# From Safety Critical Java Programs to Timed Process Models

Bent Thomsen[1], Kasper Søe Luckow[2], Lone Leth[1], and Thomas Bøgholm[1]

[1] Department of Computer Science, Aalborg University
[2] Carnegie Mellon Silicon Valley, NASA Ames

**Abstract.** The idea of analysing real programs by process algebraic methods probably goes back to the Occam language using the CSP process algebra [43]. In [16, 24] Degano et. al. followed in that tradition by analysing Mobile Agent Programs written in the Higher Order Functional, Concurrent and Distributed, programming language Facile [47], by equipping Facile with a process algebraic semantics based on true concurrency. This semantics facilitated analysis of programs revealing subtle bugs that would otherwise be very hard to find. Inspired by the idea of translating real programs into process algebraic frameworks, we have in recent years pursued an agenda of translating hard-real-time embedded safety critical programs written in the Safety Critical Java Profile [33] into networks of timed automata [4] and subjecting those to automated analysis using the UPPAAL model checker [10]. Several tools have been built and the tools have been used to analyse a number of systems for properties such as worst case execution time, schedulability and energy optimization [14, 12, 13, 19, 38, 36, 34]. In this paper we will elaborate on the theoretical underpinning of the translation from Java programs to timed automata models and briefly summarize some of the results based on this translation. Furthermore, we discuss future work, especially relations to the work in [16, 24] as Java recently has adopted first class higher order functions in the form of lambda abstractions.

## 1 Introduction

There is a growing interest in adopting Java technology in the real-time systems domain as witnessed by the large research community working on several aspects of realizing this goal. Notably, research has focused on devising appropriate real-time systems models for Java to address inherent issues such as lack of real-time tasks, high-precision clocks and a memory model not relying on (time unpredictable) garbage collection. In particular, this has led to the development of the Real-Time Specification for Java (RTSJ) [15] and the Safety Critical Java (SCJ) [33] profile.

Java is usually implemented via a translation to Java Byte Code (JBC), which is then either interpreted by a Java Virtual Machine (JVM) or further translated to native code. To accommodate the real-time execution demands of the RTSJ and SCJ programming models the underlying execution environment, the JVM, must exhibit temporal predictable behavior to allow reasoning about timeliness. One way of achieving time predictable execution of the JVM is to implement it in hardware, e.g. the aJile Systems [3] and the Java Optimized Processor (JOP) project [39]. There are also a number of software implementations of the JVM facilitating time predictable execution on time predictable commodity hardware platforms. The FijiVM [41], Hardware Near Virtual Machine (HVM) [30], JamaicaVM [2] and PicoPERC [40] are examples of this.

To address timing analysis of this environment, we have developed a tool, TetaSARTS[3], that allows the real-time system to be developed in a platform independent way. The tool is targeted at schedulability analysis of SCJ tasks taking into account a refined system model that accounts for the exact release patterns of the tasks, their relative releases, interleavings, and resource sharing. In addition, the timing model is rich enough to facilitate analysis of other properties pertaining to the verification of a real-time system including 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). TetaSARTS is the result of merging the ideas from locally developed methods for timing analysis; TetaJ [26], METAMOC [21], SARTS [14] and the TIMES [7] framework for schedulability analysis using Uppaal [22]. TetaSARTS resembles an optimizing compiler translating an SCJ system into a Network of Timed Automata (NTA) amenable to model checking. 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. It is built around a modular architecture that enables platform models to be replaced seamlessly, thereby making it possible to conduct analysis of systems running on software implementations of the JVM as well as hardware implementations of the JVM.

Although Java was not originally equipped with or designed for mathematical foundations, the theoretical underpinnings of Java have by now been explored by many researchers. In this paper we will elaborate on the theoretical underpinning of the translation from Java programs to timed automata models and briefly summarize some of the results.

The paper is organized as follows; Section 2 gives an overview of related work. Section 3 gives an overview of the Safety Critical Java programming model. Section 4 gives an overview of to implementations of the JVM supporting the SCJ programming model. Section 5 presents the theoretical model of Timed Automata. Section 6 presents an overview of the TetaSARTS tool. Section 7 presents the translation from JBC to Timed Automata and Section 8 presents our conjecture that this translation is correct. Section 9 presents various optimizations and Section 10 presents evaluation of the TetaSARTS tool. Section 11 presents the conclusions and future work, especially relations to the work in [16, 24] as Java recently has adopted first class higher order functions in the form of lambda abstractions.

[3] TetaSARTS can be downloaded at `http://people.cs.aau.dk/~luckow/tetasarts/`

## 2 Related Work

Roscoe et. al. were probably the first to analyse real programs written in the Occam language by process algebraic methods using the CSP process algebra [43]. Degano et. al. followed in that tradition by analyzing Mobile Agent Programs written in the Higher Order Functional, Concurrent and Distributed, programming language Facile [47], by equipping Facile with a process algebraic semantics based on true concurrency [16, 24]. This semantics facilitated analysis of programs revealing subtle bugs that would otherwise be very hard to find. More recently Java programs have been analyzed for correct calling order of methods using the Concurrency Workbench [29].

For analysing timing properties of systems, the traditional methods for schedulability analysis include response time analysis [17]. For each task, the response time is calculated, and the system is schedulable if the response times for the tasks are less than their respective deadlines. Tools and techniques based on the traditional method tend to be rather conservative.

The TIMES [7] tool presents a model-based, control-flow sensitive technique for schedulability analysis in which a specification for the real-time system is built as a set of tasks modeling their timing properties e.g. cost, dependencies, and deadlines. Supplementary code can be provided. This results in an NTA model which is checked using the Uppaal [10] model checker. TIMES does not perform timing analysis of the code associated with the tasks, which must be performed using external WCET analysis tools such as aiT [25], METAMOC [21], WCET Analyzer (WCA) [44] or TetaJ [26]. The aiT and METAMOC tools are targeted at timing analysis of C-programs and use respectively a combination of abstract interpretation and integer linear programming, and model checking. For Java, either WCA or TetaJ can be used. WCA makes available two techniques for timing analysis; model checking and Implicit Path Enumeration [32]. WCA, however, is targeted at the JOP [39], a JVM implementation in hardware. For dedicated schedulability analysis of Java programs, SARTS [14] can be used which also employs a model-based technique itself inspired from TIMES.

Bandera [20] is a tool for generating automata descriptions for various model checkers such as PROMELA for the SPIN [28] model checker given the program source of the Java system. Java Pathfinder (JPF) [31] can also be used for software model checking of Java real-time systems. TetaSARTS is inspired by the idea of approaching software model checking by considering the translation process from software to finite-state models as an optimising compiler.

## 3 The SCJ Real-Time Programming Model

Safety critical applications have different complexity levels. To cater for this the SCJ programming model is based on tasks grouped in missions, where a mission encapsulates a specific functionality or phase in the lifetime of the real-time system as a set of schedulable entities. The SCJ specification lets developers tailor the capabilities of the platform to the needs of the application through three compliance levels. Level 0, provides a simple, frame-based cyclic executive model which is single threaded with a single mission. Level 1 extends this model with multi-threading via periodic and aperiodic event handlers, multiple missions, and a fixed-priority preemptive scheduler (FPS). Level 2 lifts restrictions on threads and supports nested missions.

A mission encapsulates a specific functionality or phase in the lifetime of the real-time system as a set of schedulable entities. For instance, a flight-control system may be composed of take-off, cruising, and landing each of which can be assigned a dedicated mission. A schedulable entity handles a specific functionality and has release parameters describing the release pattern and temporal scope e.g. release time and deadline. The release pattern is either periodic or aperiodic.

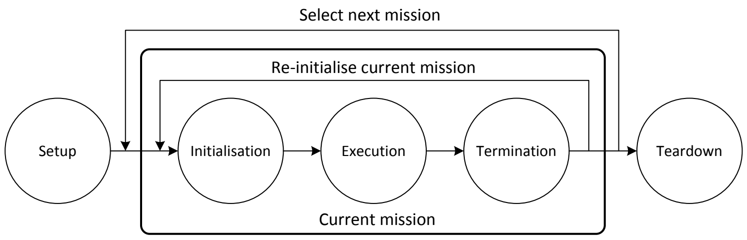

Fig. 1: Overview of the mission concept [36].

The mission concept is depicted in Figure 1 and contains five phases;

**Setup** where the mission objects are allocated during start-up of the system. This phase is not considered time-critical.

**Initialisation** where all object allocations related to the mission or to the entire applications are performed. This phase is time-critical in applications with mode changes consisting of a sequence of missions.

**Execution** during which all application logic is executed and schedulable entities are set for execution according to a pre-emptive priority scheduler. This phase is time-critical.

**Cleanup** is entered if the mission terminates and is used for completing the execution of all schedulable entities as well as performing cleanup-related functionality. After this phase, the same mission may be restarted, a new is selected, or the Teardown phase is entered. This phase is time-critical in applications with mode changes consisting of a sequence of missions.

**Teardown** is the final phase in the lifetime of the application and comprises deallocation of objects and release of locks etc. This phase is not time-critical.

SCJ introduces a memory model based on the concept of *scoped memory* from the RTSJ, which circumvents the use of a garbage collected heap during real-time execution, easing the verification of timing properties of SCJ systems.

## 4 Real-Time Execution Platforms

The SCJ programming model provides a structuring framework for applications with hard real-time requirements. Next such applications need an execution platform. For applications written in C this is usually a hardware processor. However, Java applications are typically translated into JBC which are then either interpreted or further translated into native code before execution, also called ahead-of-time (AOT) execution, or during, also called just-in-time (JIT) execution. This approach entails a time predictable implementation of each JBC.

The simplest way to ensure a time predictable execution of each Jave Bytecode is to implement the JVM in hardware. This is the approach taken by the JOP [39]. The JOP is implemented on an FPGA (Altera Cyclone EP1C6Q240 or EP1C12Q240). The JOP has its own micro code instruction set with most JBC having a one-to-one mapping. However, some are more complex and are implemented as a sequence of JOP micro codes, some are even implemented in Java. The end result is that for each JBC its execution can be bounded and its WCET be determined. Important for WCET analysis of programs executing on the JOP is that the JOP does not feature data caches, but features a method cache which must be taken into account.

The HVM [45, 30] is a lean JVM implementation intended for use in resource-constrained embedded devices with as low as 256 KB ROM and 20 KB RAM. It features both iterative interpretation, Java-to-C compilation (AOT), and a hybrid of the two. The HVM employs *JVM specialisation*; a JVM is produced specifically for hosting the JBC program of a given application. This is done using the Icecap-tools Eclipse-plugin, which analyzes the JBC program and produces an executable for the target platform. The analyses and transformations can be extended, and incorporate a number of (static) optimizations for improving performance of the JVM and for reducing its size. This includes receiver-type analysis for potentially devirtualising method calls and intelligent class linking which computes a conservative set of classes and methods that are used in the application. Only this set will be embedded in the final HVM executable. It also conservatively estimates the set of JBC that will actually be used. The HVM is self-contained and does not rely on the presence of an OS or a C standard library. The HVM has been ported to the Atmel AVR ATmega2560 microcontroller, Arduino and Lego EV3 [30].

## 5 Timed Automata

This section presents an overview of the Timed Automata formalism, based on [5, 11, 9]. A Timed Automaton is a finite state machine extended with a finite set of non-negative real-valued `clock` variables. Traditionally, vertices in the graph are called `locations`, which are connected by `edges`. The set of clocks is denoted by $C$. Clocks are distinguished from usual program variables in that their operations are limited to inspection and reset to zero. For traditional Timed Automata, clocks implicitly increase their values with rate one as time progresses, that is, if time elapses by $d$, all clocks synchronously advance by $d$. Formally, a `clock valuation` over the set of clocks, $C$, is a mapping $v : C \to \mathbb{R}_+$, where $\mathbb{R}_+$ denotes the set of non-negative reals. $\mathbb{R}_+^C$ denotes the set of all clock valuations. Then, for a valuation $v \in \mathbb{R}_+^C$ and a time delay, $d \in \mathbb{R}_+$, $v+d$ is the clock valuation that for each $c \in C$ assigns $v(c)+d$. For a set of clocks $X \subseteq C$, $v[Y]$ is the valuation assigning to each $x \in Y$ zero (i.e. it is a reset of $x$) and $v(x)$ when $x \notin Y$. A Timed Automaton can have conditions on the clock values called `guards` for edges and `invariants` for locations. In general, conditions that depend on clock values are `clock constraints` and $B(C)$ is the set of conjunctions over simple constraints of the form $x \sim c$ (or $x - y \sim c$), where $x, y \in C$, $c \in \mathbb{N}$ and $\sim \in \{<, \leq, = \geq, >\}$. When a clock constraint on an edge is satisfied, that edge is capable of being fired. Firing of an edge happens instantaneously. In locations, clock constraints are used to constrain the time spent in that location.

**Definition 1 (Timed Automaton).** *A Timed Automaton is a tuple $\mathcal{A} = \langle L, l_0, \Sigma, C, E, I\rangle$, where $L$ is a set of locations, $l_0 \in L$ is the initial location, $C$ is the set of clocks, $\Sigma$ is a set of (co-)actions (which are denoted by ! and ?, respectively) and the internal $\tau$-action, $E \subseteq L \times B(C) \times \Sigma \times 2^C \times L$ is the set of edges between locations with a guard, an action, and a set of clocks to be reset. $I : L \to B(C)$ is the map assigning to each location an invariant i.e. a clock constraint.*

In the following, $l \xrightarrow{g,a,r} l'$ denotes $\langle l, g, a, r, l'\rangle \in E$, where $l, l' \in L$, $g \in B(C)$, $a \in \Sigma$, and $r \in 2^C$. Guards and invariants will be considered as sets of clock valuations, and $v \models I(l)$ denotes that the clock valuation $v$ satisfies $I(l)$, i.e. the clock constraints representing the invariant of location $l$.

The semantics of a Timed Automaton $\mathcal{A} = \langle L, l_0, \Sigma, C, E, I\rangle$ is a timed labelled transition system $\langle S, s_0, \to\rangle$ where states are pairs $(l, v) \in S \subseteq L \times \mathbb{R}_+^C$ with $v \models I(l)$, $s_0 = (l_0, u_0)$ is the initial state, and $\to \subseteq S \times (\mathbb{R}_+ \cup A) \times S$ is the transition relation which can be either

(i) a delay transition $(l, v) \xrightarrow{d} (l, v')$ where $d \in \mathbb{R}_+$ is a delay and $v' = v + d$ if $\forall d'$ s.t. $0 \leq d' \leq d \implies v + d' \models I(l)$; or

(ii) a discrete transition $(l, v) \xrightarrow{a} (l', v')$ if there exists an edge $l \xrightarrow{g,a,Y} l'$ such that $v \models g$, $v' = v[Y]$ and $v' \models I(l')$.

Timed Automata $\mathcal{A}_1, ..., \mathcal{A}_n$ can be composed into a Network of Timed Automata using the CCS parallel composition operator, i.e, $\mathcal{A}_1 | \cdots | \mathcal{A}_n$. Let $\mathcal{A}^j = \langle L^j, l_0^j, C, A, E^j, I^j\rangle$, with $j = 1, 2, ..., n$ be a Network of $n$ Timed Automata. The location is now defined as a vector $\bar{l} = (l^1, l^2, ..., l^n)$. The notation $\bar{l}[l_i'/l_i]$ denotes the update of location vector $\bar{l}$ where the $i$th element $l_i$ is substituted by $l_i'$. The invariant functions are composed into a single function over location vectors i.e. $I(\bar{l}) = \wedge_i I_i(l_i)$. Again, the semantics of a Network of Timed Automata can be defined as a timed labelled transition system $\langle S, s_0, \to\rangle$, where states, $S$, are now defined by the set $S = (L^1 \times L^2 \times ... \times L^n) \times \mathbb{R}_+^C$, the initial state defined by $s_0 = (\bar{l_0}, v_0) \in S$, and the transition relation, $\to \subseteq S \times (\mathbb{R}_+ \cup A) \times S$, can now either be

(i) a delay transition $(\bar{l}, v) \xrightarrow{d} (\bar{l}, v')$ where $d \in \mathbb{R}_+$ is a delay and $v' = v + d$ if $\forall d'$ s.t. $0 \leq d' \leq d \implies v + d' \models I(\bar{l})$;

(ii) a discrete transition $(\bar{l}, v) \xrightarrow{a} (\bar{l}[l'_i/l_i], v')$ if there exists an edge $l_i \xrightarrow{g,a,Y} l'_i$ such that $v \models g$, $v' = v[Y]$ and $v' \models I(\bar{l'}[l'_i/l_i])$; or

(iii) a synchronisation transition $(\bar{l}, v) \xrightarrow{\tau} (\bar{l}[l'_j/l_j, l'_i/l_i], v')$ for Timed Automata $\mathcal{A}_i$ and $\mathcal{A}_j$ if there exist edges $l_i \xrightarrow{g_i, c!, Y_i} l'_i$ and $l_j \xrightarrow{g_j, c?, Y_j} l'_j$ such that $v \models g_i \wedge g_j$, $v' = v[Y_i \cup Y_j]$ and $v' \models I(\bar{l}[l'_j/l_j, l'_i/l_i])$.

Note that the above definition follows the standard definition of the CCS parallel composition operator. This will facilitate the simulation result presented later in this paper. The definition given in [11] only allows internal transitions in clause (ii) as the NTA verified by the UPPAAL model checker are closed systems and thus the parallel composition operator has an implicit hiding operator.

## 6 TetaSARTS

TetaSARTS is a fully automated tool for conducting timing analysis, such as schedulability analysis, of JBC real-time systems taking into account the particular execution environment consisting of either a software implementation of the JVM on a commodity hardware platform or a hardware implementation of the JVM. TetaSARTS employs a model-based technique for making a control-flow sensitive analysis of the JBC real-time system. It keeps a tight correspondence between the actual real-time system application code and the model used for analysis, by generating TA models amenable to model checking using Uppaal. A further benefit of using model checking is that a counterexample is provided in case the system is non-schedulable.

Two options are available for representing the execution environment: an *explicit representation* or an *inline representation*. The explicit representation incorporates the control-flow of the JBC implementations used by the specific JVM hosting the real-time system. To reflect the behavior of the JBC interpreter of the JVM, this scheme is modelled as well. Simulating the execution of the JVM is achieved by including TA models of the hardware. By using this option TetaSARTS is conducting schedulability analysis by simulating an abstract execution of the entire real-time system. This increases the overall complexity of the analysed system, but also provides the potential for more precise analysis since the dynamic behavior of e.g. caching and pipelining is accounted for. For the inline representation TetaSARTS inlines the execution times of each of the instructions in the model. These could be provided for various reasons; for JOP, the execution times are fixed, and can be found in the documentation. The inlined instruction execution times may also be available from a WCET analysis tool or from a measurement-based approach by using a stopwatch. The benefit of using an inline representation is simplicity; the dynamic behavior of the execution environment is not incorporated in the simulation, but potentially at the expense of precision, because cache-effects and timing anomalies inherent on many platforms, significantly influence instruction execution times.

TetaSARTS supports real-time tasks from SCJ with periodic or sporadic release patterns. However, it assumes that all real-time tasks are created as part of system initialisation, but future extensions will support the missing concepts from SCJ. It also supports synchronisation mechanisms such as synchronised methods in Java. The effect including synchronisation is reported in [14].

In the following sections we look at how an SCJ application is translated into a set of timed automata and how optimizations akin to those found in optimizing compilers can help reduce the model to cope with the inherent state space explosion.

How the set of program automata is combined with timed automata modeling the scheduler, sporadic and periodic task firing, the JVM and the hardware platform, forming a Network of Timed Automata (NTA) suitable for analysis with the UPPAAL model checker, is reported in [38]. Schedulability analysis can be performed by verifying that a deadlock state is never reachable within the feasibility interval [27]. This can only be the case if one or more of the real-time tasks do not finish within their deadlines. Thus schedulability is expressed by the Timed Computation Tree Logic (TCTL) specification $A\Box\ !deadlock$.

## 7 From Java Byte Code to Timed Automata

To translate an SCJ application to an NTA, the SCJ program is first compiled to JBC with a standard Java compile like `javac`. The resulting JBC forms the starting point for the transformation. The original Java source code is only used in relation to handling loops. TetaSARTS constructs an extended control flow graph (CFG) in the Timed Intermediate Representation (`TIR`) format (see below) for each method used in the system. The `TIR` is translated to a Timed Automaton for each method. These are then combined into an NTA called the `Program NTA`. The `Program NTA` captures the behavior of the system by simulating a control-flow sensitive execution of each real-time task in the system. Generating the `Program NTA` is a process composed of stages akin to those found in an optimising compiler.

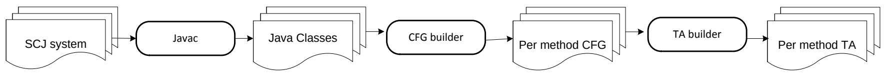

Fig. 2: From SCJ to TA

TetaSARTS initially identifies the real-time tasks of the system. With the handlers of these as entry points, TetaSARTS explores the call-graph and limits the construction of `TIR` to methods part of the reachable execution path. This reduces the overall translation time remarkably since it avoids CFG reconstruction for all methods but the relevant ones. The `TIR` is subsequently decorated with loop bound information extracted from the original source code using a comment-based approach where loop bounds are annotated using the format $//@loopbound = \langle loop\rangle$.

The output of different Java compilers including `javac`, ECJ, Jikes and GCJ, shows that all produced loop constructs are *reducible* [1], that is, they contain a single loop header that is always visited when the loop is executed. Furthermore, a reducible loop contains at least one back edge which returns control from the loop body to the loop header. To identify reducible loops, TetaSARTS employs a loop identification analysis based on the algorithm presented in [1]. When loops have been identified, extracting the loop bound from the source code is trivial since the source code line numbers are available from the JBC.

**Generating TIR** The first step in the process is, for each method used in the system, to generate the intermediate representation, `TIR`:

**Definition 2 (TIR).** `TIR` *is an extended Control-Flow Graph* $G = \langle B, L, E\rangle$ *composed of basic blocks,* $B$*, edges,* $E \subseteq B \times L \times B$*, where* $l \in L$ *decorates the CFG with extra information such as loop bounds, JVM instructions and types.*

*A basic block is a linear sequence of instructions,* $i_1, i_2, \ldots, i_n$*, that does not contain jumps nor jump targets, hence having a single entry and a single exit point. An edge,* $e = \langle b_1, l, b_2\rangle$*, connecting the two basic blocks,* $b_1$ *and* $b_2$*, denotes that a control flow path exists between the last instruction of* $b_1$ *and the first instruction of* $b_2$*. When basic blocks have been connected, the CFG is expanded with nodes/edges for each instruction in a basic block. Thus each edge in* `TIR` *is labeled with exactly one instruction.*

We also introduce the operation $succ(b) = \{b' \mid b, b' \in B \text{ and } \langle b, l, b'\rangle \in E\}$. Following the idea presented in [6] we introduce a transition system for a CFG simply by defining: $b \xrightarrow{l} b'$ *whenever* $\langle b, l, b'\rangle \in E$.

**Generating the NTA** We first introduce two sets of JBC instructions; $JBCInst$ contains all defined JBC instructions, and

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

that is, all JBC instructions used for invocation that transfer control to another method. For ease of notation, we also extend the use of *succ* to apply for use with instructions i.e.

$$succ_b(i) = \{i_{nxt} \mid \langle b, l, b'\rangle \in E \text{ and } i \in l \text{ and } \langle b', l', b''\rangle \in succ(b) \text{ and } i_{nxt} \in l'\}$$

The intuition is that $succ_b(i)$ is the set of instructions immediately following instruction $i$ in the CFG, i.e. the instructions labeling edges with origin in $succ(b)$. We omit the subscript $b$ from $succ_b(i)$ when $b$ is obvious from the context.

**Definition 3 (Left Merging TAs).** *For convenience, we define the left merging operator of two TAs,* $\lhd : TA \times TA \to TA$*:*

$$\lhd(\langle L, l_0, \Sigma, C, E, I\rangle, \langle L', l_0', \Sigma', C', E', I'\rangle) = \langle L \cup L', l_0, \Sigma \cup \Sigma', C \cup C', E \cup E', I \cup I'\rangle$$

The left merge operator is easily generalized to take a set of TA as its the second argument.

**Definition 4 (`TIR` Translation).** *Let CFG be the control-flow graph of method $m$, $chan : m \rightarrow chanName$ be the function that provides a unique channel name, $chanName$, for the method $m$, $l_0(m)$ be a unique new location for the method $m$, $l_{first}$ be the location generated by $genTA_{inst}$ for the first instruction of CFG, and similarly $l_{last}$ the location generated by $genTA_{inst}$ for the last instruction of CFG. Then*

$$TA_{CFG} = TA_{boil} \mathop{\lhd}_{\substack{i \in b \\ b \in CFG}} genTA_{inst}(i)$$

*where* $TA_{boil} = \langle \{l_0, l_{first}, l_{last}\}, l_0, \{chan(m)!, chan(m)?\}, C, E, \emptyset \rangle$,
*with* $E = \{l_0 \xrightarrow{chan(m)?} l_{first}, l_{last} \xrightarrow{chan(m)!} l_0(m)\}$,
*and* $C = \begin{cases} \{execTime\} & if \;\; Inline\ representation\ is\ used \\ \emptyset & if \;\; Explicit\ representation\ is\ used \end{cases}$

and where *execTime* is used for monitoring the inlined instruction execution times.

Generating the TA stubs for JBC instructions is parameterised on the particular type of that instruction such that

$$genTA_{inst}(i) = \begin{cases} genTA_{call}(i) & if\ i \in CallInst \\ genTA_{sim}(i) & if\ i \in JBCInst \setminus CallInst \end{cases}$$

That is, $genTA_{call}$ generates the TA stub of JBC instructions that invokes methods, whereas $genTA_{sim}$ generates the TA of all other JBCs.

$genTA_{call}$ and $genTA_{sim}$ are further parameterised depending on whether the execution environment is explicitly modelled or inlined in the `Program NTA` with static instruction execution times and without a `JVM NTA` and a `Hardware NTA`.

$$genTA_{sim}(i) = \begin{cases} genTA_{sim_{in}} & \text{if inline representation is used} \\ genTA_{sim_{exp}} & \text{if explicit representation is used} \end{cases}$$

$$genTA_{call}(i) = \begin{cases} genTA_{call_{in}} & \text{if inline representation is used} \\ genTA_{call_{exp}} & \text{if explicit representation is used} \end{cases}$$

We also define the auxiliary function $loc : TA \rightarrow Location$ that returns the initial location associated with a TA stub generated for an instruction, $edge : TA \rightarrow Edge$ that returns the outgoing edges of the initial location of a generated TA stub for an instruction, $sync : Edge \rightarrow chanName$ that returns the channel name for an edge, and $Callees : i \rightarrow M$ where $i \in CallInst$ that provides the set of potential receivers of a method call. Translating `TIR` to a `Program NTA` used along with an explicit representation of the execution environment is performed according to Definition 5.

**Definition 5 (Explicit Representation Translation).**
*For translating simple instructions, we use*

$$genTA_{sim_{exp}} : Instruction \rightarrow TA = \langle L, l_0, \Sigma, C, E, I \rangle$$

*which is defined as:*

$$genTA_{sim_{exp}}(i) = \langle \{l_i\}, l_i, \{jvm\_exec!\}, \emptyset, E, \emptyset \rangle$$

*where*

$$E = \bigcup_{\substack{\forall i_{nxt} \in \\ succ(i)}} \left\{ \left\langle l_i \xrightarrow{running[tID], jvm\_exec!, jvm\_inst := [\![i]\!]} loc(genTA_{instr}(i_{nxt})) \right\rangle \right\}$$

*Method calling instructions are translated using*

$$genTA_{call_{exp}} : Instruction \rightarrow TA = \langle L, l_0, \Sigma, C, E, I \rangle$$

*which is defined as:*

$$\begin{aligned} genTA_{call_{exp}}(i) \quad =& \langle \{loc(genTA_{sim_{exp}}(i)), l_{call}, l_{wait}, l_{ret}\}, loc(genTA_{sim_{exp}}(i)), \\ & \{jvm\_exec!\} \cup \{a!, a? | a \in chan(callees(i))\}, \emptyset, E, I \rangle \end{aligned}$$

*where*

$$\begin{aligned} E = & \; edge(genTA_{sim_{exp}}(i)) \\ & \bigcup_{\substack{\forall M \in \\ callees(i)}} \left\{ \left\langle l_{call} \xrightarrow{running[tID], chan(M)!} l_{wait} \right\rangle \right\} \\ & \bigcup_{\substack{\forall M \in \\ callees(i)}} \left\{ \left\langle l_{wait} \xrightarrow{running[tID], chan(M)?} l_{ret} \right\rangle \right\} \\ & \cup \left\{ \left\langle l_{ret} \xrightarrow{urgent} loc(genTA_{instr}(i_{nxt})) \right\rangle \right\} \end{aligned}$$

*and*

$$I = \{ \langle l_{call}, execTime == 0 \rangle, \langle l_{ret}, execTime == 0 \rangle \}$$

In Definition 5, the guard as generated by $genTA_{sim_{exp}}$, ensures that the edge can only be fired if the real-time task with ID $tID$ is set to run as governed by the scheduler. The *urgent* label means that the edge is fired immediately; when being in $l_{ret}$, time is not allowed to progress and the edge is fired instantaneously. The update statement is used for communicating the instruction, $i$, to the `JVM NTA`. Furthermore, to initiate the simulation of $i$, a synchronisation action is initiated on the *jvm_exec* channel. Whenever the `JVM NTA` is capable of processing a new instruction, it receives on *jvm_exec*. The TA stub generated by $genTA_{call_{exp}}$ makes a non-deterministic choice between all possible receivers of the call by generating an outgoing edge with a synchronisation action to the respective TA simulating the receiver. Afterwards, the process waits in $l_{wait}$ until the simulation of the callee finishes at which point the process synchronises on the same synchronisation channel, transferring control back to the caller.

For generating the `Program NTA` for use with an inline representation of the execution environment, we add the function $wcet : i \to \mathbb{N}$ that returns the statically defined WCET for instruction $i$ on the particular execution environment. The translation is performed according to Definition 6.

**Definition 6 (Inline Representation Translation).**
*Translating simple instructions is done using*

$$genTA_{sim_{in}} : Instruction \to TA = \langle L, l_0, \Sigma, C, E, I \rangle$$

*which is defined as:*

$$genTA_{sim_{in}}(i) = \langle \{l_i\}, l_i, \emptyset, \emptyset, E, I \rangle$$

*where*

$$E = \bigcup_{\substack{\forall i_{nxt} \in \\ succ(i)}} \left\{ \left\langle l_i \xrightarrow{execTime == [\![wcet(i)]\!], execTime := 0} loc(genTA_{instr}(i_{nxt})) \right\rangle \right\}$$

*and*

$$I = \{\langle l_i, execTime \leq [\![wcet(i)]\!] \ \&\& \ execTime' == running[tID] \rangle\}$$

*Translating method calling instructions is done using*

$$genTA_{call_{in}} : Instruction \to TA = \langle L, l_0, \Sigma, C, E, I \rangle$$

*which is defined as:*

$$\begin{aligned} genTA_{call_{in}}(i) = & \langle \{loc(genTA_{sim_{in}}(i)), l_{wait}\}, loc(genTA_{sim_{in}}(i)), \\ & \{a!, a? | a \in chan(callees(i))\}, \emptyset, E, I \rangle \end{aligned}$$

*where*

$$\begin{aligned} & E = edge(genTA_{sim_{in}}(i)) \\ & \bigcup_{\substack{\forall M \in \\ callees(i)}} \left\{ \left\langle loc(genTA_{sim_{in}}(i) \xrightarrow{execTime == [\![wcet(i)]\!], chan(M)!, execTime := 0} l_{wait} \right\rangle \right\} \\ & \bigcup_{\substack{\forall M \in \\ callees(i)}} \left\{ \left\langle l_{wait} \xrightarrow{chan(M)?, execTime := 0} loc(genTA_{instr}(succ(i)) \right\rangle \right\} \end{aligned}$$

Translation to an inline representation follows the same pattern as that for an explicit representation. The notable difference is the inclusion of the instruction execution times on the edges.

## 8 Correctness of Translation

In this section we conjecture that the translation of an SCJ application is correct. The correctness is stipulated through a simulation relation between `TIR` and `Program NTA`, relying on results from[18] proving the correctness of the translation from Java to JBC and [6] proving simulation between JCB and CFG.

*Conjecture 1.* For each method $m$ in an SCJ application, the `TIR` representation of method $m$ is in a simulation relation with the TA generated for $m$ using Definition 4 and the Explicit Representation Translation in Definition 5.

A proof of the above conjecture will follow the lines of [6]. There are two cases: (1) The CFG of a method $m$ can do a transition $b \xrightarrow{l} b'$ *whenever* $\langle b, l, b' \rangle \in E$ where $i \in l$ and $i \in sim_{exp}$, then $genTA_{sim_{exp}}(i) \xrightarrow{running[tID],jvm\_exec!,jvm\_inst:=[\![i]\!]} l_{loc}$ where $l_{loc} \in loc(genTA_{instr}(i_{nxt}))$.
(2) The CFG of a method $m$ can do a transition$b \xrightarrow{l} b'$ *whenever* $\langle b, l, b' \rangle \in E$ where $i \in l$ and $i \in call_{exp}$ then $genTA_{call_{in}}(i) \xrightarrow{running[tID],jvm\_exec!,jvm\_inst:=[\![i]\!]} l_{call} \xrightarrow{running[tID],chan(M)!} l_{wait} \xrightarrow{running[tID],chan(M)?} l_{ret} \xrightarrow{urgent} l_{loc}$ where $l_{loc} \in loc(genTA_{instr}(i_{nxt}))$.

*Conjecture 2.* For each method $m$ in an SCJ application, the Program TA generated for $m$ using Definition 4 and the Explicit Representation Translation in Definition 5 is in a simulation relation with the Program TA generated for $m$ using Definition 4 and the Implicit Representation Translation in Definition 6.

A proof of the above conjecture will establish a simulation between the Explicit Representation Translation and the Implicit Representation Translation, noting that when $genTA_{sim_{exp}}(i) \xrightarrow{running[tID],jvm\_exec!,jvm\_inst:=[\![i]\!]} l_{loc}$ where $l_{loc} \in loc(genTA_{instr}(i_{nxt}))$ then $genTA_{sim_{in}}(i) \xrightarrow{execTime==[\![wcet(i)]\!],execTime:=0} l'_{loc}$ where $l'_{loc} \in loc(genTA_{instr}(i_{nxt}))$ for $i \in sim_{exp}$, and similarly matching transitions can be found for $i \in call_{exp}$.

## 9 Analyses and Optimisations

To cope with the inherent problem of state space explosion, our method adopts a variety of analyses, optimisations, and transformations to reduce the size of each state and the state space that needs exploration. All transformations and optimisations are incorporated without affecting the soundness of our method.

**Inlining TAs** UPPAAL uses the CCS parallel composition operator for allowing interleaving of actions as well as allowing hand-shake synchronisations. For the parallel composition, $\mathcal{A}_1 \parallel \mathcal{A}_2 \parallel \cdots \parallel \mathcal{A}_n$, the product TA necessarily has to be constructed. This is entirely syntactical, but turns out to be computationally expensive which is the reason why UPPAAL computes the product TA on-the-fly during verification. To lower the verification time even more, TETASARTS inlines TAs wherever possible prior to the verification to reduce the size of the product TAs. Inlining TAs involves a series of steps. First a TA dependency graph is built over how TETASARTS simulates invocation of methods using synchronisation channels. Thus Definition 7 only applies for the modeling approach adopted in TETASARTS.

**Definition 7 (TA Dependency Graph).**
*A TA dependency graph $G = \langle V, E\rangle$ is a DAG where the vertices, $V$, represent the TAs of the NTA system, and edges, $E \subseteq V \times V$ represent that a dependency exists between two TAs. Let $\mathcal{A}_i$ where $i \in \{1, 2\}$ be two TAs and let $E_{\mathcal{A}_i}$ denote the set of edges in TA $\mathcal{A}_i$. $C$ denotes an arbitrary synchronisation channel. A dependency among $\mathcal{A}_1$ and $\mathcal{A}_2$ is created when there exists two edges, $\{e_{\mathcal{A}_i}, e'_{\mathcal{A}_i}\} \in E_{\mathcal{A}_i}$ where $i \in \{1, 2\}$ if*

$$e_{\mathcal{A}_1} = \langle l_{\mathcal{A}_1} \xrightarrow{g_{\mathcal{A}_1}, C!, u_{\mathcal{A}_1}, r_{\mathcal{A}_1}} l'_{\mathcal{A}_1}\rangle \qquad e'_{\mathcal{A}_1} = \langle l'_{\mathcal{A}_1} \xrightarrow{C?} l''_{\mathcal{A}_1}\rangle$$
$$e_{\mathcal{A}_2} = \langle l_{\mathcal{A}_2} \xrightarrow{g_{\mathcal{A}_2}, C!, u_{\mathcal{A}_2}, r_{\mathcal{A}_2}} l'_{\mathcal{A}_2}\rangle \qquad e'_{\mathcal{A}_2} = \langle l'_{\mathcal{A}_2} \xrightarrow{C?} l''_{\mathcal{A}_2}\rangle$$

*where $\{e_{\mathcal{A}_1}, e'_{\mathcal{A}_1}, e_{\mathcal{A}_2}, e'_{\mathcal{A}_2} \mid sync(e) = C$ where $e \in E_{\mathcal{A}_1} \cup E_{\mathcal{A}_2}\}$, that is $C$ is a channel only appearing on edges $e_{\mathcal{A}_1}, e'_{\mathcal{A}_1}, e_{\mathcal{A}_2}$ and $e'_{\mathcal{A}_2}$ in $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Assume that a dependency exists between the TAs $\mathcal{A}_1$ and $\mathcal{A}_2$ due to the existence of edges $e_{\mathcal{A}_1}$, $e'_{\mathcal{A}_1}$, $e_{\mathcal{A}_2}$, $e'_{\mathcal{A}_2}$ whose structure follows the definitions in Definition 7. A new TA $\mathcal{A}_{in}$ is created such that $\mathcal{A}_{in} = \mathcal{A}_1 \lhd \mathcal{A}_2$ except that $\{e_{\mathcal{A}_1}, e'_{\mathcal{A}_1}, e_{\mathcal{A}_2}, e'_{\mathcal{A}_2}\} \notin E_{\mathcal{A}_{in}}$. In addition, two new edges are added to $E_{\mathcal{A}_{in}}$:

$$e_{init} = \langle l'_{\mathcal{A}_1} \xrightarrow{g_{\mathcal{A}_1}, \tau, u_{\mathcal{A}_1}} l''_{\mathcal{A}_2}\rangle \quad e_{ret} = \langle l'_{\mathcal{A}_2} \xrightarrow{g_{\mathcal{A}_2}, \tau, u_{\mathcal{A}_2}} l''_{\mathcal{A}_1}\rangle$$

If the option of inlining the instruction execution times in the `Program NTA` is enabled, TETASARTS is capable of reducing the state space by aggregating edges that are fired sequentially according to Definition 8.

**Definition 8 (Sequentially Executing Instructions).**
*Let $i_1, i_2, \ldots, i_n$ be the sequence of instructions following an execution path in the program. $i_1, i_2, \ldots, i_n$ are* sequentially executing *if $\forall i_k$ s.t. $1 \leq k \leq n$ then $\mid succ(i_k) \mid = 1$*

Edge aggregation is now performed according to Definition 9.

**Definition 9 (Edge Aggregation).** *Let SeqInst be a set of sequentially executing instructions according to Definition 8. The total execution of SeqInst is then $aggWCET = \sum_{i \in SeqInst} wcet(i)$.*

*Let $SeqLoc = \{l_i \mid i \in SeqInst\}$ and let SeqEdges denote the set of edges with source location $l$ s.t. $l \in SeqLoc$ and let $\mathcal{A}$ be the TA with locations $L$ s.t.*

*$SeqLoc \subseteq L$ and edges $E$ s.t. $SeqEdges \subseteq E$. $\mathcal{A}$ is updated s.t. $L = L \setminus SeqLoc$ and $E = E \setminus SeqEdges$*

*Let $l$ and $l'$ denote the first and last location in $SeqLoc$. $e_{agg}$ is a new edge s.t. $e_{agg} = \langle l \xrightarrow{execTime==[\![aggWCET]\!]} l' \rangle$. Furthermore, the invariants of $\mathcal{A}$ are updated s.t. $\langle l, execTime \leq [\![aggWCET]\!]$ && $execTime' == running[tID] \rangle$*

**JVM NTA Specialisation** Many embedded systems do not use floating point arithmetic hence leaving out all the JBCs that handle doubles and floats. Moreover, many other JBCs are only rarely used. Due to this, our method employs an analysis that conservatively estimates the set of JBCs the program is actually using. The analysis traverses `TIR` and visits every instruction $i$. Whenever an instruction $i$ is visited such that $i \notin JBCInst_{used}$, it is added to $JBCInst_{used}$. All $TA_i$ such that $i \notin JBCInst_{used}$ are removed from the final NTA.

**Devirtualisation** 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 possible callees which, in cases with large class hierarchies, contributes significantly to the size of the state space.

TetaSARTS employs static program analyses known from optimising compilers to devirtualise virtual method calls or at least limit the amount of possibilities of dynamically-dispatched methods. The methods are used when invoking the *callees* function previously introduced. TetaSARTS makes available different approaches since the precision of devirtualisation comes at the cost of increased NTA generation time:

*Class Hierarchy Analysis (CHA)* considers 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 denoted $C$ and has subtypes $\{S_1, S_2, \ldots, S_n\}$, then only $C$ and the subtypes that override $m$ will be considered. [23]

*Rapid Type Analysis (RTA)* is an extension to CHA which combines the information about globally instantiated types and intersects it with the class hierarchy information about the callsite as obtained by CHA. [8]

*Variable Type Analysis (VTA)* makes a conservative estimate of the set of types that may possibly reach each variable in methods. [46]

## 10 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 a machine with an Intel Xeon X5670 @ 2.93 GHz and 32 GB of memory. The systems are:

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

**Sequential Computation** is composed of ten real-time tasks performing calculations using only a few conditional JBCs. This is used for demonstrating the effect of edge aggregation.

**Simple RTS** consists of nine real-time tasks performing calculations using only a few different JBCs. 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 level and methane concentration.[17][12][26]

**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.[14]

**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 [42].

For evaluating the effect of the optimisations, we have used an inline representation of the JOP execution environment. The results are shown in Table 1. As shown, all optimisations decrease the analysis time significantly. Especially

| System | Optimisations | Analysis Time | Mem. Usage |
|---|---|---:|---:|
| Class Hierarchy | CHA | 1m 44s | 65 MB |
| Class Hierarchy | RTA | 1m 7s | 52 MB |
| Class Hierarchy | VTA | 29s | 37 MB |
| Simple RTS | All | 27s | 70 MB |
| Simple RTS | No TA inlining<br>No JVM special. | 3m 59s | 360 MB |
| Seq. Computation | All | 35s | 70 MB |
| Seq. Computation | No edge aggr. | 1m 2s | 167 MB |

Table 1: The effect of the optimisations.

inlining TAs and JVM specialisation are evidently of high importance. Edge aggregation is also important and should be enabled whenever an inline representation of the execution environment is used. Using VTA for devirtualisation is also recommended. As shown, having an exact representation of the execution environment yields long verification times and high memory demands. This was also anticipated due the number of TAs and their complexity. Further results can be found in [37, 35].

Table 2 shows the results of analysing representative examples of real-time systems. The subscript indicates whether an explicit or an inline representation of the execution environment is used.

| System | Exec. Env. | Analysis Time | Mem. Usage |
|---|---|---|---|
| RTSM | $JOP_{in}$ | 11s | 19 MB |
| RTSM | $JOP_{exp}$ | 17m 19s | 166 MB |
| Minepump | $JOP_{in}$ | 1s | 12 MB |
| Minepump | $JOP_{exp}$ | 6m 18s | 62 MB |
| Minepump | $HVMAVR_{exp}$ | 15h 25m 16s | 17933 MB |

Table 2: Results obtained using TetaSARTS.

## 11 Conclusion

In this paper we have given an overview of the Safety Critical Java Profile and its programming model based on tasks grouped in missions, encapsulating a specific functionality or phase in the lifetime of a hard real-time system as a set of schedulable entities. We have given an overview of two execution platforms for SCJ, the JOP [39] and the HVM [45, 30] and we have given an overview of the TetaSARTS tool for conducting schedulability analysis of JBC real-time systems which is able to take into account the particular execution environment consisting of either a software implementation of the JVM and commodity embedded hardware or a hardware implementation of the JVM. TetaSARTS keeps a tight correspondence between the actual real-time system application code and the model used for analysis. We have briefly summarized some of the results based on this translation.

The main contribution is an elaboration on the theoretical underpinning of the translation from Java programs to timed automata models conjecturing that a simulation relation can be established between CFGs of methods in the system and their representations as TA. We conjecture the overall correctness by transitivity, relying on results from [18] proving the correctness of the translation from Java to JBC and from [6] proving simulation between JCB and CFG.

Our approach of analyzing real programs by process algebraic methods follows in the footsteps of [43] where programs in the Occam language are analyzed using the CSP process algebra and [16, 24] where Degano et. al. analyzed Mobile Agent Programs written in the Higher Order Functional, Concurrent and Distributed, programming language Facile [47].

Recently the Java language has been enhanced with anonymous higher order functions in the form of lambda abstractions. This makes Java a full-fledged higher order object oriented, functional and concurrent language. Furthermore, even embedded JVM platforms, such as JOP and HVM now have support for multi-core, and thus some level of true code migration incorporated. Thus we expect that the work in [16, 24] will become extremely relevant in the analysis of systems for the Internet-of-Things, as Java moves into this territory.

## References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Pearson Education, 2006.
2. Aicas. *JamaicaVM User Manual: Java Technology for Critical Embedded Systems*, 2010.
3. aJile Systems. Website. Last accessed 31 August 2014.
4. Rajeev Alur. Timed Automata. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 688–688. Springer Berlin Heidelberg, 1999. 10.1007/3-540-48683-6_3.
5. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
6. Afshin Amighi, Pedro de Carvalho Gomes, Dilian Gurov, and Marieke Huisman. Provably correct control flow graphs from Java bytecode programs with exceptions. *International Journal on Software Tools for Technology Transfer*, pages 1–32, 2015.
7. Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *the 1st International Workshop on Formal Modeling and Analysis of Timed Systems*, May 2003.
8. David F. Bacon and Peter 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*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM.
9. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*, volume 26202649. The MIT Press, 2008.
10. Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal – a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III : Verification and Control: Verification and Control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.
11. Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer Berlin Heidelberg, 2004.
12. Thomas Bøgholm, Christian Frost, René Hansen, Casper Jensen, Kasper Luckow, Anders Ravn, Hans Søndergaard, and Bent Thomsen. Towards harnessing theories through tool support for hard real-time java programming. *Innovations in Systems and Software Engineering*, 9(1):17–28, 2013.
13. Thomas Bøgholm, René R. Hansen, Anders P. Ravn, Bent Thomsen, and Hans Søndergaard. A Predictable Java Profile: Rationale and Implementations. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 150–159, 2009.
14. Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim Guldstrand 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*, JTRES '08, pages 106–114, 2008.
15. G. Bollella. *The Real-time Specification for Java.* Addison-Wesley Java Series. Addison-Wesley, 2000.

16. Roberta Borgia, Pierpaolo Degano, Corrado Priami, Lone Leth, and Bent Thomsen. Understanding mobile agents via a non-interleaving semantics for Facile. In *Static Analysis*, pages 98–112. Springer, 1996.
17. Alan Burns and Andy 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.
18. Egon Börger and Wolfram Schulte. Defining the Java Virtual Machine as platform for provably correct Java compilation. In Luboš Brim, Jozef Gruska, and Jiří Zlatuška, editors, *Mathematical Foundations of Computer Science 1998*, volume 1450 of *Lecture Notes in Computer Science*, pages 17–35. 1998.
19. T. Bøgholm, B. Thomsen, K.G. Larsen, and A. Mycroft. Schedulability Analysis Abstractions for Safety Critical Java. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on*, pages 71–78, april 2012.
20. J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448, 2000.
21. Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *10th International Workshop on Worst-Case Execution Time Analysis*, 2010.
22. Alexandre David, Jacob Illum, Kim Larsen, and Arne Skou. *Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1*, pages 93–119. CRC Press, 2009.
23. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.
24. Pierpaolo Degano, Corrado Priami, Lone Leth, and Bent Thomsen. Causality for debugging mobile agents. *Acta informatica*, 36(5):335–374, 1999.
25. C. Ferdinand. Worst case execution time prediction by static program analysis. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 125. IEEE, 2004.
26. Christian Frost, Casper Svenning Jensen, Kasper Søe Luckow, and Bent Thomsen. WCET analysis of Java bytecode featuring common execution environments. In *9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2011.
27. J. Goossens and R. Devillers. The Non-Optimality of the Monotonic Priority Assignments for Hard Real-Time Offset Free Systems. *Real-Time Systems*, 13:107–126, 1997.
28. Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.
29. Marieke Huisman and Dilian Gurov. CVPP: A Tool Set for Compositional Verification of Control–Flow Safety Properties. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software*, volume 6528 of *Lecture Notes in Computer Science*, pages 107–121. Springer Berlin Heidelberg, 2011.
30. HVM (Hardware near Virtual Machine). Website. Last accessed 31 August 2014.
31. JPF. Java PathFinder Tool-set, 2014. `http://babelfish.arc.nasa.gov/trac/jpf`.

32. Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.
33. Doug Locke, B. Scott Andersen, Ben Brosgol, Mike Fulton, Thomas Henties, James J. Hunt, Johan Olmütz Nielsen, Kelvin Nilsen, Martin Schoeberl, Joyce Tokar, Jan Vitek, and Andy Wellings. *Safety-Critical Java Technology Specification, Public draft.* 2013.
34. 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*, pages 41–44, 2013.
35. K. S. Luckow, T. Bøgholm, B. Thomsen, and K. G. Larsen. TetaSARTS: Modular Timing and Performance Analysis of Safety Critical Java Systems. *Concurrency and Computation: Practice & Experience*, 2014. In submission.
36. K. S. Luckow, B. Thomsen, and S. E. Korsholm. HVM-TP: A Time Predictable and Portable Java Virtual Machine for Hard Real-Time Embedded Systems. In *12th International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2014. To appear.
37. Kasper Søe Luckow. *Platforms and Model-Based Analyses for Real-Time Java.* PhD thesis, Department of Computer Science, Aalborg University, 2014, `http://people.cs.aau.dk/~luckow/thesis.pdf`.
38. Kasper Søe Luckow, Thomas Bøgholm, Bent Thomsen, and Kim Guldstrand Larsen. TetaSARTS: A Tool for Modular Timing Analysis of Safety Critical Java Systems. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, pages 11–20, 2013.
39. Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems.* Number ISBN 978-3-8364-8086-4. VDM Verlag Dr. Müller, 2008.
40. Kelvin Nilsen. Differentiating features of the PERC virtual machine, 2009.
41. Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real Time Java on Resource-constrained Platforms with Fiji VM. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 110–119, New York, NY, USA, 2009. ACM.
42. Ales Plsek, Lei Zhao, Veysel H. Sahin, Daniel Tang, Tomas Kalibera, and Jan 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, pages 95–101, New York, NY, USA, 2010. ACM.
43. Andrew William Roscoe and Charles Antony Richard Hoare. The laws of Occam programming. *Theoretical Computer Science*, 60(2):177–229, 1988.
44. Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-Case Execution Time Analysis for a Java Processor. *Software: Practice and Experience*, 40(6):507–542, 2010.
45. Hans Søndergaard, Stephan E. Korsholm, and Anders P. Ravn. Safety-Critical Java for Low-End Embedded Platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 44–53, New York, NY, USA, 2012. ACM.
46. Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280, October 2000.
47. Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In *CONCUR'96: Concurrency Theory*, pages 278–298. Springer, 1996.