number of TA instantiations.

5.2 TA Inlining

The state of an NTA is defined as $\langle \bar{l}, u \rangle$ where $\bar{l}$ denotes the vector of current locations of the TAs in the NTA, and u maps the clocks of the NTA to their current values. Each $l \in \bar{l}$ is represented using an integer in Uppaal, thus the size of *each* state is largely dependent on the number of TAs in the NTA. We can reduce the state size considerably in turn yielding reductions in memory consumption and verification time, by exploiting the structure of the JVM NTA and the JVM semantics; let $\overline{\mathcal{A}_{jvm}}$ be the set of TAs in the JVM NTA, and $\bar{l}_{jvm}$ denote the location vector of these (i.e. $\mid \overline{\mathcal{A}_{jvm}} \mid$ can potentially be the same as the number of supported Java Bytecodes in the particular JVM implementation). Let $\mathcal{A}^{\bullet}_{jbc} \in \overline{\mathcal{A}_{jvm}}$ denote the TA of an arbitrary Java Bytecode with $l^{\bullet}_{jbc} \in \bar{l}_{jvm}$ denoting its location. During simulation, all $\mathcal{A}_{jbc} \in \overline{\mathcal{A}_{jvm}} \setminus \{\mathcal{A}^{\bullet}_{jbc}\}$, will remain in the initial location while $\mathcal{A}^{\bullet}_{jbc}$ is given exclusive control. This property always holds, because the simulation of a Java Bytecode cannot be pre-empted. Thus, it is sufficient to only keep track of the location $l^{\bullet}_{jbc}$ of $\mathcal{A}^{\bullet}_{jbc}$.

This can be implemented by "inlining" all Java Bytecode TAs into the Interpretation Scheme TA analogously to the method inlining compiler optimisation, by substituting matching actions and co-actions by the corresponding Java Bytecode TA. Specifically, a dependency graph over the NTA is generated such that vertices represent TAs and an edge exists for TA $\mathcal{A}_1$ and $\mathcal{A}_2$ if there is an action, `a!`, in $\mathcal{A}_1$, and a corresponding co-action, `a?`, in $\mathcal{A}_2$. Inlining is then performed by traversing the dependency graph and recursively remove the edges forming the dependency and move the target TA of a dependency into the source.

5.3 Edge Aggregation

If the abstract timing model of the execution environment is used, the size of the Program NTA can be reduced by

aggregation of edges and removal of superfluous locations. Let $\mathcal{A}_m$ denote the TA of method m with locations $L_{\mathcal{A}_m}$ and edges $E_{\mathcal{A}_m}$. Let $l_1, l_2, \ldots l_n \in L_{\mathcal{A}_m}$ denote the locations corresponding to a maximal sequence of n instructions with the symbolic placeholders $i_1, i_2, \ldots i_n$ which are executed sequentially. The locations are connected by edges $\langle l_1, l_2\rangle, \langle l_2, l_3\rangle, \ldots, \langle l_{n-1}, l_n\rangle \in E_{\mathcal{A}_m}$. Denote the sequence of locations and edges by L_{seq} and E_{seq}, respectively. The symbolic instructions of location l_0 and l_n are conditional jumps; the former with jump target l_1 (and to some other branch) and the latter with jump targets l_{n+1} and l'_{n+1}. Further, let $wcet(l_i)$ and $bcet(l_i)$ return the WCET, and BCET of instruction i to which location l_i has been constructed.

Edge aggregation can then be performed by computing the aggregated WCET and BCET of the sequentially executed instructions as:

$$WCET_{agg} = \sum_{l_i \in L_{seq}} wcet(l_i)$$
$$BCET_{agg} = \sum_{l_i \in L_{seq}} bcet(l_i)$$

L_{seq} and E_{seq} are subsequently removed from $\mathcal{A}_m$ and replaced with a new location, l_{agg}, and edges $e_{agg} = \langle l_{agg}, l\rangle$ such that $l \in \{l_{n+1}, l'_{n+1}\}$ (see e.g. $\langle l_{agg}, l_{n+1}\rangle$ in Figure 11). The original edge $\langle l_0, l_1\rangle$ is updated to $\langle l_0, l_{agg}\rangle$.

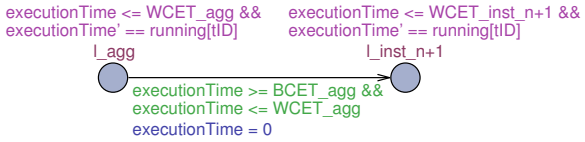


Figure 11: Excerpt of the abstract TA after performing edge aggregation.

Here, the guard is the sum of the BCET and WCET of the instructions that have been aggregated and the system is allowed to stay in l_{agg} for $WCET\_agg$ time.

5.4 Receiver Type Analysis

Dynamic polymorphism using dynamic dispatch plays an important role in Java meaning that, from a static viewpoint, the run-time type of an object can be any subclass of that type. Therefore, naively, a virtual method call site is modelled as a nondeterministic choice between all the callees of the possible run-time types which, in cases with large class hierarchies contributes significantly to the size of the state space because all possible callees need to be explored.

To reduce non-determinism and speed up both verification time and memory consumption, TetaSARTS employs receiver type analyses that limit and in some cases completely devirtualise virtual call sites. The analyses presented here are used in the *callees* function previously used in Section 4.1. TetaSARTS makes available different approaches since the precision of the receiver type analysis comes at the cost of increased NTA generation time:

Class Hierarchy Analysis (CHA) Considers only the declared type of the callee and combines it with complete information about the class hierarchy. If a virtual method call is made on method m where the declared type of the receiver is C and has subtypes $\{S_1, S_2, \ldots, S_n\}$, then only C and the subtypes that override m will be considered. [12]

Rapid Type Analysis (RTA) Is an extension to CHA, which combines the information about globally instantiated types and intersect it with the class hierarchy information about the callsite as obtained by CHA. [3]

Variable Type Analysis (VTA) Is an analysis that for each variable in the program including the local variables in methods, makes a conservative estimate of the set of types that may possibly reach each variable. [24]

6. SUPPORTED TIMING ANALYSES

The timing model facilitates various timing analyses to be conducted, which we elaborate on here. All analyses are based on information that can be retrieved from the Task Controller TAs shown previously in Figure 4.

A system is deemed schedulable if no task misses its deadline under the employed scheduling policy. For the timing model, such an analysis problem is viewed as a reachability model checking problem; the location `DeadlineOverrun` in the Task Controller TA (see Figure 4) records if the associated task at some point misses its deadline. Also, note that if this location is entered, a deadlock occurs, because the location does not have an outgoing edge. Let ϕ denote the disjunctive state formula of the $DeadlineOverrun_i$ location for all $i \in Tasks_{sys}$ i.e. $\phi = \bigvee_{i \in Tasks_{sys}} DeadlineOverrun_i$. Using the specification language of Uppaal, the schedulability analysis can be formulated as $A\Box$ *not* ϕ meaning that, for all reachable states, ϕ is not satisfied.

The worst-case processor utilisation (and idle) time of the system can also be determined provided that it is schedulable. In this case, TetaSARTS generates the *Utilisation Monitor* whose TA is shown in Figure 12 and adds two new clock variables, `idle` and `util`, for monitoring the idle and utilisation time of the processor, respectively.

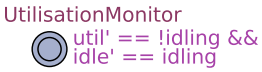


Figure 12: TA used for monitoring the utilisation and idle time of the processor.

The clocks are used as stop-watches such that whenever the scheduler has set a real-time task for execution, the `util` clock progresses while the `idle` clock is stopped. The opposite holds when no task is eligible for execution in which case the scheduler is idling. Estimating the worst-case utilisation and idle time can be done using the sup-query extension of the Uppaal specification language. A sup-query performs an exploration of the state space and records the maximum observed value, i.e. the *sup*remum, of the specified (clock) variable(s). Hence the analyses can be performed using the query $sup : util, idle$.

For WCRT analysis, TetaSARTS introduces a new clock, `wcrt`, for each Task Controller TA (see Figure 4). Whenever, the associated task is released, the clock is reset and keeps progressing while the task is executing, pre-empted or blocked. These three states are captured in the `ExecutingThread` locations and determining the WCRT can be done by finding the maximum observed value of the `wcrt` clock by using the sup-query:

$sup\{taskController.ExecutingTask\} : taskController.wcrt$

WCET analysis can be conducted in a similar way as WCRT analysis; a new clock `wcet` is introduced and is reset whenever the associated task is released (see Figure 4). Using the stop-watch expression, $wcet' == running[tID]$ ensures that the clock is only progressing whenever the task is running thus preventing blocking and pre-emption to influence the clock value. WCET analysis is performed using the query:

$sup\{taskController.ExecutingTask\} : taskController.wcet$

Worst-case blocking time is performed by introducing a clock, `blockingTime`, in the Task Controller TA which acts as a stop-watch. Whenever a task enters a synchronised region, the $blocked[tID]$ is set accordingly for those tasks identified on tID that are blocked. The `blockingTime` clock is only progressing if its corresponding task is blocked as shown in the invariant of the `ExecutingTask` location (see Figure 4). The worst-case blocking time can then be found as

$$sup\{taskController.ExecutingTask\} : taskController.blockingTime$$

7. EVALUATION

In this section, we demonstrate the applicability of TetaSARTS using representative examples of real-time systems, and evaluate on the effects of the optimisations. All results were obtained by running Uppaal on an application server with an Intel Xeon X5670 @ 2.93 GHz and 32 GB of memory. The systems we base our evaluations on are:

Class Hierarchy consists of four classes forming a class hierarchy of height four. Each class overrides the method `compute()` which performs a resource intensive calculation. Eight real-time tasks calls different implementations of `compute()`. This system is used for showing the effect of employing receiver type analysis.

Sequential Computation is a system composed of ten real-time tasks each of which performing a calculation using only a few conditional Java Bytecodes. This is used for demonstrating the effect of edge aggregation.

Simple RTS consists of nine real-time tasks each of which performing a calculation using only a few different Java Bytecodes. This system is used for demonstrating the effect of JVM specialisation and inlining TAs.

Minepump is the classic text-book example of a minepump control system that manages the operation of a water pump based on environmental conditions such as water height and methane concentration.[9][7][13]

Real-Time Sorting Machine (RTSM) is an example of an embedded real-time system that manages two motors for sorting coloured bricks based on measurements from sensory equipment.[8]

MD5SCJ is based on five periodic tasks calculating the MD5 sum of a byte array. The implementation of the MD5 task has been used in oSCJ [23].

Table 1 demonstrate resource requirements of TetaSARTS when analysing representative examples of real-time systems written in the SCJ profile for schedulability. The clock speed of the HVM + AVR configuration is set to $10MHz$ while it is set to $60MHz$ for the JOP. The *Prec* and *Abs* subscripts denote that a precise and abstract timing model has been used, respectively. In all cases, the system has been deemed schedulable.

System	Exec. Env.	Time	Memory
Minepump	HVM+AVR$_{Prec}$	15h 25m	17933 MB
Minepump	HVM+AVR$_{Abs}$	2s	11 MB
Minepump	JOP	1s	11 MB
RTSM	HVM+AVR$_{Prec}$	OOM†	OOM
RTSM	HVM+AVR$_{Abs}$	1m 2s	17 MB
RTSM	JOP	5s	15 MB
MD5SCJ	HVM+AVR$_{Prec}$	OOM	OOM
MD5SCJ	HVM+AVR$_{Abs}$	8s	17 MB
MD5SCJ	JOP	1m 23s	47 MB

†Out of memory.

Table 1: Analysis times and memory consumptions of analysing SCJ real-time systems for schedulability.

As expected, analyses times are significantly higher when a precise timing model is used, thus, there is a trade-off between analysis time and potential precision of analysis. Using an abstract timing model (or the JOP), both analysis times and memory consumptions are reasonably low to be conducted on desktop machines.

We demonstrate the results of using the supported timing analyses on the RTSM system. Table 2 shows how different adjustments of the clock speed of the processor on the execution environment renders the system schedulable/maybe not schedulable. The *maybe* not schedulable case can potentially occur due to the inclusion of stop-watches for which reachability analysis is over-approximated in Uppaal.

Exec. Env.	Clock Freq.	Schedulable
HVM+AVR	10 MHz	✓
HVM+AVR	5 MHz	×
JOP	2 MHz	✓
JOP	1 MHz	×

Table 2: TetaSARTS with various execution environments.

Analysis time and memory consumptions are the same as the results of the RTSM in Table 1. Using an approximation method like the bisection method would make it possible to estimate the clock speed that still deems the system schedulable even more precisely.

Using JOP as execution environment, the results of the processor utilisation time and idle time analyses are shown in Table 3.

System	Clock Speed	Proc. Idle	Proc. Util.	Analysis Time
RTSM	100 MHz	4.0 ms	50.3 μs (1.2%)	39s
RTSM	60 MHz	4.0 ms	83.9 μs (2.1%)	

Table 3: Processor utilisation and processor idle times.

As expected, the utilisation time is improved when the clock speed of the processor is reduced.

Table 4 demonstrates the WCRT, WCET, and WCBT of the real-time tasks of the RTSM. Analysis times are shown

in parentheses. It is beyond the scope of this paper to compare the model-based techniques of TetaSARTS with traditional techniques, see [13] for such comparisons.

RT Task	WCRT (Time)	WCET (Time)	WCBT (Time)
Per. Task 1	64.7μs (45s)	36.5μs (45s)	0μs (45s)
Per. Task 2	19.2μs (7s)	19.2μs (8s)	0μs (7s)
Spo. Task 1	4.5μs (17s)	4.5μs (8s)	0μs (17s)
Spo. Task 2	4.5μs (8s)	4.5μs (17s)	0μs (7s)

Table 4: Analysing the WCRT, WCET, and WCBT of the real-time tasks of the RTSM.

The WCBT of all tasks is $0\mu s$, thus neither will be blocked. This result implies that one could consider removing the monitors on the shared resources potentially lowering the execution times of the tasks accessing them. However, this approach changes the timing properties and hence it cannot be ensured, that there is no race condition. It would thus be necessary to complement the analysis with other analyses to verify that the system is functionally correct in terms of absence of race conditions e.g. using the tool presented in [4].

Also, analysis times of all supported analyses are reasonably low, and thus may be useful in an iterative process of development, since the effect on e.g. WCET of small changes to the code base can relatively fast be determined.

Using a real-time system consisting of two tasks, a producer and a consumer task, respectively writing and reading from a shared data structure, we demonstrate that the WCBT analysis works as shown in the results in Table 5.

RT Task	WCRT (Time)	WCET (Time)	WCBT (Time)
Producer	20.1μs ($< 1s$)	9.0μs ($< 1s$)	0.0μs ($< 1s$)
Consumer	16.5μs ($< 1s$)	11.0μs ($< 1s$)	5.5μs ($< 1s$)

Table 5: Analysing a producer-consumer real-time system.

The consumer task exhibits blocking confirming the anticipated behaviour; at its release point, the producer task has entered the critical section, thus blocking its execution.

The effect of the optimisations, shown in Figure 6, have been evaluated on the JOP execution environment. As shown, all optimisations decrease the analysis time. Especially TA inlining and JVM specialisation are of high importance and edge aggregation should be enabled whenever an abstract timing model of the execution environment is used. Using VTA for receiver type analysis is also recommended for reducing verification time and memory consumption.

System	Optimisations	Time	Memory
Class Hierarchy†	None	1m 44s	65 MB
Class Hierarchy	RTA	1m 7s	52 MB
Class Hierarchy	VTA	29s	37 MB
Simple RTS†	None	3m 59s	360 MB
Simple RTS	TA inlining + JVM special.	27s	70 MB
Seq. Computation†	None	1m 2s	167 MB
Seq. Computation	Edge aggr.	35s	70 MB

†The system without the subsequent optimisations. CHA is used for receiver type analysis.

Table 6: The effect of the optimisations.

8. CONCLUSION

In this paper, we have presented TetaSARTS; a modular and highly flexible tool for conducting timing analyses of SCJ systems. The primary purpose is schedulability analysis, but it also facilitates processor utilisation and idle time analysis, Worst Case Execution Time analysis, Worst Case Blocking Time analysis, and Worst Case Response Time analysis taking into account any pre-emption and blocking. A distinguishing feature of TetaSARTS is that the analysis supports a system model consisting of either a hardware implementation of the JVM or a software implementation of the JVM together with the underlying hardware. The execution environment model can either be *precisely* or *abstractly* represented. The former allows the actual cache and pipeline states to be simulated during instruction execution and task interleaving, thus favouring the precision of the analysis at the expense of analysis time induced by increased state space size. The latter incorporates the timing properties of the execution environment into the task model, thus reducing state space size considerably while potentially being at the expense of precision.

The enabling technique is model checking using the NTA modeling formalism of Uppaal. This paper has presented how TetaSARTS automatically constructs the timing model as an NTA from the analysed system analogously to an optimising compiler, and how the analyses are conducted using the model checker and the specification language of Uppaal.

We have demonstrated that TetaSARTS can successfully be used for analysing realistic SCJ applications such as the Real-Time Sorting Machine, the MD5 real-time system and the Minepump control system. If a precise execution environment model is used, the analysis times are high, and pushes the capabilities of model-checking as a viable technique to the extreme. In contrast, using an abstract representation, analysis times are reasonable low to facilitate frequent usage in an iterative development approach. Future work comprises investigating how to reduce analysis times even further, and make analysis of bigger systems tractable. A combination with symbolic execution is currently being explored for, among others, excluding infeasible paths.

The combination of the timing analyses facilitates multiple development approaches; if a system is not schedulable, the complementation of the other analyses can be used for debugging timing properties e.g. revealing unusual high blocking times or WCET of real-time tasks. A plug-in for the Eclipse IDE is currently under development which will facilitate this kind of use of the tool. Thus, TetaSARTS facilitates a *write once – run wherever possible* development approach where the SCJ application can be tested for schedulability on different configurations of the execution environment, and the most suited can be selected.

9. REFERENCES

[1] aJile Systems Inc., 2013. http://www.ajile.com/.

[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. In *Formal Modeling and Analysis of Timed Sys.* 2004.

[3] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.*

[4] N. Baklanova and M. Strecker. Abstraction and Verification of Properties of a Real-Time Java. 2013.

[5] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — A Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems III*, Lecture Notes in Computer Science. 1996.

[6] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. 2004.

[7] 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. *Inno. in Systems and Software Engineering*, 2013.

[8] 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 *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, 2008.

[9] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX.* Addison-Wesley Educational Publishers Inc., Boston, MA, USA, 4th edition, 2009.

[10] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proc. of the 22nd Intl. Conf. on Software Engineering*, 2000.

[11] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[12] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, 1995.

[13] 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*, 2011.

[14] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 1997.

[15] E. Hu, A. Wellings, and G. Bernat. Deriving Java Virtual Machine Timing Models for Portable Worst-Case Execution Time Analysis. *On The Move to Meaningful Internet Systems*, 2003.

[16] E. Hu, A. Wellings, and G. Bernat. XRTJ: An Extensible Distributed High-Integrity Real-Time Java Environment. *Real-Time and Embedded Computing Systems and Applications*, 2004.

[17] JSR302. The Java Community Process™Program - JSRs: Java Specification Requests - detail JSR# 302, 2011. http://www.jcp.org/en/jsr/detail?id=302.

[18] S. Korsholm. HVM, 2013. www.icelab.dk.

[19] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1997.

[20] K. S. Luckow, T. Bøgholm, and B. Thomsen. Supporting Development of Energy-Optimised Java Real-Time Systems using TetaSARTS. In *WiP Proceedings of the 19th Real-Time and Embedded Technology and Application Symposium*, 2013.

[21] M. Mikučionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, J. S. Pedersen, and P. Hougaard. Schedulability Analysis using UPPAAL: Herschel-Planck Case Study. In *Proc. of the 4th int'l conf. on Leveraging applications of formal methods, verification, and validation*, ISoLA'10, 2010.

[22] F. Pizlo, L. Ziarek, and J. Vitek. Real Time Java on Resource-Constrained Platforms with Fiji VM. In *Proc. of the 7th Intl. Workshop on Java Technologies for Real-Time and Embedded Systems*, 2009.

[23] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing Safety Critical Java Applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, 2010.

[24] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical Virtual Method Call Resolution for Java. *SIGPLAN Not.*, 2000.