Model-Based Schedulability Analysis of Hard Real-Time Java Programs using Software Transactional Memory

10th Term Software Engineering Project

Department:
Database and Programming Technologies

Authors:
Marcus Calverley
Anders Christian Sørensen

Department of Computer Science
Aalborg University

Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

Title

Model-Based Schedulability Analysis of Hard Real-Time Java Programs using Software Transactional Memory

Department

Database and Programming Technologies

Project term

Spring 2012

Project group

sw103f12

Supervisors

Lone Leth Thomsen and Bent Thomsen

Attachments

CD-ROM with source code, UPPAAL models, and PDF-version of this report.

Abstract

This report documents our work in developing a software transactional memory (STM) for real-time Java, which assigns priorities to transactions based on the tasks in which they execute. Contention is managed in such a way that the highest priority transaction does not experience retries, allowing for irreversible actions such as I/O, which is otherwise impossible using traditional STM. These properties are proven using the model checking tool UPPAAL.

The Hardware-near Virtual Machine (HVM) is chosen as the target platform due to its portability and flexibility, but it does not support multi-threading in its current state. To alleviate this, we have implemented real-time multi-threading through an implementation of Safety-Critical Java named SCJ2.

In order to determine schedulability of real-time Java programs using our STM and SCJ2, we have developed OSAT, which is a model-based schedulability analysis tool. It analyses the Java bytecode of the input program, and produces a UPPAAL model which can be used to verify schedulability. We have conducted experiments with our tool, comparing it to a similar existing model-based schedulability analysis tool named SARTS. We also compare the use of locks and our STM in a real-time setting, showing their advantages and disadvantages.

Participants

Marcus Calverley and Anders Christian Sørensen

Preface

This report assumes the reader already has knowledge pertaining to the programming languages Java and C. In addition, we expect familiarity with real-time systems theory and notation, namely regarding tasks, types of tasks, scheduling, WCET and response time analysis, although we do provide a brief recap of these. UPPAAL [1] is used extensively throughout this report, and although we give a quick introduction to the tool, a rudimentary understanding of model checking is expected.

Whenever we refer to a *programmer* in this report, we mean the person who uses the programming tools described, e.g. our analysis tool.

Enclosed with this report is a CD containing the source code developed in this project, along with the full generated UPPAAL models used in our experiments. The contents of the CD can also be found at `http://sw10.lmz.dk`.

We would like to thank our supervisors Lone Leth Thomsen and Bent Thomsen for their in-depth feedback during this project. Stephan Korsholm, the creator of the Hardware-near Virtual Machine (HVM) used in this project, also deserves thanks for his extensive support in helping us understand the inner workings of the HVM, as well as Kasper Søe Luckow who provided additional insights into the HVM and the inner workings of the tool TetaJ on which we base much of our work.

Contents

Chapter 1

Introduction

Embedded real-time software (RTS) is a class of software which drives an increasingly large amount of electronic devices, such as the anti-lock brakes in motor vehicles, pacemakers, and anti-collision detection systems in airplanes. *Real-time* refers to the notion of computations with timing constraints, i.e. computations are required to finish within a given timeframe. Fulfilling the timing requirements set up is crucial to the system and failure to do so could have catastrophic consequences, such as the failure of the brakes on a car causing it to crash.

In RTS, schedulability analysis is used to ensure that the application running on the system will, even under the worst circumstances, always fulfil the timing requirements. Schedulability analysis involves determining the time it takes to run the code of the system on the given platform. The code can be logically separated in tasks that run at certain times, e.g. every 100 ms or when the brake pedal of a car is pressed. When there are several tasks in a real-time system, these can be run as separate threads that may run concurrently. Concurrency in real-time systems may make schedulability analysis more complicated, as this means that a high priority task may preempt a low priority task, thus suspending the execution of the low priority task for the duration of the high priority task.

Schedulability analysis can be further complicated if the tasks need to communicate with each other. Shared memory is a communication method that allows threads to communicate by reading from and writing to memory that is accessible to all threads. However, one is often interested in working with a consistent snapshot of the parts of the shared memory that are needed to be able to write code that works even when another thread interrupts the execution at a critical point and changes something. To this end, lock-based mechanisms are a common way to ensure mutually exclusive access to resources where needed [2, 3]. Our experience and literature will tell us lock-based mechanisms are inherently error-prone since they are manually put in place by the programmers, they will not scale with the number of cores if not implemented carefully, they are difficult or impossible to combine, and a frequent cause for deadlocks and priority inversion [4, 3].

Recently, software transactional memory (STM) has gained interest as an abstraction for concurrency control in shared memory [3, 4, 5, 6, 7]. This allows the programmer to focus on the domain of interest rather than where to place locks to get the correct result of the computation but still achieve high concurrency. Thus STM can eliminate deadlocks, priority inversion, and other common problems that might arise when using lock-based concurrency.

However, STM introduces a new complication to schedulability analysis when used in real-time systems. The first few steps towards bringing STM to RTS have already been taken [8, 9, 10]. In this project, we create an STM that is schedulable in a real-time context in the programming language Java. Java is a high-level programming language that has been chosen for the project over other programming languages because it is gaining increased support in real-time systems, which means it is now a viable alternative to the lower level languages traditionally used for real-time systems, such as C or Ada [11]. Java is also familiar to us, which means we have a solid foundation for its use in this project.

Aside from the programming language, we also need a platform on which to run our Java code. A recent virtual machine for Java bytecode is the Hardware-near Virtual Machine (HVM) that supports several popular embedded platforms such as Atmel ATmega, National Semiconductor CR16C, and even allows running the code on a PC. [12] Using the HVM as our target system, we can thus potentially reach a range of target platforms on the market today, and furthermore the HVM is easily extensible and open source. We have chosen this platform over the JOP [13] which, although better suited for RTS, has fewer features that are useful in this project.

1.1 Problem Statement

STM provides many benefits to programmers and also has benefits especially interesting to hard real-time systems. However, STM introduces a new challenge when proving schedulability of programs. In this project, we want to implement an STM with real-time properties on the HVM, create an analysis tool that can determine if a set of tasks written in Java and containing transactions that use our STM is schedulable. We also want to prove that our STM is correct using model-based verification and test our tool through a series of experiments.

1.2 Subsidiary Goals

To elaborate on how we will accomplish the tasks set in our problem statement, we have devised a number of goals to use as milestones through the project period:

Development Process

- Select suitable development methods for the project.

Real-Time Systems

- Define real-time systems and their properties.
- Determine what is required to use the HVM in the project.

Software Transactional Memory

- Design a real-time STM for the HVM.
- Prove the correctness of our STM using model-based verification.
- Implement the STM on the HVM.

Schedulability Analysis Tool

- Design a tool that can ascertain schedulability of multi-threaded HVM programs using our STM and locks.
- Decide whether or not to use an existing tool upon which to base our tool.
- Implement the tool for use on a PC.
- Test the tool by conducting experiments on programs using our STM and locks.
- Use the tool to compare lock-based synchronisation with our STM.

1.3 Report Structure

The report is structured as follows: we begin by looking at the development process employed throughout the project period in Chapter 2, followed by a brief recap of real-time system concepts used in this report in Chapter 3. In Chapter 4, we introduce the technologies used in the project, including the concepts of STM. Our use of the HVM and the modifications we have made to it to allow its use in this project are described in Chapter 5, followed by a description of the real-time STM we have developed in Chapter 6. The schedulability analysis tool we have developed, which allows analysis of programs using this STM, is described in Chapter 7. In Chapter 8, we describe the experiments conducted with the developed tool to show its properties and properties of our STM. Finally, we give an evaluation of our work and possible directions for future work in Chapter 9, before concluding on the project in Chapter 10.

Chapter 2

Development Process

The challenges in this project were manifold. They involved the following activities: verification of proposed software transactional memory properties suitable for hard real-time systems, implementation of such properties, extending a platform to support concurrency, and developing a schedulability analysis tool for Java programs utilising these technologies.

In order to organise our work with these challenges, we tailored a development process consisting of methods from the risk-driven domain, model-driven domain, and agile methodologies in general. These methods and their applications are described in Section 2.1. The overall process and time estimates are documented in Section 2.2.

2.1 Applied Methods

We chose an agile work style due to the unknown factors and risks posed by the challenges in Section 1.2. Agile development embraces change and adaptive planning, which fit the characteristics of this project. As an example, consider if we had failed in achieving the goals of developing the STM, or if the proposed real-time properties did not verify. This would mean a drastic change of plans at that point, which could require more time in a waterfall based approach compared to taking an agile approach. [14]

2.1.1 Managing Risk

The distinct element of risk associated with this project had to be managed. This was necessary in order to be able to divert from the original plan, should the choices we made have proven infeasible.

In the software industry, this technique is known as risk-driven development. As the name indicates, it is *risk* which drives the project. It is the philosophy behind software development processes such as Unified Process (UP), which is a framework of 50 different activities [15], where

it is the practitioners' responsibility to identify and apply the relevant activities, or methods[1].

UP recommends that the first phase of the project, also referred to as the *inception phase*, is spent identifying architectural and technological insecurities which could pose as show-stoppers later in the process. We used the same technique and arrived at an ordering of which tasks to complete first:

1. Develop the STM and verify the proposed properties are valid.
2. Extend the HVM to support the developed STM.
3. Develop the schedulability analysis tool.
4. Test the schedulability analysis tool.

This ordering was based on the fact that the STM and the proposed properties were absolute musts for the remaining parts of the project. Even though the proposed properties could prove to be invalid, detecting them as early as possible would maximise the time available to adapt.

Successfully extending the HVM to support the STM was next; it required a functioning STM. However, we also had to consider the limitations of the platform during our development of the STM, so we decided that points 1 and 2 should be developed in parallel.

Developing the schedulability analysis tool and conducting the experiments relied on the STM with the proposed properties, and a functioning platform being available. Naturally, these had to be considered after points 1 and 2 given their prerequisites.

At this point, we have only described how we applied risk-driven development at the overall level, considering the tasks superficially. We also considered each task separately in a risk-driven manner, identifying specific functionalities or concepts which were important to settle first. This is explained further in Section 2.1.2.

2.1.2 Agile Development

Knowing we could have to change plans during the project, we chose to employ an agile work style. Agile development is a broad term covering all iterative and incremental methods, which embrace the fact that software requirements change. Instead of having a single cycle of analysis, design, implementation and testing, agile development employs iterations. Each iteration can be seen as a full cycle of the classic waterfall method:

1. Decide upon the goals of the iteration.
2. Analyse the goals.
3. Design a solution.
4. Implement the design.

[1] A complete list of the activities in UP is given in [15, chp. 2]

5. Test the implementation and design.

This pattern is derived from how UP, SCRUM, eXtreme Programming (XP) and other iterative methods are practiced [15, 16, 17]. We wanted to rapidly construct working proof-of-concept prototypes, and this pattern allowed us to do exactly that. In addition, it was important for us to identify implementation errors or pitfalls as early as possible. With an iteration length of one week, the pattern forced us to review the current status of our goals once per week, and given an entire project term, we determined this was a fit iteration length. For industrial software projects, iteration lengths are recommended to be between 1–4 weeks [14].

2.1.3 Model-Driven Development

In [10], we developed a prototype STM. In this project, we extended this STM with real-time properties, and to prove its correctness we used a model checker. The purpose of a model checker is described in Section 4.4, but informally it is a tool which allows one to describe certain types of systems and have the tool determine whether a given property holds. For example, one could ask the model checker to determine if a given model is deadlock-free.

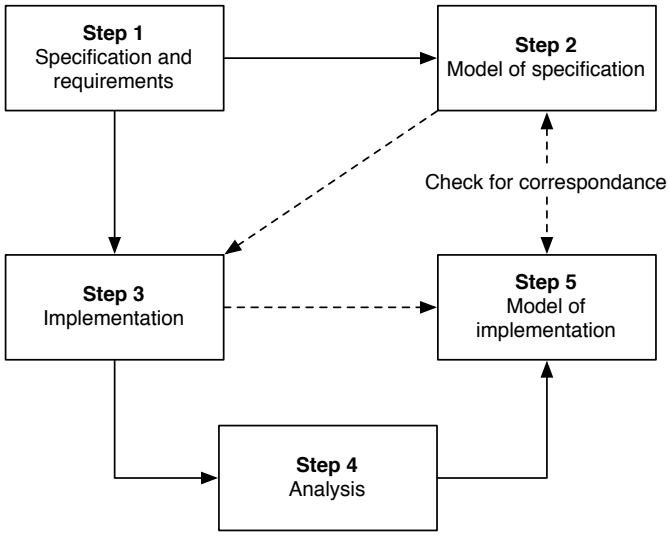


Figure 2.1: The main body of the five-step SARTS development process proposed in [18]

In order to manage this process, we followed a model-driven method proposed by SARTS. It is a process which is designed to aid the development real-time systems using model checking, and even in an iterative manner [18]. The main cycle of the SARTS process is illustrated in Figure 2.1. As it can be seen, it assumes the cycle starts with requirements and specifications of the system. Although we already have an implementation, we also have requirements in the form of the proposed claims which are not yet implemented. Constructing a model of these allowed us to abstract away from implementation details and other run-time technicalities, and

focus on the conceptual properties of the STM. Once the model was deemed correct, it was realised as a software implementation. Again as Figure 2.1 illustrates, the process encourages developers to use the model defining phase to better understand the requirements and refine the model accordingly. Should the model and/or implementation exhibit unwanted behavior, the model is further refined through another cycle.

2.2 Project Plan

This section outlines the overall plan for the project. Below is a description of how we allocated the time during the term.

February Clarify and validate project goals and problem statement.

February–March *(parallel)* Implement our proposed real-time properties in an STM verify them using model checking.

February–March *(parallel)* Extend the HVM with functionality needed to support our STM.

April–May Develop the schedulability analysis tool and conduct experiments to demonstrate both the STM and proposed real-time properties as well as the analysis tool.

By working on the STM and HVM in parallel, we could develop both the STM and HVM incrementally and continuously test how well they integrated with each other.

The STM was developed by following the SARTS method described in Section 2.1.3. The functionality and precision of the model was constantly refined, re-implemented, and subsequently tested on the HVM.

During the HVM work, functionality also gradually increased. First, proof-of-concept prototypes were used to demonstrate what we wanted to achieve was indeed possible through minimum working examples. These concepts were then implemented in the main code base, thus increasing the supported features of the HVM incrementally.

Like the other parts of the project, the schedulability analysis tool was developed iteratively. The first iteration was reserved to determine which of the existing tools we could re-use components from or be inspired by. The remaining iterations followed an incremental work style, constantly increasing the functionality.

Chapter 3

Real-Time Systems

Real-Time Software (RTS) is the class of computer software which is required to provide a *response* to an *event* within a given time frame. In this context, an event can be the occurrence of an action in the surrounding environment, such as a proximity sensor detecting an object closing in. It can also be a message from an internal clock, signalling an *interval* in time. The latter can be used to trigger an operation at specific intervals. A response is the result from a computation performed based on an event.

Examples of RTS applications are anti-lock brakes, hearing aid devices, and pacemakers. Each provides a response based on an event in the surrounding environment, e.g. anti-lock brakes allow wheels to interact tractively with the road surface while braking; hearing aid captures sounds, amplifies them, and replays them through an ear-piece; pacemakers emit small electrical shocks to stimulate the heart rate. One can then imagine how *timing* is equally as important as *functional* correctness in RTS.

In this chapter, we reiterate the properties of the task model in RTS and the fixed-priority scheduling (FPS) policy, which are applied in this project. The purpose is to briefly refresh the basics, and for an in-depth description of the task model and other scheduling policies, we refer to [11, 10].

3.1 Definition

Real-time systems can be classified as one of two types: *hard real-time* and *soft real-time*. In a hard real-time system meeting the timing requirements is essential under any circumstances, whereas a soft real-time system may occasionally miss deadlines. In this report, we are concerned with hard real-time systems only.

In RTS, logically coherent functionality is grouped into a task that, in the context of this report, is functionally the same as a thread of execution in an application, but with added timing constraints. We use the notation in Table 3.1 to describe properties of RTS tasks.

Notation	Description
B	Worst-case blocking time
C	Worst-case execution time
D	Deadline
I	Maximum interference
J	Jitter
P	Priority
R	Worst-case response time
T	Interval between release
U	Utilization of the task ($\frac{C}{T}$)

Table 3.1: Properties of the general task model.

The *worst-case execution time* (WCET) C is defined at task level. Schedulability analysis techniques use these values in determining whether a task set is schedulable or not, and as such it is important that the values are precise. Obtaining the WCET for a given task is done by either analysing or measuring the execution time the task. Analysing means calculating the amount of cycles required to execute the instructions constituting a task, while measuring them implies timing the execution of the task. Measuring the execution time of a task can provide imprecise readings, since it is difficult to determine how the code is executed—especially on modern CPUs with features such as branch prediction, caches, and pipelines [11]. Analysing the execution time of a task can be done using a computer-aided walk-through of the code, which identifies the most expensive code path and calculates the CPU cycles necessary to execute it.

The *priority* P of a task is an integer number that defines which of a set of ready-to-run tasks should be allowed to run on the processor. If a task with period p_1 and another task with period p_2 are both ready, the former should be be the one executing if $p_1 > p_2$. Priorities are selected based on the scheduling policy chosen.

When a task is executing, it may require exclusive access to a resource that is currently being held by another lower priority task that is suspended. When this occurs the task is said to be blocked while the lower priority task is resumed to continue executing until it releases the required resource. The longest period of time that a task can be blocked is denoted B. On the other hand, a task may be preempted because a task of higher priority is ready to run. This task will then *interfere* with the running lower priority task which must wait for the higher priority task to finish. The longest period of time that a task can experience interference is denoted I.

Jitter J is the time it takes for the system to switch between tasks. This involves determining which tasks are ready to run, which of them has the highest priority and context switching to the state of the new task that will be executing.

Tasks can be either *periodic* or *aperiodic*. A periodic task is a task that is released with a fixed interval of time called the *period* of the task, defined as T. An aperiodic task is a task that is released by a specific event, e.g. on input from a sensor or it can be fired by another task. A special case of aperiodic tasks is the *sporadic* task which has a bound on how often it can be fired:

its minimum *inter-arrival time*. As this is the minimum interval between releases of the task, it too is defined as T.

In hard real-time, only periodic and sporadic tasks are possible, as aperiodic tasks in general may be released an unbounded number of times and cause the processor to overload and miss deadlines.

ID	Priority (P)	WCET (C)	Period (T)	Delay
τ_1	2	50	75	10
τ_2	1	100	200	0

Table 3.2: A simple task set.

The *worst-case response time* R of a task denotes the amount of time a task requires in order to execute, that is from release to termination, and while considering the entire task set. Where WCET is defined for a single task in isolation, response time is defined for a single task with respect to the entire task set. Consider the task set given in Table 3.2: task τ_2 will be preempted by task τ_1 when it is released since the priority of τ_1 is higher. When task τ_1 is executing, τ_2 is suffering from *interference* from τ_1. τ_2 is resumed once τ_1 has terminated, and thus the observed execution time of τ_2 is greater than its WCET. Similarly, tasks that can be blocked by lower priority tasks will also have a greater response time than their WCET. The maximum time a task can be blocked is denoted B.

3.2 Schedulability Analysis

In order to determine if a set of tasks will be able to run within the timing constraints in all circumstances in a hard real-time system, schedulability analysis is performed on the tasks. Schedulability is determined by the time it takes to execute the code of the tasks on the given platform, the time between releases of each task, and the interaction between tasks if tasks are executed concurrently. If a system is *schedulable* it means that all tasks in the system are always guaranteed to meet their deadlines.

The response time of a task is influenced by the *scheduling policy* that the system uses. In this project, we consider the cyclic executive and fixed-priority scheduling (FPS) schemes. In FPS, each task is assigned a static priority, i.e. it does not change during run-time. The assignment is conducted using a rate monotonic scheme, which assigns priorites according to the periods of the task: the shorter the period, the higher the priority.

Cyclic executive requires the ordering of tasks to be defined prior to run-time. Each task is decomposed into procedures, and their execution sequence is what constitutes the schedule.

3.2.1 Schedulability Tests

In order to determine whether or not a task set is schedulable according to FPS, one of two techniques can be employed: utilisation-based testing or response time analysis.

Utilisation-based testing is the simplest of the two. The utilisation of a task set is given by $U = \frac{C}{T}$, which is then compared to the upper-bound on utilisation given in Table 3.3. If the bound holds, the task set is schedulable according to FPS. If the bound does not hold, the task set may still be schedulable, i.e. the utilisation-based test is sufficient but not necessary. [11]

N	**Utilisation bound**
1	100.0 %
2	82.8 %
3	78.0 %
4	75.7 %
5	74.3 %
10	71.8 %

Table 3.3: Utilisation bounds for task sets of sizes N using FPS scheme.

Response time analysis covers the cases where utilisation-based tests are not accurate, and it supports arbitrary deadlines, task interactions and aperiodic and sporadic tasks. It does so by considering the WCETs and response times of the tasks as described earlier in Section 3.1. An example of how a response time analysis is performed for a task set using FPS is given in Appendix A.

Chapter 4

Technology

In this chapter, we describe the technologies we have come in contact with during the course of this project. First of all, the concepts of STM are described in Section 4.1.

Next, we describe the languages Java and C in which we realised the STM concepts and schedulability analysis tool in Section 4.2. The programming languages are run on the HVM platform, which, along with the alternative JOP platform, is described in Section 4.3.

In order to verify the correctness of the STM and the validity of our proposed real-time claims, we decided to use model checking as the verification method. In order to do this, we chose to use UPPAAL [1] as model checking tool of the STM and verification engine in the schdulability analysis tool. Section 4.4 describes model checking as a method in general, how it applies in this project, and a brief introduction to UPPAAL.

We investigated two tools which aid the development of real-time systems in much the same way as our schedulability analysis tool. The first is SARTS [18], which analyses Java applications running on the JOP and generates a corresponding UPPAAL model. The second is TetaJ [19], which also analyses Java applications, but is not tied to a specific hardware platform as SARTS is. TetaJ is based on a third tool called WCET Analysis Tool (WCA), which is described together with TetaJ and SARTS in Section 4.5.

4.1 Software Transactional Memory

In this section, we introduce the concepts of STM that we use to describe the STM we have developed for the HVM in this project. The list of concepts and their definitions are derived from our previous work in [10].

Transactional Properties Each transaction in an STM must follow specific behaviour in order to provide atomicity, consistency, and isolation. Atomicity means that the results of a transaction happen entirely or not at all. Consistency is that if the system is in a consistent state before a transaction is run, it must be in a consistent state after it has run. Isolation

refers to the notion that each transaction must appear to be running in isolation from all other transactions, so that transactions may not interfere with each other.

Opacity For an STM to support opacity it must ensure that "(1) all operations performed by every *committed* transaction seem as if they were performed at a single, unique instant during its lifetime, (2) any operation performed by any *[un]committed* transaction is never visible to other transactions, and (3) every transaction always sees consistent data". [10] With these properties an STM ensures correctness as defined in [10].

Operational Structure The operational structure of an STM is how the programmer communicates that a piece of code is a transaction and what data to access transactionally. In a library-based STM, the programmer must call specific parts of an API to identify transactions. The alternative to this is to integrate the STM in the programming language, which gives new syntax that then handles the correct calls to the STM behind the scene.

Conflict Detection Two transactions that use the same shared data may conflict if they run concurrently. In that case, one transaction may need to be aborted to ensure correctness of the program. To this end, the STM must perform conflict detection on transactions. With *eager conflict detection* the STM detects conflicts as soon as they occur when one transaction tries to access shared data that is already in use by another transaction, whereas *lazy conflict detection* means that conflicts are not detected until transactions commit. A related concept is *false conflicts* which occur when the STM detects a conflict where there is none. Strictly speaking, a conflict only occurs if two or more concurrent transactions access the same shared data and at least one of them writes to the shared data.

Direct or Deferred Update Changes made to shared data in transactions can be stored in different locations. With direct updating, transactions write data directly to shared memory, and each transaction then keeps an undo-log that can be replayed whenever a transaction must abort to restore shared memory to its consistent state from when the transaction began executing, thus undoing all its changes. The alternative is deferred updates, where each transaction makes its changes locally in a redo-log and only at commit is this redo-log used to write all the changes a transaction has made to the shared memory.

Isolation An STM can ensure either *strong isolation* or *weak isolation*, where the former means that the STM guarantees consistent data is accessed when non-transactional code uses shared memory. With weak isolation, such guarantees are not given by the STM.

Nested Transactions One of the primary benefits of STM as opposed to locks, is the composability provided when transactions are nested. One way to handle this is *flat nesting* which simply means aborting the outermost transaction in case a nested transaction must be aborted. In this way, no extra resources are required to handle inner transactions. However, in case an inner transaction was aborted and could be rerun to completion without

having to abort the outer transaction, the code of the outer transaction before the inner transaction will have been rerun unnecessarily. An alternative to flat nesting is *closed nesting* which tracks nested transactions separately and allows aborting and retrying nested transactions without aborting their enclosing transactions, in exchange for higher resource use in order to track each nested transaction.

Granularity This concept describes the granularity with which the STM keeps track of transactional accesses to shared memory. With a fine granularity, e.g. tracking accesses to individual fields in an object accessed by a transaction, more metadata must be stored, but allows avoiding false conflicts when transactions access distinct parts of an object. This may not hold with coarser granularity, e.g. using an STM which only tracks objects accessed will detect a conflict even if two concurrent transactions access completely different fields of an object, but will mean that the STM uses less memory.

Static or Dynamic A static STM requires specifying which memory will be accessed by each transaction statically in the program, whereas a dynamic STM allows the STM to detect what parts of the shared memory is accessed automatically at runtime. A dynamic STM also allows creating new transactions at runtime, whereas a static STM only allows for a predetermined number of transactions.

Blocking or Non-Blocking An STM is classified as either blocking or non-blocking depending on whether or not it uses locks in its implementation. Using locks means that transactions may have to block while waiting for a lock to become available, or immediately abort themselves and retry later. With a non-blocking STM this is not the case, as "a nonblocking algorithm guarantees that if one thread is pre-empted mid-way through an operation/transaction, then it cannot prevent other threads from being able to make progress" [20].

Contention Management When two transactions conflict one of them is aborted and retried later. To determine which of the transactions is aborted a contention management strategy is used in the STM. The role of the contention management strategy is to provide fairness between transactions for some definition of fair relevant in the context in which the STM is being used. Two simple strategies are *passive* and *aggressive*, where the former aborts the transaction that detected the conflict, and the latter aborts the other transaction. Another strategy involves using fixed priorities on transactions to determine which transaction should be aborted.

4.2 Programming Languages

In this project, we used two programming languages to implement our STM and analysis tool, namely Real-Time POSIX/C [21] and Java. For the real-time aspect of our Java code we have

used a real-time Java profile [22, 23, 24]. In this section, we describe Real-Time POSIX/C and Java in a real-time context.

Developing our STM for the HVM resulted in implementation of both native low-level functions in Real-Time POSIX/C and the actual STM in real-time Java, while the analysis tool has been developed in standard Java. Real-Time POSIX/C is described in Section 4.2.1 and real-time Java is described in Section 4.2.2, followed by the reasoning behind our choice of these languages in Section 4.2.3.

4.2.1 Real-Time POSIX/C

Real-Time POSIX is a member of the POSIX standards, which are specified by the IEEE to promote compatibility between the operating systems[1]. Programming languages such as C can utilise the APIs defined by POSIX to interact with the underlying operating system, ensuring its compatibility with all operating systems complying with the same POSIX standards.

The Real-Time POSIX standard provides supporting operating systems and languages with real-time characteristics. Since C can utilise POSIX APIs, C can be used to implement real-time systems. C is known for its portability potential and expressive power. C is considered to map closely to Assembly [25] and, provided ideomatic use, C can allow for compact and highly efficient machine code, which makes it an obvious candidate for embedded systems.

Real-Time POSIX is also known as *POSIX.4*, indicating it is an extension of the functionality provided by POSIX.1–3. POSIX.1 describes basic functionality and concepts such as the notion of processes and threads, but does not describe interprocess communication, synchronisation, or scheduling. This is defined by Real-Time POSIX, also known as POSIX.4 [26]. The areas where POSIX.4 applies to this project are described below.

Thread management In POSIX.1, processes are only allowed to consist of a single thread each. POSIX.4 extends processes to contain several threads, which allows for cheaper context switches and shared address space between them [27].

Real-time scheduling Processes and threads as concurrent execution mechanisms are defined separately in POSIX.1, and the ability to schedule them using preemptive fixed-priority policy is defined in Real-Time POSIX.

Thread synchronisation Mutexes and condition are used for synchronisation between threads. Semaphores are also defined by Real-Time POSIX, but only apply to processes rather than threads. To avoid priority inversion, described in Chapter 3, Real-Time POSIX mutexes also support priority inheritance and priority ceiling protocols.

POSIX threads are managed by the functions defined in `pthread.h`[2]. This includes creating threads, initialising scheduling attributes for threads, and joining and terminating threads. The

[1] A complete list of POSIX certified products can be found at `http://get.posixcertified.ieee.org/search_certprodlist.tpl?CALLER=cert_prodlist.tpl`

[2] `http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html`

functions we use are described below, each referring to a specific line in Listing 4.1 showing its signature.

```
1 int pthread_create(pthread_t *, const pthread_attr_t *, void *(void *), void *)
2 int pthread_join(pthread_t *, void **)
3 int pthread_attr_init(pthread_attr_t *)
4 int pthread_attr_setinheritsched(pthread_attr_t *, int)
5 int pthread_attr_setschedpolicy(pthread_attr_t *, int)
6 int pthread_attr_setschedparam(pthread_attr_t *, int)
```

Listing 4.1: Excerpt of thread functions and their signatures.

`pthread_create` Creates a thread. By its signature in line 1, the first parameter is a pointer to a `pthread_t`, which becomes the handle of the thread. The second parameter is a pointer to a `pthread_attr_t` which is described further down. The third parameter is a pointer to a function, which returns `void *` and takes an argument of type `void *`. This is the function to run within the thread, and the fourth and last parameter is a pointer to the data to pass this function.

`pthread_join` Joins a thread. By its signature in line 2, the first parameter is a pointer to a thread handle of type `pthread_t`. The second parameter is a pointer to where the return values should be stored.

`pthread_attr_init` Initialises a `pthread_attr_t` which is a `struct` that describes the attributes of a POSIX thread. This prepares the `pthread_attr_t` passed as argument to be used, which can be seen in line 3.

`pthread_attr_setinheritsched` Specifies whether a given thread should inherit the scheduling policy from the surrounding process. The specific thread is given by the first parameter, which can be seen in line 4. The second parameter is an `int` indicating how the scheduling policy will be defined.

`pthread_attr_setschedpolicy` Specifies the scheduling policy for a given thread. The specific thread is given by the first parameter, which can be seen in line 5. The second parameter is an `int` indicating the scheduling policy to use, given by either `SCHED_RR` or `SCHED_FIFO`. `SCHED_RR` denotes a round-robin scheme, which applies round-robin between threads of the same priority, but otherwise favours threads of higher priority. `SCHED_FIFO` applies first-in-first-out (FIFO) between threads of the same priority, but also favours threads of higher priority. Both are preemptive fixed-priority.

`pthread_attr_setschedparam` Modifies a specific `pthread_attr_t` with the details given by a `sched_param`, which holds the priority, which is denoted by its signature in line 6.

In order to use POSIX mutexes, they must first be initialised as in line 1 in Listing 4.2. Next, the usage of `pthread_mutex_lock` and `pthread_mutex_unlock` is demonstrated.

```
1 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2
3 /* Take the mutex */
4 pthread_mutex_lock(&mutex);
5
6 /* Release the mutex */
7 pthread_mutex_lock(&mutex);
```

Listing 4.2: Initialisation of POSIX mutex.

An example of how a thread is created using the function described in this section is given in Listing 4.3. In Section 5.1 we shall see how this is applied in the HVM, and how periodic threads and delay can be implemented by using the same pattern and functions from `time.h`[3].

```
1  void worker(void *data)
2  {
3      /* Do work */
4  }
5
6  int main(const char **)
7  {
8      /* Initialize thread priority */
9      struct sched_param scheduler_parameters;
10     scheduler_parameters.priority = 10;
11
12     /* Initialise attributes. */
13     pthread_attr_t attributes;
14
15     pthread_attr_init(&attributes);
16     pthread_attr_setinheritsched(&attributes, PTHREADS_EXPLICIT_SCHED);
17     pthread_attr_setschedpolicy(&attributes, SCHED_RR);
18     pthread_attr_setschedparam(&attributes, &scheduler_parameters);
19
20     /* Declare the thread handle. */
21     pthread_t thread;
22
23     /* Start the thread sending NULL as worker argument */
24     pthread_create(&thread, &attributes, &worker, NULL);
25
26     pthread_join(&thread);
27     return 0;
28 }
```

Listing 4.3: Example of a Real-Time POSIX thread.

[3] `http://pubs.opengroup.org/onlinepubs/7908799/xsh/time.h.html`

Lines 1–4 defines the function which will run inside the newly created thread.

Lines 8–10 sets the priority of the thread.

Lines 13–15 initialises the `pthread_attr_t` variable with default values. In lines 16–18, the scheduling policy is explicitly set to round-robin and the scheduler parameters are set as well.

Line 24 creates the thread with the previously defined attributes, and stores the handle in `thread`. In line 26, the thread is joined. In line 26, the execution of `main` is blocked until the thread returns by the call to `pthread_join`.

4.2.2 Real-Time Java

Java is a high-level object-oriented language that is receiving significant interest for use in RTS [28, 24, 29, 30]. To use Java in RTS, a real-time profile can be used to provide a standard way to specify the real-time properties of the program, which can then be handled by a supported platform.

A real-time profile for Java consists of an API and a set of rules about how to program RTS in Java. The API allows creating tasks and specifying timing constraints for them, whereas the rules can specify certain features of the Java language that may be unsuitable for RTS or be difficult to analyse with regards to timing properties. For example, in the Safety-Critical Java real-time profile (see JSR-302 [31]), the use of recursion is not currently allowed, because it is difficult to analyse the running time of recursive calls. Details regarding SCJ are described in Section 5.4.

Being a high-level object-oriented language, Java allows the developers to express the application using classes and object-oriented design. This can potentially make the transition from developing general purpose Java applications to real-time Java systems easier by removing the need to learn a new language.

Another interesting feature of Java is that it is compiled to Java bytecode. Java bytecode is a platform independent low-level representation of a program. It is usually run on the Java Virtual Machine (JVM), which translates Java bytecode into machine code instructions that are run on the processor. This low-level representation is often used for static analysis of Java programs as it has a simpler syntax than the Java language.

There are also compilers for other languages than Java that can compile to Java bytecode, which means that selecting Java may potentially allow several languages to be used, as long as the analysis takes place on the generated Java bytecode. Although certain programming languages promote extensive use of certain features that complicate schedulability analysis, e.g. the frequent use of recursion in functional programming languages like Haskell to avoid maintaining a state in functions [32].

4.2.3 Our Choice

For this project, we have chosen to work with both Real-Time POSIX/C and Java. As stated earlier in this section, Java is gaining traction within the field of hard real-time systems [29]. Tools aiding this process have also emerged recently [18, 19], indicating it is gaining momentum. Java is selected for both the STM and the schedulability analysis tool. Choosing Java over another language for the STM enables us to ship the STM along side with a real-time Java profile, while still targeting every platform capable of executing Java.

Since we are targeting the HVM, which is written in C, we applied Real-Time POSIX/C in our effort to provide the HVM with real-time thread support.

The existing schedulability analysis tools we have investigated, and described later in Section 4.5, are written in Java and analyse programs written in Java as well. Accessing Java bytecode from within another JVM language such as Java itself is easy, as libraries such as the Byte-Code Engineering Library (BCEL)[4] and Java itself supports reflection and bytecode analysis [33].

4.3 Platform

In this section, we look at two platforms: the Java Optimized Processor (JOP) [13], and the Hardware-near Virtual Machine (HVM) [12]. The JOP is a processor which executes Java bytecode directly and in a time-predictable fashion which makes it a good choice for RTS. The HVM, however, is a virtual machine which executes Java bytecode and is implemented in C for several embedded hardware platforms and PC.

The HVM was chosen as the platform for our STM, while the JOP was encountered during our work with SARTS. By looking at the JOP, we also had a backup platform should the HVM not meet our requirements.

4.3.1 Java Optimized Processor

The JOP was created to be a processor for real-time Java applications, which means that its design is geared towards predictability instead of average-case speed. This means that features such as a complicated cache hierarchy or branch prediction have been left out, and instead it provides exact timing guarantees for each instruction including memory accesses. The JOP is often implemented on an FPGA mounted on a board with RAM, storage, and I/O ports.

The execution model of the JOP involves translating Java bytecode to microcode to execute each bytecode instruction in a number of cycles. In general purpose processors, such as an x86 processor, the instructions run by the processor are known as machine code, but, in the JOP, Java bytecode *is* the machine code of the processor, so no intermediate virtual machine is required to translate bytecode instructions to machine code to execute the application.

[4] `http://commons.apache.org/bcel/`

In order to support real-time applications, the Java real-time profile Safety-Critical Java (SCJ) has been implemented for the JOP in [18]. This profile allows programmers to specify tasks, periods, deadlines, and so on using a Java API. The SCJ code then contains specific calls to JOP instructions to handle creation of threads and to access the other hardware of the system for I/O. The life-cycle of a typical application using SCJ consists of an initialisation phase, the mission phase, and finally a cleanup phase. In the initialisation phase all the tasks are instantiated together with being allocated memory to be used for the duration of the runtime of the application. The threads are then started to begin the mission phase where the tasks run their logic responsible for providing the wanted behaviour of the real-time system. Finally, when the system is shut down, the cleanup phase makes sure that the tasks are stopped cleanly before the system can be powered off.

For development it is possible to run a JOP emulator on a PC to simulate execution of a system. This allows debugging on a PC before executing the code on an actual JOP.

4.3.2 Hardware-near Virtual Machine

The HVM is a virtual machine for embedded systems that can run Java bytecode generated by a standard Java compiler. Unlike the JOP, it is not a hardware platform, but serves to translate bytecode into machine code that can then be executed on the hardware platform. The HVM runs on several hardware platforms including the Atmel ATmega 2560 and the National Semiconductor CR16C. It even allows executing natively on a PC without the use of an emulator.

The HVM consists of a plug-in for the IDE Eclipse and an SDK with HVM-specific Java libraries accessible through a Java archive (JAR) file. Through Eclipse, a Java project can be exported to the HVM, which creates a number of C files containing the interpreter and the bytecode to be executed. This process is shown in Figure 4.1 which also shows the HVM SDK file `icecapSDK.jar` included in the Eclipse project. The C code emitted by the HVM plug-in can then be compiled to the target hardware platform using a C compiler. In Eclipse it is also possible to mark individual Java classes for ahead-of-time compilation which increases the size of the generated C code, but allows faster execution.

The generated C code contains hardware specific code for each target platform in a separate C file, which means that adding support for a new platform is mainly a matter of implementing the C functions to provide a form of hardware abstraction layer allowing the HVM to run on the platform.

Although the HVM is targeted at embedded platforms, it is not, as of the time of writing (April 2012), officially a real-time platform. However, in [19] the HVM interpreter was modified to have time-predictable execution so that it can be used in hard RTS, which also included the development of a real-time Java profile for use in this modified HVM.

Lastly, the HVM lacks support for threads, but it is possible to use the threads of the OS by using native C functions to implement support for them. However, this means that it is no longer possible to run the HVM on an embedded system ”bare metal”.

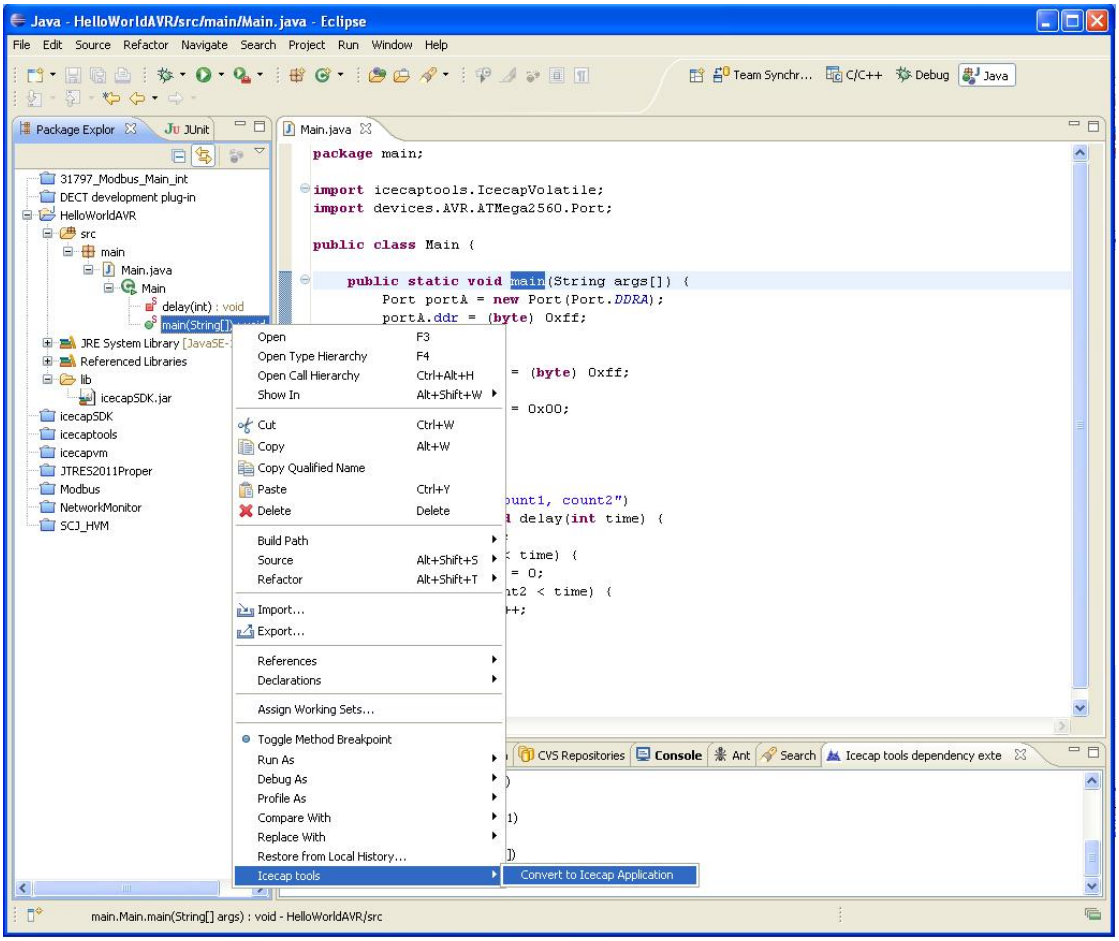


Figure 4.1: The static entry-method of the application is used to create the Icecap application containing the HVM interpreter and Java bytecode in C. [12]

4.3.3 Our Choice

For this project, we chose to use the HVM as our target platform. After a comparison of the two systems in Table 4.1, we see that the HVM appears to be more flexible than the JOP for modification. Another contributing factor is the ability to run the HVM natively on a PC, and calling native C functions from Java to implement parts of the system in C when that is more natural or low-level access to the hardware is required.

Choosing the HVM over the JOP also means that we target a wider range of hardware than just a single processor, which means a potentially wider audience for our results.

As we have pointed out, the HVM does have some shortcomings in a real-time context, and the fact that its design is not yet finalised also adds uncertainty to the project, but as demonstrated in [19], it is possible to use the HVM in a real-time context. That it is being actively maintained is also positive, and its maintainer and creator Stephan Korsholm has been available for direct

Platform	**JOP**	**HVM**
Hardware	JOP	Hardware agnostic
Native functions	Hardware	C functions
Modifiable	By reprogramming the hardware	By modifying the interpreter in C
Real-time (predictable execution time)	Yes	With modifications [19]

Table 4.1: Comparison of the features of the JOP and HVM.

support to help us understand the inner workings of the HVM.

4.4 Model Checking

Model checking is a technique to perform automated verification of finite-state reactive systems [34]. Such a system is modelled as a state machine, where transitions are equivalent to events to which the system reacts, thus changing its state. One such event could be either a modelled phenomenon or omnipresent, such as time. The semantics of a subject system is expressed using temporal logic first introduced in [35]. Since then, modelling tools such as UPPAAL [1] have been introduced, and provide a graphical user interface to construct the models in a more intuitive manner compared to writing the temporal logic by hand.

Once the model is constructed it can be queried using the model checker. In this project, we construct a high-level presentation of our STM along with the proposed real-time properties. An STM is reactive in the sense it can be triggered to open a shared variable, commit, and abort. Encoding the rules of a given system in such a way it correctly corresponds to its design is one of the main challenges, while validating this is another [36].

Employing model checking encouraged us to consider the properties of the STM from new angles. Being able to rapidly change a rule encoding and re-verify the model revealed pitfalls we would otherwise have had to construct either practical experiments or formal proofs in order to detect.

Models can be of varying detail. In this case, where it is the execution time of programs we consider, we are forced to let the generated models be of a sufficiently high detail. As an example, TetaJ [19] generates UPPAAL models from Java programs capturing it at machine instruction level. Naturally, for complex applications it would be cumbersome to manually calculate the number of machine instructions for even a small program. Model checking automates this process, and uses an algorithmic approach to explore the entire state-space of a given program.

4.4.1 UPPAAL

UPPAAL is the model checker we chose for this project since we have prior experience using UPPAAL, and its temporal logic makes it suitable to model real-time software. In UPPAAL,

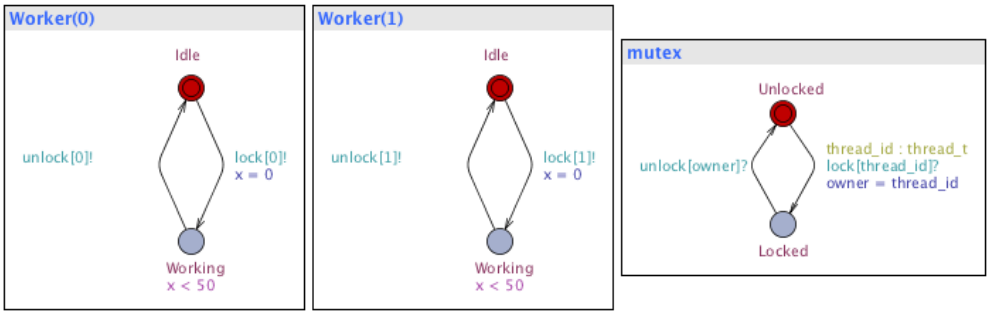


Figure 4.2: A simple UPPAAL model: two worker processes synchronising using a single lock.

models are expressed as a Network of Timed Automata (NTA), and verified using a specific querying language [1]. This section provides a brief introduction to or recap of UPPAAL, while detailed information can be found in [1].

A model can consist of several templates. A template is a single timed automaton, and an instantiation hereof is a process. A template consists of a set of locations and transitions between these locations. A location can be decorated with the following properties:

Initial It is the initial location for the template.

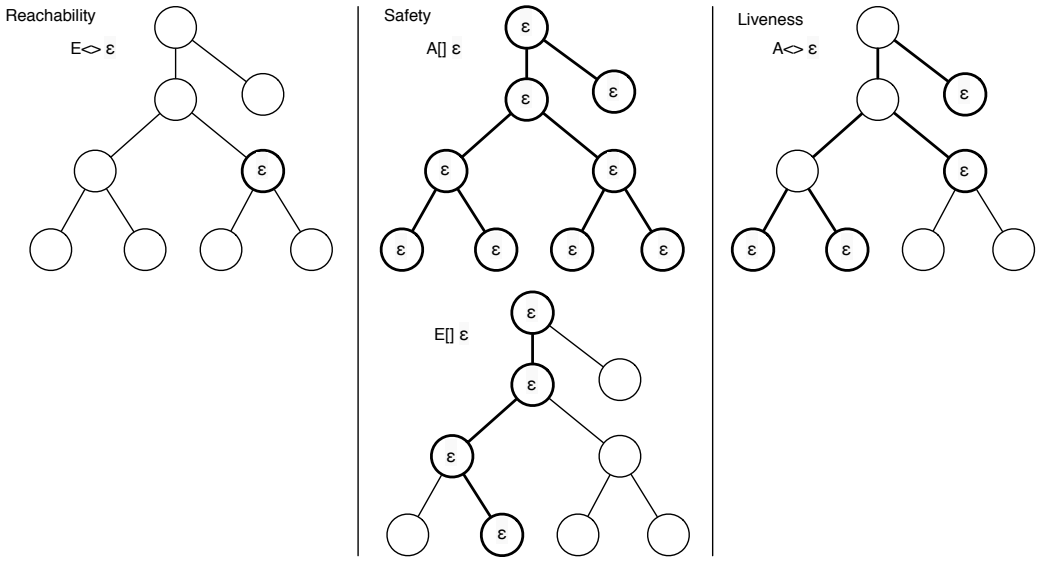
Committed When in this location, time for the entire system is not allowed to pass. The next transition must involve a process leaving a committed location, should one be in a committed location.

Urgent Less strict than a committed location. It is the equivalent of adding a new location-local clock, resetting it to zero, and assign all out-going transitions with the guard `x <= 0`. As opposed to a committed location, the system is not forced to leave the location as long as time does not pass.

Invariant An invariant must always be satisfied when in such a location. As an example, the Worker processes in Figure 4.2 have a *Working* location with an invariant `x <= 50`. This means that the clock `x` is not allowed to be greater than 50 while in this location. In the invariant field, it is also possible to set stopwatches. An example of a stopwatch is `x' == 0` indicating that the clock `x` should be stopped in that location, while all other clocks can continue unaffected.

A model can change state by taking transitions between process locations. Transitions can also be decorated with properties, and these are described below:

Selection UPPAAL allows the definition of bounded integers, for example an `int` within the bounds of [0; 10]. In Figure 4.2, the transition between `mutex.Locked` and `mutex.Unlocked` selects a value from the bounded integer `thread_id_t` non-deterministically and stores it as `thread_id`.



Notation	Description
`E<>` ϵ	Is it true for any state in any path that ϵ is satisfied?
`A[]` ϵ	Is it true for all states in all paths that ϵ is satisfied?
`E[]` ϵ	Is it true for all states in any path that ϵ is satisfied?
`A<>` ϵ	Is it true for any state in all paths that ϵ is satisfied?

Figure 4.3: An illustration of reachability, safety, and liveness queries and their meanings in UPPAAL. [19]

Synchronisation In order for processes to communicate, they can synchronise with each other. In Figure 4.2, the Worker processes synchronise with the mutex process. This happens over a channel, in this case `lock` and `unlock`. Suffixing the channel name with `!` (exclamation mark) indicates a *caller*, while `?` (question mark) indicates a *receiver*. The Worker processes both attempt to synchronise over the `lock` channel, each passing their ID 1 and 2, respectively. The mutex process receives one of the lock calls, and succeeds in doing so and receives the caller ID by non-deterministically selecting it from the bounded integer.

Updating A transition can also trigger side-effects. In Figure 4.2, the mutex process stores the matched `thread_id_t` within the `owner` variable, denoting the current holder of the mutex.

Guards Equivalent of invariants for locations, but for transitions.

Querying models in UPPAAL is done by expressing the property using a logical formula. For example, UPPAAL can verify whether the system is deadlock free: `A[] not deadlock`. Queries can either check for reachability, safety, or liveness properties. Their characteristics are illustrated and described in Figure 4.3.

4.5 Schedulability Analysis Tools

In an effort to learn from existing schedulability analysis tools, we have investigated three of such systems, which are designed to aid specific parts of the schedulability analysis. Each supports different versions of the Safety-Critical Java profile, described in [10] and Section 5.4.

A central challenge in schedulability analysis is the ability to express the upper-bound on execution times. While it is trivial to provide an unrealistically high upper-bound, tightening this is not, but results in a more realistic analysis. Modern processors are also becoming increasingly complex with features such as branch prediction, thus making it cumbersome to construct equivalent models of them. The tools we have investigated address these issues in various ways described in this section.

The JOP WCET Analysis Tool (WCA) [30] is intended to calculate the WCET of hard real-time Java programs for the JOP. It is capable of using the Implicit Path Enumeration Technique (IPET), and also a model-based approach. The features and how this is achieved are described in Section 4.5.1, but for an in-depth explanation of IPET we refer to [37].

TetaJ [19] generates a UPPAAL model from Java bytecode. This is used to calculate the WCET of hard real-time Java programs, but supports changing the model of the underlying JVM and hardware platform. TetaJ is described in Section 4.5.2. The resulting WCET from WCA and TetaJ are then used in a further analysis to determine whether it can be scheduled or not, and thus only aid in a particular part of the schedulability analysis.

SARTS [18] generates a model corresponding to the supplied real-time Java program developed for the JOP. Instead of returning the WCET as WCA and TetaJ do, the model captures the schedulability property by deadlocking if the program is not schedulable. SARTS is described in Section 4.5.3.

All three tools generate control-flow graphs (CFGs) from the Java programs, which are then mapped to UPPAAL models. A CFG in this context is given by directed graph $G = (V, E)$ where each vertex, or *basic block*, is a sequence of bytecode instructions without any branching. Edges denote connections between basic blocks, either in the form of branching, invoking methods or returning from method calls.

4.5.1 WCET Analysis Tool

WCA is designed for calculating WCETs of single tasks in real-time Java programs for the JOP. In addition, the programs must conform to the SCJ level 0 and 1 standards [23]. The JOP has known execution times for Java bytecode instructions, and its architecture simplifies the analysis even further. As we covered in Section 4.3.1, the JOP still provides caching of stack data and methods, but these features are designed to be WCET analysable. The JOP pipeline is also analysable, and allows for tight WCETs [30].

WCA is capable of performing both model-based and IPET-based WCET analysis:

Model-Based The control-flow graph (CFG) of the program is mapped onto a model, relying on the model checker to explore the entire state space in order to determine the most expensive path. In WCA, loops are modelled as illustrated in Listing 4.4. Large loop bounds can greatly increase the state space of the model, and thus the verification time. According to [38], this is one of the caveats of model-based WCET analysis. However, expressing features such as caching processor pipelines in models can be more intuitive than with IPET.

Implicit Path Enumeration Technique Given the CFG for a task, its WCET can be expressed for each basic block B_i by $WCET = \max \sum_{i=0}^{N} c_i e_i$, where N is the total number of basic blocks, c_i is the execution time of B_i, and e_i is the execution frequency [30]. Maximising the value of this expression can be accomplished by using integer linear programming (ILP) [39], which results in the maximum execution time of the basic block. According to [38], this technique reduces the analysis time for complex systems, with e.g. large loop bounds, but it is more difficult to express the hardware-specific features in this manner.

```
1 //@WCA loop=10
2 for (int i = 0; i < 10; i++)
3 {
4     // Work
5 }
```

Listing 4.4: Loop bounds must be defined statically in the source code in order for WCA to detect them.

Listing 4.4 shows how loop bounds are defined in the Java source code in order for WCA to detect them. Data Flow Analysis (DFA) [40] is also implemented in order to detect loop bounds automatically.

WCA includes a Java framework called *JOP libgraph* to construct CFGs from BCEL. JOP libgraph was developed to be used with WCA which targets the JOP, but it is not directly coupled to the JOP. As such, it can be used to model a control-flow graph for any Java program in basic blocks, instructions, and branching. JOP libgraph is used by TetaJ, which maps the generated CFGs onto their own object model.

4.5.2 TetaJ

TetaJ calculates WCET for real-time Java programs scheduled by the cyclic executive scheme. As such, it does not support multiple threads, and thus only a restricted subset of the SCJ profile [19]. However, its architecture was novel at the time of development (2011) in that it allowed for exchanging the underlying platform model, resulting in a flexible way of calculating WCETs across different hardware platforms. The architecture is illustrated in Figure 4.4, and the components are described below.

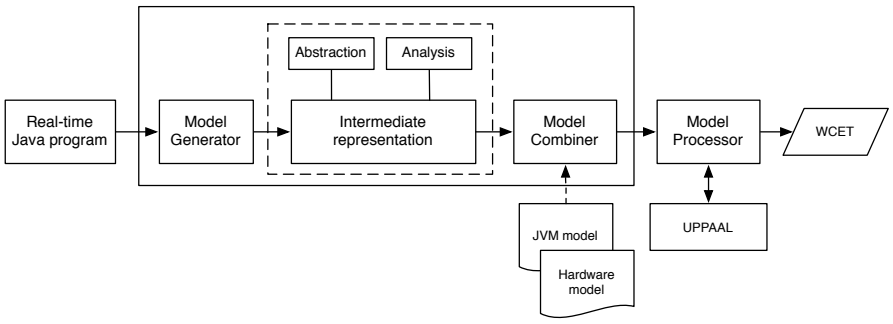


Figure 4.4: The architecture of TetaJ, showing how the process is divided into components and made pluggable.

Model Generator Extracts a CFG from the Java bytecode and generates a UPPAAL model from it. Besides the control flow, the CFG also captures bytecode instructions for each code block, loop bound annotations and instruction-to-source code mapping. The initial generation of the CFG is made using JOP libgraph, which is described in Section 4.5.1 as a part of WCA. Based on the JOP libgraph CFG, TetaJ maps this onto its own intermediate representation. This allows for decorating instructions, basic blocks, and edges with metadata such as loop bounds. An example on how loop bounds are defined for TetaJ is given in Listing 4.5.

Each edge carries information about whether it is a loop entry or exit edge. Loop bound information is attributed directly to such edges, and is extracted from the source code by conducting a loop bound analysis, which in practice searches the source code files for loop bound annotations.

TetaJ also ships with an optimisation analysis which can reduce the complexity of conditional basic blocks. Implementing new analysis techniques can be done by implementing a new class, which implements the supplied `IAnalysis` interface.

Model Combiner Combines the hardware, JVM and program models into one model. The purpose is not only to consider the program model and every instruction it consists of, but also taking into account the underlying platform. Having this information allows for determining a realistic WCET for the application on the given platform.

Model Processor Queries the combined model using UPPAAL to output the WCET for the application.

```
1 //@loopbound=10
2 for (int i = 0; i < 10; i++)
3 {
4     // Work
5 }
```

Listing 4.5: Loop bounds must be defined statically in the source code in order for TetaJ to detect them.

4.5.3 SARTS

SARTS generates UPPAAL models from real-time Java programs developed for the JOP. Instead of calculating the WCET, SARTS determines whether or not a given program is schedulable without any further analysis [18]. The generated model is encoded in such a way that the safety query `A[] not deadlock` is equal to whether or not the program is schedulable.

The flow of the SARTS analysis mimics that of TetaJ, except for the pluggable hardware and JVM models. The flow is described below and illustrated in Figure 4.5.

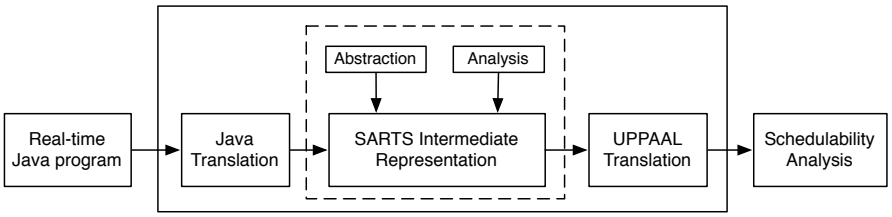


Figure 4.5: The architecture of SARTS, showing the responsibility and order of each component.

Java Translation SARTS uses BCEL for accessing the Java bytecode of the compiled program. From this, SARTS creates a class graph with the structure illustrated in Figure 4.6. Child nodes in the figure denote specialised classes, and parent nodes generalised classes. Each class contains a set of methods for which CFGs are generated.

SARTS represents the CFGs using a specialised object model: the SARTS Intermediate Representation (SIR). The basic blocks are of a specific type given their meaning in the program. As an example, a basic block containing a loop is of the type `LoopBasicBlock`. It contains fields denoting loop bounds and possible outgoing edges, which are specific for this type of basic block. The SIR class hierarchy is depicted in Figure 4.7.

SARTS performs analysis on the SIR in order to optimise and to decorate the CFGs with loop bound information. A basic block in SIR corresponds to one bytecode instruction, which results in a very large state space for even small programs. Because SARTS uses stopwatches for preemption of tasks, basic blocks can be collapsed into bigger blocks consisting of multiple bytecode instructions and their execution times. This reduces the state space significantly, and thus the verification time.

UPPAAL Translation UPPAAL models are generated from the SIR. Each method is represented by a UPPAAL template, and method invocations are modelled using channel synchronisation between these. The system also contains a *PeriodicThread* and *SporadicThread* template, which are responsible for driving the methods called by the periodic and sporadic threads of the system. Finally, a scheduler template is included statically, which employs the preemptive fixed-priority scheduling scheme.

Schedulability Analysis The generated model captures the schedulability property through its safety property of being deadlock-free. If the model is deadlock-free, the system is schedulable, and vice versa.

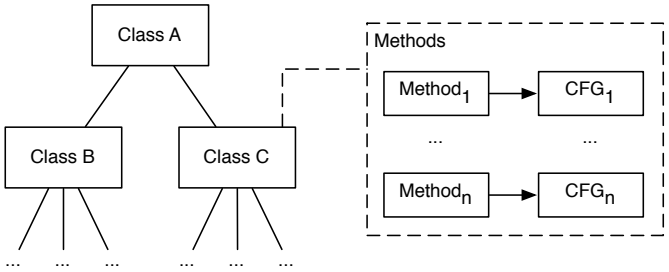


Figure 4.6: The SARTS Intermediate Representation is a graph containing every class in the program, and child nodes represent specialised classes. A CFG is generated for each method in the classes.

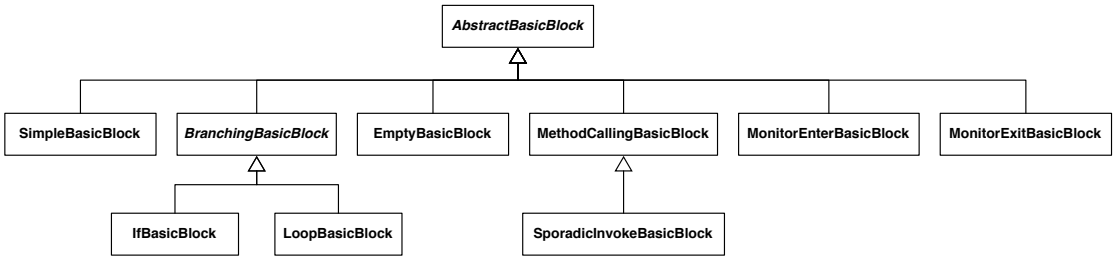


Figure 4.7: The basic block class hierarchy in the SARTS Intermediate Representation (SIR).

4.5.4 Our Choice

From what is described about WCA, TetaJ, and SARTS in this section, and from inspecting their code, we have identified key components which would assist us in achieving our goals. We wanted to focus on achieving a working STM suitable for real-time systems running on the HVM, rather than implementing CFG generators and UPPAAL model generators from scratch. WCA is JOP specific as a tool, and as is SARTS. Their code base is closely tied to the JOP architecture and timing details, while TetaJ is agnostic of the underlying platform.

The structure of the TetaJ source code was more suited for changing rather than that of SARTS. As stated, SARTS is closely tied to the JOP, and TetaJ has a more stringent notion of being hardware agnostic. Having identified the phases where the bytecode processed and the intermediate representation is generated in TetaJ, we decided to use the TetaJ CFG generator.

In SARTS, multi-threaded programs are modelled in an intuitive manner. Methods are still represented by separate templates, and as are the threads invoking the methods. Having the thread model expressed in a separate template yields a greater separation of concern between

the templates. It also heightens the intuitive construction of the model by having thread and scheduler logic intertwined into the program control flow.

In conclusion, TetaJ does not support multi-threading or any form of synchronisation mechanisms, but its object model and CFG generators are not directly coupled to the underlying platform as SARTS is. Reusing the object model and CFG generators from TetaJ along with multi-threading concepts from SARTS, we will create a hybrid of the two tools which we can use to add our STM functionality.

Chapter 5

Hardware-near Virtual Machine

With our choice of the HVM, we addressed its shortcomings in relation to RTS and multi-threading. At the time of writing (June 2012), the HVM does not have support for multi-threading and thus has no use for synchronisation mechanisms like the STM developed in this project. However, there are ongoing efforts to implement the Safety Critical Java (SCJ) level 1 profile (see JSR-302 [31]) which requires multi-threading. Since we are not able to use these unfinished efforts in our project, we have invested time to provide the necessary features ourselves to be able to use the HVM in this project. This chapter describes the work we have done to this end.

For the purposes of this project we have extended the HVM with threading via POSIX threads as described in Section 5.1. This allows us to use the HVM to run multi-threaded applications on a standard Linux installation. Running the HVM on Linux has the benefit of easier debugging, as we can connect the GNU Project Debugger[1] to the running HVM to investigate bugs we encounter. On a PC we can also access print functions to print strings to the screen both from C and Java code, which further helps our efforts.

To close some of the gap between running on a standard Linux installation and an RTS platform, we have used RTLinux [41] as a test platform as described in Section 5.2.

Memory management is another vital part of real-time systems, with which we have gained some experience in the HVM that is discussed in Section 5.3. We also look at how to allow the use of the SCJ API found in [18] in the HVM in Section 5.4.

5.1 Multi-Threading

In this section, we describe how we added thread support for the HVM using POSIX threads in C. Our efforts resulted in several C functions that can be called from Java as native function that can be used to manage the threads. We implemented functions to be able to create threads that

[1] `http://sources.redhat.com/gdb/`

will execute the `run()` method of a Java object that implements the `Runnable` interface, run these threads, and wait for them to finish executing.

The most complicated function required to implement this functionality is the functionality to start a thread which, in our code, means looking up the `run()` method, creating a thread local call stack for Java, and starting the HVM method interpreter in the thread. The threads have access to the same memory as the rest of the Java application, making synchronisation mechanisms such as locking and STM useful.

The code that handles thread execution is shown in Listing 5.1.

```
1  void *dispatchRunnable(void* arg)
2  {
3    unsigned short methodVtableIndex;
4    unsigned short* vtable;
5    unsigned short clIndex;
6    const MethodInfo* methodInfo;
7    int32 isrMethodStack[50];
8
9    struct thread_data* data = (struct thread_data *) arg;
10
11   clIndex = getClassIndex(data->runnable);
12   methodVtableIndex = findMethodVTableIndex(JAVA_LANG_RUNNABLE, 0, clIndex);
13   vtable = (unsigned short*) pgm_read_pointer(&classes[clIndex].vtable, unsigned
         short**);
14   methodInfo = &methods[pgm_read_word(&vtable[methodVtableIndex])];
15
16   isrMethodStack[0] = (int32)(pointer)data->runnable;
17
18   /* Execute the run() method of the Runnable instance */
19   enterMethodInterpreter(methodInfo, &isrMethodStack[0]);
20
21   return 0;
22 }
23
24 void runNativeThread(int thread)
25 {
26   pthread_create(&threads[thread].thread_id, &threads[thread].attr, &
         dispatchRunnable, (void*) &threads[thread]);
27 }
```

Listing 5.1: Running a Java `Runnable` in a POSIX thread with the HVM.

In this code, there are two C functions: `dispatchRunnable(void* arg)` and `runNativeThread(int priority)`. The former is run by the POSIX thread when it is created in the latter, where the function pointer to `dispatchRunnable` is sent as an argument in line 26 to `pthread_create` that is the POSIX threads function to create and run a new thread. The `threads` array used for the other arguments of the call in line 26 is an array that we have created to hold thread specific data in C.

The code for the `dispatchRunnable` function accesses internal functions of the HVM interpreter to look up the `run()` method of the `Runnable` instance in line 11-14. This method is then run with a thread local method stack in line 19. This code was provided by Stephan Korsholm.

To create and run a thread from Java the thread data is first created with a call to the native function `int createNativeThread(Runnable object, int priority)`. The returned integer is then used to run the thread using `void runNativeThread(int thread)` which calls the C function described above. Once the thread is running it is possible to wait for it to finish executing with a call to `joinNativeThread(int thread)` which is implemented with code in Listing 5.2.

```
1 void joinNativeThread(int thread)
2 {
3     pthread_join(threads[thread].thread_id, NULL);
4 }
```

Listing 5.2: Joining a POSIX thread.

Using these functions it is possible to create threads from Java with the HVM running on a Linux system, however, to close the gap between running the HVM on a time-predictable embedded system and a Linux system that does not provide any real-time guarantees, we have used RTLinux. A downside to our implementation of threads in the HVM is that it requires an operating system with POSIX threads, so it is not possible to run the code on the embedded platforms described in [12]. This means that we are only able to run our code on a PC until threads are officially supported in the HVM.

One problem that we have experienced with our thread extension is that there is certain code in the HVM that is prone to race conditions. Especially the memory allocation code is not thread safe, and can cause the heap to become corrupt and crash threads that try to concurrently allocate new instances of objects. In our code, we have worked around this issue by avoiding allocation of new memory once the threads have been started, and as such have been able to run threads for extended periods of time without experiencing individual threads crashing when race conditions happened as we experienced when allocating memory in concurrent threads.

5.2 RTLinux

RTLinux is a microkernel that wraps a standard Linux kernel to make it possible to use Linux for RTS. It does this by making it possible for tasks to preempt the Linux kernel, and as tasks are given higher priority than the Linux kernel, this means that tasks will run predictably on the hardware without being interrupted by the kernel to handle e.g. hardware interrupts. Interrupts are instead queued and handled as software interrupts when no real-time tasks need to use the system resources. [42]

In this project, we have used RTLinux for running the HVM, since it allows us to test our code on a real-time platform which supports the POSIX threads we have used to implement threads in the HVM.

5.3 Memory Management

As of the time of writing (June 2012), the HVM does not feature a garbage collector. Thus, any memory instantiated in the heap during a run of an application will not be automatically garbage collected while the application runs.

The limitations in memory handling in the HVM mean that we must either develop our algorithms for a future version of the HVM that will support real-time garbage collection (which may never come), or ensure that our code does not rely on the presence of a garbage collector to run correctly. The former case still allows using the HVM for the project, as we are able to instantiate memory as long as there is still free room in the heap, but when the heap runs dry individual threads, or even the entire application will crash. However, the latter option will, in theory, allow our code to run indefinitely, but means we have to take greater care in development so that allocation of memory is bounded.

5.4 Safety Critical Java Profile

A real-time profile for Java gives programmers an API for creating real-time applications. The SCJ2 profile introduced in [18] gives programmers access to create tasks and give them periods, deadlines and delay their startup. It also provides a clean way of shutting down the system by allowing clean-up methods in individual tasks.

For this project, we have implemented the SCJ2 profile for the HVM using POSIX threads as described in Section 5.1. We have done this to be able to compare our results with those of SARTS. This section describes our implementation of SCJ2 for the HVM.

5.4.1 API

The API of SCJ2 available to the programmer consists of several classes:

RealtimeSystem is a non-instantiable class that has static methods to start and stop all tasks. The method `start()` runs all periodic tasks and blocks until they stop, `stop()` stops all periodic tasks and cleanly shuts down all sporadic tasks and blocks until they are done, and `fire(int event)` asynchronously runs the sporadic task registered with the event number given as the argument.

PeriodicThread is an abstract class that is extended by the programmer when wanting to add a class of periodic tasks to the system. When the constructor is called, the object is added to

the list of tasks that will be started when `RealtimeSystem.start()` is called. The argument to the constructor is an instance of `PeriodicParameters` described below.

PeriodicParameters is a class used to describe the parameters of execution for periodic tasks. It has fields for the period, deadline, and delay with which to offset initial execution of the task. Each of these times are defined by an integer that expresses the time in microseconds.

SporadicThread is an abstract class that is extended by the programmer when wanting to add a class of sporadic tasks to the system. The constructor takes as its argument an instance of `SporadicParameters` described below.

SporadicParameters is a class used to describe the parameters of execution for sporadic tasks. It has fields for the integer that is used when firing the event using `RealtimeSystem.fire(int event)`, the minimum inter-arrival time between runs of the task, and the deadline. The times are expressed in microseconds with integers.

Using this API the programmer can create tasks by extending the relevant class and there add the code that a task will execute in a `run()` method. This method will be called whenever the task runs, either every period for periodic tasks, or every time the event is fired for sporadic tasks. The `run()` method returns a Boolean value to indicate whether or not it is ready to be shut down to ensure that the task is shut down cleanly when stopping the system. For a periodic task, returning **`false`** means that the task will be executed next period until it returns **`true`** after which the `cleanup()` method is run. For sporadic tasks, when the system is shut down all events are fired to allow the `cleanup()` method of the task to be run, unless the `run()` method has returned **`false`** on its last run.

```
1  public class MyPeriodicTask extends PeriodicThread {
2    private int i;
3
4    public MyPeriodicTask(PeriodicParameters pp) {
5      super(pp);
6      // Perform task initialisation here.
7      i = 1;
8    }
9
10   @Override
11   protected boolean run() {
12     // This code is run every period once the mission phase is started.
13     devices.Console.println(String.valueOf(i++));
14     RealtimeSystem.fire(2);
15     return true;
16   }
17
18   @Override
19   public boolean cleanup() {
20     // Cleanup code run on shutdown.
```

```
21      i = 0;
22      return true;
23    }
24  }
```

Listing 5.3: An example of a periodic task.

In Listing 5.3, a new periodic task is defined, which runs the code in line 12–15 every period. In line 14, the task fires the sporadic task that has event number `2`, which demonstrates how sporadic events function. The `cleanup()` method in line 18–23 is run when `RealtimeSystem.stop()` is called. A sporadic task has a similar structure except `SporadicThread` must be extended instead of `PeriodicThread` and `SporadicParameters` must be sent to the super-constructor instead of `PeriodicParameters`. To start the system using a periodic and sporadic task, the code in Listing 5.4 can be used.

```
1 public static void main(String[] args) {
2   new MyPeriodicTask(new PeriodicParameters(100000, 100000, 250000));
3   new MySporadicTask(new SporadicParameters(2, 100000, 100000));
4   RealtimeSystem.start();
5 }
```

Listing 5.4: Example main method of a real-time system using SCJ2.

In lines 2 and 3, the periodic and sporadic tasks are initialised by calling their constructors. It is not necessary to store the task objects in the main method, as the SCJ2 code will automatically keep track of the tasks in the system. The parameters given to the periodic task are: 100 millisecond period, 100 millisecond deadline, and an initial delay of 250 milliseconds before the task is run the first time after the call to `RealtimeSystem.start()`. The sporadic task has these parameters: the number `2` as the event number, 100 millisecond inter-arrival time and 100 millisecond deadline. The inter-arrival time of the sporadic task has been set to 100 milliseconds, because this is the period with which it can be fired from the periodic task in the `MyPeriodicTask` code above. In line 4, the system then begins the mission phase by calling `RealtimeSystem.start()`. The main method will be blocked in this call until all threads have stopped.

5.4.2 Implementation

Our implementation of SCJ2 for the HVM is based on the SCJ2 code from [18]. It is implemented partly in Java, and partly in the native C functions used to control the POSIX threads from Java. We use the `createNativeThread`, `runNativeThread`, and `joinNativeThread` native functions from Section 5.1 to manage the POSIX threads, which have been extended for SCJ.

Much of the SCJ2 code from [18] could be used unmodified, so we have primarily focused on removing the JOP specific code and replacing it with code that can run on the HVM. The API

provided to programmers using SCJ2 is the same whether using the JOP version from [18] or the version presented here.

Periodic Tasks

Periodic tasks are managed by the SCJ2 code through a class called `RtThreadAdapter` that uses native methods to work with the POSIX threads. An instance of `RtThreadAdapter` is created for each periodic task, and this `RtThreadAdapter` instance holds a reference to the `PeriodicThread` of the periodic task. The constructor of the `RtThreadAdapter` initialises the native thread through the call in Listing 5.5.

```
1 this.threadId = createNativeThread(this, priority, period, offset);
```

Listing 5.5: Calling the native function to initialise a POSIX thread.

This call will initialise the thread data with the information required to run the thread when the mission phase begins. The arguments given are: the `Runnable` instance which contains the `run()` method that is executed in the new thread, the priority of the task, the period of the task, and the offset or delay before the first release of the task after the mission phase has started. The native call returns the unique thread id of the data that must be used in subsequent calls to start or join with the thread. The first argument sent is the `RtThreadAdapter` instance itself. The `RtThreadAdapter` class contains the `run()` method that is shown in Listing 5.6.

```
1  @Override
2  public void run() {
3    for (;;) {
4      if (shutdown && runReturn) {
5        if (!cleanupReturn) {
6          cleanupReturn = re.cleanup();
7        } else {
8          break;
9        }
10     } else {
11       runReturn = re.run();
12     }
13     waitForNextPeriod(threadId);
14   }
15 }
```

Listing 5.6: Periodic thread loop.

The `run()` method consists of an infinite loop that executes the `run()` method of the `PeriodicThread` in line 11, and then waits for the next period in line 13. In line 4–9, the handling of the shutdown procedure is done. This consists of executing the `cleanup()` method each period

until it returns true after which the code exits from the loop and the thread is terminated. The method `waitForNextPeriod` used in line 13 is native and implemented with the C function in Listing 5.7.

```
1 void waitForNextPeriod(int thread_id)
2 {
3   struct thread_data *data = &threads[thread_id];
4
5   add_timespec(&data->next_release, &data->next_release, &data->period);
6   clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &data->next_release, &data->
        remaining_time);
7 }
```

Listing 5.7: C native function to wait for next release of a periodic task.

In this code, the thread data is found in the array of thread data using the thread ID in line 3. In line 4, the data is then used to add the period to the time of the last release and put it in the `next_release` variable. The resulting absolute time is then used in line 5 to sleep until the next release after which the call returns. The `remaining_time` variable can be used, should the sleep be interrupted, to indicate the remaining time of the sleep period. However, we assume that no interruptions will occur.

Sporadic Tasks

Similar to the `RtThreadAdapter` for periodic tasks, sporadic tasks use instances of `RtSwEventAdapter` to interface with the POSIX threads that run the individual tasks.

Due to it only being possible to create a limited amount of POSIX threads, we chose not to create a new thread every time a sporadic task is fired. Instead we use the two native methods `waitForSignal(int threadId)` and `signalNativeThread(int threadId)` in Listing 5.8. These methods use a semaphore that is stored in the thread data kept in C.

```
1 void signalNativeThread(int thread_id)
2 {
3   struct thread_data *data = &threads[thread_id];
4   sem_post(&data->sem_signal);
5 }
6
7 void waitForSignal(int thread_id)
8 {
9   struct thread_data *data = &threads[thread_id];
10  sem_wait(&data->sem_signal);
11 }
```

Listing 5.8: Native implementation of code to handle blocking until a sporadic thread is released.

In lines 3 and 9, the thread data is accessed which is then used in lines 4 and 10 to increment and wait for the semaphore to be incremented, respectively.

This native functions are used in the Java code in Listing 5.9.

```
1 public void fire() {
2   signalNativeThread(threadId);
3 }
4
5 @Override
6 public void run() {
7   while (!shutdown) {
8     waitForSignal(threadId);
9     handle();
10  }
11 }
```

Listing 5.9: Java code to handle firing of events.

The method `fire()`, defined in lines 1–3, is called when a call to `RealtimeSystem.fire(int event)` is made. This will then increment the semaphore of the sporadic thread. The `run()` method, defined in lines 6–11, runs in its own POSIX thread and loops until the system is shut down. The `waitForSignal(threadId)` call will block while the semaphore is `0` until another task increments it by calling `fire()` after which the `handle()` method is called to run the actual logic of the sporadic thread.

Due to this implementation, when an event is fired multiple times before it can run to completion, the event will be released the requested number of times successively. This is because the `sem_wait` call will immediately return until the semaphore reaches `0`.

Chapter 6

Software Transactional Memory Development

Developing an STM requires careful consideration of which properties are desired and their implications. In Section 4.1, we presented design and implementation parameters derived from research within the field of STM. In this chapter, we describe our implementation of a real-time STM for the HVM, and, using these design and implementation parameters, we describe its properties.

In [10], we argued for a non-blocking STM with real-time properties. This work is described in Section 6.1 along with claims that we made in [10] about how it would behave in a real-time setting. Since then, however, we decided on the HVM platform, which has required changes to the design and implementation choices that we had made. The resulting STM is described in Section 6.2 along with the reasoning for our changes and how the STM fulfils the claims we have made. In the process of ensuring we indeed fulfil the claims, we have taken a model-based approach. The model of our STM is documented in Section 6.2.5 and then used to verify relevant properties in Section 6.2.6.

6.1 Early STM Prototype

The non-blocking STM that we created in [10] was designed for the OpenJDK[1] JVM. We used it as a basis in the beginning of this project to create an early prototype with support for priorities. However, using the OpenJDK JVM meant that we had no access to real-time functionality and thus we could only speculate on how it would behave in a real-time context. We begin by describing the design and implementation details of this early non-blocking STM.

STMs for Java such as those found in [3, 7] use Java bytecode rewriting to allow using the

[1] `http://openjdk.java.net`

STM with a simple annotation such as in line 1 of Listing 6.1.

```
1 @atomic
2 public void addValue(int v)
3 {
4   counter = counter + 1;
5 }
```

Listing 6.1: Using Java annotations to make a method transactional.

However, because this is just syntactic sugar, we chose to use a lower-level approach by having the programmer use the STM by calling all library functions directly as in Listing 6.2.

```
1  do
2  {
3    try
4    {
5      c.txNew();
6
7      // This is the body of the transaction
8      c.txWrite(counter, c.txRead(counter) + 1));
9
10     c.txCommit();
11   }
12   catch (AbortException e)
13   {
14     if (c.txRethrowAbort())
15     {
16       throw e;
17     }
18   }
19 } while (!c.txIsCommitted());
```

Listing 6.2: Example of calling the STM library directly. [10]

This code consists of an explicit retry-loop, that makes the transaction start anew whenever it is aborted by the `txWrite`, `txRead`, or `txCommit` methods throwing an `AbortException`. The transaction body only actually consists of line 8, where a shared counter is incremented by one. The if-statement in lines 14–17 is there to support flat nesting by rethrowing the exception if the transaction is nested. In this way it will continue to be thrown until it reaches the outermost transaction which will then retry from the beginning. The rest of the handling of flat nesting consists of ignoring the calls to `txNew` and `txCommit` if the transaction is nested, so that a transaction is only started and committed at the beginning and end of the outermost transaction.

The code shows that the usage of our STM requires a lot of boilerplate code (all lines except 7–8) before writing the actual contents of the transaction, where the methods `txRead` and `txWrite`

must be used to access data in shared memory. However, since the boilerplate code could be generated automatically, it would be possible to attain the same operational structure as other STMs using bytecode rewriting.

Conflicts in the STM are detected when a call to `txRead` or `txWrite` tries to acquire a shared object that is already acquired by another transaction. Each shared object is encapsulated in a `TValue` object, which stores transactional metadata about who has acquired a variable and a new and old version of the object. To make the STM non-blocking, the `TValue` object is further wrapped in an `AtomicReference` object, which allows lock-free compare-and-set through the Java built-in `java.util.concurrent.atomic` package on supported platforms [43]. The transaction states are also wrapped in `AtomicReference` objects so that transactions can change states atomically.

```
1  public <T> void write(AtomicReference<TValue<T>> value, T newValue) throws
       AbortException
2  {
3    TValue<T> v = value.get();
4    TValue<T> result = new TValue<T>(this, v.oldValue, newValue);
5
6    if (v.owner != this && v.owner != null)
7    {
8      if (v.owner.state.get() == State.LIVE && v.owner.priority > this.priority)
9      {
10       // Current owner has higher priority and should not be aborted
11       abort();
12     }
13     v.owner.state.compareAndSet(State.LIVE, State.ABORTED);
14     if (v.owner.state.get() == State.COMMITTED)
15     {
16       result.oldValue = v.newValue;
17     }
18   }
19
20   if (state.get() != State.LIVE || !value.compareAndSet(v, result))
21   {
22     // Transaction has been aborted, or someone else acquired the variable
23     abort();
24   }
25 }
```

Listing 6.3: Transactional code for writing to a shared object.

In Listing 6.3, the old metadata is first retrieved through the `AtomicReference` object, and a new metadata object is created in line 4, which has a reference to the new value. In line 6, the transaction checks that it is the current owner of the variable. If not, it must acquire it by first trying to abort the current owner if it is running and does not have a higher priority than

the active transaction. If the current owner has already committed, the value written by that transaction must be used as the old value in line 16, in case the current transaction is aborted. In line 20, the transaction then atomically tries to acquire the variable by replacing the old metadata with the new metadata that it has just created. If this fails, it means that another transaction managed to acquire the variable in the mean time, which means that the transaction must abort. This is an essential part of the structure of the STM making it non-blocking.

From the code, it can be seen that readers and writers are not distinguished, as there is only a single field recording who has acquired the variable. This means that false conflicts occur when two readers try to acquire the same variable, since one of them will be aborted when the other tries to acquire the variable. The object granularity of opening the entire object for reading or writing means that false conflicts can occur when two different fields are written by individual transactions as well.

The implementation of the metadata in `TValue` also means that the STM uses direct updates, as the metadata is globally accessible. However, the effects are undone not by the current transaction when it aborts, but by the next transaction that acquires that variable, as it creates new metadata to replace it. This also means that there is no strong isolation, because both values are globally accessible.

To summarise, the properties of the STM are concluded below:

Operational Structure To use the STM, the programmer must call our library functions manually. Shared objects must be manually wrapped in `TValue` and `AtomicReference` objects.

Conflict Detection Conflicts are detected early and false conflicts can occur with read-only accesses by transactions.

Direct or Deferred Update Updates are stored directly in shared memory, and, in case of an abort, rolled back by the transaction following the aborted transaction.

Isolation The STM does not guarantee strong isolation, as shared data in an inconsistent state can be accessed outside of transactions. The STM thus has weak isolation.

Nested Transactions Nested transactions are supported through flat nesting.

Granularity Object-level granularity is used, since ownership of shared variables is defined at object-level.

Static or Dynamic The STM is dynamic, as it allows creating transactions and accessing shared memory based on runtime information.

Blocking or Non-Blocking Relying on hardware instructions to support non-blocking synchronisation primitives, the STM is non-blocking.

Contention Management Contention is handled by the priority of the competing transaction: if a shared variable has already been acquired by another transaction, it will be aborted if the requesting transaction has a higher priority than the current owner.

6.1.1 Real-Time Claims

In [10], we argued that if we could prove that the following claims hold for our STM running on a real-time platform, then it would be possible to perform a schedulability analysis on a program using the STM:

Claim 1:

By assigning non-blocking transactions priorities inherited from the tasks starting them, and preventing other than the highest-priority transaction from committing, we can allow higher priority tasks to start executing their critical sections faster than with lock-based synchronization.

This first claim is fulfilled by the fact that a non-blocking algorithm will always be able to run to completion if no other threads interfere, whereas a lock-based approach might require a lower priority task to be scheduled to allow it to release any locks it is holding that prevent the current task from obtaining the lock and continue executing.

Claim 2:

By using distinct priorities for threads, only one of the running transactions at any point in time will have the highest priority and will always be allowed to commit successfully.

This claim relates to the first claim, as it ensures that no other threads of equal priority will be allowed to interfere, so that they might continually abort the transactions of each other causing a livelock. If the system is scheduled using FPS only one task at each point in time will have the highest priority of the released tasks, and thus be able to run.

Claim 3:

Using static analysis, we can determine sets of transactions that can run concurrently without conflicts. If the schedulability analysis takes this into consideration it will lead to a tighter bound on the worst-case response time, as there might be fewer retries to consider.

This claim was put forward before we chose a model-based approach to verifying schedulability. Using modelling, the schedulability analysis will consider which tasks can conflict and take this into account when proving schedulability.

Claim 4:

Since the transactions of the running task with the highest priority in our system are effectively inevitable, I/O can be performed here. If applications are designed to use only the highest priority thread to perform I/O that must be in critical sections, then our idea becomes viable for a wider range of real-time software.

This final claim is related to allowing I/O in transactions. This claim would be fulfilled with the STM, since the transaction-related code of the highest priority transaction would not be preemptible by any other task, and thus it would not be able for any other transaction to cause the highest priority transaction to abort.

6.2 HVM STM

In this section, we describe our efforts to bring our STM to the HVM platform. This consisted of first attempting to modify the prototype, and when discovering that the design was fundamentally incompatible with the platform, a new STM design was created specifically for the HVM, but reusing many of the principles from our early prototype.

6.2.1 Porting the prototype STM

Once we had chosen the HVM as our platform, we began porting our prototype STM to the HVM. This consisted of implementing the features described in Chapter 5 and the `AtomicReference` class using native functions to access hardware compare-and-set instructions. However, once this work was complete, we found that due to the lack of garbage collection in the HVM, our initial design would only be able to run for a limited time, since our algorithm would continuously allocate new transactional objects on the heap. This meant that the heap would eventually run out of space causing tasks to crash.

In order to solve this, we first tried to see if it was possible to manually clean the heap of unused transactional objects using native C functions for the HVM. However, we were unable to identify a pattern that let us reliably clean up transactional objects at certain points in the execution of transactional method calls. Furthermore, at that time the HVM memory allocation code to create new objects on the heap was not thread-safe and would regularly cause memory corruption during test runs of our prototype.[2]

6.2.2 New Design

Due to the problems we encountered with our original prototype, we decided to try a new approach which used a fixed amount of objects and modified them directly, so that in the mission phase no new objects would be created. This approach allows us to run the STM reliably without memory corruption and without eventually running out of heap space.

The new design uses a blocking approach, because we were not able to rely on modifying metadata by creating new metadata objects as we did with the prototype. A global transactional lock ensures atomicity of metadata operations instead of using compare-and-set, which is what makes the STM blocking.

In order to retain some of the properties of the original design, however, the ability to revoke ownership of shared data was implemented to still allow high priority tasks to execute faster by at most only having to wait for the metadata operation of a lower priority task to complete as opposed to waiting for the entire critical section of the lower priority task to finish as would have been the case with lock-based concurrency.

[2]The HVM gained thread-safe memory allocation in April 2012. [12]

The new design also let us improve on the operational structure of the STM as we decided to use a different approach to accessing shared data: when accessing a shared object, a single call to open the object is made where the transaction acquires the object, instead of having to read or write by calling STM library functions. The reasoning behind this is that readers and writers are not distinguished, so separate calls for these operations are not necessary.

When a transaction in the old design aborted another transaction, it would directly modify the state of the competing transaction to abort it. In the new design, however, we avoided accessing the states of other transactions as transaction memory is reused when new transactions are started to avoid allocating new objects. This means that transactions in the new design must instead validate that they still have ownership of previously acquired objects when opening new shared objects to ensure opacity. The transaction must abort and retry if validation fails due to revokation of its ownership of one or more shared objects by a higher priority transaction.

6.2.3 Implementation

The implementation of the new design includes several classes:

`AbortException` The `AbortException` is similar to the `AbortException` of the prototype, but a single instance per task is reused to avoid allocating new objects when transactions abort.

`Lock` This singleton class is a helper class which gives access to a global lock object used by all transactions when modifying global metadata.

`OpenSet` Each transaction has its own instance of this class to store which shared objects are opened by the transaction. Details of how the open set avoids continually allocating new heap space are shown below.

`TContext` Each task that wants to use transactions must have an instance of this class, which handles communication with the STM and flat nesting by keeping track of the transaction depth.

`Transaction` The `TContext` object of a task has an instance of a `Transaction` object that contains the main functionality of our STM.

`TValue` Shared objects must be wrapped in a `TValue` instance which stores metadata about which transaction has acquired the object.

In the prototype, the `Transaction` class was instantiated for each new transaction and retry. However, since we are using flat nesting, only one transaction can be active at any time in each task, so in the new design we reuse the `Transaction` object by merely resetting its state and clearing its `OpenSet` instance.

To compare the implementation with the non-blocking prototype, we show how the shared objects are acquired by a transaction in Listing 6.4.

```
1  public <T> void open(T destination, TValue<T> value) throws AbortException
2  {
3    if (!openSet.contains(value))
4    {
5      synchronized (Lock.getInstance().lock)
6      {
7        if (value.getOwnerPriority() > priority.getPriority() || !validate())
8        {
9          // Unable to acquire object; other transaction has higher priority or
               this transaction has been invalidated
10         abort();
11       }
12       value.acquire(priority.getPriority(), destination);
13     }
14     openSet.add(destination, value);
15   }
16 }
```

Listing 6.4: Java code to open a shared object transactionally.

In the non-blocking implementation a new `TValue` object was created in the beginning to update the metadata, which replaced the old `TValue` object if the object was successfully acquired. The `TValue` objects were immutable. In the new design, we have made the `TValue` objects mutable, however, so it is possible to acquire the object by changing the existing metadata and thus we avoid allocating a new object.

To ensure opacity, the value is only acquired if the transaction still has ownership of all values in its open set. If any of the values have been lost, it means that a higher priority transaction has acquired one or more of its values, and it is necessary to abort to ensure that no inconsistent data is used. Validation happens in line 7, where the call to `validate()` iterates over the entire open set to check that all `TValue` objects have not changed ownership.

The first argument to the `open` method is a thread-local store which is used to hold a copy of the object. This copy can then be read and modified in the transaction and when committing the modified data is written back to shared memory. This means that deferred updates are used in the new design instead of the direct updates used in the original prototype.

To copy the data of the global object, we have implemented a native function in the HVM to perform a shallow copy of the object as shown in Listing 6.5.

```
1 int16 n_javax_stm_TValue_shallowCopy(int32 *sp)
2 {
3   Object *src = (Object *) (pointer) sp[0];
4   Object *dst = (Object *) (pointer) sp[1];
5
6   unsigned short ci = getClassIndex(src);
7   unsigned short size = classes[ci].dobjectSize >> 3;
```

```
8
9   while (size > 0)
10  {
11      *dst++ = *src++;
12      size--;
13  }
14
15  return -1;
16 }
```

Listing 6.5: Native HVM function to shallow copy an object.

In this code, the two arguments are first accessed through the `sp` stack pointer array. The first argument is the source object, and the second argument is the destination and they are saved to the `src` and `dst` pointers in line 3 and 4. To perform the shallow copy, it is necessary to figure out the size of the source object. This is done by looking up the class in the HVM `classes` array, which contains this information about all classes. When the object size has been retrieved in line 7, the data is copied byte for byte in the while-loop in lines 9–13.

From Java the function is accessed through the signature in Listing 6.6.

```
1  private static native void shallowCopy(Object source, Object destination);
```

Listing 6.6: Signature of the Java method to access the native shallow copy function.

The use of this function, however, requires that the destination space has already been allocated in the heap. Usually, objects are allocated using the **`new`** keyword in Java, but this will call the constructor of the class to create a new object, which may have side effects. To avoid having to call the constructor, we have created a native function for the HVM which merely allocates the memory in the heap without calling the constructor. This code is shown Listing 6.7.

```
1 int16 n_javax_stm_TValue_allocateInstance(int32 *sp)
2 {
3   Object *class = (Object *) (pointer) sp[0];
4   unsigned short classIndex = *(unsigned short *) ((unsigned char*) class + sizeof(
        Object));
5
6   handleNewClassIndex(sp, classIndex);
7
8   return -1;
9 }
```

Listing 6.7: Native HVM function to allocate an instance on the heap.

Using this code from Java, it is possible to allocate the memory for the local store, which is then ready to be used as the destination for the shallow copy when the global object is opened. The argument to the function is the `Class` object of the class that should be instantiated. In Java every class has a `Class` instance that describes the class. To allocate a memory for a new instance the code is used from Java as in Listing 6.8.

```
1  MyObject threadLocalStore = allocateInstance(MyObject.class);
```

Listing 6.8: Using the `allocateInstance` native function from Java.

6.2.4 Properties

Based on the design and implementation described, the STM is classified as follows:

Operational Structure The same operational structure is used as in the prototype, except that writing and reading shared objects occurs after opening the object which copies it to a thread-local store, and shared objects are only wrapped in `TValue` objects and not `AtomicReference` objects as well.

Conflict Detection As in the prototype, conflicts are detected early and false conflicts can occur with read-only accesses by transactions.

Direct or Deferred Update Updates are stored locally in a thread-local copy of the shared object. The copy is written back to shared memory when committing, thus deferred updating is used by the STM.

Isolation Unless all access to the shared object is protected by the global transaction lock, it may be modified while using it if a transaction commits its changes while non-transactional access takes place. Thus the STM has weak isolation.

Nested Transactions As in the prototype, nested transactions are supported through flat nesting.

Granularity As in the prototype, object-level granularity is used, since ownership of shared data is defined at object-level.

Static or Dynamic As in the prototype, the STM is dynamic, as it allows creating transactions and accessing shared memory based on runtime information.

Blocking or Non-Blocking The STM is blocking, because it uses a lock to ensure mutually exclusive access to the shared object metadata.

Contention Management As in the prototype, contention is handled by the priority of the competing transaction: if a shared object has already been acquired by another transaction, it will be aborted if the requesting transaction has a higher priority than the current owner.

From this list, we remark that the most significant change from the prototype to the new design is that the new design is blocking. This means that we must rethink our first claim in Section 6.1.1, which is dependent on a non-blocking design. We still want to be able to say that higher priority tasks will be able to start their critical sections faster than with lock-based concurrency control, but guaranteeing this property requires further investigation into the behaviour of the STM.

With the non-blocking design, the high priority transaction would be able to revoke ownership of any acquired objects in the same time as if their were not acquired. This is also true for the blocking design, as long as another task does not hold the global transaction lock and is thus in the process of acquiring an object or committing its results. If the transaction lock is held, the lower priority task must be allowed to run until it releases the lock. This behaviour is similar to that of lock-based concurrency, except for the fact that the lock is held only while committing or acquiring an object. With lock-based concurrency the lock would be held for the duration of the critical section, and thus a high priority critical section would be blocked until the lower priority critical section is completed. An even worse case is when several locks are required in the high priority critical section that are held by several lower priority critical sections. In this case, the high priority critical section must wait for each of these lower priority critical sections to run to completion. With our STM design, there is only one lock, and at most one lower priority transaction can block the high priority transaction in its entire span, and only for the duration of acquiring an object or committing.

The time it takes to acquire an object depends on the size of that object, as the object needs to have a shallow copy taken to be acquired. The size of each object in the HVM depends on the number and type of the fields of the object class. Simple types such as `int` and `long` are stored directly in the object, while fields that point to other objects such as instances of `String` or user-defined classes are stored as 32-bit pointers.

The time it takes to commit depends on the number of objects in the open-set, which must be shallow copied back to the global counter. This time can be derived by inspecting the number of acquired objects and how long it took to acquire them.

Claim 2 and 4 are proven using a UPPAAL model of our STM, whereas Claim 3 has been replaced by our model-based approach to schedulability analysis taken in this project.

6.2.5 STM Model

In this section, we describe the model of our STM. The model was used to ensure that the implementation of the STM is correct, and to verify certain properties. It is also used later in the development of our schedulability analysis tool.

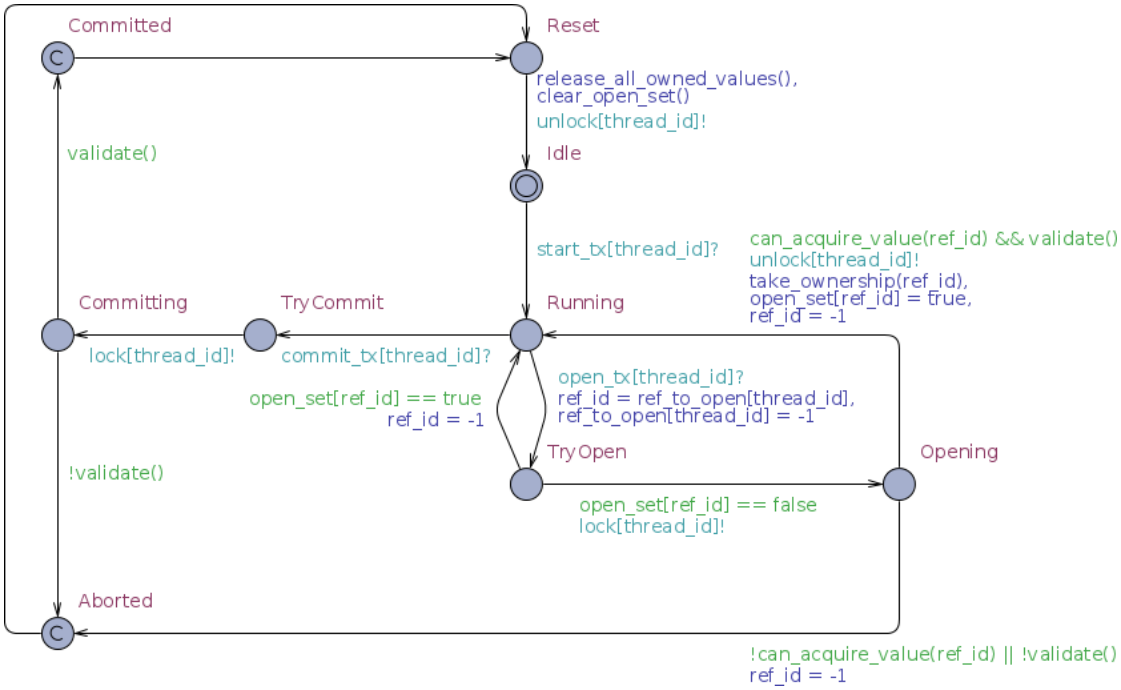


Figure 6.1: The UPPAAL template modelling our STM.

The transaction template of our model is shown in Figure 6.1 and has several locations:

Idle The initial location where the model is ready to start a new transaction.

Running The state where the actual body of the transaction is run. At any point during the transaction body it may attempt to open a variable or commit.

TryOpen The state when the body of the transaction attempts to open a specific shared variable with a specific `ref_id_t`.

Opening If the `ref_id` has not already been acquired by the transaction, it will attempt to acquire it from this location.

TryCommit The state when the body of the transaction attempts to commit the transaction.

Committing From here, the transaction either commits or aborts depending on whether it has been invalidated by another transaction.

Committed When the model is in this state, it means that the transaction was successfully committed. The location is committed, as it does not represent any code in the implementation.

Aborted Similarly, this state means that the transaction was aborted, and the location is also committed because it does not represent any code in the implementation.

Reset From this state, the variables of the model are reset to be ready for the start of the next transaction.

Since the STM uses flat nesting, only one transaction can be running in each thread. To accommodate this, a template instance is created for each thread in the system, which can then communicate with the STM using urgent channels. The communication can be either `start_tx`, `open_tx`, or `commit_tx`, which represent starting a new transaction, opening a shared variable, and committing the transaction, respectively. When opening a shared variable, the identifier of which variable to open must be sent as well. For this we use a global array `ref_to_open` with the size of the number of threads allowing each thread to set its entry to the identifier of the variable it wants to acquire. The transaction then reads the identifier from this array and saves the result in a local variable `ref_id` for easy access in following transitions. To reduce the state-space, these values are reset to -1 as soon as possible.

The `can_acquire_value` function is used to check that the value is not already owned by a higher priority transaction. If it is, then the transaction must abort.

The template also uses a global lock to model the transaction lock of the implementation. This lock is used by synchronising with the channels `lock` and `unlock`. If the lock is currently held by another transaction, the instance must wait in the TryOpen or TryCommit locations until the lock is free and listening on the `lock` channel.

The open set of each transaction is modelled as an array with the total number of shared variables as its size. Whenever a value is acquired, the corresponding entry in the array is set to **`true`** to indicate that the value has been acquired. The open set is used whenever the transaction needs to validate where ownership of all values in the open set are checked to ensure that no other transaction has acquired any of its values.

Nested transactions in the model are not handled expicitly, but rather by the convention that nested transactions do not use `start_tx` or `commit_tx` when they start or commit. Thus, only the outermost transaction uses `start_tx` and `commit_tx`, and inner transactions only use `open_tx` to open the variables they need.

6.2.6 Verification

To verify that the remainder of our claims hold, we have created the following queries to assert the necessary properties of our STM model:

1. `A[] not deadlock`
2. `A[] not Transaction(0).Aborted`

The first query ensures that there are no deadlocks in the system. This query is important in a lock-based STM where it must not be possible for any interleaving to cause the STM to crash.

The second query allows us to verify that the transaction with the highest priority, which is the transaction with `thread_id` 0, cannot be aborted under any circumstances. Proving this

query means that the STM will always be able to ensure that at least one transaction out of a number of transactions can commit. Thus, if a number of transactions are running concurrently, then it is assured that the one with the highest priority of these will always commit. It also shows that our fourth claim in Section 6.1.1 is true.

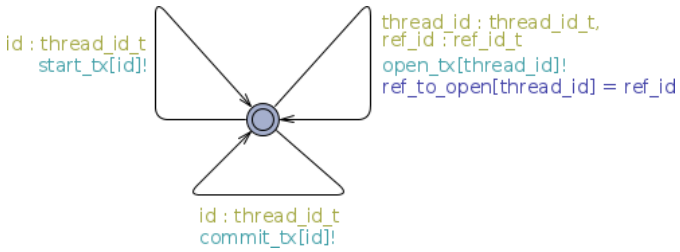


Figure 6.2: The driver template used to run the STM model.

To verify these queries, we have created a generic system of two transactions with two shared variables, where a "driver" template is used to randomly start transactions, open variables, and commit the transactions. This template is shown in Figure 6.2. Two transactions are sufficient to show correctness of an STM since conflicts occur with at least two transactions contending for the same shared variable [44]. Running with more transactions provide the same results, except for UPPAAL taking longer to explore the state-space of more transactions interacting.

Both queries were proven to be satisfied using UPPAAL. Verifying that the transaction with the highest priority can never be aborted has thus proven that Claim 2 and 4 are fulfilled with our STM.

Chapter 7

Schedulability Analysis Tool Development

In this chapter, we describe the development of our schedulability analysis tool which we have named OSAT. OSAT allows analysis of Java bytecode using the real-time profile SCJ2 described in Section 5.4, and our STM described in Section 6.2.

7.1 Requirements Analysis

With our choice of using TetaJ as the code-base for OSAT in Section 4.5.4, we identified which features we had to implement to extend TetaJ and develop OSAT. By looking at what TetaJ is already capable of, and what we want to achieve as described in Chapter 1, we see that the main feature that we need to add is analysis of programs using multi-threading and our STM.

SARTS already does schedulability analysis of multi-threaded Java programs for the JOP using their own implementation of SCJ. To be able to compare our results with SARTS, we implemented their version of SCJ for the HVM in Section 5.4 and decided on a similar approach to analysing Java programs:

- Use control-flow analysis of the main method of the Java program to find all instantiations of the `PeriodicThread` and `SporadicThread` classes.
- Perform control-flow analysis for each of the `run()` methods of the threads. This step will also perform special handling of STM-related functionality to use our STM-model in the schedulability analysis.
- Generate a UPPAAL model using the control-flow analyses and add in templates for a real-time scheduler and handling of each periodic and sporadic task.

The resulting schedulability analysis would be more like that of SARTS than TetaJ. Where SARTS can verify if a program will always be able to execute its tasks within their deadlines, TetaJ calculates a WCET for one round of running the tasks using cyclic executive scheduling. Using the approach of SARTS thus makes sense for multi-threaded programs using SCJ.

To identify transactions in the program, we decided to add a specific annotation to the code before each transaction. TetaJ and SARTS already use Java code comments as annotations to be able to specify loop bounds in the Java programs. However, these annotations rely on having access to Java source-code files, which we found unnecessary, since the remainder of the analysis only relies on access to the compiled Java class files. An alternative to using code comments as annotations is to use static method calls as described in [45], e.g. writing `RealtimeSystem.loopbound(20);` right before a loop instead of `//@loopbound=20` which is how TetaJ reads loop-bounds. Using this approach, we are also able to analyse other languages than Java that have Java bytecode compilers, since the schedulability analysis is now only dependent on Java class files. However, the HVM is intended for executing bytecode compiled from Java [12], and we have not experimented with other JVM compatible languages. Unfortunately, the addition of static method calls as annotations increases the execution time since the method calls will incur an overhead in entering and exiting the methods even if they contain no code to be run. One solution to this would be to modify the bytecode to remove these method calls in the analysis and before deploying the code to the platform, but that was not added as a system requirement for OSAT.

7.2 Design and Implementation

We now present our design for OSAT and its implementation. We begin by giving a general overview of the flow through the tool, followed by an individual description of each of the components that make up our implementation.

7.2.1 Main Flow

The analysis of a Java program using OSAT takes place through several different steps. In this section, we describe these steps and how they relate to give a better understanding of where and how our modifications to TetaJ are incorporated to create OSAT.

1. **Load Main Method**
   The process begins with loading the `main` method of the program to analyse. The bytecode in the class file is opened using BCEL and a CFG of the `main` method is generated to access its code.

2. **Find Threads**
   This step consists of walking through the code of the `main` method using the CFG generated

in the first step. During this walk, we search for places where subclasses of `PeriodicThread` or `SporadicThread` of our SCJ implementation are instantiated and read the task properties sent via `PeriodicParameters` or `SporadicParameters` as arguments.

3. **Analyse Threads**
   In this step, the `run` method of each discovered task is analysed and a CFG is created for the `run` method and for each method used. The CFGs are then analysed to discover transactions, set bounds on loops, and locate firings of sporadic tasks.

4. **Generate UPPAAL Model**
   This step is where the UPPAAL model is generated with a template for each method in the program. Various UPPAAL model constants are also defined and arrays used in the verification are initialised. To start the `run` methods in the UPPAAL model, a template for each task is created which manages the temporal properties such as deadline, period, and minimum inter-arrival time.

5. **Combine UPPAAL Model**
   In this step, the generated UPPAAL model is combined with various static UPPAAL templates that each reside in their own UPPAAL model files. These templates are necessary to complete the UPPAAL system, and include a model of the scheduler, a model of the platform, and the model of our STM.

6. **Output UPPAAL Model**
   Finally, the resulting XML-file is saved to disk and can be opened with UPPAAL to perform the schedulability analysis.

The first and second steps are where we add the SARTS thread-capability to TetaJ for it to be able to analyse each thread in the program. The third step is basically a loop around how TetaJ analyses a single-threaded program, where we have added additional analysis phases to incorporate our unique functionality. In the fourth step, we added the templates to handle the threads, where TetaJ only has to analyse the `main` method. The fifth and sixth steps are the same as in TetaJ, where the TetaJ model combiner program is used to combine several UPPAAL models into one, where the resulting UPPAAL model is then verified using UPPAAL.

The flow and architecture of OSAT is also illustrated in Figure 7.1. We adopted the idea of a *combiner* phase from TetaJ, which allows us to plug in different models for the scheduler and STM. This is useful in the event the tool should consider a different scheduling scheme than the preemptive fixed-priority scheme we provide, or an STM with different properties than the one we provide.

In the following sections, we describe the details of each step to explain how OSAT works.

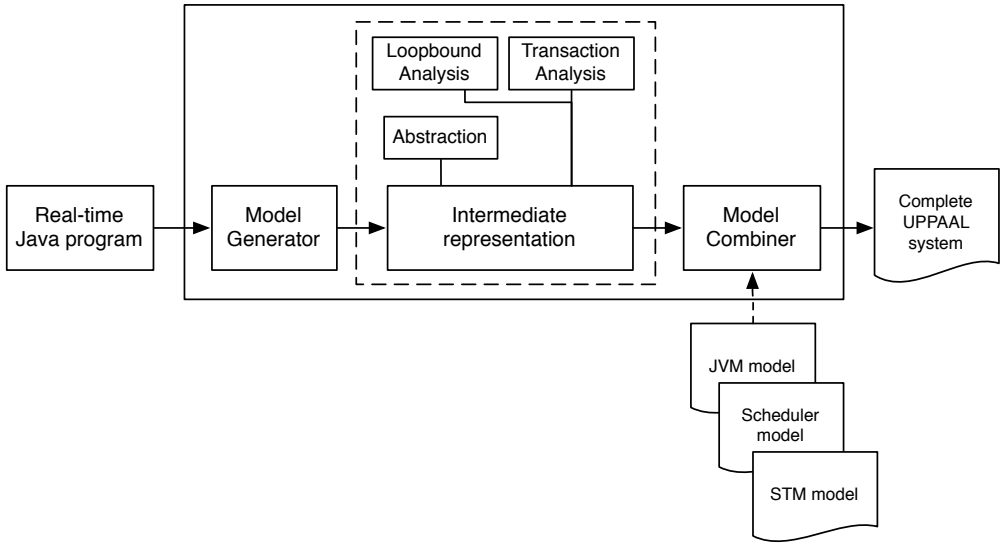


Figure 7.1: The flow and architecture of OSAT showing a similar pluggable architecture found in TetaJ.

7.2.2 Load Main Method

In programs analysable with TetaJ, the `main` method is where the mission phase is executed. However, in the multi-threaded SCJ implementation, the mission phase takes place when the periodic threads are all running and the `main` method is blocked in the call to `RealtimeSystem.start()` after it has instantiated all the tasks. This means that OSAT must first perform an analysis of the `main` method to discover all the tasks of the system, and then generate the UPPAAL model based on the `run` methods of the individual tasks. By using this approach, we attain the same functionality in this step as found in SARTS.

In our approach, only the `main` method is analysed, and thus if the `main` method initialises threads in a separate method that is called, these threads will not be detected. This means that it is a convention that the `main` method must have the form shown in Listing 7.1.

```
1  public static void main(String[] args)
2  {
3    // Code that is part of the initialisation phase
4    ...
5
6    // Thread initialisation
7    new MyPeriodicThread1(new PeriodicParameters(1000000, 1000000, 250000));
8    new MyPeriodicThread2(new PeriodicParameters(2000000, 1000000, 0));
9    new MyPeriodicThread3(new PeriodicParameters(1500000, 1500000, 0));
10   new MySporadicThread1(new SporadicParameters(1, 1000000, 100000));
11   ...
12
13   // More code that is part of the initialisation phase
```

```
14    ...
15
16    // Start mission phase
17    RealtimeSystem.start();
18  }
```

Listing 7.1: An example of the initialisation phase in SCJ2.

The first code in line 4 is where any initialisation code can be run. In lines 7–11, all threads of the system are initialised, and then in line 14, more initialisation code can be run. Finally, the mission phase is started by running all the periodic threads in line 17. The most important part is that all threads are instantiated in the `main` method only. It is possible to run code in between instantiations of threads, but this is not shown in the example. The execution time of the initialisation phase is not relevant to the schedulability analysis.

7.2.3 Find Threads

After the `main` method has been loaded, we use the same CFG generator that is used to create UPPAAL templates in TetaJ to find the thread instantiations. The output of the CFG generator contains a list of all instructions in the analysed method, which we then iterate over to locate all **`new`** statements. When a **`new`** statement is encountered, the names of the super-classes of the class being instantiated are compared to the SCJ periodic and sporadic task classes `javax.scj.PeriodicThread` and `javax.scj.SporadicThread`. Any real-time task will extend one of these classes and be instantiated in the `main` method by our convention.

```
1  if (statement instanceof StackPush)
2  {
3    StackPush pushStatement = (StackPush)statement;
4    switch (currentThread.getType())
5      {
6      case Periodic:
7        currentThread.setPeriodicParameter(pushStatement.getValue().getIntValue());
8        break;
9      case Sporadic:
10       currentThread.setSporadicParameter(pushStatement.getValue().getIntValue());
11       break;
12   }
13 }
```

Listing 7.2: Excerpt of our code to extract thread parameters.

Another convention, that is also found in SARTS, is that the constructor of each task takes, as its first argument, an instance of either `PeriodicParameters` or `SporadicParameters` depending

on whether it is a periodic or sporadic task. The constructors of these classes take the arguments described in Section 5.4. To allow OSAT to access the parameters without performing data-flow analysis, the arguments must be passed as integer literals, and not as variables. Using this convention we can access the numbers with the code in Listing 7.2.

This code is run as part of the loop that iterates over each bytecode and after an instantiation of either a subclass of `PeriodicThread` or `SporadicThread`. We can then access the numeric value of each argument in lines 7 and 10 through the bytecode statement `getValue` method, which is implemented in the JOP libgraph bytecode analysis library from WCA. The `setPeriodicParameter` and `setSporadicParameter` methods are implemented by us in our `ThreadInfo` class to set the parameters while holding the state of which parameters have been set and which is the next one. One `ThreadInfo` instance is created for each thread found in the `main` method.

After all threads have been found in the `main` method, we assign the priorities of the tasks. In the implementation of SCJ, tasks are given priorities based on whether they are sporadic or periodic, and on their deadlines. A sporadic task always has a higher priority than a periodic task, and, of two sporadic or periodic tasks, the task with the lowest deadline has higher priority. In OSAT, we have implemented this as a `compareTo` method that can compare two `ThreadInfo` objects. We then just iterate over the threads using the code in Listing 7.3 to set the priorities.

```
1 int priority = 1;
2 for (ThreadInfo ti : threads) {
3   ti.setPriority(priority++);
4 }
```

Listing 7.3: Thread priority assignment.

Here the iterator used will automatically iterate over the threads in sorted order by priority because the `ThreadInfo` class implements the Java `Comparable` interface.

7.2.4 Analyse Threads

We now have information about all the threads that are created in the `main` method and can begin to analyse the `run` methods of the threads. For each thread, a CFG is created of the `run` method and, recursively, all methods called from the run method. This means that any method reachable from the `run` method will have a CFG generated. Using these CFGs, we then perform different analysis steps to add our functionality to the generated UPPAAL model:

Transaction Analysis

The first analysis that we perform is the transaction analysis. To be able to recognise transactions, each transaction must have the form described in Section 6.2, and must further be prepended with a call to `RealtimeSystem.transactionStart()`. The `transactionStart` method

is an empty static method, and calls to this method are looked for in the transaction analysis. When a transaction is found, the CFG is modified to remove the do-while-loop and try-catch of the transaction and replace calls to the STM with special instructions that will become synchronisation points with our transaction template in the UPPAAL model that is generated later. Removing the do-while loop and try-catch block of the transaction is done because these parts of the transactions are handled by the transaction template.

The transaction analysis can be seen as a finite-state machine with four states depending on what is searched for in the CFG of a method:

TxStart is when a call to `RealtimeSystem.transactionStart()` is being looked for. This is the state that the analysis is in whenever it is searching for the next transaction.

TxNew is when a call to the `txNew` method of the `TContext` object is looked for. While the analysis is in this state, all nodes encountered are removed from the CFG to ignore the first part of the do-while loop and try-catch block around the transaction.

TxCommit is the state that signifies that the call to the `txCommit` method of the `TContext` object is looked for. This is the body of the transaction, and here we also look for calls to `txOpen` of the `TContext` object, where new shared variables are opened. Calls to `txOpen` are rewritten in the CFG so that they will become synchronisation points with the transaction template later. The fully qualified name of the first argument of the call, which is the shared variable to open, is used to identify which shared object the transaction is trying to open.

TxIsCommitted is after the call to `txCommit` of the `TContext` object has been located, and the call to `txIsCommitted` of the `TContext` object is searched for. The call to `txIsCommitted` signifies the end of the transaction do-while loop and thus also the end of the transaction. In this state all bytecode instructions are removed from the CFG to remove the catch block of the try-catch block and the remainder of the do-while loop. After the `txIsCommitted` call is found, the analysis goes back to the `txStart` state to search for the next transaction.

Since the fully qualified name of the shared object to open is used to identify when two transactions are opening the same object, it is necessary that all shared objects are accessed through static fields, so that the shared objects can be recognised correctly without a data-flow analysis. Whenever a call to `txOpen` is found, a static map of all shared objects is searched to see if the shared object with that name has been seen earlier. If not, a new `RefInfo` object is created, which holds the name of the shared object and a unique integer ID, which is later used in the generation of the UPPAAL model to open the shared object in the transaction template. If the shared object already has an entry in the map of shared objects, we use that so that two different transactions are able to conflict when opening the same shared object in the UPPAAL model.

Loop Bound Analysis

The next analysis that is performed is the loop bound analysis. TetaJ already does a loop bound analysis by accessing the Java source files of the program to read specially formatted comments on the lines before loops begin. However, as mentioned in Section 7.1, we decided to change this analysis to use static method calls for the annotations like in the transaction analysis. Loop bounds are set using a call to `RealtimeSystem.loopBound(int upperBound)`. As in the arguments to `PeriodicParameters` and `SporadicParameters`, the argument to this method must use an integer literal to specify the upper bound of the number of loop iterations. This number is then recorded to be used later in the generation of the UPPAAL templates for each method, so that loops are correctly bounded.

Sporadic Task Firing Analysis

As described in Section 5.4, sporadic tasks are fired by a call to `RealtimeSystem.fire(int eventNumber)`. When this call is made in the bytecode of a method, the sporadic task with the corresponding event number must be fired in the UPPAAL model. During the sporadic task firing analysis, we identify all these calls so that they can generate a synchronisation with the corresponding sporadic thread template in the UPPAAL model to release the task. Locating the calls is done in the same manner as locating loop bounds in the loop bound analysis.

7.2.5 Generate UPPAAL Model

At this point, the generated CFGs are translated into UPPAAL timed automata. The general idea is to let the progress of the automata denote the execution of bytecode instructions, while at the same time considering the execution time of each instruction and the temporal properties defined for each thread.

The UPPAAL system is generated through a series of steps. Figure 7.2 illustrates this process, which is further described throughout this section.

Generating Thread Models

In order to preserve the concepts of threads in our model, threads are captured in separate templates. The characteristics of periodic and sporadic threads differ in their execution, where sporadic threads are one-off and started as a response to a signal, and periodic threads are repetitive. The structure of these templates are based on similar templates found in SARTS.

In Figure 7.2, **Main** uses the `[threadInfo]`s when generating the thread templates. We provide a template generator for each of the types:

`PeriodicThreadTemplateGenerator` Generates a periodic thread template. Its constructor takes a `ThreadInfo` object as an argument and uses it to initialise the properties of the

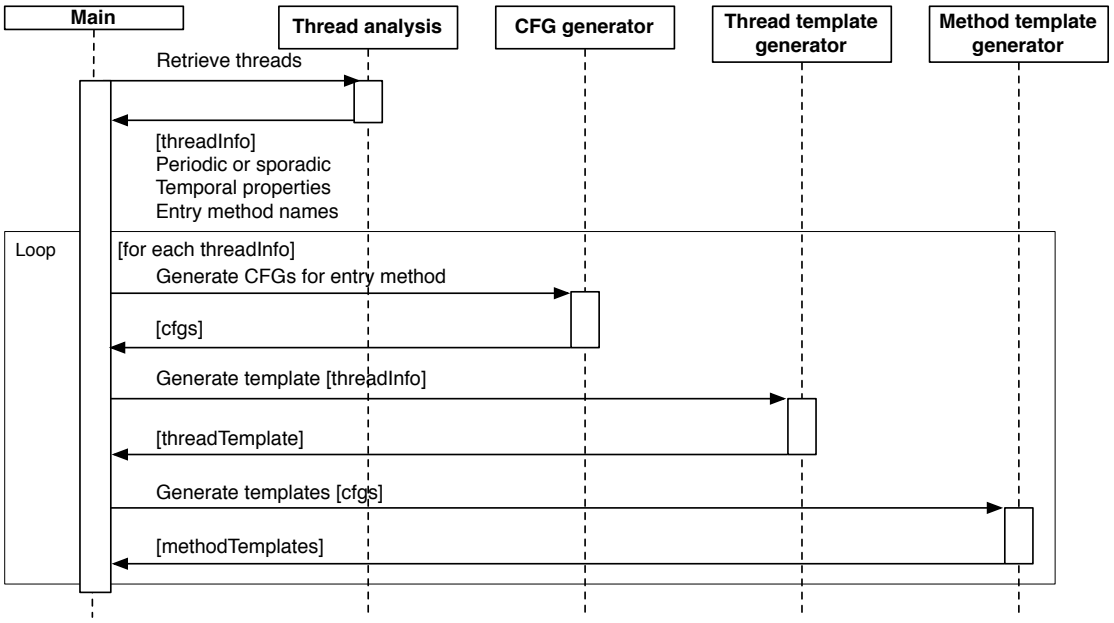


Figure 7.2: A sequence diagram showing the process of generating a UPPAAL system for a compliant Java program using OSAT.

thread template. The temporal properties used are deadline, period, and offset. Besides these, it also uses the fully qualified name of the `run` method.

`SporadicThreadTemplateGenerator` Generates a sporadic thread template. Its constructor takes a `ThreadInfo` object as an argument and initialises the properties of the thread template based on these information. The specific properties used are the event identifier, deadline, and minimum inter-arrival time. The fully qualified name of the `run` method is also used.

The periodic thread template uses the temporal properties to behave as a periodic thread: *deadline* to detect whether the thread finishes in time, *period* to control the interval of execution, and *offset* to delay thread execution. The fully qualified method name is used to start the automaton equivalent of the `run` method. For now, we only describe how the thread templates are generated. The connection between these and starting the `run` method templates is described in the next section.

An example of a periodic thread template is given in Figure 7.3.

`Initial` → `CheckForOffset` This transition is taken once the scheduler synchronises on the `GO` channel. This is a broadcast channel, and as such all thread templates will participate in this synchronisation. This is the equivalent of the SCJ mission start.

`CheckForOffset` → `Tmp1` When the invariant in `CheckForOffset` becomes false and the guard

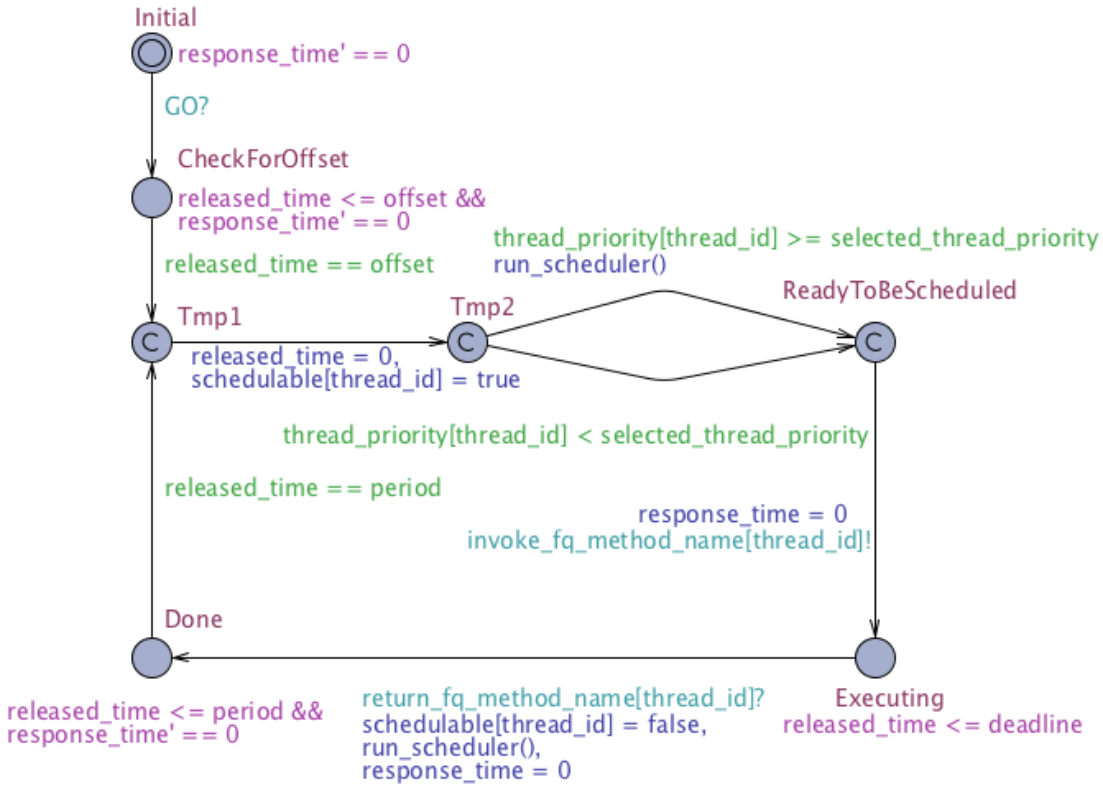


Figure 7.3: The base template for a periodic thread. The channels `invoke_fq_method_name` and `return_fq_method_name` are used to invoke and synchronise on return from `run` methods. The names are placeholders for automatically generated fully qualified method names based on the actual program.

transition `released_time == offset` guard is valid, the automaton enters the `Tmp1` location, symbolising the offset of the thread has passed.

`Tmp1` → `Tmp2` The thread is now released and schedulable, hence the `released_time` clock is reset and the thread is marked schedulable by `schedulable[thread_id] = true`.

`Tmp2` → `ReadyToBeScheduled` If the thread currently running has lower priority, the function `run_scheduler()` is called. This function is described in Section 7.2.6, but at this point it suffices to say it switches the context to this thread.

If the thread currently running has higher priority, the automaton still enters the `ReadyToBeScheduled` location, but without forcing the context switch.

`ReadyToBeScheduled` → `Executing` At this point, the actual code in the thread is signaled to run. This is achieved by having the thread automaton synchronise on a special channel intended to signal the associated `run` method: `invoke_fq_method_name`. Note that this channel name is only a placeholder, and the name of the channel will change according

the fully qualified name of the `run` method in an actual program.

Notice that in the previous un-commited locations, the `response_time` clock is stopped by the stopwatch `response_time' == 0`. Now that the thread is executing, this clock is started and measures the response time of the current thread.

`Executing` → `Done` The invariant in `Executing` forces the out-transition to be taken once the time since release is greater than or equal to the deadline of the thread. However, the out-transition can be taken if and only if the automaton responsible for the actual thread code returns from execution, denoted by the synchronisation on the `run_fq_method_name` channel. If this transition cannot be taken at any point in time, the thread has effectively missed its deadline.

The `response_time` clock is also reset, letting the intermediate value of the clock be the response time of the thread.

`Done` → `Tmp1` Once the thread is done executing *and* its period has passed, the thread is made schedulable again by entering `Tmp1` once again.

Sporadic threads are modeled in much the same way, except sporadic threads are signaled to execute programmatically whereas periodic threads are signaled by time. The base model for sporadic threads is given in Figure 7.4.

Much of the logic behind the periodic thread template also applies for the sporadic thread template, except that sporadic threads do not have offsets and periods, but are signaled to be released and a minimum inter-arrival time:

`Idle` → `ReadyToBeFired` The thread is made ready to fire when the scheduler synchronises on the `GO` broadcast channel, and marked fireable by `fireable[thread_id] = true`.

`ReadyToBeFired` → `ReadyToBeScheduled` Sporadic threads are released whenever another thread synchronises on the `fire` channel array, where the `thread_id`th element denotes the specific thread.

At this point, the sporadic thread is ready to be scheduled and synchronises on both the invoke and return channel, similar to how periodic threads invoke their associated `run` methods.

Generate Method Templates

We already established how CFGs are generated using the `run` method for each thread as entry point. That is, for each tread in the Java program under analysis, we have a complete list of every method invoked by this thread and a CFG for each of these including the `run` method. This is illustrated in Figure 7.5.

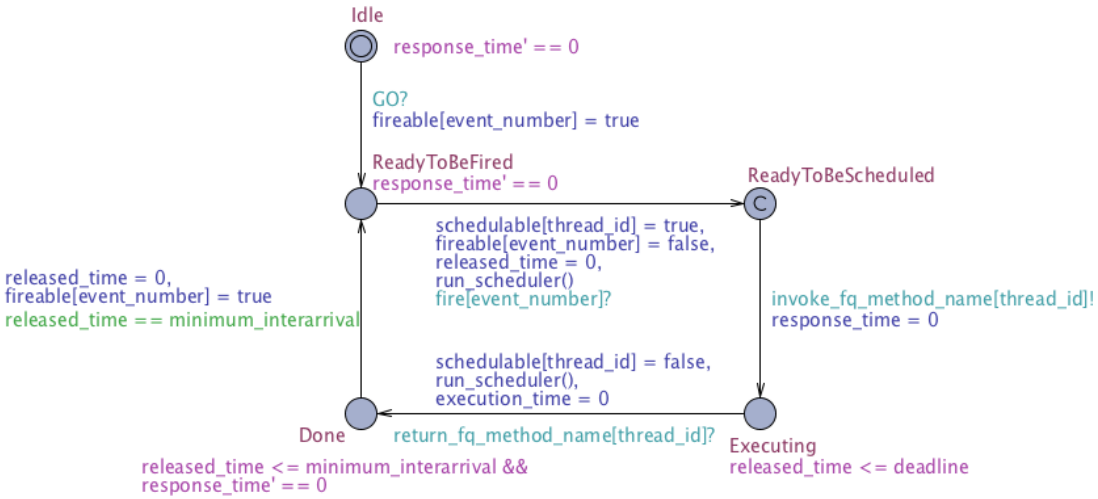


Figure 7.4: The base template for a sporadic thread. The channels `invoke_fq_method_name` and `return_fq_method_name` are used to invoke and synchronise on return from `run` methods. The names are placeholders for automatically generated fully qualified method names based on the actual program.

The final UPPAAL system will consist of a set of templates, where each invoked method in the Java program is represented by a single template and two channels: `invoke_<fq_name>` and `return_<fq_name>` for synchronising upon invocation of and returning from a given method respectively. A simple example is given in Figure 7.6.

The use of the `jvm_execute` channel denotes the invocation of a bytecode instruction, which is handled in a separate JVM model described in Section 7.2.6.

However, the mapping between methods and UPPAAL templates illustrated in Figure 7.6 is not sufficient for multi-threaded programs. If two threads were to invoke `Invokee` at the same time, only one of them would succeed since only one can synchronise on the invoke channel.

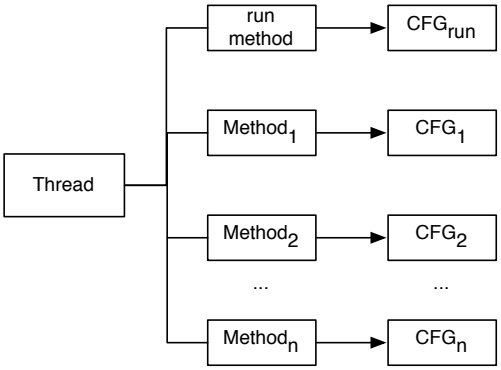
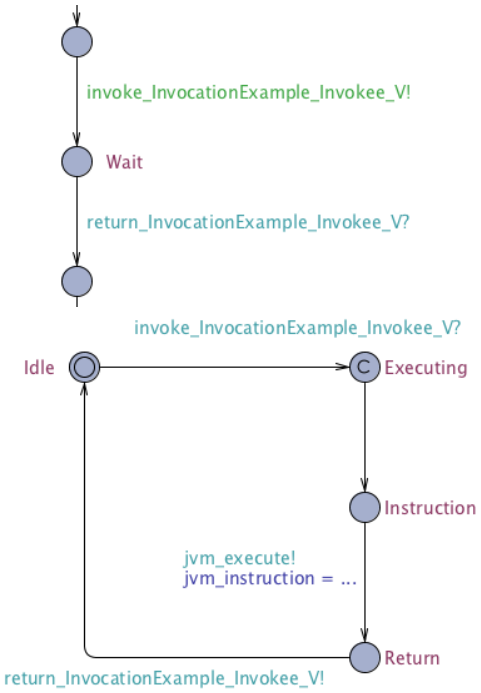


Figure 7.5: For each thread in the Java program being analysed, OSAT generates a CFG for the `run` method and each of the invoked methods.

```
1  class InvocationExample
2  {
3      public void Invoker()
4      {
5          Invokee();
6      }
7
8      public void Invokee()
9      {
10         // Do something
11     }
12 }
```

(a) Example Java code showing a simple method invocation.



(b) Corresponding UPPAAL templates capturing the method invocation found in **(a)**.

Figure 7.6: Methods are mapped to UPPAAL templates. Invocation of and returning from methods is modeled through synchronisation channels.

Furthermore, a single instantiation of each method template would also result in a synchronised invocation of methods.

Clearly, this does not capture the notion of concurrency in the given program. To address this, OSAT does the following:

- Generate `invoke` and `return` channels as arrays of size equal to the number of threads.
- Instantiate one of each method template per thread parameterised by thread IDs.

Figure 7.7 is an excerpt of a generated model, which uses this approach. Notice how `jvm_execute` has also become a channel array now. This is because we also instantiate a JVM template per thread, in order to isolate the execution of each thread entirely. However, using this approach naively would enable the system to perform execution of bytecode instructions in parallel—each of the JVM models will synchronise with the respective method templates, and since there is one of each per thread, they will execute in parallel which is not correct behavior. We overcome this by introducing UPPAAL stopwatches in the model, which is described in Section 7.2.6 where we document the inclusion of JVM, scheduler, and transaction models.

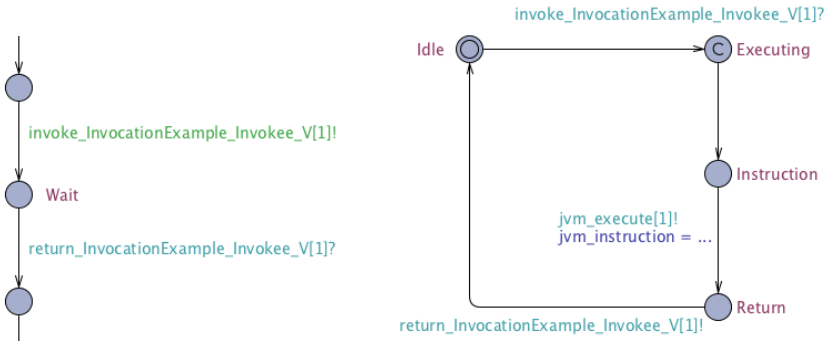


Figure 7.7: To support concurrency in the UPPAAL system, `invoke` and `return` channels have been made arrays of size equal to the number of threads.

Generate Transaction Interaction

When programs using the our STM are analysed, the transaction analysis step described in Section 7.2.4 decorates the CFG with extra information. This information is used to determine when a transaction is started, when shared values are opened, when transactions are committed, and when they are aborted. In order to support the STM concepts in the generated model, OSAT includes the transaction template we defined in Section 6.2. An overview of the template extended to be able to interact with the remaining system is provided in Figure 7.8.

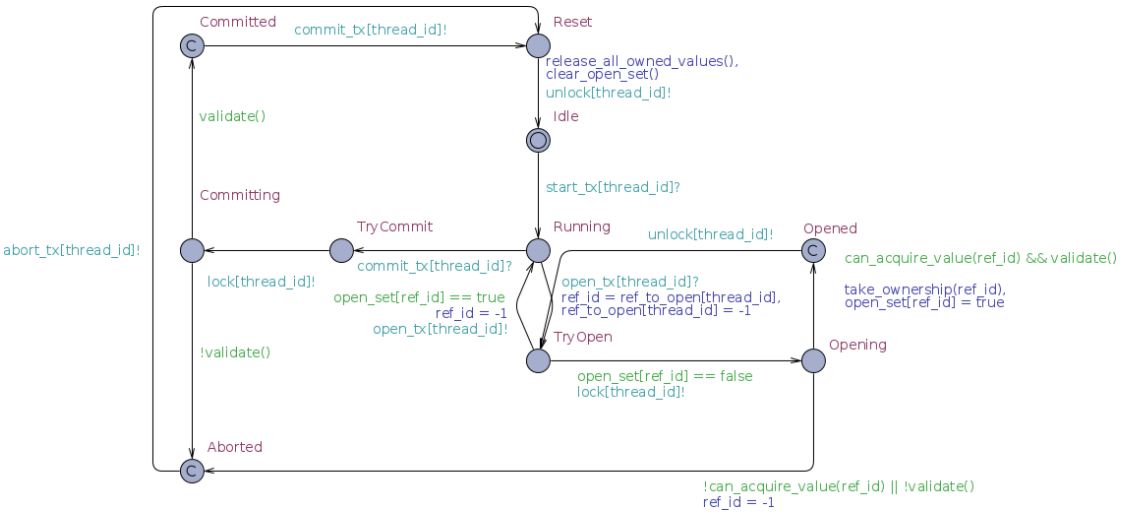


Figure 7.8: The STM model has been extended to contain an `abort` channel in order to interact with the generated models.

Before the transaction analysis, invocations of `txStart`, `txOpen`, and `txCommit` are represented as regular method invocations in the CFG. Using the decorated transaction information, the generated template will become transaction-aware and able to interact with the included STM model: for every encounter of `txStart`, the template will synchronise with the transaction model

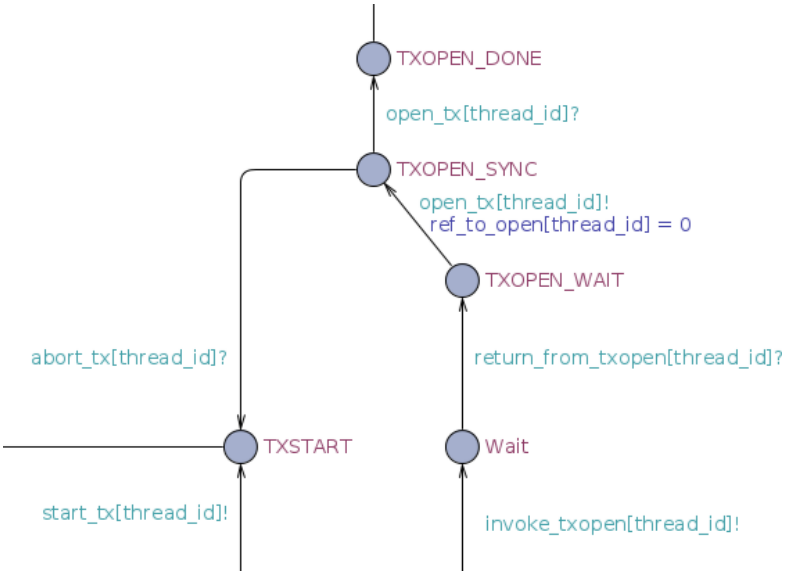


Figure 7.9: An excerpt of a method template, where invocations of STM related methods are extended with synchronisations in order to interact with the STM model.

on the `start_tx` channel. For every `txOpen`, the template will synchronise on the `open_tx` channel using an intermediate variable to hold an identifier of the requested shared variable. For every `txCommit`, the template will synchronise on the `commit_tx` channel, and lastly, if the transaction is aborted, the transaction model synchronises back to the method template. Aborting resets the execution of the invoking method, which is illustrated in Figure 7.9 including examples on how STM actions are mapped.

The `TXSTART` location is entered whenever `txStart` is invoked, and, as it can be seen, it synchronises on the `start_tx_` channel. This signals the transaction template to enter its `Running` location. The out-transition from `TXSTART` indicates execution of some bytecode instructions, but when `txOpen` is invoked, the method template synchronises on the `invoke_txopen` channel[1]. The identifier of the requested shared variable is stored in the `ref_to_open` array, which is then read by the transaction model. Should the opening logic in the transaction model prohibit this shared variable from being opened, the transaction model will synchronise back on the `abort_tx` channel, thus resetting the execution of the transactional block. If it succeeds, the transaction template synchronises back to the method template on the `open_tx` channel again, thus indicating it was successfully opened.

[1]The channel name has been shortened for readability purposes. OSAT automatically inserts a generated name, which is a combination of the fully qualified method name and a unique identifier.

7.2.6 Combine UPPAAL Model

In order to complete the UPPAAL system, additional static templates that we have created must be included. This is achieved in this step, by combining the generated UPPAAL system with the static templates. This is similar to both TetaJ and SARTS.

In TetaJ, this step is performed in a separate Java program that is part of the TetaJ program suite, but we have chosen to integrate the process into the main flow of OSAT, as in SARTS, to avoid having to call an extra program after having run the first steps of the analysis.

The models included are:

Scheduler Maintains the execution and preemption of threads in the system. It is based on the fixed-priority preemptive scheduling scheme.

StaticJvm Responsible for simulating the execution of bytecode instructions on the platform while preserving the worst-case execution times of these.

Transaction Whenever transactions are created and interacted with in the analysed program, the transaction template is used to respond to these interactions.

The scheduler model is responsible for starting threads when they are supposed to be started. That is, when they are schedulable. Figure 7.10 shows the scheduler template which is injected into the generated UPPAAL system.

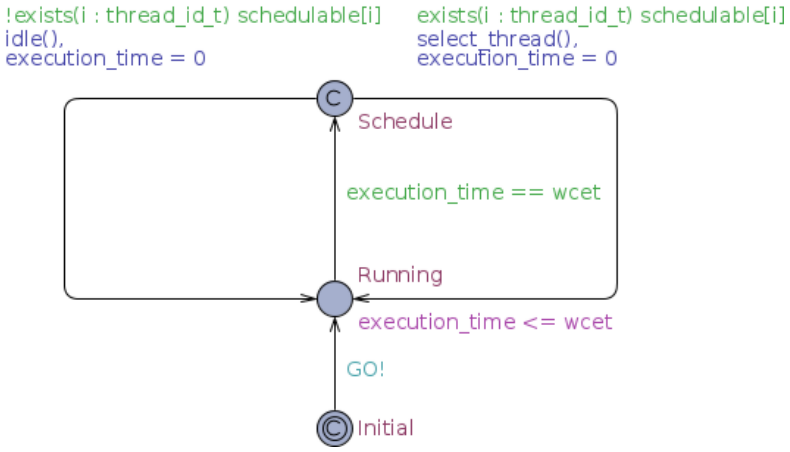


Figure 7.10: The scheduler template which is injected into the generated UPPAAL system as-is.

The invariant in `Running` and the guard on the out-transition simulates the execution time of the scheduler, which is 1 cycle. While this is clearly not enough for a scheduler, it is the number used by SARTS. In future work, the actual WCET for the scheduler of the chosen platform must be used to give a sound analysis.

it is not important in the context of this project as we do not consider jitter in our timing analysis.

Exiting from the `Schedule` location, the scheduler can take one of two transitions depending on whether or not there is a schedulable thread. If not, the `idle()` function is called, which simply marks every thread as *not* running. If there is a thread schedulable, the `select_thread()` function is called, which is where the fixed-priority preemptive scheduling logic resides. The code is provided in Listing 7.4.

```
1  void select_thread()
2  {
3      int i;
4      selected_thread = -1;
5      selected_thread_priority = -1;
6      for (i : thread_id_t)
7      {
8          if (schedulable[i] && thread_priority[i] > selected_thread_priority)
9          {
10             selected_thread = i;
11             selected_thread_priority = thread_priority[i];
12         }
13     }
14
15     for (i = 0; i <= NUM_THREADS; i++)
16     {
17         running[i] = 0;
18     }
19     running[selected_thread] = 1;
20 }
```

Listing 7.4: The source code of `select_thread`, which contains the fixed-priority preemptive scheduling logic for the scheduler template.

The function iterates through every thread identifier in the system, defined by the `thread_id_t` bounded integer type, and leaves `selected_thread` with the identifier of the schedulable thread with the highest priority. Lastly, the selected thread is marked as running. Marking a thread as running allows the associated JVM model to execute its code.

The JVM template is then included. It simulates the execution time of executing bytecode instructions, and is the lowest level of abstraction in the UPPAAL system. An example on how this interaction works is given in Figure 7.11. First, the method template to the left sets `jvm_instruction = JVM_ICONST_3` indicating this is the bytecode instruction to execute. Then it synchronises with the JVM template using the `jvm_execute` channel parameterised by the identifier of the calling thread. Once the JVM model enters the `JVM_Execute_ICONST_3` location, time elapses on the `execution_time` clock if and only if the thread is marked as running—that is, when `running[thread_id] == 1`. This is denoted by the stopwatch syntax `execution_time' ==`

`running[thread_id]` indicating the rate of which the clock increases, which in turn means the execution is stopped when `running[thread_id] == 0`.

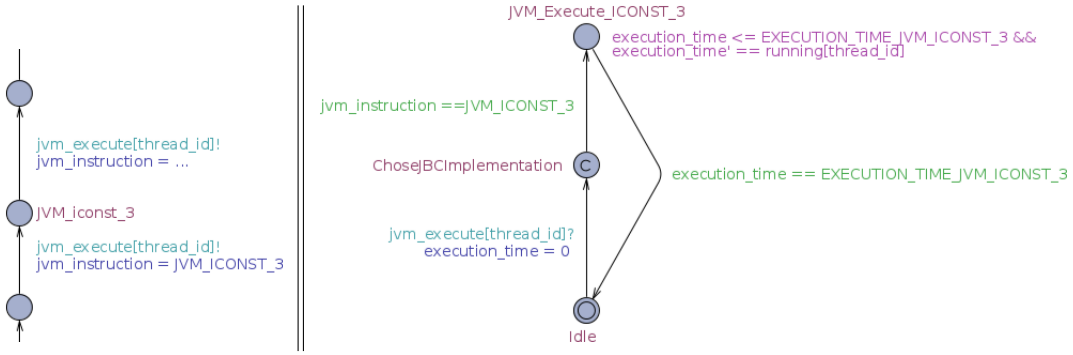


Figure 7.11: Excerpts of the generated UPPAAL system showing how method templates interact with the JVM template, and how stopwatches are used to control execution of the instructions. The JVM model to the right is simplified; showing only how one bytecode instruction is encoded in the template.

In TetaJ, the JVM template communicates with a model of the hardware to provide exact timings for the platform. However, we have simplified this by removing the hardware model, and instead set the timings directly in the JVM template as described above. The transformation from the TetaJ static JVM model to our version was performed using a Python script. This script edited the XML of the static UPPAAL model to remove all synchronisation with the hardware model, and add timing guards and invariants along with the stopwatches we use.

The transaction template is also included. How the generated system interacts with the STM model is described in Section 7.2.5.

7.2.7 Output UPPAAL Model

The final stage of OSAT is where the combined UPPAAL model is written to an XML-file on the disk. Using this file, it is possible to perform the actual schedulability analysis of the program by using UPPAAL to check that the model is deadlock-free. Any deadlocks indicate that it is possible for the system to miss a deadline, thus proving that the system is not schedulable. However, if the system is proven deadlock-free by UPPAAL, then all tasks will always be able to finish executing within their deadlines thus proving that the system is schedulable.

Chapter 8

Experiments

In this chapter, we describe experiments we have performed with OSAT. We have conducted three experiments that show different aspects of OSAT comparing it to SARTS and how our claims in Section 6.2 are fulfilled in practice:

Response Time Comparison with SARTS In an example found in [46] with a periodic and two sporadic threads, we compare response times of the tasks with UPPAAL in models generated by SARTS and OSAT. This experiment shows how similar models generated by OSAT are to those generated by SARTS.

Response Times of Lock-Based and STM-Based Tasks With an example of our own construction, we compare response times of tasks that use lock-based synchronisation with response times of the same tasks using our STM. With this experiment, we show what kind of overhead our STM incurs on a program and how a high priority task critical section is affected by critical sections of other tasks.

Fault-Tolerance Through an example constructed from our previous example, we show how our STM provides certain fault-tolerance to tasks using transactions instead of lock-based synchronisation.

8.1 Response Time Comparison with SARTS

In this section, we use a small example program called `ConditionalSporadic` found in [46] to test OSAT and compare the results with the output of SARTS. Because OSAT uses the same timings as SARTS, the resulting schedulability analyses should match. The test program consists of one periodic task that fires one of two sporadic tasks every period. The code for the periodic task is shown in Listing 8.1 and the code for the two sporadic threads is shown in Listing 8.2.

```
1 package ConditionalSporadic;
2 import javax.scj.*;
```

```
3
4  class ExamplePeriodic extends PeriodicThread{
5    public ExamplePeriodic(PeriodicParameters pp) {
6      super(pp);
7    }
8
9    boolean b = true;
10
11   public boolean run() {
12     if(b)
13       RealtimeSystem.fire(1);
14     else
15       RealtimeSystem.fire(2);
16
17     return true;
18   }
19 }
```

Listing 8.1: Periodic thread that fires one of two sporadic threads every period.

```
1  package ConditionalSporadic;
2  import javax.scj.*;
3
4  class ExampleSporadic extends SporadicThread{
5    public ExampleSporadic(SporadicParameters sp) {
6      super(sp);
7    }
8
9    int j = 0;
10
11   protected boolean run() {
12     RealtimeSystem.loopBound(2);
13     for(int i = 0; i<2; i++); // @WCA loop=2
14     return true;
15   }
16 }
```

Listing 8.2: Sporadic thread that loops twice doing nothing and then returns.

For analysis with OSAT, the code in Listing 8.2 has been modified to use our loop bound annotation, since the SARTS one of `//@WCA loop=2` is not recognised by OSAT.

Using a model generated with SARTS of the original code (without our loop bound annotation), UPPAAL was able to verify that the system is schedulable using the system definition given in Listing 8.3. In this definition, the periodic task has a period of four microseconds which is equal to 240 JOP cycles at 60 MHz. Since the sporadic tasks are released by the periodic thread at most once every period, these have minimum inter-arrival times of 4 microseconds as well.

Task	Priority	SARTS	OSAT
Periodic	1	193	307
Sporadic	2	69	162
Sporadic	3	69	162

Table 8.1: Response times in JOP cycles for the tasks of the ConditionalSporadic program found with SARTS and OSAT.

```
1 new ExamplePeriodic(new PeriodicParameters(4));
2 new ExampleSporadic(new SporadicParameters(1, 4));
3 new ExampleSporadic(new SporadicParameters(2, 4));
```

Listing 8.3: Task definitions for the example real-time system.

In [18], it is shown that a traditional schedulability analysis is not able to verify that the system is schedulable because the response time of the periodic thread would be too high if both the sporadic threads were to be released every period. However, using SARTS, the system can be proven schedulable because the periodic task can only trigger one of the sporadic tasks each period, thus reducing the response time to schedulable levels.

With this experiment, we want to show that OSAT reaches the same conclusion as SARTS with its generated model. However, since the code used for OSAT contains an additional static method invocation in the sporadic tasks to provide the loop bound annotation, we cannot compare the results directly. Instead, we compare the response times of each task and then show that the generated model of OSAT is equivalent to that of SARTS.

To read the response times of the tasks, we modified the SARTS model by hand to add a clock that runs only when the task is released and is reset after the task has finished executing, which means that it records exactly the response time of the task. In Figure 8.1, the modified SARTS template for a periodic thread is shown. The clock we have added to the template is called `response_time`, and to ensure that it only runs in the `ExecutingThread` location, we have added a stopwatch of `response_time' == 0` to all other non-committed locations in the template.

In a similar fashion, we have also added this to the sporadic thread template, so that we can also query the response time of the two sporadic threads.

To find the response time of each task, a `sup` query per task was used: `sup:PeriodicThread1.response_time` gives the response time of the periodic task, while `sup:SporadicThread2.response_time` and `sup:SporadicThread3.response_time` give the response times of the sporadic tasks.

Using the same procedure to get the response times of the tasks in the model generated by OSAT, we get the response times shown in Table 8.1. In this table, we see that the response times of the tasks analysed with OSAT are higher than those found with SARTS. As mentioned, this is due to the invocation of the static `RealtimeSystem.loopBound` method that we use in OSAT. The

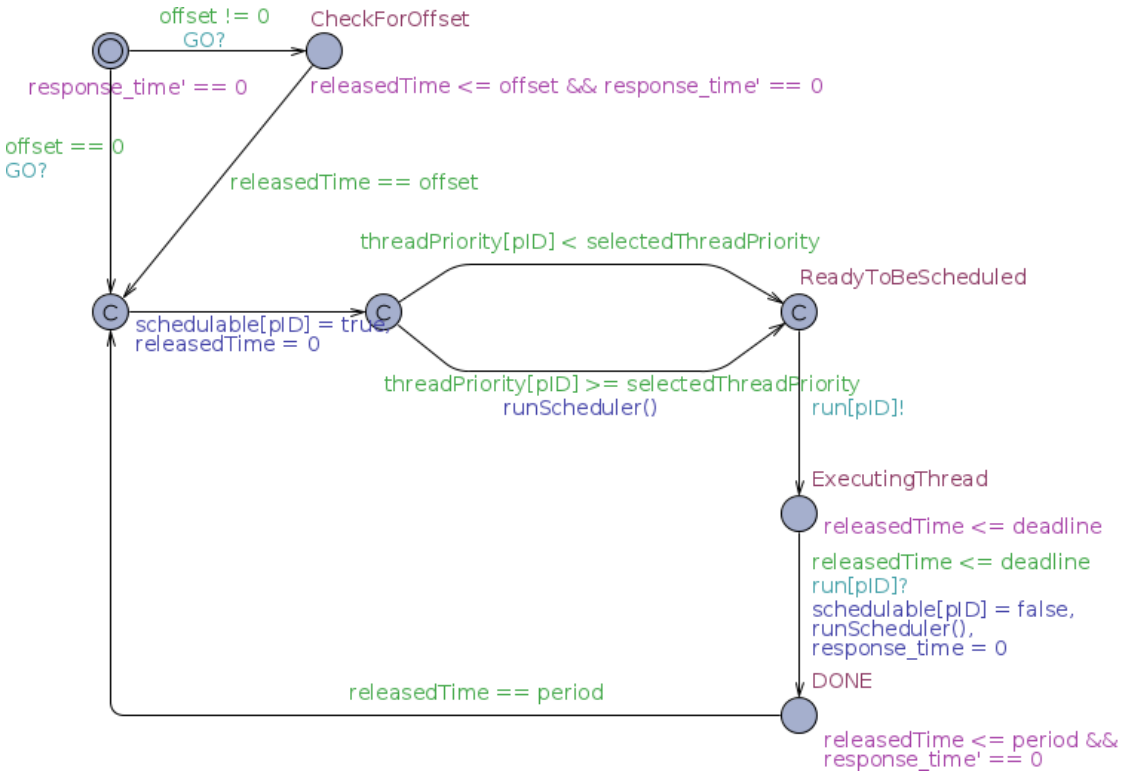


Figure 8.1: The modified periodic thread template in the generated SARTS model.

cost of the STATICINVOKE bytecode for the static method invocation is 75 cycles, to which we must add the cost of pushing the argument to the method onto the stack. In this case, the argument is an integer in the range from -1 to 5, which means that this is an ICONST bytecode which has a cost of 1 cycle[1]. Finally, the void RETURN bytecode to return from the static method costs 21 cycles, so in total the cost of using our loop bound annotation is 97 cycles.

Subtracting the method invocation from the response time found with OSAT gives us $162 - 97 = 65$, which is lower than the 69 cycles found with SARTS. The reason for this is that the compiler we have used to generate the bytecode that we analyse with OSAT is different from the one used with SARTS. The difference is found in the for-loop, where our newer OpenJDK 7 compiler has optimised the loop by performing the evaluation of the loop-condition at the end of the loop and starting the loop with a GOTO, whereas the program compiled for SARTS used an older Java compiler which performed the loop-condition evaluation at the beginning and then performing a GOTO at the end of the loop. This means an additional GOTO is used at the last iteration of the loop, costing 4 cycles and thus gives us the same number of cycles: $69 - 4 = 65$.

Now that we have accounted for the differences in the analysis of the sporadic tasks, we will look at the differences between the analyses of the periodic task:

[1]The general BIPUSH bytecode for pushing an integer literal onto the stack costs 2 cycles.

Because the periodic task fires one of the sporadic tasks, its response time contains the response time of the sporadic tasks. So we will start by performing the same subtractions we did for the sporadic tasks: so for the SARTS generated version we get $193 - 4 = 189$ and for OSAT we get $307 - 97 = 210$. This still leaves a difference of 21 cycles. These remaining cycles are due to SARTS not returning from the static method invoke of the `RealtimeSystem.fire` method. In OSAT, however, we do account for the `RETURN` bytecode, so this adds 21 cycles to the response time. By subtracting these we get $210 - 21 = 189$ which is exactly the number we arrived at for the SARTS generated version.

These calculations show that OSAT and SARTS provide equivalent models, when taking into account certain differences in the environment as we have done above.

8.2 Response Times of Lock-Based and STM-Based Tasks

In this experiment, we look at the response times of tasks when using lock-based synchronisation and when our STM is used. To provide a simple workload for the systems used in the experiment, we have created two periodic threads of the same class that change a static field using the synchronisation mechanism to be tested. The code for the lock-based threads is shown in Listing 8.4 and the code for the STM-based threads is shown in Listing 8.5.

```
1  public static Integer sharedCounter = 0;
2
3  @Override
4  protected boolean run() {
5    synchronized (sharedCounter) {
6      int current = sharedCounter;
7      RealtimeSystem.loopBound(3);
8      for (int i = 0; i < 3; i++) {
9        current = increment(current);
10     }
11     sharedCounter = current;
12   }
13   return true;
14 }
15
16 private int increment(int value) {
17   return value++;
18 }
```

Listing 8.4: The code run by the lock-based periodic tasks.

```
1  public static TValue<Integer> sharedCounter = new TValue<Integer>(0);
2
3  private Integer store;
```

```
4  private TContext c;
5
6  @Override
7  protected boolean run() {
8    RealtimeSystem.transactionStart();
9    do {
10     try {
11       c.txNew();
12       c.txOpen(store, sharedCounter);
13       RealtimeSystem.loopBound(3);
14       for (int i = 0; i < 3; i++) {
15         store = increment(store);
16       }
17       c.txCommit();
18     } catch (AbortException e) {
19       if (c.txRethrowAbort()) {
20         throw e;
21       }
22     }
23   } while (!c.txIsCommitted());
24   return true;
25 }
26
27 private int increment(int value) {
28   return value++;
29 }
```

Listing 8.5: The code run by the STM-based periodic tasks.

Basically, the code of these two thread classes increments the shared integer `sharedCounter` by three. The lock-based version uses Java monitors with the **`synchronized`** keyword, while the STM-based version uses our STM to wrap the increment calls in a transaction. The tasks are created with the system definition in Listing 8.6.

```
1 new WorkerThread(new PeriodicParameters(125, 125, 10)); // Priority 2
2 new WorkerThread(new PeriodicParameters(125)); // Priority 1
```

Listing 8.6: Definition of the real-time system properties for the experiment.

UPPAAL is able to verify schedulability for each of the systems with the query `A[] not deadlock`. In Table 8.2, the results of the schedulability analyses for the systems are shown. The time UPPAAL took to verify the queries has also been added, and shows that there is a considerable overhead to verifying schedulability when using our STM.

Since the systems are schedulable, we can compare the response times of their tasks to see more precisely how the schedulability is achieved and the differences between the response

System	Verification Time	Result
Lock-Based	5 seconds	Satisfied
STM-Based	120 seconds	Satisfied

Table 8.2: Schedulability analyses of the described real-time systems.

Type	Priority	With Offset	No Offset
Lock-Based	1	1496	1494
Lock-Based	2	867	747
STM-Based	1	6473	4654
STM-Based	2	2327	2327

Table 8.3: Response times for the tasks of the ConditionalSporadic program.

times of the tasks using locks and the tasks using our STM. The response times for all tasks are shown in Table 8.3.

The offset, which is defined for the high priority task in Listing 8.6 to be 10 microseconds, ensures that the low priority task is released before the high priority task, thus making it possible for the low priority task to acquire the shared resource before the high priority task. For the lock-based version, this means that the high priority task is blocked while the low priority task finishes its critical section, resulting in the higher response time of the high priority task when the blocking occurs. The low priority task essentially has the same response time with or without the offset, as the difference of two cycles is due to the scheduler, which we have given a WCET of 1 cycle, runs two times more when the low priority task is released first. For the version with no offset, both tasks are released at the same time, and the response time of the high priority task is thus the WCET of the task, whereas the low priority task has a response time of $747 * 2 = 1494$ because it must first wait for the execution of the high priority task to finish before it can use the processor. These interleavings are illustrated in Figure 8.2.

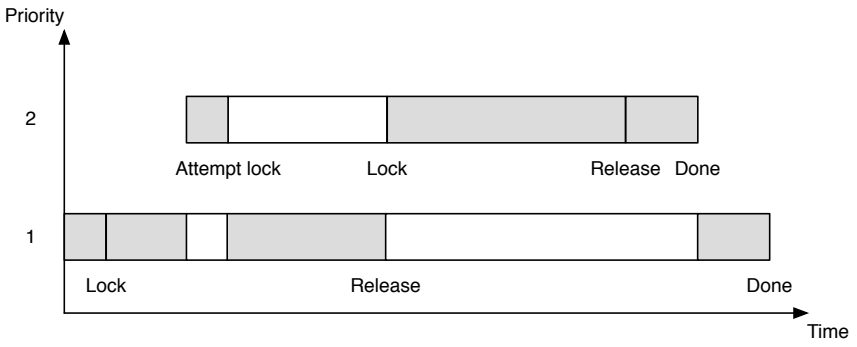


Figure 8.2: Interleavings showing how the high priority task is blocked by the low priority task.

For the STM-based program, we see that the tasks have higher execution times. This is due to the overhead of the STM, as each call to the STM library consists of at least two virtual

method calls which cost over 100 cycles each on the JOP. We see that the version with no offset again provides the case of a response time equal to the WCET for the high priority task and a response time equal to two times the WCET for the low priority task. However, when the offset is introduced, we see that the response time of the high priority task does not increase. This is because the STM allows the high priority task to preempt the ownership of the shared object and execute without delay. Due to the preemption of the ownership, the low priority task is invalidated and must abort and retry its transaction, which increases its response time. This situation is shown in Figure 8.3.

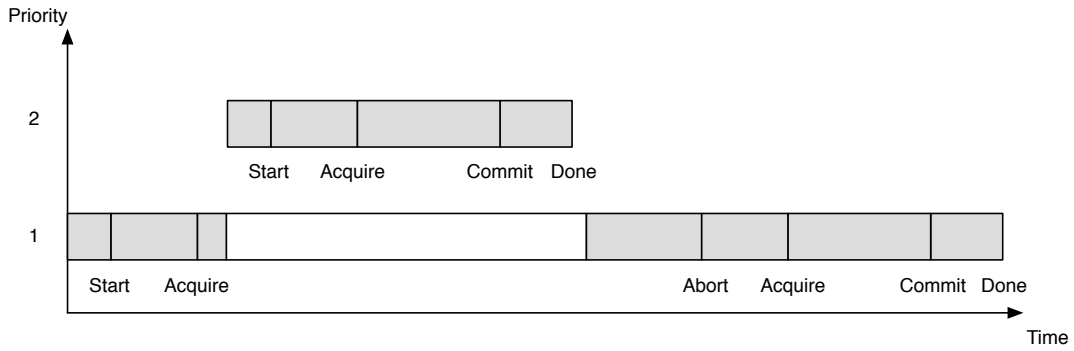


Figure 8.3: Interleavings showing how the high priority task preempts the low priority transaction before the end of its critical section.

The result of having exactly the same response time for the high priority task with the offset as without the offset may not always hold, however. As we remarked in Section 6.2.4, a low priority transaction may block the execution of a higher priority transaction if it is in the process of acquiring a variable or committing and thus holds the global transaction lock. Although the effects of this event are limited, since it is only possible for a transaction execution to be blocked once per execution, and even then it will in the worst case be blocked only for the time it takes to commit the transaction of lower priority that takes the longest to commit. This is potentially much less than in the lock-based program, where the high priority task could be blocked for the entire length of a critical section of a lower priority task.

From our claims in Section 6.1.1, we expected that the STM-based program would allow the high priority task to complete its execution with less time spent blocked than the lock-based program. The experiment confirmed this by comparing with a lock-based program that we used as a baseline to show the amount of blocking that was avoided by using STM instead of locks.

8.3 Fault-Tolerance

For our final experiment, we again use the lock-based and STM-based programs to show what effect it would have on a real-time system if a low priority task misbehaved in a critical section. In our experiment, we use the UPPAAL models of the programs to simulate that the low priority

task of each program crashes in its critical section, however, the same logic applies should the low priority task enter an infinite loop or exhibit similar unexpected behaviour. If the priority of a task denotes the importance of that task, it may be useful for the higher priority tasks of the system to continue executing unaffected. This experiment shows how that is possible using our STM.

In Figure 8.4, we see a small part of the critical section modelled in the UPPAAL template for the `run` method of the tasks (see Listing 8.4 and Listing 8.5). This is part of the loop where the method `increment` is called. To simulate the thread crashing, we have added a guard so that only the high priority thread with the `thread_id` of `2` is allowed to continue, as shown in Figure 8.5.

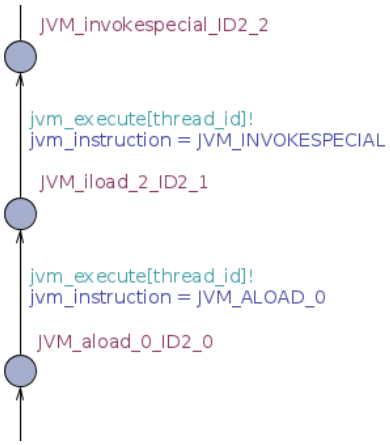


Figure 8.4: Part of the original worker thread template in the body of the for-loop.

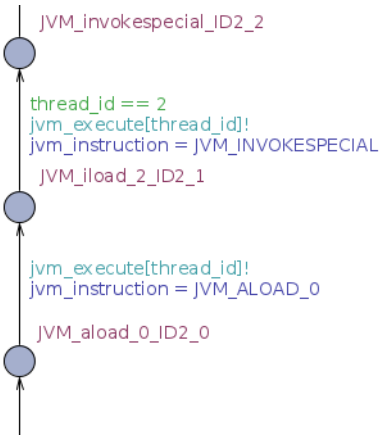


Figure 8.5: The modified template which makes the low priority thread hang at the invokation of the `increment` method.

With the modified model, we see that UPPAAL is no longer able to prove that the system is

schedulable. What is more important in this experiment, though, is that none of the tasks can run to completion. By using the queries `E<> Thread1.Done` and `E<> Thread2.Done`, UPPAAL tells us that these properties are *not* satisfied, which means that it is not possible for either of the tasks to finish executing even their first release.

By performing the same change to the model of the STM-based program, we again find that UPPAAL is not able to prove that the system is schedulable. This is of course because the low priority thread cannot complete its execution, which is verified by UPPAAL telling us that the query `E<> Thread1.Done` is not satisfied. However, with our STM, it is possible for the high priority task to continue executing, because it is not being blocked by the low priority thread. UPPAAL can verify this with the query `A<> Thread2.Done`, which it indeed proves to be satisfied, and running the query `sup:Thread2.response_time` shows us that the response time of the high priority thread remains unchanged at 2327 cycles.

Chapter 9

Evaluation and Future Work

In this chapter, we evaluate the choices we have made throughout this project. As a part of each section, we also suggest future efforts which could improve our work.

We begin by discussing the development of our STM and the implications of introducing STM in a real-time setting in Section 9.1.

This is followed by an evaluation of our work on the HVM in Section 9.2, which has undergone several modifications in order to meet the requirements set up in this project.

Then we evaluate on our tool OSAT in Section 9.3. Reusing functionality from TetaJ and SARTS had it strengths and weaknesses, which are documented here.

Lastly, we evaluate the development process and applied methods used throughout the project in Section 9.4.

9.1 Software Transactional Memory

The blocking STM we designed during this project period has made schedulability analysis more difficult than the analysis we considered for our non-blocking design since it introduced the ability for tasks to block each other, which the non-blocking design did not allow. We were able to reason about the worst-case time a transaction could be blocked, but due to time constraints we could not implement an exact analysis of this in OSAT.

Using OSAT, we discovered that the time it takes to access and execute the code of our STM is significantly higher than using locks. Especially when critical sections are small, we see that the time consumed to execute it consists mostly of running our STM code. In an improved version of our STM, it might be interesting to look at cutting these costs, for example by inlining the method calls of the STM to reduce the number of method invocations that are very costly to execute. Another way to reduce the overhead could be to implement the transactional memory in hardware as in [9], however, the focus on this project was to make an analysable STM, so performance was not considered to be of any concern.

Even with a high overhead, we still find STM to be a valuable addition to real-time systems development, since it guarantees deadlock freedom, minimises priority inversion, provides composability through nesting, and allows for fault-tolerance as demonstrated in Section 8.3. Another useful feature would be to add the syntactic sugar shown in Listing 6.1. This would decrease the amount of boilerplate code needed to use our STM, but would require an additional step to transform the code into using the STM library.

9.2 Hardware-near Virtual Machine

We extended the HVM with multi-threading to be able to use synchronisation mechanisms. However, by using POSIX threads to implement multi-threading, we limited the HVM so it was only able to run on a POSIX OS, which meant that we could not run our code on an embedded platform. Although we still had embedded real-time systems in mind when developing our code, it has not been tested on such a platform. Originally, we had planned to create a larger case-study to test our STM and schedulability tool in a more realistic setting, as done in [19, 18], but we opted to use the experiments in Chapter 8 instead since we would not be able to test the case-study in practice. In addition, the experiments demonstrate specific properties in a very concise manner, which is a strength compared to a case-study where conclusions could be harder to derive.

Multi-threading was added to the HVM by implementing SCJ2 from SARTS. This approach made it easy to compare our work with SARTS, since we had the same API for real-time programs developed for our project as those developed for SARTS on the JOP. Implementing the JOP specific thread-handling code as native functions on the HVM allowed us to reuse much of the SCJ2 code. As such, the approach we took to implement multi-threading proved to be of great success to us, by providing us with a version of the HVM that could be used as a solid foundation for further development.

Before April 2012, the HVM was prone to race conditions in the heap allocation code, which meant that our threads could crash randomly if such race conditions were encountered. This is one of the reasons for HVM being a high-risk choice for this project, but by designing our code to avoid memory allocation in running threads, we were able to work around this problem. With a new release of the HVM in April 2012, this problem was fixed as part of the efforts to implement SCJ with multi-threading on the HVM for all supported platforms. However, by looking through the code-base of the HVM, we still see patterns that are difficult to reason about for worst-case execution time of Java bytecode instructions running on the HVM. In TetaJ, the HVM interpreter was modified to be able to perform a worst-case analysis, but due to the changes to the HVM since, we did not reuse those efforts. OSAT allows for plugging in a model of the platform, so using a model of the JOP we were able to compare our results with those of SARTS. In order to analyse programs targeting the HVM, a new model of the HVM has to be developed.

Being a software interpreter of Java bytecode, the HVM proved to be a good choice for this project. One of the main reasons for this is its open source nature that allowed us to access and modify the inner workings of the HVM. This would have been more difficult had we chosen to use the JOP, which runs Java bytecode in hardware, as our target platform. We also had the benefit of being able to communicate directly with the creator of the HVM, Stephan Korsholm, which allowed us to get help when we discovered problems that we suspected were due to the platform, such as the thread-crashes experienced when allocating memory due to the race condition in the heap allocation code.

9.3 OSAT

We anticipated the time saved from re-using the TetaJ code base would be useful when extending it with concepts from SARTS, and the time saved from re-using SARTS concepts would be useful when tying the generated model together with our STM model. The advantage of doing this was the fact we did not have to create our own object model for a control flow graph (CFG), let alone the UPPAAL model generator. Instead, we modified the object model of TetaJ to contain information on transactions which we utilise in the model generator. As we describe in Section 7.2.4, it was easy to add new analysis procedures which was how we decorated the generated CFGs with transaction details.

We recognise STM usage by looking for invocation of the static methods, indicating the start of a transactional block, creation of a new transaction, opening of shared variables, and committing and aborting a transaction. The method invoked when starting a transactional block is empty, but it still counts as a method invocation in the analysis. This also applies for loop bound definitions, which are empty methods as well. It enables OSAT to operate on class-files only without the need of access to the corresponding source code files. This is a great advantage in that OSAT could potentially analyse programs written in other languages targeting the JVM. As mentioned in Section 7.1, it would be possible for OSAT to remove these invocations from the CFG, thus removing the execution time overhead and still being independent of the source files. The resulting bytecode could then be written to separate class-files, which would be deployed to the platform as optimised bytecode. The overhead could also be reduced by using variable assignments, e.g. `RealtimeSystem.loopBound = 20;`, which are cheaper to execute than method invocations, but removing the method invocations before deployment would completely remove the overhead of the annotations.

UPPAAL was used to great success for this project. In addition to TetaJ and SARTS, UPPAAL has also been used to analyse the schedulability of transactions in [44] but without the containing program. This indicates that UPPAAL is a good base for schedulability analysis of multi-threaded real-time programs using a real-time STM, which has been proven to be true in this project.

Another shortcoming of OSAT, which is due to it being based on TetaJ, is that the analysis

does not consider the time it takes to execute methods of Java library classes, e.g. `java.lang.String`. The same is true for SARTS, where the SCJ2 and JOP library classes are also exempt from analysis. To be able to compare our results with SARTS, we also exempt SCJ2 from being analysed in OSAT. However, this means that all three tools are unsound in this regard. Thus, before OSAT can be considered sound, these shortcomings must be addressed in future work.

In OSAT the JVM model is parameterised like in TetaJ, with the possibility of defining custom worst-case execution timings for each bytecode instruction. We use the timings of the JOP in order be able to compare our analysis with that of SARTS. However, to use OSAT with the HVM, it is necessary to use HVM specific timings.

Further inaccuracy can be found when transactions validate upon opening shared variables and committing, which takes linear time with respect to the current size of the open set. We have prepared for this in both OSAT and STM model, but it is still short of exact timings for objects with a variable number of fields and types. At this point, we provide a fixed timing for this in order to demonstrate the concept, but future work is needed to determine the exact time required.

9.4 Development Process

The assumptions we made about the project being characterised by unknown factors and risks led us to choose a risk-driven agile work style. For example, during the work on the HVM we often hit dead-ends and problems which took more than a week to solve. Based on our risk assessments, the HVM was one of the first things we started to work. If we had begun working on the HVM at a later stage, assuming it would be straight-forward, we could risk facing problems that would take too long to solve compared to the time available.

Since we followed an agile work style, we did not define an exhaustive specification of what we wanted to create, other than stating the basic functional requirements listed in Section 1.2. Neither did we go through a full phase of analysing the problem domain of using the HVM or STM for this project, but instead we analysed and solved problems in smaller steps. If we had provided a full specification of the STM, for example, stating it should be a non-blocking STM to begin with, we could have ended up implementing other pieces of the project which relied on this fact. Due to limitations in the HVM, we decided to change the STM and make it blocking instead. Approaching the parts of this project which had risks and unknown factors in this manner made it easy to change plans as work progressed. In a waterfall-based process, we could end up spending a large amount of time speculating on which limitations we would run into, and which quirks would prevent us from achieving what we wanted.

The model-driven method we applied in developing the STM also proved to be a valuable technique. We did have an STM implementation from the 9th project term, but without the proposed real-time properties, and we found it difficult to convince ourselves that the properties were sound by testing the implementation. Constructing a model of the STM and the real-time

properties made us think about the actual problem instead of the challenge of implementing it, and the model was also used to verify the soundness of the properties. In addition to being an easier way of grasping the STM concepts, the model we constructed from the initial STM implementation also helped prove the correctness of our STM.

At times, we worked through iterations without upholding the one-week iteration length. We are not aware of any negative impact of this, and suspect it is more likely to be a problem in larger teams, where the continuous iteration meetings help keeping everyone up-to-date. Being a group of two members, we worked closely during every phase of this project, and did not notice any negative side-effects.

Chapter 10

Conclusion

In this project, we continued our earlier work [10] in bringing STM to hard real-time systems. In our earlier work, we defined claims which an STM must fulfil in order for it to be suitable in hard real-time systems. We have developed an STM in this project, which fulfils these claims. To verify the validity of the claims, we have used the model checker UPPAAL to construct a model of the STM and express UPPAAL queries that capture our claims.

The STM we have developed respects the priorities assigned to the tasks in the system, and uses them to prevent transactions in lower priority tasks to abort transactions in those of higher priority. This enables the highest priority thread to perform irreversible actions such as I/O, which is useful in embedded real-time systems. However, our STM does not discriminate between reads or writes, thus creating the possibility of contention in read-only systems. In order to make the STM more efficient, future work could involve allowing for multiple concurrent readers.

We chose the HVM as the target platform due to its portability and flexibility. Since it did not support multi-threading, we have extended it with support for this using POSIX threads. To provide an API for the programmer to utilise this, we implemented the Safety-Critical Java profile (SCJ) SCJ2 from SARTS [18].

To determine whether a given real-time program using our STM and SCJ profile is schedulable or not, we have developed a schedulability analysis tool (OSAT) which is capable of generating a UPPAAL model of the supplied program. This model can then be used with UPPAAL to verify whether or not the program is schedulable. OSAT allows plugging in a static model of the platform to become platform agnostic, like TetaJ which is used as the code base for OSAT. To add support for concurrency, which is not present in TetaJ, we implemented concepts from SARTS, and furthermore, we have extended it with features to support the STM we have developed.

Being platform agnostic, OSAT is able to consider any target platform, provided timings for each bytecode instruction can be supplied or analysed. In this project, we have considered the

HVM as a target platform for our STM. However, in order to compare OSAT with SARTS, from which the concurrency concepts are adopted, we have chosen to use the JOP timings in our experiments.

Proving the schedulability of programs using STMs with model checking is a novel technique we have developed based on the work of SARTS and TetaJ. Estimating the execution time of STMs in real-time systems manually has been proposed before [47, 48, 49], but OSAT automates this process by using model checking.

OSAT was tested through three experiments, each showing specific properties of our tool and STM. In the first experiment, we compare models generated by OSAT and SARTS. We do this by comparing response times of each task in the models. Through reasoning, we found the models to be equivalent, providing us with a baseline for response time analysis. In the second experiment, we compare the use of locks and STM. This shows that using STM results in a higher execution time, but on the other hand minimises the blocking time of higher priority threads. The final experiment demonstrates an interesting property gained from using our STM compared to locks that low priority threads cannot prevent progress for the entire system by allowing higher priority threads to preempt critical sections. The experiments also revealed that verifying our STM-based programs required more CPU time compared to lock-based programs. This is due to the fact that transactions are not synchronised in their critical sections as tasks using locks are, thus yielding a larger state space within the model.

Throughout the project, we employed an agile risk- and model-driven development process to structure and organise our work. This minimised the risks and impact of unknown factors which were associated with the nature of the project.

Manually calculating response times for tasks in a system using STM would be complex, considering the many interleavings threads and transactions can display. Model-based schedulability is used to automate the process. However, OSAT still has limitations in its analysis in that it underestimates the time required for scheduling and leaves out certain parts of SCJ2, transactions, and built-in Java libraries. Before OSAT can be considered sound in its analysis, these missing parts must be added. These additions have been prepared for in OSAT to encourage this development.

Bibliography

[1] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1:134–152, 1997. 10.1007/s100090050010.

[2] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *Queue*, 6:16–25, September 2008.

[3] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41:253–262, October 2006.

[4] Simon P. Jones. *Beautiful Concurrency*. O'Reilly Media, Inc., 2007.

[5] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, May 1993.

[6] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[7] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with java stm. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-3)*, 2010.

[8] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time java. In *In 26th IEEE Real-Time Systems Symposium*, 2005.

[9] Martin Schoeberl, Florian Brandner, and Jan Vitek. Rttm: real-time transactional memory. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 326–333, New York, NY, USA, 2010. ACM.

[10] Marcus Calverley and Anders Christian Sørensen. Towards software transactional memory in hard real-time java systems, 2011. 9th term pre-specialisation project. Available online: `http://sw9.lmz.dk`.

[11] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, fourth edition, 2009.

[12] Stephan Korsholm. Hvm lean java for small devices. `http://icelab.dk`. [Online; accessed 30 Mar 2012].

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

[14] Craig Larman. *Agile and Iterative Development – A Manager's Guide*. Pearson Education, 2003.

[15] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[16] Ken Schwaber and Jeff Sutherland. The scrum guide: The definitive guide to scrum, rules of the game. `http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf`. [Online; accessed 22 Jan 2012].

[17] *eXtreme Programming Explained: Embrace Change*. Addison-Wesley, us ed edition, 2001.

[18] Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 106–114, New York, NY, USA, 2008. ACM.

[19] Christian Frost, Casper Svenning Jensen, and Kasper Søe Luckow. Wcet analysis of java bytecode featuring common execution environments. Master's thesis, Aalborg University, 2011.

[20] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[21] X/Open Base Working Group. Real-time extension in unix. `http://www.unix.org/version2/whatsnew/realtime.html`. [Online; accessed Dec 6 2011].

[22] Oracle-Sun Developer Network. An introduction to real-time java technology: Garbage collection. `http://java.sun.com/developer/technicalArticles/Programming/rt_pt2/`. [Online; accessed Dec 2 2011].

[23] Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, 2009.

[24] Oracle Java Community Process. Real-time specification for java (rtsj). `http://www.rtsj.org/specjavadoc/book_index.html`. [Online; accessed Oct 9 2011].

[25] Alan Mycroft. Programming language design and analysis motivated by hardware evolution. In *SAS*, pages 18–33, 2007.

[26] Michael Gonzalez Harbour. Real-time posix: An overview, 1993.

[27] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[28] Oracle Corporation. Java real-time system. `http://java.sun.com/javase/technologies/realtime/index.jsp`. [Online; accessed Dec 2 2011].

[29] CISS. Indlejrede systemer skal tale java. `http://ciss.dk/dk/projekter/afsluttede_projekter_-_case_stories/indlejrede_systemer_skal_tale_java.htm`. [Online; accessed Dec 28 2011].

[30] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a java processor. *Softw. Pract. Exper.*, 40(6):507–542, May 2010.

[31] The Open Group. Jsr 302: Safety critical java technology. `http://jcp.org/en/jsr/detail?id=302`. [Online; accessed Mar 5 2012].

[32] Haskell. `http://www.haskell.org/`. [Online; accessed Mar 9 2012].

[33] Inc. Oracle. Trail: The reflection api. `http://docs.oracle.com/javase/tutorial/reflect/index.html`. [Online; accessed 22 Apr 2012].

[34] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *POPL*, pages 117–126, 1983.

[35] Edmund Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin / Heidelberg, 1982. 10.1007/BFb0025774.

[36] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Toward a provably-correct implementation of the jvm bytecode verifier. In *In Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, pages 403–410, 1998.

[37] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration, 1995.

[38] Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based wcet analysis. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany. also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6.

[39] Peter Puschner and Anton Schedl. Computing maximum task execution times - a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997.

[40] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[41] Wind River. Rtlinux. `http://www.rtlinuxfree.com`. [Online; accessed 20 Mar 2012].

[42] Rtlinux howto: Why rtlinux. `http://tldp.org/HOWTO/RTLinux-HOWTO-3.html`. [Online; accessed 24 Apr 2012].

[43] Oracle. Package java.util.concurrent.atomic. `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html`. [Online; accessed Dec 20 2011].

[44] C. Belwal and A.M.K. Cheng. Schedulability analysis of transactions in software transactional memory using timed automata. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 1091 –1098, nov. 2011.

[45] Erik Yu-Shing Hu, Guillem Bernat, and Andy Wellings. A static timing analysis environment using java architecture for safety critical real-time systems. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, WORDS '02, pages 77–, Washington, DC, USA, 2002. IEEE Computer Society.

[46] Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Schedulability analyzer for real-time systems (sarts). `http://sarts.boegholm.dk/old.php`. [Online; accessed 20 May 2012].

[47] António Barros and Luís Miguel Pinho. Managing contention of software transactional memory in real-time systems. Technical report, CISTER Research Center, Polytechnic Institute of Porto, Portugal, 2010.

[48] Sherif F. Fahmy, Binoy Ravindran, and E. D. Jensen. Response time analysis of software transactional memory-based distributed real-time systems. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 334–338, New York, NY, USA, 2009. ACM.

[49] S.F. Fahmy, B. Ravindran, and E.D. Jensen. On bounding response times under software transactional memory in distributed multiprocessor real-time systems. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 688 –693, april 2009.

Appendix A

Example: Response Time Analysis for FPS

Here we give a short example of how to calculate the response time for a task using response time analysis as described in Chapter 3. The formula for calculating the response time of a task is given in Equation A.1 which is a recurrence relation that converges on a fixed point. More details on how the formula is constructed is given in [10].

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

Given the task set of independent tasks in Table A.1, we can establish the worst-case response time for the highest-priority task a equals its computation time, that is $R_a = 2$ since no blocking can occur. The response times of the remaining tasks are calculated using Equation A.1. The iterations needed for calculating the response time for task b is defined here:

Task	**Period (T)**	**Computation time (C)**	**Priority (P)**
a	4	2	2
b	10	3	1

Table A.1: Example task set.

$$\begin{aligned} w_b^0 &= 3 \\ \\ w_b^1 &= 3 + \left\lceil \frac{2}{4} \right\rceil 3 \\ w_b^1 &= 6 \end{aligned}$$

$$
\begin{aligned}
w_b^2 &= 3 + \left\lceil \frac{6}{4} \right\rceil 3 \\
w_b^2 &= 9
\end{aligned}
$$

$$
\begin{aligned}
w_b^3 &= 3 + \left\lceil \frac{9}{4} \right\rceil 3 \\
w_b^3 &= 12
\end{aligned}
$$

$$
\begin{aligned}
w_b^4 &= 3 + \left\lceil \frac{12}{4} \right\rceil 3 \\
w_b^4 &= 12
\end{aligned}
$$

The fourth iteration w_b^4 yields the same result as the previous iteration w_b^3, and thus the response time for task b in the example task set equals 12. Finally, asserting whether or not the task is schedulable depends on whether $R \leq T$ for the given task, and since $12 \leq 12$ task b is deemed schedulable with respect to both its own computation time and the worst-case interference from task a.