Model-Based Schedulability Analysis of Real-Time Systems

Master Thesis
SW10 - Spring 2008
Project Group: d601a

AALBORG UNIVERSITY

Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg Ø
Telephone 99 40 99 40
http://www.cs.aau.dk

Title:
Model-Based Schedulability Analysis of Real-Time Systems

Theme:
Real-Time Embedded Systems

Semester:
Software Engineering
Master Thesis
Feb. 4^{th}, 2008 - June 10^{th}, 2008

Group:
d601a

Members:
Thomas Bøgholm
Henrik Kragh-Hansen
Petur Olsen

Supervisors:
Bent Thomsen
Kim Guldstrand Larsen

Copies: 6

Report - pages: 117

Appendices: 3

CD-ROM: 1

Total pages: 137

Abstract:

This report describes the implementation of SARTS, a model-based schedulability analysis tool for hard real-time systems. SARTS translates hard real-time systems, implemented in Java, to an abstract time preserving model for the modeling tool UPPAAL.
The system being analyzed must be implemented in SCJ2, a safety critical profile for Java developed in this project, based on SCJ. The target hardware is the time predictable Java processor JOP, developed specifically for hard real-time systems.
Several experiments have been conducted to illustrate the accuracy of SARTS compared to existing tools and techniques. It is shown that a model-based approach can result in a more accurate analysis, than possible with traditional approaches.
SARTS has successfully been used to verify the schedulability of a real-time sorting machine consisting of two periodic and two sporadic tasks. Future improvements and research directions for SARTS are presented.
This document, source code, and other relevant material can be found on the enclosed CD. The content of the CD and any updates can also be found at `http://sarts.boegholm.dk/`.

Preface

The following master thesis is written during the spring 2008, by three Software Engineering students, at the Computer Science Department at Aalborg University.

When the words *we* and *our* are used, it refers to *the authors* of this report and *he* refers to *he/she*.

When code is presented, it may differ from the actual source code. It may have been modified and have had some details removed to make it fit into the report. This has been done to heighten the legibility of the code, and to focus on essential functionality. The entire source code can be found on the enclosed CD.

The first time an abbreviation is used, the entire word or sentence is written, followed by the abbreviation in parentheses. Throughout the rest of the report, the abbreviation is used. Abbreviations and their meanings are listed in Appendix A.

This report is based on the research performed in [6], which is included on the enclosed CD. Knowledge about the information presented in this report is assumed, in particular Chapters 2, 5, 6, 7, and 9. This covers real-time systems in general, modeling tools, and a test case implementation of a real-time application. These subjects are briefly introduced in this report.

We would like to thank Bent Thomsen and Kim Guldstrand Larsen for supervising this project. Furthermore we would like to thank Anders P. Ravn, Martin Schoeberl, Hans Søndergaard, and Stephan Korsholm for feedback and interesting discussions, and Alexandre David for feedback on UPPAAL technicalities.

Thomas Bøgholm Henrik Kragh-Hansen

Petur Olsen

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Embedded systems are in widespread use in everyday devices and critical systems. Most embedded systems have real-time requirements, i.e. they must respond to events timely, to guarantee a certain degree of quality. Real-time systems are divided into soft and hard systems, where missed deadlines in soft systems is undesirable, it must not happen in hard systems. When hard real-time systems are used as traction control in cars or as a part of a nuclear power plant, missed deadlines can cause loss of lives. Missed deadlines can also cause a significant financial loss, e.g. in banking systems.

The dependability of embedded systems is therefore often very important, and it is desirable to verify that certain properties hold, e.g. that the traction control always reacts within a given time.

The traditional approach to verifying that no deadline misses occur, is to use the Worst Case Execution Time (WCET) of the tasks, in different analyses, such as utilization test. The nature of these analyses is to assume everything can go wrong, and make sure the system has enough computational power to cope with this situation. This approach often results in a very pessimistic analysis, since the worst case assumed might never occur. Due to this pessimistic nature, a new approach is desirable.

Several modeling tools exist, where the general idea is to model the system, and verify that certain properties hold. Some tools also allow the developer to check whether deadlines are missed, based on a scheduling strategy and a WCET for each task; other tools must then be used to estimate this WCET. A tight correspondence between the model used in these tools and the actual implementation is required, in order to rely on the guarantees given.

This project focuses on improving the schedulability analysis of real-time systems. The approach developed in this project, is to translate an existing implementation of a real-time system to a model for a model checker, and using the model checker to verify that deadline misses never occur.

The purpose of this thesis is to:

> *Develop a model-based schedulability analysis to achieve a more accurate result than possible with traditional approaches.*

This schedulability analysis must consider blocking, interference, context switch etc. This improves the accuracy of the analysis, while ensuring a tight correspondence between the model and the actual implementation.

This project targets Java as the language for developing hard real-time systems. Some of the arguments for using Java is the high level of abstraction. Another advantage of Java, is its popularity; a lot of new programmers learn Java as their first, and sometimes only, programming language. Thus it is desirable to minimize the transition from developing standard applications to developing real-time systems.

Processors executing Java bytecode have been developed [2, 31]. Implementing the Java Virtual Machine (JVM) directly in hardware, should increase the performance of Java [30], making it comparable to executing C programs. However, the most important feature is predictability of execution time. These processors focus on a predictable WCET, rather than a fast average execution time.

The contribution of this project is a tool called Schedulability Analyzer for Real-Time Systems (SARTS). SARTS performs a fully automatic translation of real-time Java applications into Uppaal models, on which schedulability analysis is performed. This analysis targets the time predictable Java Optimized Processor (JOP). Some problems exist with such an analysis, but we believe these can be solved with further research.

The remainder of this report, is structured as follows:

Part I provides an introduction to real-time systems, including schedulability analysis theory. A proposal for a hard real-time Java profile, SCJ2, is presented and implemented for JOP. Finally a case study, implemented in SCJ2, has been introduced. This case study is used as an example throughout this report.

Part II describes the implementation of SARTS. Initially relevant technologies are presented. The design of SARTS is described followed by the actual implementation. Finally optimizations to this implementation are presented.

Part III describes the results and future directions of SARTS. Several experiments have been conducted, illustrating the accuracy and scalability of SARTS. Further improvements, which have not been implemented, are also discussed. Finally this project is concluded upon, followed by future research areas.

Part I

Real-Time Systems

Chapter 2

Real-Time Systems

A real-time system is a system which must respond timely to events. *"The correctness depends not only on the logical result of the computations, but also on the time at which those results are produced."* [41]. This means that predictability in execution time is more important than performance in execution time, when developing real-time systems.

This chapter provides general terminology regarding real-time systems and a brief introduction to real-time concepts, which are used in the rest of this report. This is to give the reader a short introduction to real-time systems, and to ensure a common notion of terminology. For more details regarding real-time systems in general, and a more thorough explanation of the different concepts introduced in this chapter, refer to [7]. Furthermore a development process is proposed for real-time systems.

A real-time system contains a set of tasks, which have some sort of timing constraints. To describe these tasks the notation in Table 2.1 is used.

Notation	**Description**
B	Worst-case blocking time of the task
C	Worst-case execution time (WCET) of the task
D	Deadline of the task
I	The interference time of the task
J	Release jitter of the task
N	Number of tasks in the system
O	Offset of the task
P	Priority assigned to the task
R	Worst-case response time of the task
T	Minimum time between task releases (period)
U	The utilization of each task (equal to C/T)
a - z	The name of a task

Table 2.1: Standard notation of real-time concepts [7, p. 467]

The focus in this project is on *hard* real-time systems, where deadline misses may result in a critical failure, and must be avoided; hard real-time systems are used to monitor nuclear power-plants, airplanes, and other critical applications. Critical failures may result in loss of lives, financial loss, or cause environmental damage. Real-time systems may also be *soft*, *firm* or a mixture of the three [7], however, these are not discussed further. The remainder of this report focuses on hard real-time systems and *real-time system* refers to *hard real-time system*, unless otherwise noted.

2.1 Types of Tasks

In order to specify timing constraints in real-time systems, different types of tasks are introduced. In general a task can be either periodic or aperiodic. Each tasks may have a static *priority*, P, or it can be assigned dynamically at runtime, depending on the scheduling strategy. This is discussed further in Chapter 3.

A periodic task is released with a fixed interval, known as its *period*, T. It has a *deadline*, D, which must be less than or equal to the period, $D \leq T$. A periodic task might have an *offset*, O, which represents an offset for when the periodic task should be released after the initialization of the system.

An aperiodic task is not released with a fixed interval, but can be released at any time. This is not suitable in a hard real-time system, because it is not possible to guarantee that the system is schedulable, e.g. if it is released more often than its WCET. A specialization, called a sporadic task, is introduced, which includes a *minimum inter-arrival time*, denoted by T. It deviates from the period of a periodic task, by not necessarily being released with a T interval, it only guarantees that it will not happen more often. Aperiodic and sporadic tasks are released by *events*, either software or hardware generated events. Aperiodic tasks are not discussed further, and only sporadic tasks are used in the remainder of this report. Sporadic tasks also have a deadline similar to periodic tasks. A sporadic task has no offset.

The WCET, C, of a task must be calculated based on the actual implementation of the task and the executing hardware. The WCET must always be less than or equal to the deadline, $C \leq D$.

In general D, T, and O must be independent of the actual implementation, and specified in the requirements for each task. P is dependent on the scheduling strategy. The remainder of the notations in Table 2.1 are dependent on the actual implementation of the system, and the underlying hardware.

2.2 Execution Time

By definition deadlines must not be missed. To guarantee this property, predictability of execution is necessary. In order to achieve this, the underlying hardware must be predictable. JOP is a Java processor targeting real-time systems, with focus on predictable execution. Standard processors focus on optimizing the average case, where the worst case is more important in a real-time environment. JOP is described further in Chapter 5.

As mentioned, the average execution time is not the main factor when optimizing an application. Since the system is allocated enough execution time to execute its worst case execution path under all circumstances, anything using less time than that, makes the processor idle for the rest of the allocated time. This makes it important to minimize the WCET.

As an example, consider the choice of sorting algorithm. Many systems use quicksort as sorting algorithm because of its good average execution time, which is $O(n \log n)$; at worst case quicksort runs in $O(n^2)$ [11]. This makes quicksort a bad choice in a real-time system because of the bad worst case. An algorithm such as heapsort is a more suitable choice with a WCET of $O(n \log n)$ [11].

When calculating the WCET of a task, the execution time of each bytecode must be known. If it is a linear task, it is trivial to calculate the WCET if the execution time of each bytecode is known, i.e. the WCET of each bytecode is added. However, it is no longer trivial when branches, loops, method invocation etc. are introduced, i.e. determining the maximum amount of iterations in a loop is non trivial. A simple loop is shown in Listing 2.1, where the loop bound is ten, provided that i is not altered inside the loop. The WCET of such a loop is simply the loop body multiplied by ten.

```
1 for (int i = 0; i < 10; i++)
2     ...
```

Listing 2.1: A simple for loop

However, the loop bound of the for loop shown in Listing 2.2 may change at runtime, depending on the size of the array.

```
1 for (int i = 0; i < array.length; i++)
2   ...
```

Listing 2.2: A non trivial for loop

To compensate for this, most WCET analyzing tools force the programmer to annotate loop bounds [14, 17, 33]. This is also the approach in this project, because automatic loop bound annotation is still a research topic e.g. [17] where Java Modeling Language (JML) is used to specify application independent annotations used to derive provably correct loop bounds.

However, knowing the WCET of each task is not enough to guarantee that deadlines are not missed, for this, a schedulability analysis must

be performed. Given a set of periodic and sporadic tasks, their WCET, and a scheduling strategy, a schedulability analysis can determine whether it is possible to schedule these tasks, such that deadline misses never occur. If the worst case execution path is schedulable, then all other paths are schedulable as well, since they by definition have a shorter execution time. Blocking and interference complicates the schedulability analysis even further. Schedulability analysis is described in further details in Chapter 3.

2.3 Development Process

In [6], we proposed a development process, with the purpose to aid the development of hard real-time systems. The process is depicted Figure 2.1.

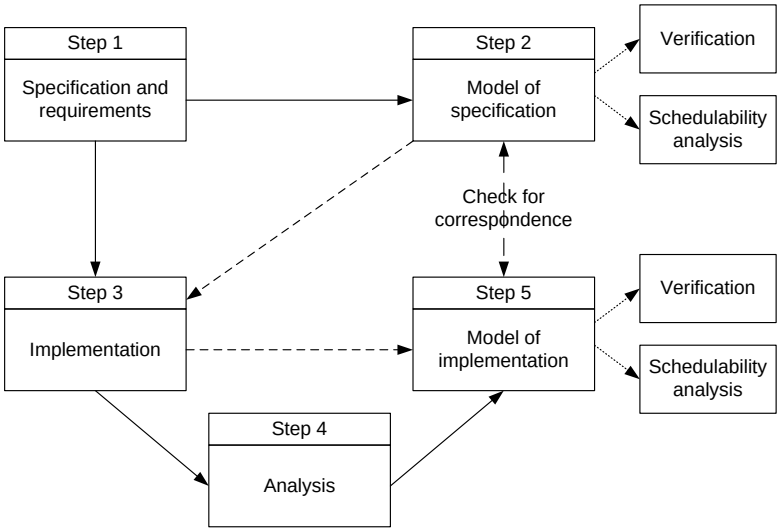


Figure 2.1: Real-time development process

The idea is to specify the system requirements and the real-time constraints in step 1, these are then modeled in step 2. This initial model is a cost effective way to indicate whether it is possible to develop a real-time application corresponding to the requirements. The system is then implemented in the desired target language in step 3. Step 4 is used to perform analysis on the implementation, e.g. WCET analysis. The actual implementation is then translated to the input language of a modeling tool, on which verification is performed, using the analysis performed in step 4. This allows a schedulability analysis, to ensure no deadline misses. Several tools exist to aid the developer in each of these steps, but none support the entire development process. The tools also have different limitations, and target different languages. For a more thorough description of the development process and different tools supporting the different steps, see [6].

2.4 Summary

In this chapter basic concepts of real-time systems have been introduced, and a proposal for a development process for real-time systems has been presented. The terminology presented in this chapter is used throughout the report.

Some of the problems with WCET analysis have been pointed out. WCET analysis is needed to perform a schedulability analysis. The next chapter describes schedulability analysis in general.

Chapter 3

Schedulability

To ensure that a real-time system meets its deadlines, the system must be schedulable, ensured through schedulability analysis. A system is said to be *schedulable* if it can be guaranteed never to miss a deadline. *Scheduling* is the act of choosing which task to allow execution and when to preempt it. A *schedulability analysis* is performed to verify that the system can be scheduled by a given scheduling strategy. Different scheduling strategies include [7]: Fixed-Priority Scheduling (FPS), Earliest Deadline First (EDF), and Value-Based Scheduling (VBS). FPS assigns a fixed priority to each task. EDF is a dynamic scheduling strategy, where the highest priority is assigned to the task with the closest absolute deadline. VBS is a more advanced scheduling approach where a value is assigned to each task, which can also cope if the system is overloaded.

As mentioned, SCJ2 uses a fixed-priority preemptive scheduler, with deadline monotonic priority ordering. This section focuses on traditional approaches to analyzing a system running under such a scheduler. Some restrictions are put on the system under analysis. The deadline of a task must be less than or equal to its period. Sporadic tasks are supported, but must be supplied with a minimum inter-arrival time, and will be treated as periodic tasks with period equal to their minimum inter-arrival time. A WCET for each task must be supplied, WCET is described in Section 2.2. The initial formulae described, assume that context switches are instant, furthermore the scheduler execution time is ignored. How to incorporate the scheduler and context switches in the analysis is discussed in Section 3.4.

The formulae are based on the assumption that the critical instant is the worst possible situation. The critical instant is the point in time where all tasks are released at the same time. Considering the critical instant is safe because introducing an offset can only prevent a critical instant.

This section describes traditional approaches to schedulability analysis, and gives an introduction to *model-based schedulability analysis*.

3.1 Utilization Test

Analyzing the utilization of the processor for a given set of tasks can give an indication of schedulability. The utilization test of a system is performed by using the formula:

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq U(N) \tag{3.1}$$

where

C_i is the worst-case execution time of task i

T_i is the period of task i

and

$U(N) = N(2^{\frac{1}{N}} - 1)$

is the utilization bound for N tasks

Note that the utilization is only dependent on the number of tasks and is a lower bound on guaranteed processor utilization available, assuming tasks are independent and non-blocking [7, 22]; if this condition holds, the system is schedulable. Even though these assumptions are not always true, this solution is often used because of its simplicity [7].

This is a sufficient but not necessary test. Meaning that if a system passes the test, it is guaranteed to be schedulable, but it can still be schedulable even if the test fails.

A small example of a system failing the utilization test, even though it is schedulable is shown in Table 3.1. The system consists of two periodic threads, and their combined utilization is 1, and the threshold for the utilization test for two tasks is 0.828. The priorities in Table 3.1 has been assigned using rate monotonic priority assignment; higher priority value means a higher priority.

Process	**Period,** T	**WCET,** C	**Priority,** P	**Utilization,** U
a	20	10	1	0.50
b	10	5	2	0.50

Table 3.1: Utilization example

However, the actual behavior of the execution of these tasks is depicted in Figure 3.1, and it can be seen that no deadlines will be missed, the systems is therefore schedulable. Note that this is still assuming that tasks are independent and non-blocking.

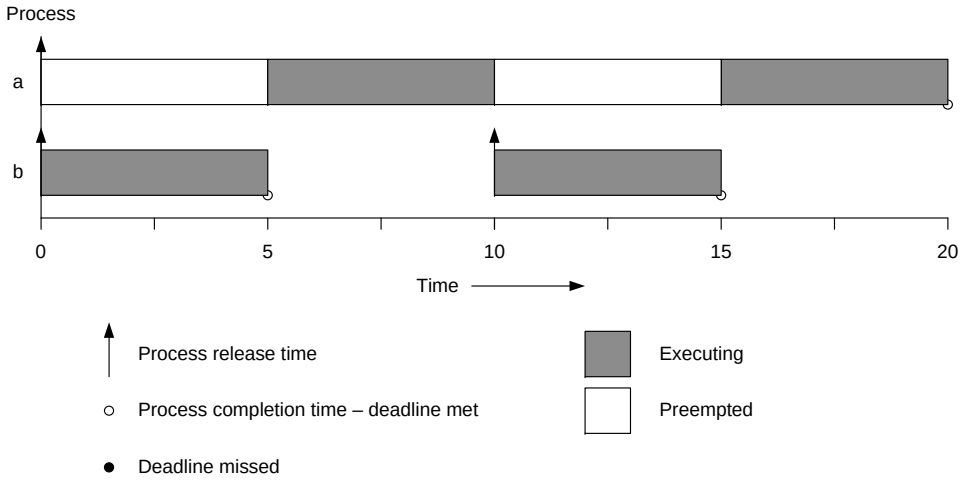


Figure 3.1: Time-line for utilization example

3.2 Response Time Analysis

A response time analysis gives a sufficient and necessary test for schedulability. It incorporates the maximum *interference* a task can experience in one period. For a general task the response time is calculated as:

$$R_i = C_i + I_i \tag{3.2}$$

where

R_i is the worst-case response time of task i
C_i is the worst-case execution time of task i
I_i is the maximum interference of task i by higher priority tasks

The interference is dependent on the number of times the task is interfered and the execution time of the task interfering it. The maximum interference a task, i, can experience from a higher priority task, j, is given by the formula:

$$Maximum\_Interference = \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3.3}$$

This calculates the maximum times i can be interrupted by j, and multiplies this with the WCET of j. By summing the interference of all higher priority tasks, the response time can be calculated as:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3.4}$$

where

$$hp(i) \text{ is the set of tasks with higher priority than } i$$

If the response time for all tasks is less than its deadline, the system is schedulable.

As an example a response time analysis can be performed on the example in Table 3.1. First, the response time for each task is calculated:

$$R_a = 10 + \left\lceil \frac{R_a}{10} \right\rceil 5 = 10 + 10 = 20$$
$$R_b = 5$$

Then the response time is compared to the deadline, which in this case is equal to the period.

$$R_a <= T_a = 20$$
$$R_b <= T_B = 10$$

It can be seen that the response times are lower than or equal to the deadlines, and the system is therefore schedulable.

This test is as mentioned sufficient and necessary, meaning the system is schedulable if and only if the test succeeds. This test gives a better result that the utilization test, but still assumes tasks to be independent and non-blocking.

3.3 Blocking

Allowing tasks to block, possibly preventing higher priority tasks from executing, is needed in almost all meaningful applications [7]. Blocking is used to synchronize tasks and allow tasks to pass data between each other.

A small example of a system which is schedulable according to the utilization test is shown in Table 3.2. However, since the utilization test does not consider blocking, this is not sufficient. A time-line example of the system executing is depicted in Figure 3.2.

Process	**Period,** T	**WCET,** C	**Priority,** P	**Utilization,** U
a	20	10	1	0.50
b	5	1	2	0.20

Table 3.2: Blocking system example

The lower priority task a contains a blocking region which is longer than the period of the higher priority task b, which therefore misses its deadline. This is a small example showing the problem with blocking tasks.

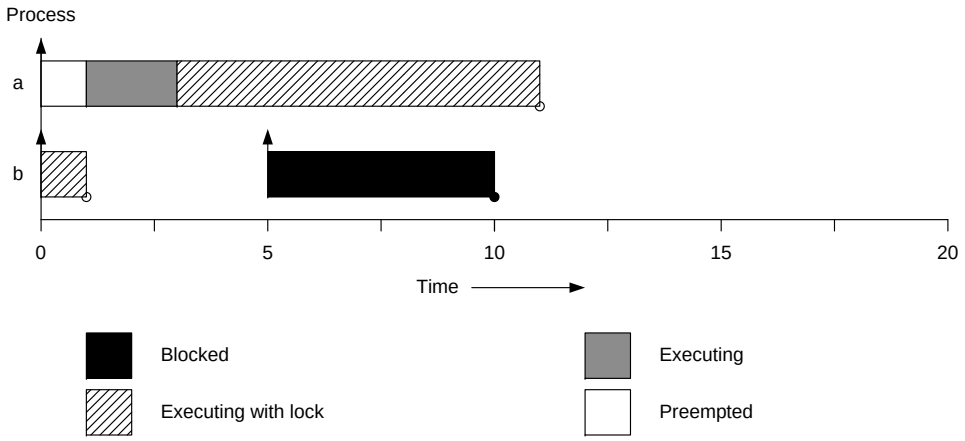


Figure 3.2: Time-line for a blocking system

The scheduler implementation used in this project implements blocking regions by disabling interrupts, this is essentially an immediate ceiling priority protocol, with the ceiling for all locks set to maximum (higher than the highest priority task.) This simplifies the analysis, since the maximum blocking time becomes the largest execution time for all blocking regions in all lower priority threads. Given by the formula:

$$B_i = \max_{j \in lp(i)} K_j \tag{3.5}$$

where

$lp(i)$ is the set of tasks with lower priority than i

K_j is the execution time of the longest blocking region in task j

Incorporating this into the response time analysis, gives the formula:

$$R_i = C_i + \max_{j \in lp(i)} K_j + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3.6}$$

This is a sufficient and necessary analysis which incorporates interference from higher priority tasks and blocking from lower priority tasks. There are still two problems with this analysis: The scheduler and context switch are not incorporated, and it might be very pessimistic since it assumes maximum interference and blocking. The next section discusses the incorporation of the scheduler and context switches.

3.4 Incorporating Context Switches

The mentioned analyses have ignored the execution time required by the scheduler, and have assumed the context switches to be instant. Noted in If these times are a certain order of magnitude lower than the periods of the tasks, it can be said to be fair to ignore these as long as some slack is available in the processor utilization. However, if these values are significant they need to be considered in the analysis. The boundaries for *a certain order of magnitude* and *some slack*, is not discussed in the literature.

Generally three blocks need to be considered. Firstly, the execution time the scheduler needs, to choose which task to schedule. Second, the time it takes to perform a context switch to the chosen task. Finally, the time it takes to perform a context switch away from the executing task. Incorporating these into the analysis gives the formula:

$$R_i = CS^1 + C_s + C_i + \max_{j \in lp(i)} K_j + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_s + C_j) \quad (3.7)$$

where

CS^1 is the execution time required for a context switch to a task

CS^2 is the execution time required for a context switch away from a task

C_s is the execution time of the scheduler

This formula includes the initial context switch to task i. Each interference from higher priority task includes two context switches: one to the higher priority task j, and one back to the current task i. The incorporation of context switch and the execution time of the scheduler of a task is depicted in Figure 3.3.

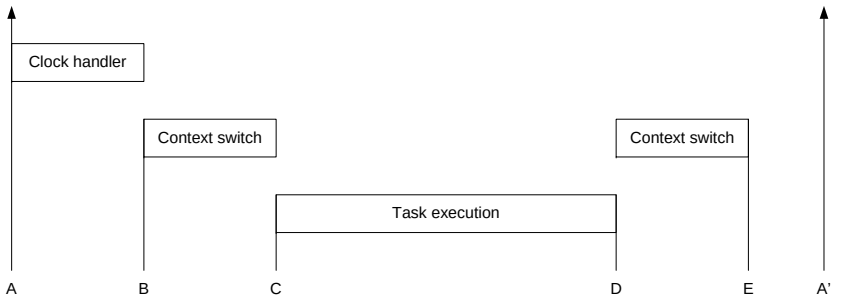


Figure 3.3: Overheads when executing tasks [7, p. 643]

- A - C is the initial context switch to the task, including the scheduler, represented by $CS^1 + C_s$ in the formula.

- C - D is the execution of the actual task. The task may be interrupted by higher priority threads between C and D. D is the completion of the task, this must happen before the deadline for the task.
- E is the completion of the context switch away from the task, represented by CS^2 in the formula.
- A' is the next release of the task

This analysis incorporates much of the jitter and overheads of execution into the schedulability analysis, but the pessimistic nature is still a problem. The worst case assumed is very conservative, and only considers local execution. By analyzing the application in a different manner, the worst case might be lowered based on the communication between tasks, e.g. sporadic tasks may have dependencies which prohibit them from being fired under certain conditions.

3.5 Model-Based Schedulability Analysis

Existing tools for WCET analysis of Java code, like Volta [14] and WCET Analyzer (WCA) [33], only calculate the WCET for a single task. The schedulability analysis is then performed based on the WCET for all tasks in the system. However, this might lead to a pessimistic result, because the entire system is not analyzed as a whole, and dependability between tasks is not detected. As described in Section 3.1, a utilization test can only be used if tasks are independent and non blocking. These aspects are not considered when using WCA or Volta, and a utilization test might consider a system to be schedulable even though it in practice is not, as depicted in Figure 3.2. Other aspects like context switches and the scheduler cost must also be included in the schedulability analysis if they are significant.

This section describes the approach to model-based schedulability analysis developed in this project.

The idea behind model-based schedulability analysis is to create a model on which analysis can be performed. The model must be a translation of the implemented system, in order to guarantee a tight correspondence.

This model must represent the control flow of the system, and must be decorated with information like WCET and blocking. The properties for each tasks must also be a part of the model, i.e. deadline, period etc. This model should then be used to verify that the implementation will never miss its specified deadlines.

The hypothesis is that this type of analysis can draw knowledge of the execution paths and connection between different tasks, to make a less pessimistic worst case. The model could reflect that two sporadic tasks would never be released at the same time, avoiding the critical instant. It might

also be possible to derive that the worst case execution path never occurs while a sporadic task is released, thus lowering the actual worst case response time.

The following examples illustrate how a model-based approach might result in a more accurate schedulability analysis compared to using the traditional approaches.

Example 1. Listing 3.1 contains a small example of a branching, which may result in a pessimistic schedulability analysis.

```
1 if(condition)       // 100
2   fire(a)           // 50
3 else
4   complexMethod()   // 500
```

Listing 3.1: Pessimistic WCET

The traditional way to compute the WCET for this small example would use the following equation:

$$\begin{aligned} wcet &= 100 + max(50, 500) \\ &= 600 \end{aligned}$$

Where the numbers indicates the amount of clock cycles used to compute the result. This is the correct WCET for this small code in an isolated environment, however when traditional schedulability analysis is performed it is assumed that the sporadic task invoked by `fire(1)` is released as often as possible, i.e. at its minimum inter-arrival time. However, when the sporadic thread is invoked, the code in Listing 3.1 only uses 150 clock cycles, thus reducing the Worst Case Response Time (WCRT). This can be detected using model-based scheduling analysis, because the system as a whole is analyzed.

Example 2. A small system consisting of one periodic thread and two sporadic threads is shown in Table 3.3. This system is not schedulable according to the utilization test, and not even by creating the more accurate time-line test for the execution of the system, because the utilization is higher than 1.

Process	**Period,** T	**WCET,** C	**Priority,** P	**Utilization,** U
a (periodic)	20	10	1	0.50
b (sporadic)	20	10	2	0.50
c (sporadic)	20	10	3	0.50

Table 3.3: Dependency of sporadic tasks example

However, if the periodic thread invokes the sporadic threads based on the code shown in Listing 3.2. Following the control flow, it can be derived that process b and c are mutually exclusive, i.e. only one can be fired each period. The control flow is a part of the model, and process b and c are therefore never released at the same time in a model-based schedulability analysis. When it is known that b and c cannot be fired at the same time, it can be shown that the system actually is schedulable, when context switches and blocking is not included.

```
1 if(condition)
2   fire(b)
3 else
4   fire(c)
```

Listing 3.2: Dependent sporadic threads

Example 3. Listing 3.3 contains a small example of two if branches, which are dependent on each other. If `condition1` is true then `condition2` is also true, and vice versa. This kind of dependability might be analyzed through static analysis, such as predicate abstraction. Including this analysis in the model of the system, might result in a more tight WCET of the task.

```
1  if(condition1){        //100
2    simpleMethod1()      //90
3    condition2 = true    //10
4  } else {
5    complexMethod1()     //990
6    condition2 = false   //10
7  }
8
9  if(condition2)         //100
10   complexMethod2()     //1200
11 else
12   simpleMethod2()      //100
```

Listing 3.3: Dependent control flow

The naive WCET calculation for the code in Listing 3.3 is:

$$\begin{aligned} wcet &= 100 + max(90 + 10, 990 + 10) + 100 + max(1200, 100) \\ &= 100 + 1000 + 100 + 1200 \\ &= 2400 \end{aligned}$$

However, if it the dependability in the control flow is analyzed, the WCET

calculation can be refined to:

$$\begin{aligned} wcet &= max(100 + 90 + 10 + 100 + 1200, 100 + 990 + 10 + 100 + 100) \\ &= max(1500, 1300) \\ &= 1500 \end{aligned}$$

Resulting in a less pessimistic, but still accurate WCET, as it is still an upper bound for the actual execution time.

3.6 Summary

Different approaches to schedulability analysis have been described, with different granularity of accuracy. Several approaches ignore the scheduler cost, because it is assumed to be insignificant. However, this is believed, by the authors, to be a wrong approach to hard real-time systems.

Model-based schedulability analysis allows more information about the executing system to be included in the analysis. Including context switches, blocking, scheduler cost, and control flow.

A well defined development environment and a predictable underlying hardware is desirable when performing schedulability analysis. The next two chapters describe a Java profile for hard real-time systems and a time predictable Java processor.

Chapter 4

Hard Real-Time Java Profile

This chapter describes the implementation, of the profile used in this project. The implementation is a further development of Safety Critical Java (SCJ), introduced in [34]. SCJ is a suggestion for an implementation of a safety critical profile for Java, which is in line with the JSR-302 specification request [27]. It is intended to fulfill the highest criteria of the DO-178B standard [28], used in airborne systems and equipment certification. The profile is not a subset of Real-Time Specification for Java (RTSJ) [9] like Ravenscar-Java [20]. RTSJ and Ravenscar-Java have been described in [6] and are not discussed further.

The implementation in this project is a reflection of the hands-on experience gained in [6], but it still corresponds to the initial ideas introduced in [34]. The profile introduced in this report is referred to as SCJ2, which is a further development of the JOP implementation of SCJ. JOP is described in Chapter 5, for further information see [32].

The overall idea is a simple, but sufficient, profile targeted hard real-time systems. The profile has no direct notion of priority and WCET of each thread, this is handled by tools supporting the profile. Such tools have been developed [14, 33] and are described in this report. Using tools to aid the development process relieves a burden from the developer, which should focus on the development of the application instead of tedious tasks that could be performed by a tool. Memory management is not a part of the profile, but it is assumed to be analyzed by external tools, the memory model is described further in Section 4.3.

The implementation of SCJ2 is described in the following sections, and it is pointed out when it differs from the original profile.

4.1 Execution of an SCJ2 Application

The singleton class `RealtimeSystem`, in Listing 4.1, represents the runtime system.

```
1  public class RealtimeSystem {
2    private RealtimeSystem()
3
4    public static void start()
5    public static void stop()
6    public static void fire(int event)
7    public static RelativeTime currentTime()
8    public static RelativeTime currentTime(
9                                  RelativeTime destination)
10 }
```

Listing 4.1: The Representation of the Real-Time System

A running application, implemented in SCJ2, can be in one of three states. These different states are depicted in Figure 4.1.

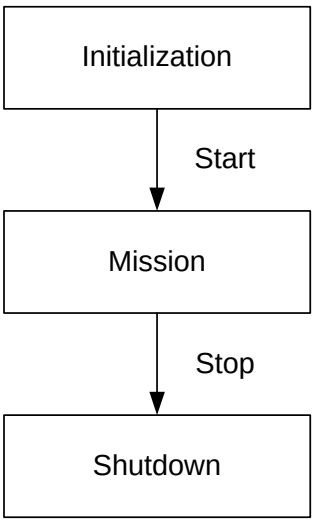


Figure 4.1: Application states [34]

- **Initialization:** This phase is used to instantiate all threads, and the memory used in the immortal memory. This phase has no timing constraints, and is done in the `main` method. When everything has been initialized, `RealtimeSystem.start()` is invoked, and the mission phase is started.
- **Mission:** This is where the application is run, where deadlines must not be missed. It will stay here until `RealtimeSystem.stop()` is invoked.
- **Shutdown:** Before entering this phase, it is ensured that all threads are not in a critical state, before the application is shut down.

4.2 Schedulable Entities

In order to keep the simplicity in SCJ2 only two types of threads are possible; periodic and sporadic threads. The class structure is depicted in Figure 4.2. The two types of threads have two methods in common:

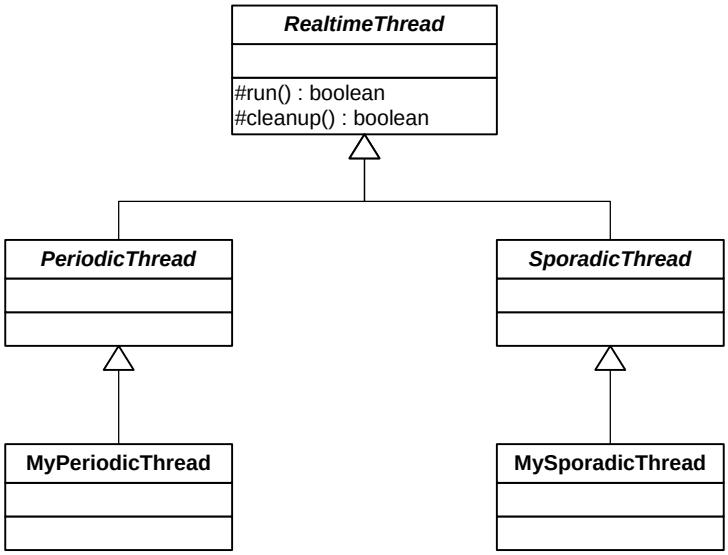


Figure 4.2: Class structure [34]

- `run()` This method is abstract, and is implemented with thread logic. The Boolean return value indicates whether the thread is ready to be shut down, i.e. it is not in a critical state.
- `cleanup()` This method is called instead of the `run()` method during shutdown, when all threads have returned true from their `run` method.

4.2.1 Release Parameters

The constructors of periodic and sporadic threads must be supplied with `PeriodicParameters` and `SporadicParameters` respectively. This is to encapsulate the required arguments in one simple class, which increases the readability of the code. When inheriting from periodic or sporadic thread, the release parameters are easily distinguishable from application parameters. In SCJ, each parameter is specified directly in the constructor, instead of aggregating these in a single object. Aggregation is also used in RTSJ and Ravenscar-Java. The `PeriodicParameters` class encapsulates release parameters for a periodic thread, the definition is shown in Listing 4.2.

```
1  public class PeriodicParameters {
2    public PeriodicParameters(RelativeTime period,
3                              RelativeTime deadline,
4                              RelativeTime offset)
5    public PeriodicParameters(RelativeTime period,
6                              RelativeTime deadline)
7    public PeriodicParameters(RelativeTime period)
8
9    public final RelativeTime getPeriod()
10   public final RelativeTime getDeadline()
11   public final RelativeTime getOffset()
12 }
```

Listing 4.2: PeriodicParameters

The period represents the release period of the periodic thread and the deadline represents the deadline from when the thread has been released. It should be noted that the JOP implementation of the profile does not support periods larger than about 35 minutes, because JOP uses a 32 bit integer to represent microseconds, and Java only supports signed integers. This is a limitation in the specific implementation for JOP and not the profile as such.

The offset defines how long time should pass before the initial release of this periodic thread, after entering the mission phase. The default value for the deadline is equal to the period and for the offset is zero.

The `SporadicParameters` class encapsulates release parameters for a sporadic thread, the definition is shown in Listing 4.3.

```
1  public class SporadicParameters {
2    public SporadicParameters(int event,
3                              RelativeTime minInterarrival,
4                              RelativeTime deadline)
5    public SporadicParameters(int event,
6                              RelativeTime minInterarrival)
7
8    public final int getEvent()
9    public final RelativeTime getMinInterarrival()
10   public final RelativeTime getDeadline()
11 }
```

Listing 4.3: SporadicParameters

The minimum inter-arrival time of the sporadic thread is specified together with a deadline. The event to invoke the specific sporadic thread must always be specified, and must be unique, because each sporadic thread is bound to an event in a one-to-one relation. A sporadic task must be released using the static method `RealtimeSystem.fire(int event)`, from Listing 4.1. The corresponding sporadic thread is then scheduled. The original SCJ profile uses strings as events instead of integers, however due to

problems with analyzing strings, integers have been chosen instead. The programmer is advised to define constant integers to represent the different sporadic events. This gives a similar level of explicitness as using strings. The default value for deadline is equal to the minimum inter-arrival time.

4.2.2 Representation of Time

In the original proposal of SCJ, only relative time is available [34], thus limited to applications where absolute time is not important. The class `RelativeTime` is used to represent time, similar to RTSJ, however in SCJ the class has been made immutable. The idea is to prevent changing the periods, deadlines etc. of the running threads.

SCJ2 does also only use relative time, however it has not been made immutable. Time is important in real-time systems, and we believe it is desirable to modify time objects during runtime instead of creating new objects. Methods to compare, add, and subtract two instances of a `RelativeTime` object have been implemented. In order to prevent the developer from changing periods, deadlines etc. of threads at runtime, static analysis could be used to ensure `RelativeTime` objects used in the release parameters are not modified in the mission phase. Another solution could be to make an immutable version of `RelativeTime`, used in the release parameters. The definition of `RelativeTime` is shown in Listing 4.4.

```
1  public class RelativeTime {
2    public RelativeTime(RelativeTime time)
3
4    public final void set(RelativeTime time)
5    public final int compareTo(RelativeTime time)
6    public final boolean equals(RelativeTime time)
7    public final RelativeTime add(RelativeTime time)
8    public final RelativeTime add(RelativeTime time,
9                                  RelativeTime destination)
10   public final RelativeTime subtract(RelativeTime time)
11   public final RelativeTime subtract(RelativeTime time,
12                                  RelativeTime destination)
```

Listing 4.4: RelativeTime

RTSJ uses a long to represent milliseconds and an integer to represent nanoseconds internally, and allows the programmer to set these directly. The internal representation in SCJ2 is a long to represent nanoseconds, this reduces the size of a `RelativeTime` object, and reduces the computation of compare, add and subtract methods.

Java does not support unsigned value types, and a long nanoseconds is therefore able to represent almost 300 years. An SCJ2 application can still run indefinitely, as long as it does not measure any time interval longer than 300 years, which should be enough as long as the developer is aware of this.

However, the internal representation is not revealed to the developer, and he must always use `RelativeTime` objects in order to manage time in SCJ2. The methods to add and subtract time return a newly allocated `RelativeTime` object with the new value. The methods are overloaded with a destination argument, where an already existing instance of `RelativeTime` can be supplied to avoid the allocation of a new object.

All the methods and constructors only allow `RelativeTime` objects, so in order to actually instantiate a `RelativeTime` object, some container classes have been implemented, shown in Listing 4.5.

```
1  public class RelativeTimeSeconds extends RelativeTime{
2    public RelativeTimeSeconds(int seconds)
3  }
4  public class RelativeTimeMilli extends RelativeTime{
5    public RelativeTimeMilli(int milliseconds)
6  }
7  public class RelativeTimeMicro extends RelativeTime{
8    public RelativeTimeMicro(int microseconds)
9  }
10 public class RelativeTimeNano extends RelativeTime {
11   public RelativeTimeNano(long nanoseconds)
12 }
```

Listing 4.5: Inheritance of RelativeTime

These container classes have been added to improve the explicitness of the code, to reduce the confusion of whether the time is specified in e.g. nanoseconds or microseconds. It also enables the developer to represent e.g. milliseconds directly instead of multiplying by 1.000.000 to convert to nanoseconds. This gives some overhead in computation cost and memory usage, however it increases the readability of the code, and follows the object oriented paradigm, which is one of the strengths of Java.

The original proposal of SCJ contains a method `int currentTimeMicros()` to get the amount of microseconds since the system was started. This seems like a very JOP specific method, because JOP uses an integer to represent microseconds internally. The problem with this approach is it wraps around each 35 minutes, and it is then the programmers' task to compensate for this. This is undesirable as it may lead to errors. In SCJ2 this method is replaced by `RelativeTime currentTime()` which is consistent with the representation of time, this is also overloaded to take a destination object to prevent creation of a new object. This method is modified to return the time in long, to avoid the early wrap around. Even though the profile supports nanoseconds, the actual JOP implementation only supports microseconds, this is only a limitation in the implementation and not the profile.

4.3 Memory Model

The original proposal of SCJ uses a memory model similar to Ravenscar-Java, with immortal memory, which is initialized in the initialization phase, and then a scoped memory area for each executing thread. The size of the scoped memory area must be defined as a parameter to the thread, however it is intended to be analyzed by external tools. The use of immortal memory and scoped memory is to avoid garbage collection, since no real-time garbage collector has been available.

However, a real-time garbage collector has been developed for JOP [29]. Using a garbage collector results in an easier, and more powerful, computational model for the developer, because it allows objects to be shared between tasks. The use of garbage collector also increases the similarity to standard Java, and should therefore increase the productivity of the developer. The garbage collector acts as an additional thread, which must be accounted for in the schedulability analysis.

However, as discussed in Section 5.3, the garbage collector implemented for JOP is not ready for industrial use. The current implementation of SCJ2 does therefore not support a memory model, i.e. no objects should be created during the mission phase. Doing so might result in memory depletion.

4.4 Restrictions

Some restrictions are imposed on the language, in order to simplify analysis. Recursion is not allowed, because it can result in unpredictable behavior. However, it could be possible to introduce some restrictions on recursions, but this is omitted due to simplicity. Handling or preventing recursion is considered future work.

Dynamic class loading has been disabled, in order to allow a static analysis of possible methods which can be invoked at runtime. Furthermore the standard Java libraries are disallowed, because these libraries are not optimized for real-time systems, and might have an unpredictable WCET. These restrictions are currently not enforced by the profile.

4.5 Performance Optimized Version of SCJ2

The notion of `RelativeTime` to represent time, is believed by the authors to be a clean object oriented design, however this is at the cost of execution time and memory consumption. An additional version of SCJ2 has been implemented, where `RelativeTime` is omitted, and it is replaced with an integer representing time in microseconds. This reduces the granularity of

time. Similar the `RelativeTime currentTime()` method is replaced with `int currentTimeMicros()`.

This means that the largest intervals which can be represented are about 35 minutes. Furthermore it is now the developer's responsibility to correctly add and subtract time while preventing wrap around.

Both versions of SCJ2 is included on the enclosed CD.

4.6 Summary

SCJ2 is a further development of SCJ, with the introduction of periodic and sporadic parameters, another representation of time, and a new memory model.

The use of a class to represent time instead of an integer, as JOP does internally, results in some overhead, but prevents wrap around, and conforms to the object oriented world. However, a version with integers to represent time has also been developed, to support a more performance oriented approach.

Even though the garbage collector is not ready for use, it is still desirable. With the use of a garbage collector and object oriented encapsulation of e.g. relative time, the use of the profile is more similar to standard Java than e.g. Ravenscar-Java and the original SCJ. This should increase the productivity and reduce the step from developing standard Java applications to real-time Java systems. However, an important note is that optimized programming is still important to reduce production cost of the underlying hardware, since optimized applications require less computational power.

The profile is kept simple with only a small amount of classes, to support hard real-time development. The profile is supposed to be accompanied by tools, e.g. to guarantee that deadlines are not missed. SARTS is a proposal for a schedulability analysis tool targeting SCJ2.

SCJ2 is implemented on the time predictable Java processor JOP, described in the next chapter.

Chapter 5

Java Optimized Processor

When developing hard real-time applications the underlying hardware is very important, due to timing constraints. This chapter describes the Java Optimized Processor (JOP) which is used as underlying hardware in this project. Further information about JOP can be found in [30, 31, 32].

5.1 Overview

JOP is a Java processor executing Java bytecode as native instruction set. The processor is implemented on an FPGA. Since the JVM is a CISC, some bytecode instructions are too complex to implement directly in hardware. JOP implements each bytecode in an instruction set called microcode, making JOP a RISC architecture. Each bytecode is translated into one or more microcodes. Implementing the processor on a FPGA makes the distinction between hardware and software somewhat blurry. It enables the developer to implement some bytecode instructions in hardware and some in software. E.g. the instruction, `new`, which allocates a new object on the heap is implemented in software to ease memory management.

JOP is developed for hard real-time applications with strict time constraints, and has been used in industrial applications.

5.2 Predictability

JOP is developed to be a real-time processor. This is reflected in its time predictability features i.e. the execution time of each bytecode instruction is known. This includes an analysis of each instruction on the microcode level, so by making a bytecode level analysis of a Java application the WCET can be calculated. To improve the predictability, features used in mainstream processors which improve the average case execution time, have been removed in favor of features to improve the WCET. The features removed include branch-prediction and data caches. Since caches are important to

improve the performance of a processor, two types of predictable caches are implemented: Stack cache and method cache.

Stack Cache. The stack cache is implemented as a two-level stack cache. The first level contains the top two elements of the operand stack in two on-chip registers directly accessible by the ALU. The second level contains the rest of the operand stack, the context of the caller, including the return address, and the method local variables, in an on-chip stack accessible in one clock cycle concurrently with ALU operations. This cache is a part of the thread context and needs to be saved and restored on context switches. This can be incorporated into a WCET analysis since the maximum stack size can be analyzed from the call graph and the size of the operand stack of a given method is statically known at each instruction, as a consequence of restrictions on the Java class file format [33]. The call graph can be statically analyzed since recursion and dynamic class loading are disallowed in SCJ2, as described in Section 4.4.

Method Cache. The method cache contains the entire method body of the currently executing method. This ensures that a bytecode fetch will never result in a cache miss, since no jumps can be made outside a method body. All cache misses are grouped together at method invoke and return. The size of each method is known, and the execution time needed to fill the cache can be analyzed and added to that of a method invoke and return. The method cache comes in several forms: single method cache, two-block cache, and variable block cache.

The version of JOP used in this project is configured with a variable block cache. The variable block cache is split into several blocks of a given size. A single method can span several blocks, but a block can only contain bytecode from a single method. The cache is implemented as a circular buffer, where a method spanning beyond the end of the buffer continues from the beginning. If a method is not present in the buffer it is added after the most recently added method, possibly replacing other methods. Using a variable block cache makes the choice of number of blocks and block size a major design decision, since it can greatly affect the performance of the application running. By default JOP is configured with a 4KB cache divided into 16 blocks. This configuration is used in this project, and in all experiments executed.

A problem with the method cache arises when interrupts are allowed. When an interrupt is generated, the scheduler is invoked. This involves loading the scheduler logic into the method cache. The scheduler might decide to switch to a different thread, which involves loading different methods into the method cache. This means that if a method can be interrupted, the method cache can be flushed, complicating the analysis.

These two types of caches are implemented to improve the performance while maintaining a time-predictable processor. This time predictability enables the WCET for each method to be analyzed. By running this analysis on the run method for each thread in the system, the WCET for each thread can be found. General theory about schedulability analysis is discussed further in Chapter 3.

5.3 Memory Model

Standard Java supports a garbage collector, which collects objects which are no longer referenced. This approach is not readily available in hard real-time systems. The problem is that the garbage collector interrupts the system and can prevent it from executing, causing deadline misses. Even a real-time garbage collector is not trivial to incorporate into the application. The amount of memory which is allocated must be analyzed to ensure that the garbage collector can be allocated enough execution time to collect it all. JOP supports two memory models: Scoped memory areas and garbage collection.

Scoped Memory. The scoped memory model uses a shared immortal memory area, and a scoped memory area for each thread. Objects allocated in the immortal memory area will live throughout the lifetime of the application, and are shared between threads. Each thread has an associated area, in which it can create objects. The area is cleared whenever a thread enters or exits the area. References from the immortal memory into a scoped memory are not allowed, and references from one scope into another are not allowed.

Garbage Collection. The real-time garbage collector is implemented as a regular thread and collects objects which are no longer referenced, including references from the immortal memory. Since the garbage collector is a regular thread it needs to be incorporated into the schedulability analysis. This approach has one globally shared heap, and an immortal area. References from the immortal area into the heap are allowed, meaning that objects can easily be passed from one thread to another.

One of the advantages of the garbage collector, compared to scope memory, can be seen when developing a producer/consumer application. One thread acts as a producer and places objects in a list, another thread acts as consumer removing objects from the list. The list must reside in immortal memory to make it accessible from both threads. Using scoped memory it is not possible to add objects to the list, since this would make a reference from the immortal area into the scope; using scoped memory, only value

types can be shared across threads. This makes the garbage collector a better solution when trying to make it easier for the developer.

Even though the garbage collector has been implemented, it is not ready for industrial applications since the analysis of it has not been completely implemented yet. There are also some problems with the current implementation of scoped areas in JOP and SCJ2. Due to these problems this project abstracts away form memory management, since it is a research project in itself. All implementations and experiments executed in this project do not allocate new objects during the mission phase to avoid memory related problems.

5.4 Measurement

Even though JOP is said to be time predictable, it is important to do tests and measurements, to ensure the values found are actually correct. It is possible to measure the actual number of clock cycles used by JOP. This can be used in controlled experiments to measure the actual time used to execute a given piece of code.

The code shown in Listing 5.1 is used to determine the actual amount of clock cycles used. The clock cycle counter is read before and after the execution of the code, and the difference is printed to the output stream. The variable, `to`, represents the amount of clock cycles used to read the counter.

```
1  int ts, te, to;
2  ts = Native.rdMem(Const.IO_CNT);
3  te = Native.rdMem(Const.IO_CNT);
4  to = te-ts;
5  ts = Native.rdMem(Const.IO_CNT);
6
7  measuredMethod();
8
9  te = Native.rdMem(Const.IO_CNT);
10 System.out.println(te-ts-to);
```

Listing 5.1: JOP measurement

This code prints out the number of clock cycles used by JOP to execute the method `measuredMethod()`. It is important to note that the time measured is not necessarily the worst case, but the execution time used in the concrete execution. By controlling the variables in the system, the worst case branch of execution can often be forced, and the WCET can be measured.

To simplify the code examples using measurement the syntax in Listing 5.2 is used. This simplification is only done on the code presented, not the

code executed.

```
1 timeBegin();
2
3 measureMethod();
4
5 timeEnd();
```

Listing 5.2: Simplified measurement

Using this functionality it is possible to conduct several experiments, testing the actual execution time. Some anomalies have been found in the actual execution time compared to the specified execution time. For instance the instruction `iinc` is specified as using 8 clock cycles, but measuring it on JOP shows it as taking 4 clock cycles. This is because the `JOPizer`, which translates class-files into JOP-files, translates `iinc` into two `iload`, one `iadd`, and one `istore`, which add up to 4 clock cycles. When deploying JOP in an industrial setting, all such anomalies have to be found, and specified to ensure the analysis is not optimistic.

5.5 Summary

Even though JOP is said to be time predictable, there are still some problems which need to be accounted for. The method cache is not as predictable as first assumed, and this needs to be included into the analysis. There are several problems with the memory models available on JOP, and non of these are used in this project to focus on the problem at hand. Despite these problems, JOP provides a time-predictable platform for real-time systems and is suitable for the analysis developed in this project.

The next chapter describes a case study of a real-time system, developed for JOP using the SCJ2 profile.

Chapter 6

Case Study

This chapter presents a case study of a real-time system implemented in SCJ2. It was originally designed and implemented in [6], where it is described in further details.

The system is a sorting machine called Real-Time Sorting Machine (RTSM). The machine is built in LEGO, using motors and sensors monitored by JOP.

The overall idea of RTSM is to sort candy available in two different colors, white and blue. The candy is suitable for sorting because the shape fits into the LEGO context, not being too big or small, and it is available in white and blue, which are two rather distinguishable colors. The candy is from now on referred to as objects.

The design of RTSM is depicted in Figure 6.1.

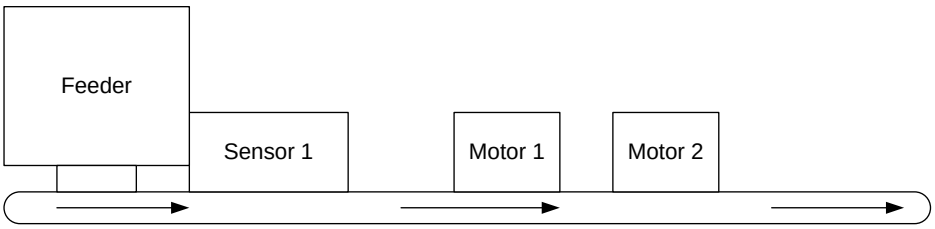


Figure 6.1: Design of RTSM

The objects are placed in the *Feeder* which places one at a time on the conveyor belt with some space between. This ensures a constant flow of objects to be sorted, and prevents two objects from being right beside each other.

The conveyor moves the objects to the right, as the arrows indicate. When the object leaves the *Feeder* it passes by *Sensor 1*, which determines whether it is a white or a blue object. *Sensor 1* on the figure is actually two sensors placed on each side of the conveyor belt, and then encapsulated in a small cube. The cube is built to prevent external light from interfering with the sensors. The use of two sensors has two purposes. Firstly, the sensors

emit light to each other, keeping a constant high amount of light, easing the detection of objects breaking this light, and therefore spot when objects pass by the sensor. Secondly, the object might not always be in the middle of the conveyor belt, making the light it reflects either higher or lower than expected, this is compensated for, by using the average of the two sensor inputs.

Based on what color is detected, either *Motor 1* or *Motor 2* must be activated when the object is in front of the motor. Activation of a motor involves pushing the object into the correct bin, thereby sorting the objects.

The implementation contains two periodic and two sporadic tasks. The periodic task `PeriodicReadSensor`, reads the input from the sensors, to determine whether an object has passed by, and of which color. Each time an object is detected, the time of detection is added to the corresponding list. The periodic task `PeriodicMotorSpooler` reads this list, and fire a sporadic event when the object must be pushed off the conveyor belt. Two instances of `SporadicPushMotor`, one for white and one for blue, starts either *Motor 1* or *Motor 2* when invoked.

RTSM was originally implemented for SCJ, but it has been updated to use SCJ2. RTSM is implemented for both versions of SCJ2. When conducting experiments the performance optimized version is used.

Additionally a simplified version of RTSM has been developed, which contains the same worst case path, but paths resulting in a lower WCET have been omitted. This version is referred to as $\text{RTSM}_{\text{SIMPLE}}$.

All three versions are available on the enclosed CD, in the projects `RTSM`, `RTSMNoRelativeTime`, and `RTSMNoRelativeTimeSimple` respectively. A video of the actual machine is also available on the enclosed CD.

RTSM is a simple example of a hard real-time system, but it is sufficient as a case study. It includes periodic and sporadic tasks, blocking regions, and dependencies between tasks. These are interesting properties of a system, when performing schedulability analysis.

Chapter 7

Project Focus

The focus of this project is to develop a tool for schedulability analysis of hard real-time Java systems, named Schedulability Analyzer for Real-Time Systems (SARTS). Rather than relying on slack in the schedulability analysis, the philosophy is to include more aspects of the system being analyzed, to refine the analysis.False positives are not acceptable, and the goal of the project, is that the result of a final version of SARTS provides safe guarantees about the schedulability of the analyzed system. The result must not be more pessimistic than what is possible with traditional approaches, presented in Chapter 3.

SARTS is a model-based approach to schedulability analysis. The system being verified is translated into an abstract time preserving model for Uppaal.

An additional goal of SARTS is to automate the verification process, where the system is automatically translated from Java to a finite state model. This reduces human errors in the verification process, and it allows the developer to focus on the actual implementation of the system. Furthermore the developer needs no knowledge of model checking in order to use SARTS.

The next part describes the implementation of SARTS.

Part II

Implementation

Chapter 8

Technology Choices

The translation to a finite state model is done by inspecting the compiled Java program, i.e. Java class files, together with the Java source code, gathering the information needed. This information includes control flow of the system together with loop bounds and execution time of each bytecode. In order to achieve a more accurate analysis, information about blocking should be included in the model as well. Additionally, information about the number of tasks in the system and the parameters for these tasks should be available, in order to simulate the system correctly.

In order to perform the translation the following information must be retrieved:

- Control flow
- Execution time for each bytecode
- Loop bounds
- Blocking information
- Task parameters

The following sections describe tools and frameworks which could be used to retrieve these information. Additionally different model checkers are introduced. A summary describes which of these tools are used in the implementation of SARTS.

8.1 Soot

Soot [10] is a Java optimization framework capable of inspecting and changing Java class files. Soot has been used in various projects for analysis and transformations of Java applications [16], e.g. Bandera [21] and Polychrony [18].

Soot provides four internal representations of the program to be analyzed, Baf, Jimple, Shimple and Grimp, each with some advantages in different situations:

- **Baf** is an abstract representation of Java bytecode with no constant pool and reduced instruction set.
- **Jimple** is a three address code representation of the Java bytecode.
- **Shimple** is a static single assignment form of the Jimple representation.
- **Grimp** is a Java-like representation of the Jimple code.

Soot gives the ability to freely switch between the intermediate representations. A more thorough explanation of the intermediate representations can be found in [6].

All four representations make abstractions over the actual bytecode, and change the flow of the original application. This makes Soot difficult to use, since SARTS requires the exact flow of the bytecode. Additionally, even though Soot has been used in various projects, it still very much lacks documentation, making it difficult to use to its full potential.

8.2 BCEL

The Byte Code Engineering Library (BCEL), provides a simple API for analyzing and modifying Java class files. BCEL is currently used in various projects, e.g. static analysis in FindBugs [24] and in AspectJ [13] for pointcut identification and code injection.

BCEL provides an internal representation, which describes the concrete class in low level detail. This internal representation provides access to all the information stored in Java class files, such as classes, methods, and bytecode. Additionally, a thorough documentation of BCEL is provided as a detailed Javadoc documentation of the API and a manual providing details about the Java class file format and some project examples using BCEL.

BCEL provides a simple and powerful representation of Java bytecode, for performing transformation and analysis.

8.3 Java 6 API

With the newest Java version three new APIs are made available. These provide the ability to customize the Java compiler and to perform source code analysis on Java programs, from within Java programs. The three APIs are: *Java Compiler API*(JSR 199), *Pluggable Annotation Processing*

API(JSR 269), and *Compiler Tree API* [19]. This section gives a brief introduction to the functionality of these APIs.

The compiler API enables developers to invoke the Java compiler from within Java programs. The API provides several classes for accessing and configuring the Java compiler.

The annotation processing API enables developers to create custom annotation processors and plug them into the compiler. The processor is configured to accept different annotations, and is invoked when the compiler encounters an element annotated with the corresponding annotation.

The compiler tree API enables the developer to create visitors, to visit the syntax tree of the Java application being compiled.

Combining these tree APIs developers can analyze the Java source code, and retrieve the information needed.

8.4 Modeling Tools

Several modeling tools for verification already exist. These tools are highly optimized and the tool developed in this project does therefore use an existing modeling tool. Several existing tools have been described in [6], including Uppaal [39], TIMES [37], Bandera [21], Moby/RT [25], and Java PathFinder [23]. These tools all have different advantages and disadvantages, but some of them has some interesting features, making them more suitable for this project.

A tool like Bandera [8] translates Java source code to an intermediate representation, on which slicing and abstraction is performed. This intermediate representation is then translated to the input of a model checker, to perform verification. Using this technique different model checkers are supported. However, Bandera has no notion of time, which is critical in real-time systems. Bandera is therefore not suitable in this project, but the idea of an intermediate representation of the implemented system, before a translation to a specific model checker is used.

A tool like TIMES [3] already supports a schedulability analysis of a real-time system. However, TIMES is intended for an initial modeling tool, corresponding to step 2 in Figure 2.1. TIMES has a lot of restrictions on what computation is actually possible by periodic threads. This might be suitable for an initial model, but seems too restrictive when automatically doing full translating from code to a model, and is therefore not used in this project. TIMES includes no context switch or scheduler cost in the schedulability analysis, which may be considered significant in some systems.

Polychrony [18] is another interesting tool, but is not described in [6]. It allows translation of Java to its input language SIGNAL [38] targeted hard real-time systems. However, they do not mention how WCET is handled. The support for Java is only a plug-in to the existing Polychrony, however

it has not been possible to locate this plug-in and actually try it in practice, and it is therefore not used in this project.

The modeling tool used in this project is UPPAAL because it has several advantages, when performing schedulability analysis. It has a notion of time through clocks, and it furthermore supports stopwatches, which allow time to stop. How these features are used is discussed in further details in Chapter 11. UPPAAL is developed at Aalborg university which also results in some advantages. The authors are familiar with UPPAAL, and it is easy to get support on specific topics about UPPAAL from the employees at the university.

8.5 Summary

The following paragraphs describe which tools are used to retrieve which information.

Execution time. In BCEL, the actual bytecode for each method is available as a list of instructions. The WCET for each bytecode, executed on JOP, is specified in [32].

Control flow. Control flow concerns the flow of control between instructions in the analyzed code.

In BCEL, the instructions to where control may flow, for each instruction, is readily available and it is straight forward to analyze control flow using BCEL.

Loop bounds. The loop bound annotations are available in the Java source code only, as these are written as comments which are ignored during compilation into Java class files. When analyzing control flow, loop bounds must be considered and should influence the model. The loop bounds are essential to the execution time of the loop and hence need to be incorporated into the model. As described in Section 2.2, loop annotations must be specified by the developer.

A mapping from each bytecode to line number in the source code is available in the class files and hence in BCEL, and using this information, the loop bound annotations can be retrieved.

Blocking. Each method is marked with a synchronized flag, and two bytecode instructions, `monitorenter` and `monitorexit` enclose synchronized regions of code. No extra effort is needed to retrieve this information as this is readily available in BCEL.

Task parameters. Task parameters are derived from the periodic and sporadic parameters used in the Java profile described in Section 4.2.1. These values must then be used to parameterize the tasks in the model as described in Section 11.4.

Retrieving the task parameters for periodic and sporadic threads is doable in BCEL, but this is unfeasible due to the low level representation of BCEL.

A better representation is using the Java 6 APIs. An abstract syntax tree of the main method is parsed to retrieve information about instantiation of tasks, and their release parameters.

The majority of the analyses are done using BCEL, because of close the correspondence between the actual execution and the generated model, where Soot performs abstractions over the bytecode. Another reason for not using Soot is the lack of documentation. Since Soot is a powerful analysis framework, which would aid in some optimizations of the analysis, a mapping from the current intermediate representation to the Soot representations might be useful, but considered future work.

Java 6 API is used to inspect the source code for task parameters, a more appropriate level for retrieving this information.

The next chapter describes the design of SARTS.

Chapter 9

Design

SARTS translates a real-time Java system, into a corresponding UPPAAL model, in order to perform schedulability analysis. The architecture of SARTS is depicted in Figure 9.1.

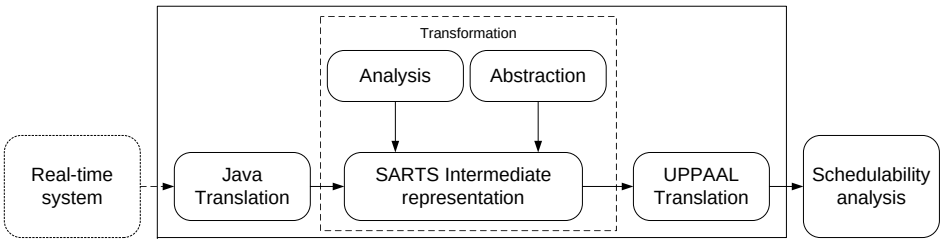


Figure 9.1: Architecture of SARTS

The real-time system must initially be implemented in Java, using the SCJ2 profile described in Chapter 4. This system is translated to SARTS Intermediate Representation (SIR); the intermediate representation developed for SARTS. How this translation is performed is described in Chapter 10.

This intermediate representation is an abstraction of a Java program, and is described in Section 9.1. Different kinds of analyses and abstractions can be performed on this intermediate representation. This could be to gain knowledge of dependencies in the control flow of the program, which can result in a more accurate schedulability analysis, as described in Section 3.5. The analysis and transformations implemented in the current version are, decorating SIR with WCET, and collapsing SIR to a more compact representation, as described in Section 13.2.

SIR is translated to a UPPAAL model, on which the actual schedulability analysis is performed. How this is performed is described in Chapter 11.

9.1 SARTS Intermediate Representation

The purpose of the intermediate representation is to represent the implemented Java application in a more abstract form. An abstract model of the intermediate representation is depicted in Figure 9.2. The actual implementation is located in the `intermediate` package.

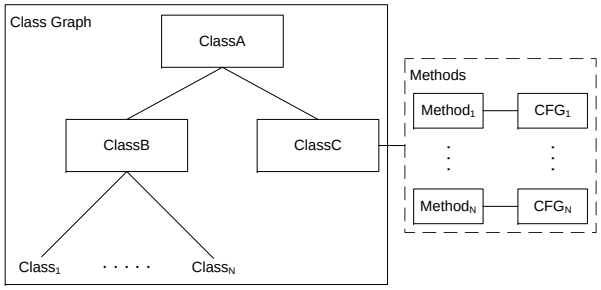


Figure 9.2: Intermediate representation

The class graph represents all classes associated with the specific implemented real-time system, representing the class hierarchy, i.e. a child node represents a specialization of a class. The root node in the class graph must therefore be `java.lang.Object`. Each class contains a set of methods, each of which contains a control flow graph (CFG) for the specific method, along with the actual instructions in the method implementation.

As an abstraction to the actual Java bytecode, the concept of basic blocks is introduced. A Basic block contains a list of the Java bytecode instructions it represents and the cost of executing these along with extra information e.g. loop bound in the case of a loop basic block. The different types of basic blocks and their class hierarchy is depicted in Figure 9.3.

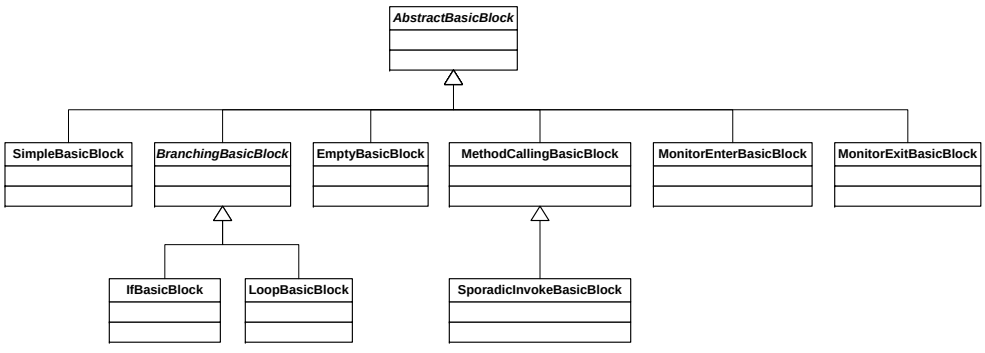


Figure 9.3: Basic block hierarchy

`AbstractBasicBlock` has a WCET attribute, denoting the execution time of the Java bytecode the block represents. This is used in the schedu-

lability analysis, and it enables basic blocks to be merged into a new basic block and then calculate the new WCET for the new block. The different types of basic blocks are described below:

- **SimpleBasicBlock:** This is a sequence of bytecode instructions with exactly one predecessor and exactly one successor.
- **MethodCallingBasicBlock:** This represents a method invocation. It contains a list of possible methods, which can be invoked.
  - **SporadicInvokeBasicBlock** This is a special case of a method invoke, where a sporadic task is invoked. It is supplied with the `event` corresponding to the sporadic task invoked.
- **BranchingBasicBlock:** This is an abstract class, representing a branching in the control flow.
  - **IfBasicBlock:** Represents an if branch, and therefore contains two outgoing edges.
  - **LoopBasicBlock:** Represents any kind of a loop. It contains a `LoopBound` object, which has a list of possible entrance nodes to the loop, and a list of possible exit nodes. It also contains an estimated loop bound for the actual loop, specified by the developer of the real-time system.
- **MonitorEnter- and MonitorExitBasicBlock:** Represents when a synchronized region is entered or left.
- **EmptyBasicBlock:** These blocks do not represent actual instructions, and are added for convenience reasons, e.g. one is added in the beginning and the end of a method.

The next chapter describes how Java is translated to SIR.

Chapter 10

Java Translation

The class `sarts.generator.JavaTranslator` is used to translate Java to SIR. This class is instantiated with the main class of the analyzed system, and the SIR, contained in a `IntermediateData` object is returned from the method `buildIR()`. This method uses three visitors to build the intermediate representation. First the entire class hierarchy is built using the visitor `ClassGraphBuilder`. This creates a class graph, `ClassGraph`, which is traversed, and for every method of every class a control flow graph is created consisting of basic blocks, representing the actual bytecode. This translation is done using the two visitors, `LocationBuilder` and `TransitionBuilder`. Where `LocationBuilder` creates the basic blocks, and `TransitionBuilder` creates the relations between them. The visitors are implemented as BCEL bytecode visitors. The three visitors are described in the following sections.

10.1 Class Graph Builder

The class builder recursively traverses every method of every class. The only visit method implemented is `visitLoadClass`, which is an abstraction over all bytecode instructions which can cause the JVM to load a new class. The type of the class which can be loaded is added to a queue, and parsed as well. For every class parsed the parent class is parsed as well, this way a complete class hierarchy is built. The reason for building the complete class hierarchy and not only the classes directly accessed, is that inherited methods must be parsed as well. Only the relevant parts of this class hierarchy are translated to UPPAAL, as described in Section 13.1.

10.2 Location Builder

The location builder creates basic blocks corresponding to the bytecode. For most instructions, a `SimpleBasicBlock` is created using the helper method

`createBasicBlock`. Several instruction require special handling, some of these are described below.

Branch Instructions. The visit method `visitBranchInstruction` is an abstraction over all jump instructions, conditional and unconditional. If the index of the jump is positive, i.e. the jump is forward, a `IfBasicBlock` is created. If the index is negative, i.e. the jump is backward, it means that this jump is part of a loop, and a `LoopBasicBlock` is created. The target of the loop is used as an identifier of the loop, so all instructions targeting the same instruction are part of the same loop. A `LoopBound` object is created for this target, and all loop instructions targeting this target are added to this object. The loop bound for this loop must be found in the source code and added to the `LoopBound` object. The `LoopBound` object is used in the translation to UPPAAL, where the loop bound counters need to be incremented on all possible loop nodes.

This approach to loop identification, relies on specific patterns generated by the compiler. If a different compiler is used, which generates loops differently, this approach might not work.

Switch instructions are not supported in the current implementation and an error is printed if such instructions occur.

Invoke Instructions. The visit method `visitInvokeInstruction` is an abstraction over all invoke instructions. There are three special cases of invoke instructions which need to be handled: Native method invocation, firing of sporadic events, and other method invocations.

A special case is native method invocations, which are translated to the correct instructions.

Calls to `RealtimeSystem.fire(int)` result in a firing of a sporadic event, and must be translated as such. This is handled by finding the actual event being fired in the source code, and creating a `SpoadicInvokeBasicBlock`. Other method invocations are handled by creating a `MethodCallingBasicBlock`.

Monitors. The visit methods `visitMONITORENTER` and `visitMONITOREXIT`, correspond to the instructions `MONITORENTER` and `MONITOREXIT`. These are handled by creating a `MonitorEnterBasicBlock` or `MonitorExitBasicBlock`, respectively.

Java Implemented Bytecode. Some bytecode instructions are implemented in Java. These are translated into invocation of the correct method in the class `com.jopdesign.sys.JVM`. For instance the bytecode `NEW` is translated into an invocation of the method `JVM.f_new`. This is handled by creating a `MethodCallingBasicBlock` to the correct method, instead of a `SimpleBasicBlock` for these instructions.

10.3 Transition Builder

The transition builder creates the flow in the control flow graph. It does this by adding edges between the basic blocks created by the location builder.

`LoopBasicBlock`s are handled specially, since all jumps inside loop which exit the loop need to be found. This is needed by UPPAAL to know when a loop has been exited, so it wont be exited too early, and to be able to reset the loop bound counter.

The next two chapters describe how schedulability analysis is performed using UPPAAL and how SIR is translated to UPPAAL models.

Chapter 11

Uppaal Schedulability Analysis Model

This chapter describes the different principles behind the Uppaal schedulability analysis model. How SIR is translated to Uppaal is described in Chapter 12, this translation is then combined with the Uppaal model described in this chapter, in order to provide a schedulability analysis for the entire system.

A short introduction to the syntax and terms used when creating a model in Uppaal is described in Appendix B, and a more thorough description of Uppaal is available in [5].

11.1 Stopwatches

The Uppaal model used in this project uses a new feature in the development version of Uppaal called stopwatches. The stopwatch feature enables the developer to stop one or more clocks independent of the rest. Clock can be stopped by adding an invariant to a location in the template, depicted in Figure 11.1. The expression must evaluate to either 0 or 1, where 0 stops the given clock, and 1 enables it.

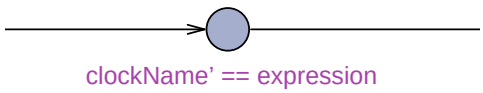


Figure 11.1: Stopwatch syntax

The idea is that when a thread is in a state which requires execution time, the following invariant is added:

```
executionTime <= WCET &&
executionTime' == expression
```

Listing 11.1: Execution invariant

Where executionTime is reset on all incoming and outgoing transitions. `WCET` is the execution time of the corresponding Java bytecode(s). All outgoing transitions have the guard `executionTime == WCET`. The clock `executionTime` is stopped if the current thread is not running, and the thread is therefore in the given state until it has been executing for `WCET` time. This allows preemption by simply stopping the clock of the currently executing thread.

11.2 Global Declarations

The following declarations are shared among all templates in the model, that is all tasks, classes, and the scheduler. These are shown in Listing 11.2.

```
const int periodicThreads;
const int sporadicThreads;
const int totalThreads = periodicThreads +
   sporadicThreads;
const int schedulerID = 0;

typedef int[1,periodicThreads] PeriodicID;
typedef int[periodicThreads + 1, totalThreads] SporadicID
   ;
typedef int[1,totalThreads] ThreadID;

bool schedulable[ThreadID];
bool fireable[SporadicID];
int threadPriority[ThreadID];
int running[totalThreads+1];
int selectedThreadPriority = -1;

chan fire[SporadicID];
chan run[ThreadID];
broadcast chan GO;

void runScheduler(){ ... }

bool synchronized = false;
bool interruptWaiting = false;
int monitorDepth = 0;
void monitorEnter(){ ... }
void monitorExit(){ ... }
```

Listing 11.2: Global declarations

- **Lines 1-4:** Constants set to the amount of periodic tasks, sporadic tasks, total tasks, and the scheduler ID.

- **Lines 6-8:** Defines new types for periodic, sporadic and thread ID.

- **Line 10:** An array indicating which threads are ready to be scheduled. Only periodic threads which has not completed their period, and sporadic threads which have been fired, are ready to be scheduled.

- **Line 11:** Indicates whether a sporadic thread is ready to be fired, i.e. its minimum inter-arrival time since last invoke of fire has passed.

- **Line 12:** Constants set to the priority of each thread. This priority is assigned using the deadline monotonic priority assignment.

- **Line 13:** An array used to denote which thread is currently executing, where 1 indicates executing and 0 stopped. This is a single processor system so only one thread can execute at a given time. The scheduler index is 0. The rest of the indices in the array correspond to the periodic and sporadic threads in the system.

- **Line 14:** Represents the priority of the currently running thread.

- **Line 16:** Used to signal when a sporadic thread is fired.

- **Line 17:** Used to invoke the run method corresponding to the thread.

- **Line 18:** A broadcast channel used to ensure all threads are in the correct state, when initializing the system.

- **Line 20:** Stops the currently running thread and starts the scheduler by setting the 0 index in `running` to 1, and the rest to 0.

- **Line 22:** Used to determine whether the executing thread is in a synchronized state, thus disabling interrupts.

- **Line 23:** If interrupts occur when a thread is in a synchronized state, the interrupt is saved until the thread leaves the synchronized state.

- **Lines 24-26:** Used when synchronized states are entered and left, and updates synchronized accordingly. Nesting is controlled by `monitorDepth`. If a synchronized region is left the scheduler is invoked if interrupts are waiting.

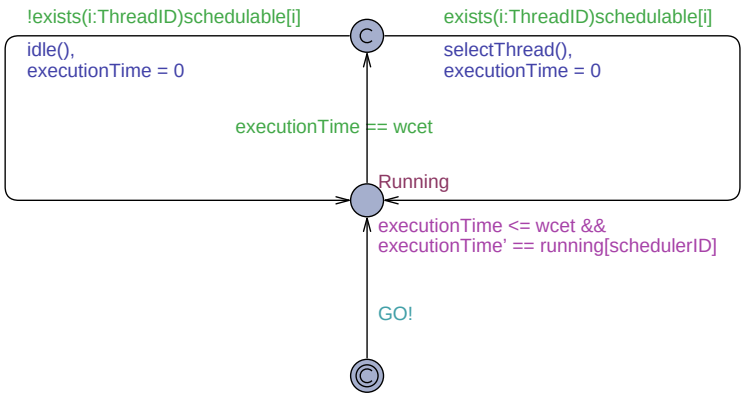


Figure 11.2: Scheduler model

11.3 The Scheduler

The purpose of the scheduler is to schedule the thread with the highest priority, which is ready to be scheduled. The scheduler model is depicted in Figure 11.2.

Initially the broadcast channel `GO!` is signaled to ensure all threads are in their correct state. The scheduler then executes for `wcet` time. If there are any schedulable threads, the highest priority thread is selected, by setting the corresponding index in the `running` array to 1. If no threads are schedulable all indices in `running` are set to 0. This is handled by the two methods `selectThread()` and `idle()` respectively.

11.4 Periodic Thread

For each periodic and sporadic thread in the Java program, a base model is added. The base model for the periodic thread is depicted in Figure 11.3. This model must be supplied with parameters to determine its ID, period, deadline, and offset corresponding to the actual Java implementation of the thread.

Initially the thread waits if an offset is specified. If the thread has a higher priority than the currently running thread, it stops the currently running thread and starts the scheduler, by calling `runScheduler()`. In the actual Java implementation, it is not the threads responsibility to notify the scheduler, but instead the scheduler's responsibility. However, this implementation is not suitable in UPPAAL, since it would make the model unnecessarily complicated. The implementation of `runScheduler()` is shown in Listing 11.3.

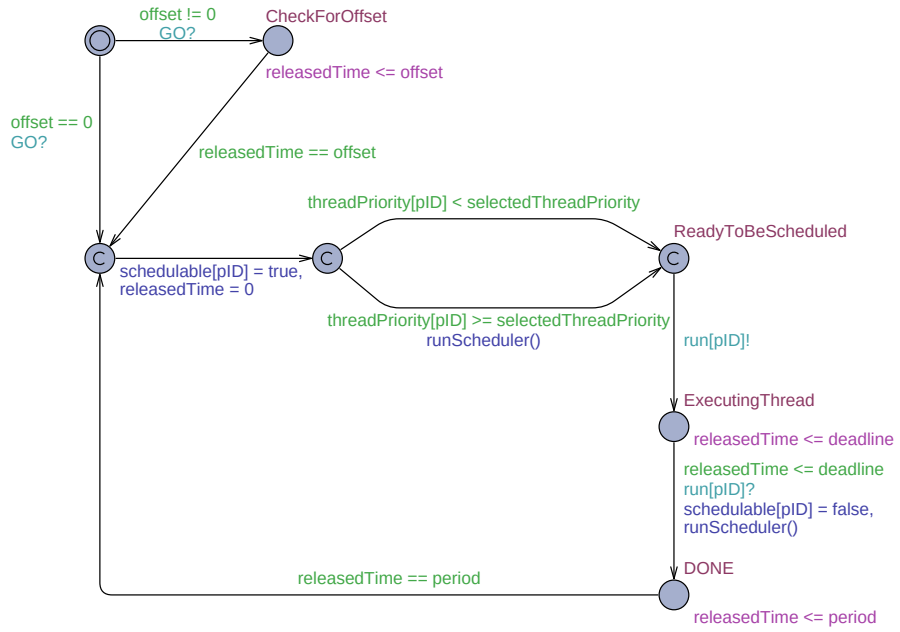


Figure 11.3: PeriodicThread base model

```
1  void runScheduler(){
2          int i;
3          if (synchronized){
4                  interruptWaiting = true;
5          } else {
6                  for (i = 0; i <= totalThreads; i++){
7                          running[i] = 0;
8                  }
9                  running[schedulerID] = 1;
10         }
11 }
```

Listing 11.3: Implementation of runScheduler

- **Lines 3-4:** If the system is in a synchronized region an interrupt is scheduled.
- **Lines 5-10:** Otherwise all threads are stopped and the scheduler is started.

The `run` channel is signaled with the current ID of the thread, `run[pID]!`, it then waits for a response on the same channel, which indicates the thread is done with its run logic. It is then ensured that it has completed before its

deadline, otherwise it results in a deadlock, and the system is not schedulable. The scheduler model is invoked to determine which thread to schedule next. The same procedure continues for each period of the periodic thread.

11.5 Sporadic Thread

The sporadic model is similar to the periodic model, except it must be invoked by signaling `fire` with the sporadic ID. The base model is depicted in Figure 11.4. This model must be supplied with parameters for its ID, minimum inter-arrival time, and deadline. When the minimum inter-arrival time since the last release of this task has passed, the `firable` array is set to true for the specific task, and it is ready to be fired again.

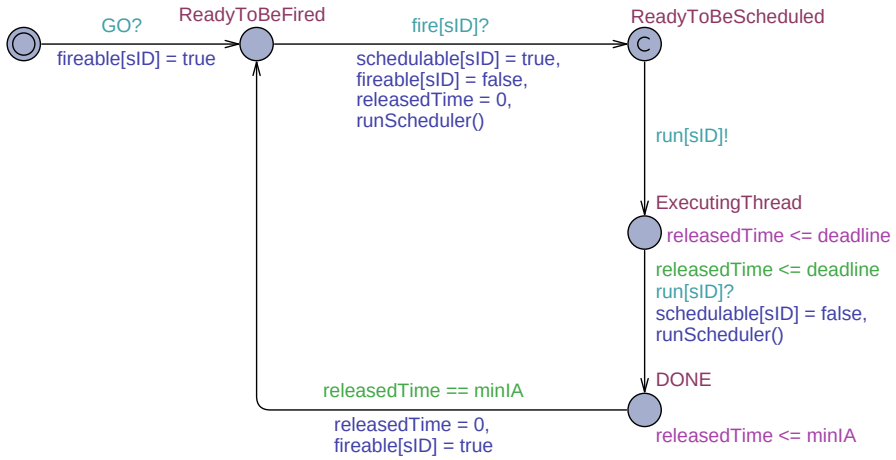


Figure 11.4: SporadicThread base model

11.6 Performing Schedulability Analysis

The scheduler and the concept of periodic and sporadic tasks in the UPPAAL model have been discussed in this chapter, how each of the methods are translated to UPPAAL is described in Chapter 12. If a periodic or sporadic task misses its deadline, then the model is in a deadlock state, because the invariant of the `ExecutingThread` in Figure 11.3 or 11.4, is violated. If a deadlock cannot occur in the system, then the system must be schedulable, because no deadlines are can be missed. In order to determine the schedulability of the model the following query is used:

$$A[]not\ deadlock$$

If this property is satisfied, the system is schedulable.

11.6.1 Decidability

In general, the reachability problem for timed automata extended with stopwatches is undecidable [1], where it is decidable for timed automata without stopwatches. Let A be a timed automata using stopwatches. If A can be reduced to a timed automata without stopwatches, then reachability problems for A are decidable.

This section describes how a SARTS model can be reduced to a model without stopwatches.

SARTS generates a system of timed automata extended with stopwatches, where stopwatches are used to model preemption. In the model, each basic block requires one location in the model and a number of outgoing transitions, depending on the type of block, e.g. branching. In each location the clock representing the execution time associated with the running thread, can be stopped to simulate preemption. The model generated is deterministic in the sense that preemption can only happen at integer points in time i.e. when threads are released and when executing threads are done executing. The use of stopwatches allow simpler models, in terms of locations and transitions, to express preemption with the granularity of one clock cycle in the model. Although since the preemption occurs only when periodic tasks are released, when sporadic threads are fired by periodic threads, and when a thread is done executing, actual preemption at this granularity is very unlikely. This can be equally modeled without the use of stopwatches, by explicitly adding a *preempted* location to each basic block and thereby allow preemption before entering a non-preemptive state, where the execution time for this single basic block is elapsing. To illustrate how it is modeled, a simple example of two sequential instructions is depicted in Figure 11.5 using stopwatches, and how this can be modeled without stopwatches is depicted in Figure 11.6.

The idea here is that it is not possible to preempt the model when executing an instruction, e.g. when it is in state `Inst1`, but it is possible to preempt it before the next instruction is executing, e.g. in state `Inst1Done`. Interrupts are performed by setting the `interruptWaiting` variable to true. The currently executing task then preempts itself and starts the scheduler, which invokes the correct thread again, i.e. the thread with the highest priority is invoked. This pattern must then be applied to all states in the model, in order to convert from a model with stopwatches, to one without. Preemption is therefore only possible between instructions, but splitting basic blocks into blocks which require one clock cycle each, a preemption granularity equal to the model using stopwatches, is achieved, but without the use of stopwatches. This, however, makes the models very large, and will in turn influence verification time. Since it is possible to model this without using stopwatches, the reachability problem is decidable. The reduction from a model with stopwatches to one without is only an indication of how it can

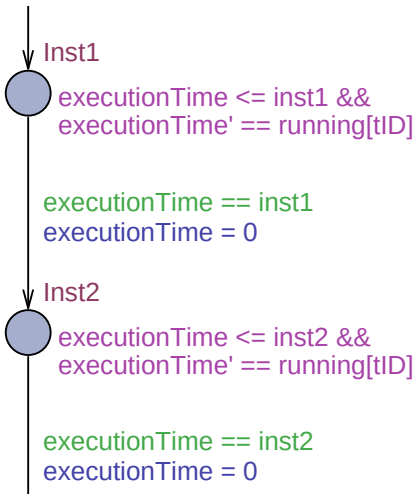


Figure 11.5: Preemption with stopwatches

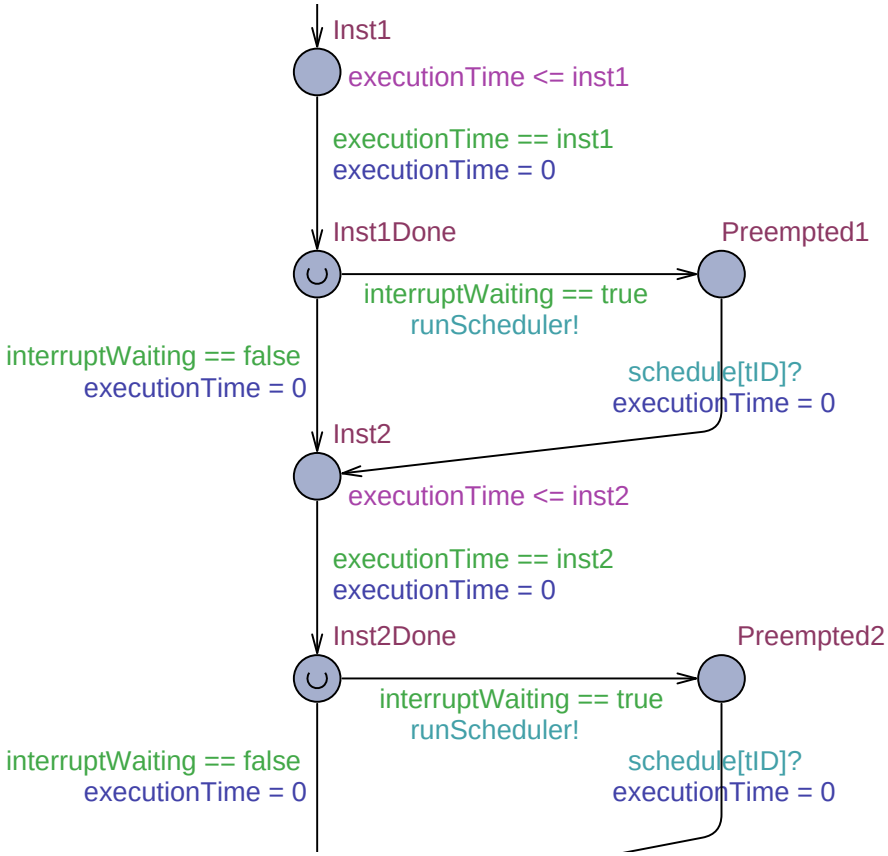


Figure 11.6: Preemption without stopwatches

be performed, and not exactly a proof. It is an open question whether the performance is significantly increased by using stopwatches, since the model is significantly smaller in the number of locations and transitions.

The next chapter describes the actual translation of basic blocks from SIR to UPPAAL.

Chapter 12

UPPAAL Translation

The UPPAAL translation is located in the `sarts.uppaal.translation` package. It consists of a set of visitors, which visit the entire intermediate representation, described in Section 9.1. A UPPAAL template is then created for each method in the system. This template represents the CFG built of the different types of basic blocks. The corresponding UPPAAL template for each of the basic blocks is described in the following sections.

12.1 Simple Basic Block

A basic block modeled in UPPAAL is depicted in Figure 12.1. The state is named `BasicX` where `X` is an increasing variable used to ensure each state has a unique name. An invariant is added to ensure the model stays in this state as long as the WCET of the represented bytecode. Whether the given task is executing is denoted by `executionTime' == running[tID]` using stopwatches as described in Section 11.1. The execution time is reset on the outgoing edge to ensure the model stays in the next state in the correct amount of time.

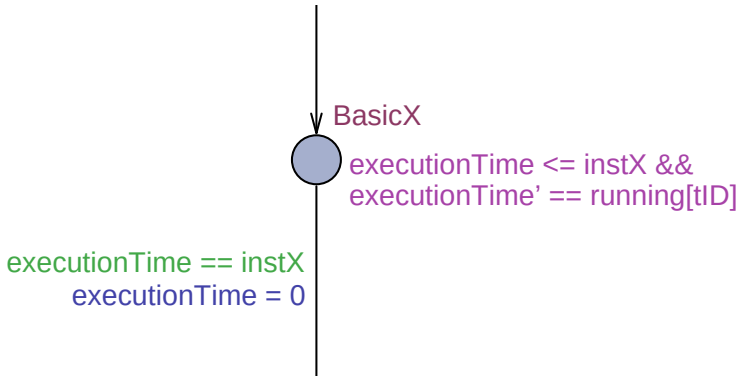


Figure 12.1: Basic Block

Each basic block, uses the same notation to represent the correct amount of time spent in a state. When describing the rest of the basic block types, only additional information added to the template is described.

12.2 Method Calling Basic Block

When a method invocation is performed, it corresponds to switching to another template in Uppaal. This is modeled as depicted in Figure 12.2. A channel for each method in the real-time system is added to the Uppaal model, and the correct method must be invoked. At compile time it is not always possible to uniquely identify, which implementation of a method will be invoked. This is due to dynamic dispatching, described further in Section 15.1. This is handled by adding all possible implementations of the specific method to the Uppaal template, denoted by `methodName1` to `methodNameN`. The template then changes state to `running_MethodCallingX` and waits until the method returns, by signaling on the corresponding channel. Note that the `running_MethodCallingX` and `returningFrom_MethodCallingX` locations are added during the translation to Uppaal, and are not present in SIR.

Using this design ensures that Uppaal considers all possible method candidates for this call. However, this might be pessimistic and an approach to refine the set of method candidates is discussed in Section 15.1.

The variable `methodSwitchCost` is set to the cost of fetching the method returned to, into the cache. In the current implementation method caches are always assumed to miss. How the use of a method cache could be implemented is discussed in Section 15.4.

12.2.1 Sporadic Invoke Basic Block

A special case of method invocation is when `fire` is invoked, corresponding to releasing a sporadic task. How this is modeled is depicted in Figure 12.3. When releasing sporadic tasks, their minimum inter-arrival time must be met, which is ensured by checking whether the specific sporadic thread is ready to be fired, using the `firable` array, described in Section 11.5.

If the sporadic thread is not ready to be released, the template enters a committed state and continues to loop, resulting in a livelock. Uppaal then backtracks and select another branch in a previous branching block. If the sporadic thread is ready to be released, `fire[Y]` is signaled and the scheduler is run in order to determine which task to execute. Note that this design assumes that the minimum inter-arrival time is not violated in the actual implementation, which might not be true in the actual real-time system. However, this is a problem in the specification of the system or the implementation of the real-time system, discussed in Section 15.3.

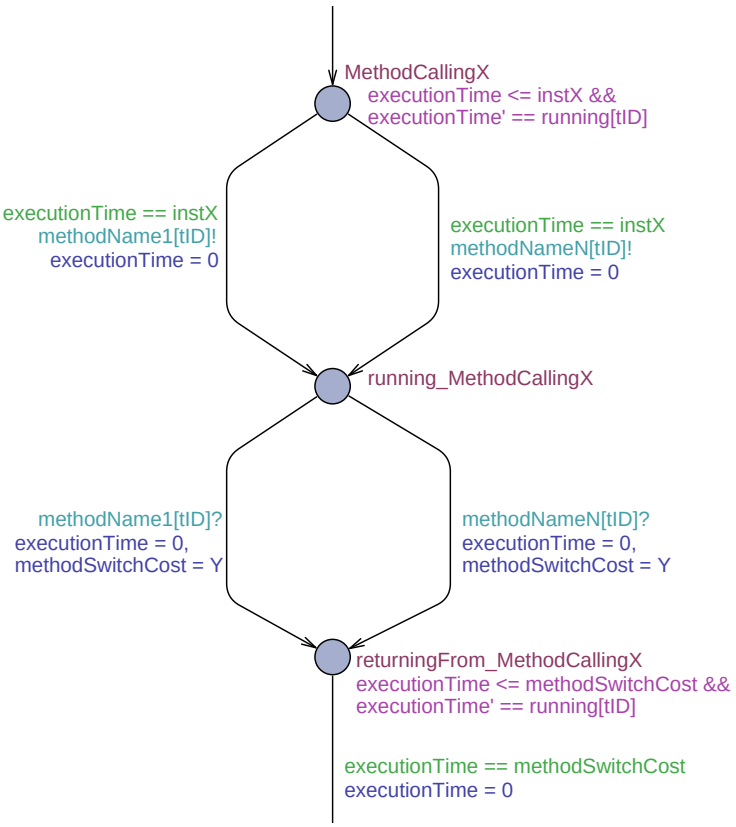


Figure 12.2: Invoke of a standard method

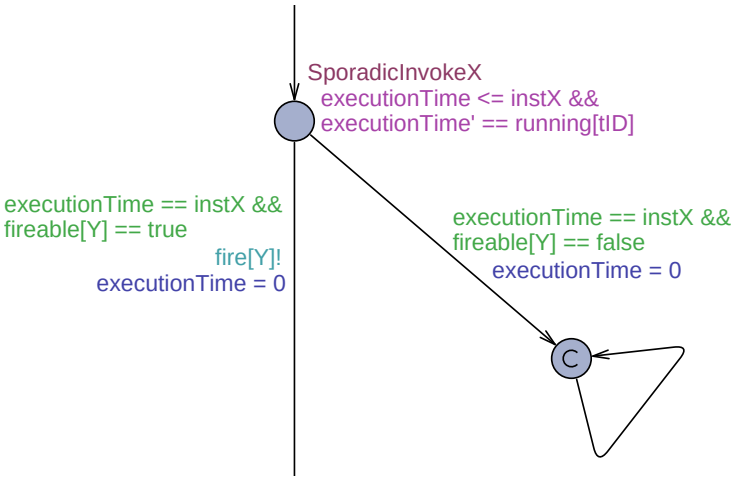


Figure 12.3: Invoke of a fire method

12.3 If Basic Block

When reaching an *If* location, UPPAAL performs a nondeterministic choice between transitions, illustrated in Figure 12.4.

`ThenBody` and `ElseBody` each represent one or many blocks, in the figure, abbreviated for readability, and `EndIf` represents the next basic block after the if block.

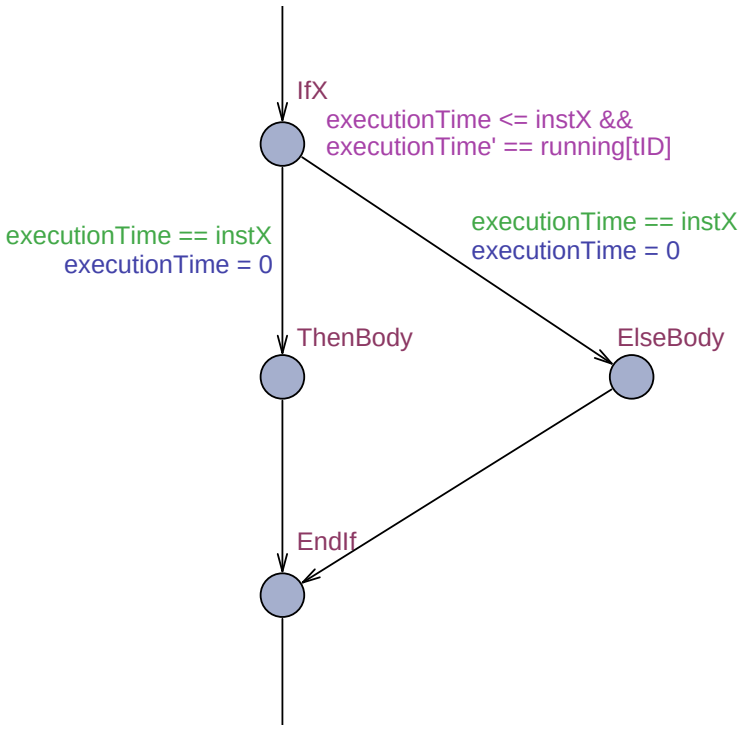


Figure 12.4: If basic block

12.4 Loop Basic Block

When loops are modeled in UPPAAL they must be supplied with a loop bound annotation from the source code, which in this project is assumed to be specified by the programmer. The code inside the loop is then run as many times as the loop bound indicates, as depicted in Figure 12.5. Each time the loop is entered the loop bound variable is incremented, and reset when the loop is complete.

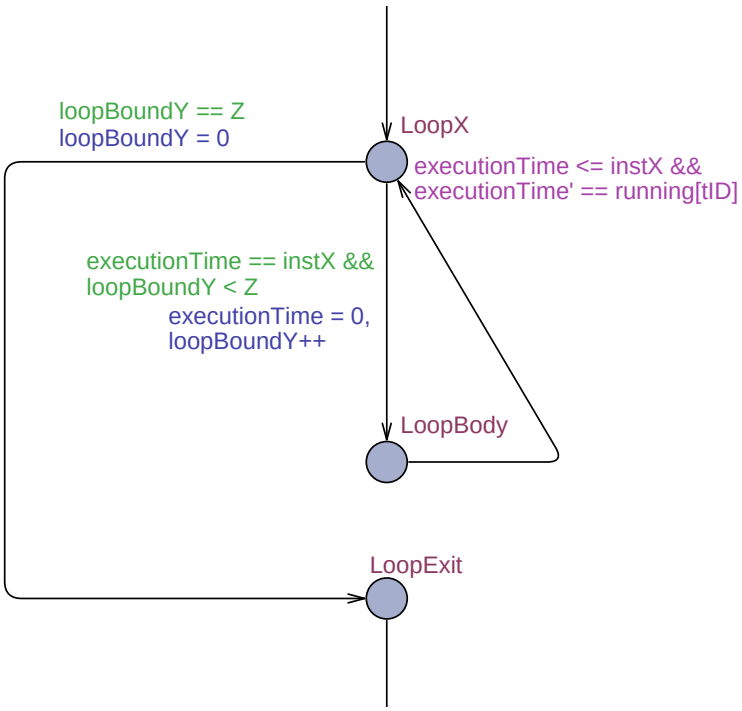


Figure 12.5: Loop basic block

12.5 Synchronization

JOP disables interrupts inside a synchronized region, interrupts occuring during a synchronized region is handled when the region is left. Nesting of synchronized regions are allowed, and interrupts are enabled again when the last synchronized region has been left. How this is modeled in UPPAAL is depicted in Figure 12.6 and 12.7.

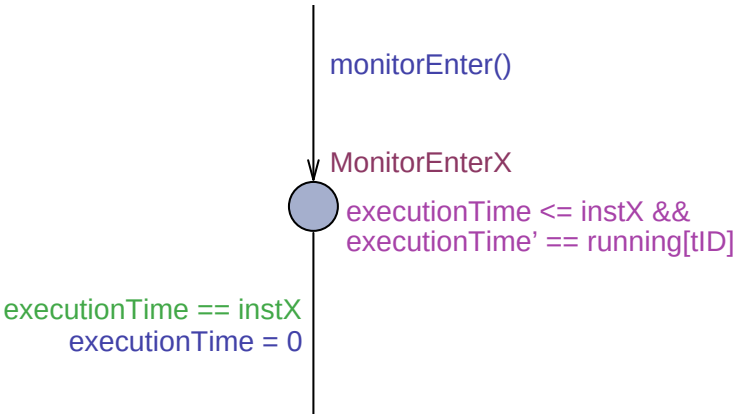


Figure 12.6: Monitor enter basic block

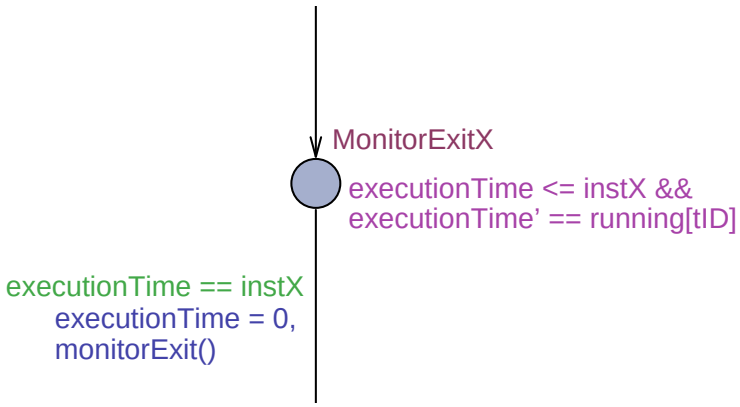


Figure 12.7: Monitor exit basic block

Where the `monitorEnter` method is added when entering a synchronized region, disabling interrupts. The `monitorExit` method is added when leaving a synchronized region, enabling interrupts if the last synchronized region is left. If interrupts have been requested during a synchronized region, the scheduler is invoked. The implementation of `monitorEnter` and `monitorExit` are shown in Listing 12.1.

```
1  void monitorEnter(){
2          synchronized = true;
3          monitorDepth++;
4  }
5
6  void monitorExit(){
7          monitorDepth--;
8          if (monitorDepth == 0){
9                  if (interruptWaiting == true){
10                         interruptWaiting = false;
11                         runScheduler();
12                 }
13                 synchronized = false;
14         }
15 }
```

Listing 12.1: Implementation of monitorEnter and monitorExit

The depth of the synchronized region is incremented each time `monitorEnter` is invoked and decremented each time `monitorExit` is invoked. If the depth is 0, meaning the outer synchronized region has been left, the scheduler is invoked if interrupts are waiting. The implementation of `runScheduler` is shown in Listing 11.3. Synchronized methods are supported by adding `monitorEnter` and `monitorExit` to the incoming and outgoing transitions when a `MethodCallingBasicBlock` is translated.

12.6 Empty Basic Block

As described in the design, empty basic blocks are only added for convenience reasons and do not represent any bytecode. A basic block is modeled like a basic block, where the waiting time is set to 0, because no bytecode is executed. An empty block is added to the beginning and the end of each template, to represent method invoke and return. On a method invoke, the cost of loading the method into the method cache is added to the cost of the first empty basic block.

This concludes the explanation of the implementation of SARTS. To improve performance of the verification process, several optimizations have been implemented. These optimizations are described in the next chapter.

Chapter 13

Optimizations

This chapter describes some of the optimizations which have been implemented, in the current version of SARTS. The idea is to describe why these optimizations help and how much. The optimizations presented in this chapter mainly focus on reducing the state space of the model, and in general reduce the execution time needed to verify the system. Some optimizations additionally result in a more accurate analysis, while still being an over approximation. Additional optimizations, which have not been implemented yet, are discussed in Chapter 15.

When experiments have been conducted in this project, the computer being used to verify the models, has the following specification: Sun Fire X4100, with two 2.4 GHz CPUs (Dual Core AMD Opteron 275) and 4096 MB RAM running Suse Linux Enterprise Desktop 10 - 64bit. The version of UPPAAL used is 4.1.0 (rev. 3425), February 2008.

13.1 Remove Unused Templates

In the initial phase, the Java implementation is translated to SIR, described in Chapter 10. Each time an object is created, the class is translated to SIR together with all parent implementations of that class. Translating a class to SIR includes translating all methods in the class to SIR. This results in a lot of unused methods being translated to SIR. Before SIR is translated to UPPAAL, it is checked whether a method is used in the actual execution of the system. This includes hiding the methods invoked during scheduling and sporadic invoke, because this is handled by a default template in UPPAAL, described in Chapter 11. This greatly reduces the amount of templates in the final UPPAAL model, increasing the throughput of the verification, as less states are considered. $\text{RTSM}_{\text{SIMPLE}}$ has been translated to a UPPAAL model with and without unused templates being removed. How this reflects the amount of templates and verification time is shown in Table 13.1.

This reduction does not compromise the accuracy of the verification,

Configuration	Number of templates	Verification time
Standard	460	Out of memory
Templates removed	17	1m 55s

Table 13.1: Removal of unused templates

because only unused methods are removed. It has not been possible to get the verification to terminate without this optimization, simply because the system runs out of memory. The system ran out of memory after 75 minutes.

13.2 Collapse Basic Blocks

When Java is translated to SIR each bytecode is represented by a basic block, this results in long sequences of basic blocks. Preemption is possible during a basic block, because of stopwatches, i.e. a basic block with a WCET of 100, and a sequence of 10 basic blocks with a WCET of 10 represents the same system, in regards to timing behavior. Sequences of basic blocks are therefore collapsed into one basic block, where the WCET of each block is added together. Collapsing basic blocks results in a more compact model, which result in a faster verification. A more compact model is also an advantage when manually inspecting the models, due to a more simple representation increasing the overview of the model. How collapsing of basic blocks affects the number of total states in the system, when translating $\text{RTSM}_{\text{SIMPLE}}$, is shown in Table 13.2.

Configuration	Number of states	Verification time
Standard	383	2m 36s
Collapsing of basic blocks	209	1m 55s

Table 13.2: Collapsing of basic blocks

13.3 Remove Invalid Traces

As mentioned, the schedulability analysis is performed by verifying that the translated model contains no deadlocks. A method for reducing the amount of traces, examined by UPPAAL, is to introduce livelocks, thereby eliminating an invalid trace. This technique has been used when invoking sporadic tasks.

It is assumed that the minimum inter-arrival time for a sporadic task is not violated, as it can result in unpredictable behavior. If a trace to a firing of a sporadic task, where the minimum inter-arrival time requirement is not met, it is assumed that this is not a valid trace, since this would violate the

requirements. This is handled by entering a livelock, as depicted in Figure 13.1.

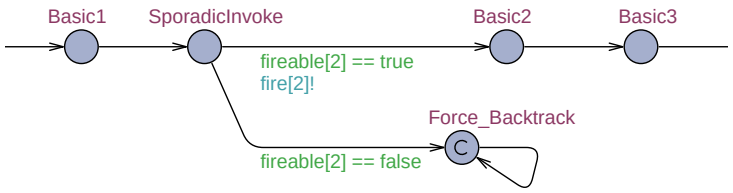


Figure 13.1: Invoke of sporadic task

The old approach is depicted in Figure 13.2, where the trace continues instead of forcing a backtrack. This approach has two flaws, the state space is increased, because an invalid trace is completed. Furthermore, this trace might result in a more pessimistic analysis, because the trace, which leads to this sporadic invoke might have a higher WCET or blocking regions.

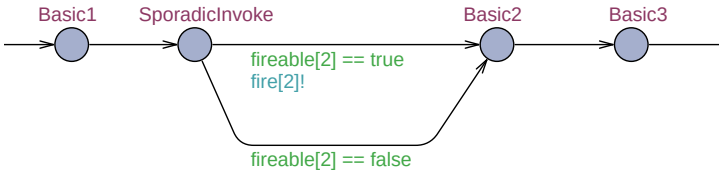


Figure 13.2: Old invoke of sporadic task

The approach used in the current implementation, Figure 13.1, assumes that other alternatives are available, i.e. a previous branch exists, not leading to firing of the sporadic event. If no such alternative is available the analysis can result in a false positive, this problem is discussed in Section 15.3.

13.3.1 Experiment

In order to illustrate the improvement of introducing a livelock to remove invalid traces, a small example is used. The system consists of a periodic and a sporadic task. The sporadic run method is empty and the periodic run method is shown in Listing 13.1. This example is not a realistic example, but illustrates the improvements of introducing livelocks.

The idea behind the example is that the `if` condition on line 3 is only true once in the loop, and the minimum inter-arrival time of the sporadic task is therefore not violated.

```
1 protected boolean run() {
2   for (int i = 0; i < foo.length; i++){ //@WCA loop=10
3     if (foo[i] == true){
4       RealtimeSystem.fire(2);
5       complexMethod();
6     }
7   }
8   return true;
9 }
```

Listing 13.1: Invalid trace removal example

If the backtrack state is removed, `complexMethod` is in the worst case executed 10 times instead of 1, thus resulting in a pessimistic analysis. The result of the experiment is shown in Table 13.3.

Configuration	Verification time	Result
Standard	0.31s	Satisfied
No sporadic backtrack	2.32s	MAYBE NOT satisfied

Table 13.3: Removal of invalid traces

As expected the addition of the livelock state reduces the state space, and thus reduces verification time. This is a very simple example, but it illustrates how the verification time needed is significantly higher without the optimizations. When the sporadic backtracking is removed, UPPAAL returns **Property is MAYBE NOT satisfied**, and the system is determined as not schedulable by SARTS.

13.4 Limit Total Execution Time

It might be possible to determine whether a system is schedulable before UPPAAL has examined the entire model, by limiting the total execution time of the model. Note this is not the execution time of the verification, but the amount of time each trace in the system is examined regarding its clocks. This is handled by adding a global clock to the system, and then add a bound for when the scheduler should stop. The idea is that if this time is met, and no deadlocks have been detected, then the system must be schedulable.

This value is determined by least common multipe (LCM) of the periods of all periodic tasks in the system, referred to as *hyper period*. The maximum load on the processor is when all the tasks are released, known as a critical instant [7], a point in time which will be repeated each hyper period. If no deadlocks have occurred in all possible traces to this point, the system must be schedulable. Worth noting is how the choice of periods can influence

LCM, e.g. prime numbers are not a good choice. However, this value is not sufficient if one or more of the periodic tasks have an offset. It also requires that none of the sporadic tasks are running, because if they are, the next hyper period will not be equal to the first.

How this can be added to the scheduler template is depicted in Figure 13.3, where X denotes LCM of the periods of all periodic tasks in the system.

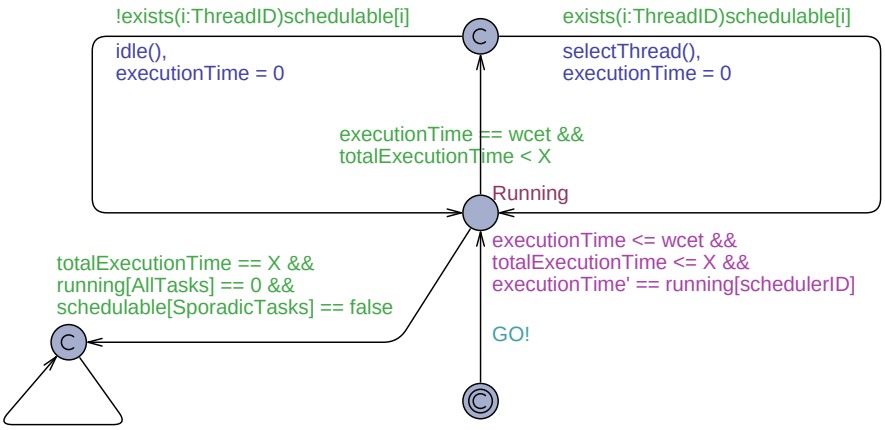


Figure 13.3: Limit total execution time

When X time has passed, the system enters a livelock state. None of the tasks in the system must be running, ensured by `running[AllTasks] == 0`. All sporadic tasks must either have completed their execution or not have been released, ensured by `schedulable[SporadicTasks] == false`. This ensures that the sporadic tasks are not in the `ExecutingThread` state, depicted in Figure 11.4 Section 11.5. Note that `AllTasks` and `SporadicTasks` in the figure represents IDs of periodic and sporadic tasks respectively. If these additional guards are added to the scheduler, and the system is verified to be schedulable, it is sufficient. However, the system might be determined not to be schedulable if one of the sporadic tasks has not been completed, but the system might be schedulable anyways. A less restrictive addition to the scheduler template is depicted in Figure 13.4.

This modification ensures that the system enters an idle state after X amount of time has passed. If the system is determined to be schedulable with this approach, it is indeed schedulable. Note if the system is determined to not be schedulable with this restriction, removing it does not help.

The optimization described in this section is not implemented; it is not automatically generated as a part of the translation. However, the optimization in Figure 13.3 has been used in all the experiments performed in Chapter 14.

How these approaches affect the verification time is shown in Table 13.4. Note that all the experiments have been conducted with a depth first search

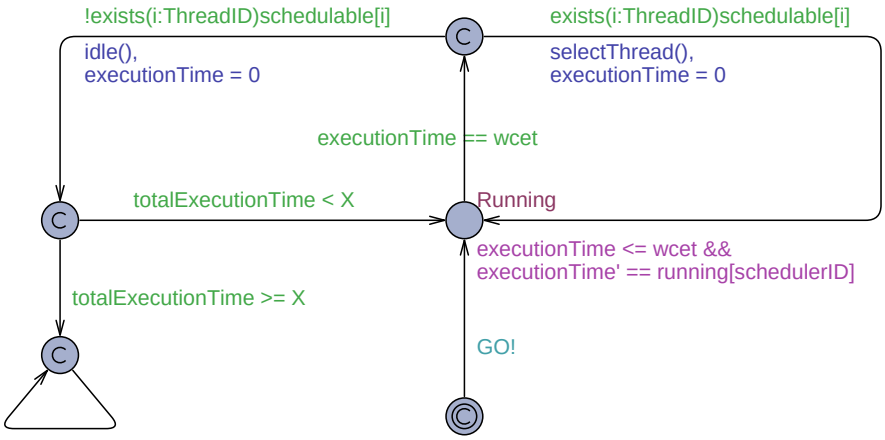


Figure 13.4: Limit total execution time - less restrictive

and aggressive state space reduction, to reduce the verification time. The most restrictive bound yields a significantly lower verification time. It is therefore advised to use the approach from Figure 13.3, and it has also been used in all the experiments conducted in this report.

Configuration	Verification time	Result
Figure 13.3 approach	51s	Satisfied
Figure 13.4 approach	401m 21s	Satisfied
No bound	712m 54s	Satisfied

Table 13.4: Limitation of total execution time of $\text{RTSM}_{\text{SIMPLE}}$

This concludes the implementation of SARTS and the different optimizations implemented. A list of limitations of the current implementation can be found in Appendix C. The next part describes the results and future directions of SARTS.

Part III

Results

Chapter 14

Experiments

This chapter documents a series of experiments conducted, to evaluate the implementation of SARTS. SARTS is used to perform a full schedulability analysis, where tools like WCA and Volta support only an isolated WCET analysis for each method. A direct comparison is therefore not possible. However, a modified version of SARTS is available in `sarts.MethodWCET`, which allows WCET analysis for a specific method, in order to allow a comparison with WCA and Volta. Volta is based upon WCA and should return the same WCET result, and only provides a more user friendly output. However, JOP has been updated with new execution times, and WCA has been updated accordingly, whether the current version of Volta is updated is unknown. WCA and Volta uses the same approach to WCET analysis, and it is the approach which is interesting to compare to SARTS and not the specific tool. SARTS is therefore only compared to WCA in the experiments.

14.1 WCET Calculation

WCA only supports WCET calculations for a single method. These values are then used in traditional approaches, described in Chapter 3, in order to determine whether the system is schedulable. However, the utilization test requires the tasks to be independent and non-blocking. The experiment presented in this section is used to illustrate that SARTS and WCA result in the same WCET estimate. A small method with nested loops is shown in Listing 14.1. This method is used as an example to illustrate the accuracy of WCA, and is therefore used in this experiment.

The interesting part of the method is whether the tool can determine the most time consuming path, i.e. the first loop. The method is analyzed with WCA, SARTS, and executed on JOP, where the actual amount of clock cycles used is determined. The method is invoked with `b = true` in order to hit the worst case path.

```
1  public static int measure(boolean b, int val) {
2    int i, j;
3    for (i=0; i<10; ++i) {          //@WCA loop=10
4      if (b) {
5        for (j=0; j<3; ++j) {       //@WCA loop=3
6          val *= val;
7        }
8      } else {
9        for (j=0; j<4; ++j) {       //@WCA loop=4
10         val += val;
11       }
12     }
13   }
14   return val;
15 }
```

Listing 14.1: Method example

The result of the experiment is shown in Table 14.1. WCA and SARTS

	WCET
JOP	1369
WCA	1553
SARTS	1553

Table 14.1: WCET results for measure

return, as expected, the same result, because they both use the most resent bytecode instruction costs for JOP. However, this seems to be a pessimistic result, which is not entirely true. The execution of the code on JOP does not include `return val;` which is translated to `iload_1` and `ireturn`, which cost 1 and 23 clock cycles respectively. `return val;` is not included because the `timeBegin();` and `timeEnd();` are inserted after line 1 and 13 respectively. The difference between the calculated and the measured WCET is now 160 clock cycles. This is caused by the anomaly with the `iinc` instruction, as described in Section 5.4, which is measured to 4 clock cycles, but specified as 8 clock cycles. In the worst case path, `iinc` is used 40 times, which is a difference of 160 clock cycles, which explains the difference. So if `iinc` was updated to take 4 clock cycles and `iload_1` and `ireturn` were included, the measurement would be equal to the calculated WCET.

14.2 Conditional Sporadic Events

This experiment shows how dependencies between sporadic tasks are detected in SARTS. The dependency consists of a periodic task firing two sporadic tasks in mutually exclusive branches.

The system tested consists of one periodic thread and two sporadic threads. The run method for the periodic thread is shown in Listing 14.2. The periodic class `Experiment2` fires either event 1 or event 2, but never both in the same period. The period and minimum inter-arrival times are set to 4 microseconds. The sporadic tasks have the same WCET.

```
1  public class Experiment2 extends PeriodicThread {
2          public boolean run() {
3                  if(b) {
4                          RealtimeSystem.fire(1);
5                  } else {
6                          RealtimeSystem.fire(2);
7                  }
8                  return true;
9          }
10 }
```

Listing 14.2: Conditional sporadic invoke

The WCET for each run method can be seen in Table 14.2. The period calculated into clock cycles is 240. Calculating the processor utilization for this system gives:

$$\left(\frac{161}{240}\right) + \left(\frac{64}{240}\right) + \left(\frac{64}{240}\right) = 1.20 \tag{14.1}$$

This shows that the system uses more than the available processor time, and is therefore not schedulable according to any of the approaches described in Chapter 3. Running SARTS on this system will correctly show is as being schedulable, since the model checker can deduct that the two sporadic events will never be fired at the same time. A time-line for the execution of the system can be seen in Figure 14.1.

	WCET
Periodic thread	161
Sporadic thread	64

Table 14.2: WCET for the threads

14.3 Scaling

Several experiments have been conducted to illustrate the scalability of SARTS. The experiments consider only the time used to verify the system in UPPAAL, because the translation time is insignificant. The different optimizations described in Chapter 13 have all been enabled in the experiments. Both the standard version and $\text{RTSM}_{\text{SIMPLE}}$ have been verified, with and without the use of `RelativeTime`, to reduce verification time.

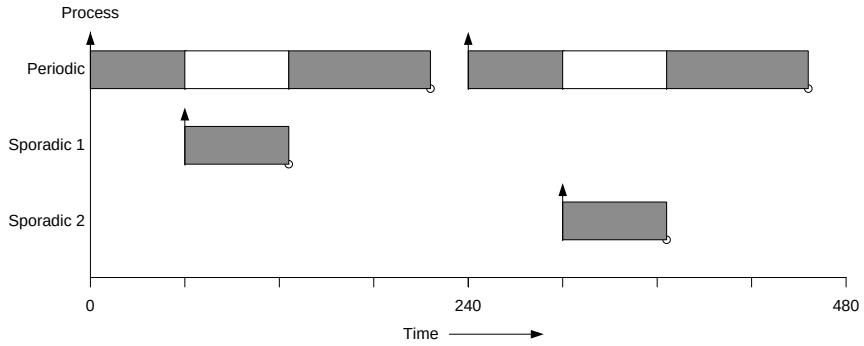


Figure 14.1: Time-line for conditional sporadic invoke

Two different compilers, `javac` and `Eclipse`, are used, as these generate different bytecode. This make the generated models different, influencing the verification time.

The execution of the verification for each of the configurations is shown in Table 14.3. These results indicate that SARTS does not scale well.

System	Compiler	Verification time	Result
RTSM	Javac	27h 15m 26s	Satisfied
RTSM	Eclipse	5h 42m 10s	Satisfied
$\text{RTSM}_{\text{SIMPLE}}$	Javac	14m 29s	Satisfied
$\text{RTSM}_{\text{SIMPLE}}$	Eclipse	1m 55s	Satisfied

Table 14.3: Verification time of RTSM

However, it is possible to reduce the time needed to verify the systems, using the options available in UPPAAL. Additional tests have therefore been conducted. Table 14.4 is the same experiments where a depth first search instead of breath first search is used, and aggressive state space reduction is enabled.

System	Compiler	Verification time	Result
RTSM	Javac	6h 30m 01s	Satisfied
RTSM	Eclipse	1h 28m 29s	Satisfied
$\text{RTSM}_{\text{SIMPLE}}$	Javac	4m 23s	Satisfied
$\text{RTSM}_{\text{SIMPLE}}$	Eclipse	51s	Satisfied

Table 14.4: Verification time of RTSM using depth first search and aggressive state space reduction

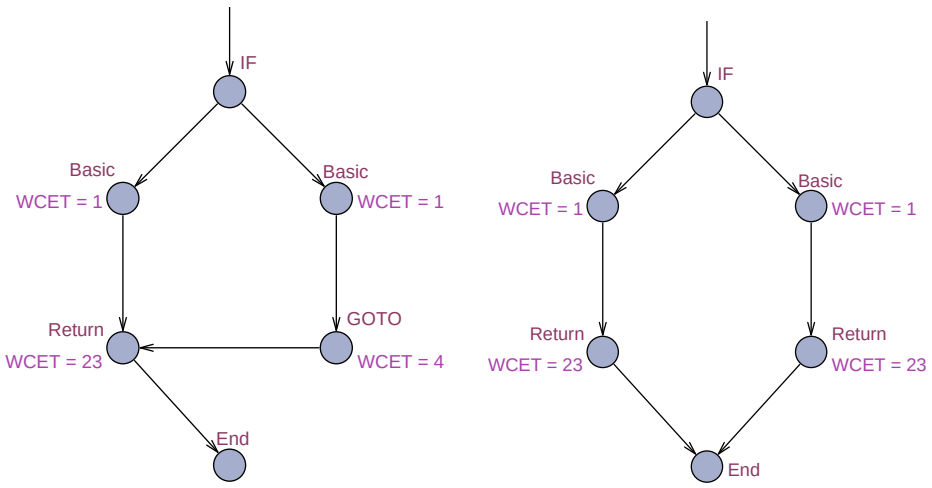
UPPAAL also supports a convex-hull approximation option, reducing verification time at the cost of an over approximate answer. If UPPAAL using convex hull determines the property to be satisfied, then it is also satisfied

without the approximation. The result of this experiment is shown in Table 14.5.

System	Compiler	Verification time	Result
RTSM	Javac	52s	Satisfied
RTSM	Eclipse	37s	Satisfied
$\text{RTSM}_{\text{SIMPLE}}$	Javac	16s	Satisfied
$\text{RTSM}_{\text{SIMPLE}}$	Eclipse	9s	Satisfied

Table 14.5: Verification time of RTSM using convex-hull approximation

An interesting indication of the poor scalability is the difference in verification time even though it is the same system, but with a different compiler. The generated models for $\text{RTSM}_{\text{SIMPLE}}$ have been inspected, to detect the difference. The only major difference is in the `isEmpty()` method in `BoundedBuffer`, where the final if statement is translated differently. The two different models are depicted in Figure 14.2.



(a) (b)

Figure 14.2: (a) Compiled with Javac (b) Compiled with Eclipse

These two models are semantically equivalent, disregarding execution time. The Javac version has a single return statement and a jump to this statement from the other branch, the Eclipse version has two return statements. In the Eclipse version both branches have the same cost. The state of the model when the template enters the `end` location is therefore independent of the previous branch, and the branch becomes insignificant. If this optimization is introduced in the Javac model, it has a similar verification time as the Eclipse version. This indicates how small changes in the model can drastically change the verification time needed.

14.4 Summary

Experiments have been conducted to test the accuracy and scalability of SARTS. It has been shown that SARTS is capable of determining a precise WCET estimate of a method, compared to executing it on JOP. This estimate is comparable to WCA. It has been shown that SARTS is capable of producing a less pessimistic result than possible with traditional approaches. These two results show that it is possible to derive locally accurate WCET estimates, combining this with information about the control flow of the actual execution of the system, results in a more accurate analysis. However, SARTS has some issues regarding scalability, even small changes in the program can drastically change the verification time needed. Several optimizations are available in Uppaal, which can reduce the verification time, while still providing sound results.

Further improvements to increase the accuracy and scalability of SARTS are discussed in the next chapter.

Chapter 15

Improvements

In this chapter, some bottlenecks of the current implementation are considered and some suggestions for improvements are presented. These improvements have not been implemented, but are considered as future work suggestions. The goal is to lower the state space of the model in order to speed up verification, but also to get a more tight result while still being sound.

15.1 Method Invocation

In a program, the virtual methods may cause more methods to be considered at runtime, what is known as dynamic dispatching, i.e. finding the correct implementation of a method. In Java, methods are by default virtual as described in Section 16.2. The current implementation of SARTS will make UPPAAL consider all possible methods at a given method call. This makes the resulting analysis pessimistic, as the method with highest execution cost and/or particular interleaving will dominate the analysis and also increase the state space since more choices are considered. Reducing the set of possible methods which can be invoked on a method call, is therefore an improvement to the state space and possibly a more tight analysis.

As an example, consider the Java code snippet in Listing 15.1. The class diagram in Figure 15.1 shows the relationship between Triangle, Drawable and several other classes. The draw methods considered in this piece of code, if translated into a model, will be from the classes: `Drawable`, `Character`, `Shape`, `Square`, and `Triangle`, while the correct implementation actually will be from Triangle.

```
1 Drawable d = new Triangle();
2 d.draw();
```

Listing 15.1: Java code snippet

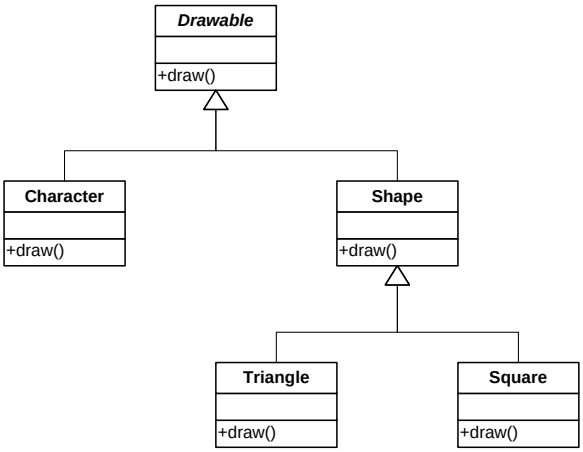


Figure 15.1: Class diagram of Drawable

15.1.1 Static analysis

Static analysis can be used to solve this problem, by minimizing the set of methods considered in a method call. This must be an over approximation to preserve the correctness of the analysis. Knowing the concrete type of all objects at any given time in the program, will provide enough information to identify the concrete method at a method call. To be able to reduce the set of method implementations considered at a given virtual method call, the concrete type of the object on which the method is specified must be known. Thus, the problem of identifying method implementation can be solved if the concrete type of the object is known.

An analysis for type inference for Java is presented in [40]. The analysis presented is used for checking down casts of objects, but the technique can be used for performing concrete class analysis as well; this analysis would provide a conservative estimate of the type of each object in the program [40].

Additionally, this analysis could be extended by selecting from this conservative set of types, the type of an object appropriately in each execution path. This amounts to generating UPPAAL code, which would execute abstractions of the actual code, which only tracks the possible types of given objects.

15.2 Predicate Abstraction

Predicate abstraction is widely used in conjunction with model checking, to verify safety properties of systems [4]. This makes it obvious to incorporate predicate abstraction in SARTS.

The advantages of predicate abstraction are twofold: are tighter analysis

and a reduced state space. Knowledge about predicates can limit branches in the verification process, reducing the state space, as well as possibly removing invalid traces.

The idea is to create abstractions over values in the Java program, and track the value of these predicates during the verification in UPPAAL.

A suggestion to how predicate abstraction can be applied to the UPPAAL model is presented in the remainder of this section.

The predicate abstraction can be inserted into the UPPAAL model by creating variables for each predicate. These variables are represented using three-valued logic, the values being: `TRUE`, `FALSE` and `UNKNOWN`. A new type and some constants defined for this purpose are shown in Listing 15.2. All condition variables are of type `Condition`, and are assigned the value `TRUE`, `FALSE`, or `UNKNOWN`.

```
1 typedef int[0,2] Condition;
2
3 const Condition TRUE = 0;
4 const Condition FALSE = 1;
5 const Condition UNKNOWN = 2;
```

Listing 15.2: Types and variables for predicate abstraction

When a branch is encountered in the model, the appropriate branch needs to be taken if the value of the predicate is known. If the value is unknown, a nondeterministic choice is made, and the variable is updated to the correct value. This is depicted in Figure 15.2.

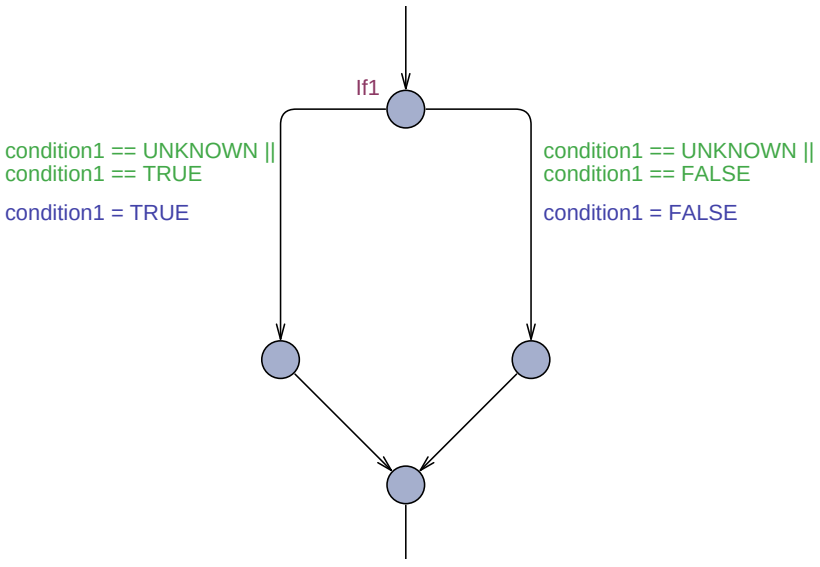


Figure 15.2: Branch with predicate abstraction

Whenever a variable is updated in the Java program the predicate in the

Uppaal model needs to be updated as well.

15.2.1 Experiment

This experiment calculates the WCET for a method using SARTS and WCA, and compares it to the actual execution time on JOP. The method tested is shown in Listing 15.3. Both simple methods contain only a single `i++`, and both complex methods contain a loop which iterates 10 times.

```
1  public void testMethod() {
2    if(condition1){
3      simpleMethod1();
4      condition2 = true;
5    } else {
6      complexMethod1();
7      condition2 = false;
8    }
9
10   if(condition2)
11     complexMethod2();
12   else
13     simpleMethod2();
14 }
```

Listing 15.3: Predicate abstraction experiment

Two measurements are conducted on JOP, with `condition1` set to `true` and `false` respectively. The execution time is calculated using WCA, using standard SARTS, and finally using SARTS with predicate abstraction. The model using predicate abstraction is constructed manually as described above. The method cache has also been manually inserted in the model, to make the result consistent with WCA and JOP, as described in Section 15.4.3.

Configuration	**Execution time**
JOP - true	446
JOP - false	438
WCA	590
SARTS- no abstraction	590
SARTS- with abstraction	467

Table 15.1: Execution times

The calculated times can be seen in Table 15.1. It can be seen that without abstraction, SARTS calculates the same value as WCA. With abstraction added the value calculated by SARTS is 21 higher than the actual execution. This is the problem with the return instruction not being added.

The method is a void method, a return from a void takes exactly 21 clock cycles, making the result exactly equal to that of the actual execution.

15.3 Sporadic Thread

To reduce the number of traces explored while model checking, live locks are placed in the model where invalid traces would be explored, discussed in Section 13.3. Section 13.3 deals with invalid traces regarding the firing of sporadic tasks, which requires the adherence to the specification of minimum inter-arrival time of sporadic tasks. It is currently not possible to verify that a trace will always reach the end of a run method whenever a run method is entered. This would be expressed in UPPAAL as $A[](p \text{ imply } E <> q)$, which means that reaching p it should always be possible to reach q; this is not allowed in UPPAAL because nesting of quantifiers is not allowed. This is a problem because if all traces from a periodic task release reach firing of a sporadic task, where minimum inter-arrival time requirement is not met, the system will be considered schedulable; based on wrong assumptions. Unfortunately, both solutions presented in Section 13.3 may result in optimistic analysis when the specification of minimum inter-arrival times are not adhered to. An example where SARTS fails in detecting an unschedulable system is presented in Table 15.2. This system contains three threads, one sporadic thread S and 2 periodic threads $P0$ and $P1$. In this system, the minimum inter-arrival time specification of S is ignored by $P0$, and S will be fired every 10 milliseconds, preventing $P1$ to meet its deadline. The time-line of the system is depicted in Figure 15.3, where it can be seen that the deadline of $P1$ will be missed and the system is not schedulable.

In SARTS, the backtracking construction will make the system enter a livelock at the second release of $P0$, because the minimum inter-arrival time of S is has not elapsed. Using the old construction skipping the firing of the sporadic thread when the minimum inter-arrival time requirement is violated, will cause $P1$ to meet the deadline because S will not be fired.

Process	**Period,** T	**WCET,** C	**Deadline,** D
P0	10	5	9
S	20	5	10
P1	20	5	20

Table 15.2: False positive example

The cause of this problem is the system not adhering to the requirements specified, an error, which is not considered in SARTS, where it is assumed that this specification is adhered to. This should be ensured before performing model checking.

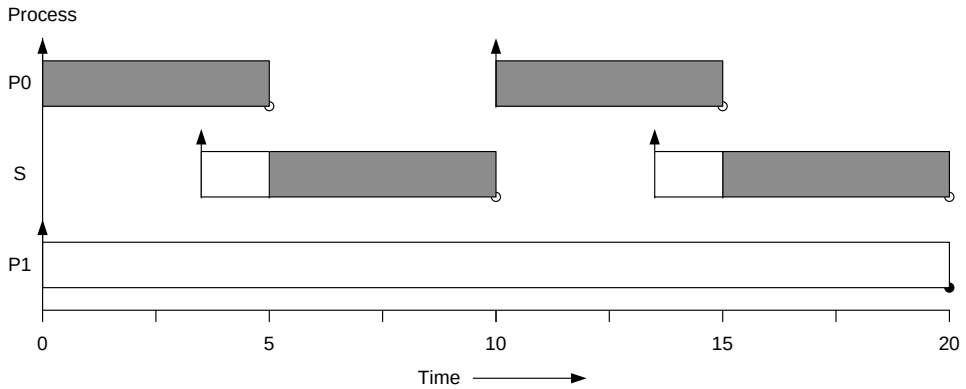


Figure 15.3: Time-line for false positive

15.4 Method Cache

The current implementation does not consider the method cache implemented in JOP. To ensure an over approximation, a method miss is always assumed. This results in a pessimistic analysis, and could be improved by supporting the method cache as it is implemented on JOP, as described in Section 5.2. The introduction of a correct representation of the method cache results in a more accurate analysis, at the cost of verification time. A suggestion to how the method cache could be implemented is presented in this section.

Additional variables and functions must be added to the UPPAAL model, in order to track the state of the method cache. The variables added are shown in Listing 15.4, note that the values are specified in words, which is 4 bytes on JOP.

```
1  const int CACHE_SIZE = 1024;
2  const int BLOCK_COUNT = 16;
3  const int BLOCK_SIZE = CACHE_SIZE / BLOCK_COUNT;
4  const int METHODS = 10;
5  typedef int[0,METHODS] MethodId;
6
7  const int MAXSIZE = CACHE_SIZE;
8  typedef int[1,MAXSIZE] MethodSize;
9  const MethodSize ACTUALSIZE[METHODS] =
10       { 10, 10, 30, 120, 40, 90, 70, 110, 30, 60};
11
12 typedef int[0,BLOCK_COUNT-1] Index;
13 Index current = 0;
14 MethodId cache[BLOCK_COUNT];
```

Listing 15.4: Method cache variables

- **Line 1:** Specifies the size of the cache in words.

- **Line 2:** Specifies how many blocks the cache is divided into.
- **Line 3:** The amount of words which can be stored in each block.
- **Line 4:** The total number of methods in the system.
- **Line 5:** Method identifiers where 0 denotes no method.
- **Line 7:** The maximum amount of words a method can contain.
- **Lines 8-10:** The size of each method in the system, measured in words.
- **Lines 12-14:** An array representing the cache, with the initial index set to 0.

The variables `CACHE_SIZE` and `BLOCK_COUNT` depend on how JOP is configured. In this example JOP is configured with a 4KB cache divided into 16 blocks. The amount of methods and the size of them, specified in `METHODS` and `ACTUALSIZE`, depend on the actual system being analyzed. On method invoke and method return, it must be determined whether the method is present in the cache. A simple linear search is performed through the `cache` to determine whether the specified `MethodId` already exists in the cache. This is performed using the `present` function presented in Listing 15.5.

```
1  bool present(MethodId id) {
2    int c;
3    for (c = 0; c < BLOCK_COUNT; c++)
4      if(cache[c] == id)
5        return true;
6    return false;
7  }
```

Listing 15.5: Determine whether a method is in the cache

If the specified method does not exist in the method cache, it must be fetched from memory, possibly replacing other methods. This is performed using the `replace` function shown in Listing 15.6.

```
1  void replace(MethodId id) {
2    int c = current;
3    int s = (ACTUALSIZE[id-1]+(BLOCK_SIZE-1))/BLOCK_SIZE;
4    cache[c] = id;
5    c++;
6    if (c == BLOCK_COUNT) c = 0;
7    while (--s > 0) {
8      cache[c] = 0;
9      c++;
10     if (c == BLOCK_COUNT) c = 0;
11   }
```

```
12    current = c;
13  }
```

Listing 15.6: Insert the method into the cache

- **Line 2:** The current index in the cache is stored.
- **Line 3:** The amount of blocks the method needs is calculated.
- **Line 4:** The current index is set to the ID of the selected method.
- **Lines 5-11:** The remaining indices, corresponding to the block size of the method, are set to 0.
- **Line 12:** The current index in the cache is updated.

These methods must be used when invoking and returning from methods in the UPPAAL model, to simulate to the actual method cache on JOP.

15.4.1 Method Invoke

The cost of invoking a method is dependent on the method type, the exact formulae for each type are available in [32]. As an example, a private method is invoked with `invokespecial`, and the WCET for this instruction is defined by the formula:

$$74 + r + \begin{cases} r-3 & : \quad r > 3 \\ 0 & : \quad r \leq 3 \end{cases} + \begin{cases} r-2 & : \quad r > 2 \\ 0 & : \quad r \leq 2 \end{cases} + \begin{cases} l-37 & : \quad l > 37 \\ 0 & : \quad l \leq 37 \end{cases}$$

In the version of JOP used in this project, the number of clock cycles used to read a word from memory, r, is 1, and the formula can therefore be reduced to:

$$74 + 1 + \begin{cases} l-37 & : \quad l > 37 \\ 0 & : \quad l \leq 37 \end{cases}$$

As it can be seen, the method cache only improves the performance if the load time, l, is greater than 37. The load time l is defined by:

$$l = \begin{cases} 6 + (n+1)(2 + c_{ws}) & : \quad \text{cache miss} \\ 4 & : \quad \text{cache hit} \end{cases}$$

Where the cache wait state, c_{ws}, is 0 in the version of JOP used in this project, and n is defined by the size of the method in words. The size of the method must be at least 15 words, in order to benefit from the method cache.

If the size of a method is 50 words, the load cost l is 108 on a miss, and the cost of the invoke is therefore 146. This is an increase of 71 clock cycles compared to a hit.

15.4.2 Method Return

The cost of returning from a method is dependent on the return type, the exact formulae for each type are available in [32]. As an example, the cost of returning void is defined by:

$$21 + \begin{cases} r-3 & : \quad r > 3 \\ 0 & : \quad r \leq 3 \end{cases} + \begin{cases} l-9 & : \quad l > 9 \\ 0 & : \quad l \leq 9 \end{cases}$$

The load cost, l, is now related to the size of the method which is returned to. Even a method size of 1 word results in a load cost of 10, and therefore benefits from a cache hit. If the size of the method is 10, the load cost is 28 on a miss, and the cost of the return is therefore 40. This is an increase of 19 clock cycles in case of a miss.

15.4.3 Method Cache Experiment

A small experiment is conducted to show how this improvement would affect the analysis.

The experiment is conducted on different versions of SARTS and the result is compared to WCA and the actual execution on JOP. Listing 15.7 contains a method named `run`. This is a static version of the run method of the sporadic thread in RTSM, introduced in Chapter 6.

```
1  protected static boolean run(){
2    if (state == IDLE){
3      motor.setMotorPercentage(Motor.STATE_FORWARD,
4                               false, 100);
5      state = FORWARD;
6    } else if (state == FORWARD){
7      motor.setMotorPercentage(Motor.STATE_BACKWARD,
8                               false, 100);
9      state = BACKWARD;
10   } else if (state == BACKWARD){
11     motor.setMotorPercentage(Motor.STATE_BRAKE,
12                              false, 100);
13     state = IDLE;
14   }
15   return true;
16 }
```

Listing 15.7: Method cache example

The invoke of `setMotorPercentage` results in further method invocations, and utilizing the method cache implemented on JOP results in a lower execution time, compared to no method cache.

The worst case path is when `state` equals `BACKWARD`, and the last two parameters in `setMotorPercentage` are set to true and a negative value respectively. The results for this method are shown in Table 15.3.

	WCET
JOP	713
WCA	801
SARTS(a)	817
SARTS(b)	801
SARTS(c)	738

Table 15.3: WCET results for run method

The calculation of the WCET of the method has been performed using different modifications of SARTS, referred to as (a), (b), and (c). SARTS (a) is the result of the current implementation, where method cache is not included, and it is therefore a pessimistic result. The current implementation of WCA supports a two block method cache. SARTS (b) is a manually modified version of the translated model, to represent a two block method cache, and it provides the same result as WCA. The final SARTS (c) configuration is a manual modification of the translated model, to represent the actual method cache on JOP. The problem with the experiment performed on JOP is similar to the one described in Section 14.1, where `return true;` is not included, which costs 24 clock cycles. However, there is still a difference of 1 clock cycle. This difference may be because of some pessimism introduced in the formula to calculate the cost of loading in the method into the cache. However several tests indicate that this pessimism is maximum 1 clock cycle, and is introduced because the size of a method is measured in words consisting of four bytes.

This experiment shows how the introduction of a method cache can reduce the pessimism of the WCET analysis, and still be an upper bound of the actual execution cost. However, an important note is that the use of a method cache in WCA might result in an optimistic analysis, because each method is analyzed isolated. When a thread is preempted the contents of the method cache can not be guaranteed once the thread is scheduled again. SARTS includes the entire execution of the system, and it is therefore possible to know the exact content of the method cache at a given state in the system, and it will therefore not result in an optimistic analysis, which might be the case with WCA.

15.5 Scheduler

In the current implementation, the scheduler cost is not considered and is set to the cost of one clock cycle in the Uppaal model. This should in a final implementation be parameterized with the actual cost of running the scheduler.

The scheduler cost can be expressed by the formula:

$$schedulercost = S_{in} + S_{out} + C(N)$$

where

- N is the number of threads in the system.
- S_{in} is the cost of switching to the scheduler, and save the context of the currently executing thread.
- S_{out} is the cost of switching from the scheduler to the selected thread, and restoring its context.
- $C(N)$ is the scheduler cost given the number of threads in the system.

By analyzing the system, the number of threads in the system is known. The scheduler contains loops dependent on the number of threads in the system. These loop bounds can be derived from this thread count, and thus the worst case execution time for the scheduler, $C(N)$ can be calculated.

S_{in} and S_{out} depend on the current state of the system; the content of the method cache, the current stack size, and the thread switched from and to. The method cache and the stack is currently not implemented in SARTS, and the scheduler cost is therefore not possible to incorporate in the analysis. How the method cache can be implemented is described in Section 15.4. The size of the stack can be calculated by analyzing push and pop instructions and tracking the call graph.

Adding scheduler cost to the UPPAAL model will make the analysis more accurate.

15.6 Soot

An improvement to SARTS would be to incorporate Soot into the generation of the intermediate representation and thereby enable further analysis of the system being translated, e.g. type inference, as discussed in Section 15.1.1.

Using Soot in this manner poses two problems: mapping the intermediate representations of Soot to SIR, and mapping the analysis results into the model checker. Mapping Soot to SIR can be done in two ways, reimplementing the translator to use Soot, and creating the map directly, or implementing a mapper which traverses the Soot intermediate representations and SIR to create a map. The first solution would be the better since it is cleaner, however SARTS is not implemented using Soot as described in Section 8.5, so the second solution would be viable as well. Using Soot directly also makes it possible to use Soot Java Bytecode optimizations on the system before it is uploaded to the hardware platform. This would improve the execution time of the system, since the underlying hardware probably

has no JIT compiler performing optimizations. Since this might simplify the system it might also lower the time required to perform verification.

15.7 Collapsing

In the current implementation all branches and loops from the Java code are present in the UPPAAL model. This is done to maintain the control flow of the actual application. This, however, might be unnecessary. Only the areas of the control flow which contain instructions which are not local are actually needed. A local instruction is an instruction which does not affect other tasks, or the system overall. A branch where only local instructions exist in both paths could be collapsed to reduce the state space of the model. This could be done by calculating the worst case path through the branch and creating a single basic block, requiring a WCET equal to that worst case. This way the state space could be significantly reduced, since all areas of the system which do not affect other tasks can be collapsed into a single block. The solution suggested is problematic as described in Section 16.1, where a blocking region can be moved past a point where it would have prevented a higher priority thread from executing. A way to circumvent this problem is to perform changes in both the model and the program itself, i.e. the Java class file, by padding the cheapest branch, adding execution time. As an example, consider a simple branching if-statement. The cheapest of the two branches, cheapest in terms of execution time, could be padded with `nop` instructions with the execution time of 1 clock cycle, such that the branch is execution time symmetric. Since selecting either branch is of no significance to the execution time, this branch can be collapsed into one block.

This technique will add execution time, but the WCET is preserved. However this is not a problem for the overall system, since average execution time is not important. This technique could also be extended, by the developer being able to choose where to pad branches, inside an IDE, e.g. Eclipse. In the case of loops it may be possible to apply a dynamic padding such that a loop with upper iteration bound of 1000 could only terminate at 10 different execution times. This could be done by querying the number of cycles used by a thread before and after a loop.

15.8 Summary

Several improvements have been proposed. Preliminary tests indicate the benefits of these improvements.

Some improvements increase the accuracy of the verification, albeit at an increased verification time. Whether the gained accuracy is worth a slower verification time needs further research to determine. Researching and implementing these improvements is considered future work.

The next chapter decribes problems and issues with the presented approach.

Chapter 16

Reflections

This chapter describes a problem with the model-based schedulability approach used in SARTS, and issues with Java as a programming language for hard real-time systems.

16.1 WCET and Blocking

While model-based schedulability seems like an attractive approach, there are some problems when blocking is introduced. The problem is that assuming the worst case execution path can postpone a blocking region which would have affected a higher priority thread. An example of an actual execution is depicted in Figure 16.1. It can be seen that the lower priority thread enters a blocking region which prevents the higher priority thread from executing, causing a missed deadline. The execution path verified by the model checker is depicted in Figure 16.2. Here the lower priority thread has a longer execution time prior to entering the blocking region, allowing the higher priority thread to preempt it. This execution path will not result in a missed deadline.

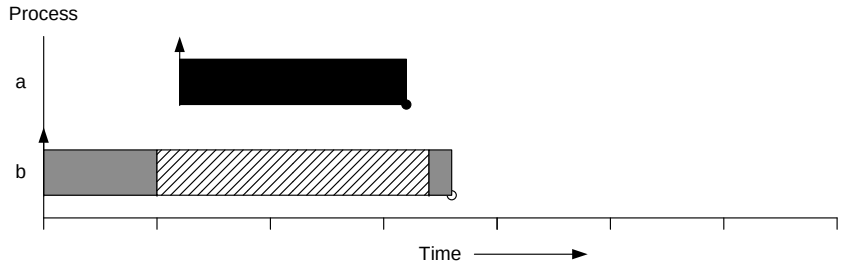


Figure 16.1: Actual execution

Unless this problem is solved, the model-based schedulability approach is not sound. A safe approach to solving this problem is to verify every possible

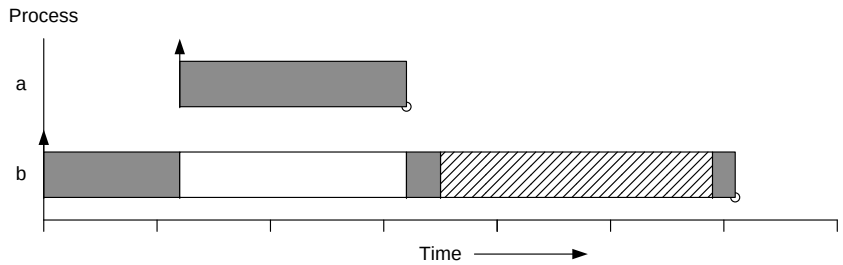


Figure 16.2: Execution verified

path generated by an execution time in the discrete interval between best and worst case execution time.

This problem disappears if the best and worst case always is the same, which in return results in a very deterministic model, thus reducing the verification time needed. This can be achieved by using the technique described in Section 15.7.

Considering these extra paths in the model will significantly increase the complexity of verifying the model. It is believed that the complexity can be reduced by using approaches such as collapsing, but applying this will require further research.

16.2 Programming Language

This section discusses the use of the Java programming language for embedded real-time systems, both from the perspective of the analysis developed, but also from the general perspective of using Java for embedded real-time systems.

Using Java in real-time embedded systems poses some challenges due to the design of the language. This includes the heavy use of the heap and thus the garbage collector, but also the ability to add meta data, such as loop annotations.

16.2.1 Language Usage

Among reasons for using Java for real-time embedded systems is the great number of developers who learn Java as their first language. Although this is true, some characteristics of the Java language and embedded systems may require changes in the way programs are written.

16.2.1.1 Performance

Programming for performance is one way of lowering the cost of a system. Lower performance requirements allows cheaper hardware and thus lower

production costs e.g. when a company produces millions of devices, cutting off a few cents on each item can be considered significant.

In general, coding for efficiency means that the developer should not rely on clever optimizing performed by the compiler, since many optimizations in Java are difficult to safely perform [36]. An example, presented in [36], illustrates simple optimizations, can easily be performed by the developer.

```
1  for(int i=0; i<size()*2; i++){...}
```

Listing 16.1: Code example

In this example, `i<size()*2` is evaluated for every iteration of the loop. This can be optimized manually by the developer, by storing the value `size()*2` evaluates to. This optimization is platform independent, and can thus be used regardless of the underlying hardware. This example assumes that `size()` has no side effects and returns a constant value.

In Java, methods are virtual by default, which gives the worst execution time of method invokes. Optimizations can be applied by declaring methods `private`, `final` or `static` which will remove the need for dynamic dispatching.

Besides knowing what general optimizations to apply, knowledge about the target device could help the developer optimize the code. Using JOP, some instructions are not implemented in microcode directly, but are implemented as static Java methods. These instructions, in addition to not being supported by hardware have the overhead of requiring a method call. This means that using e.g. floats or longs may be surprisingly much slower than expected, and also have an impact on the method cache.

The choice of compiler can greatly affect the execution time. The general philosophy in Java is to leave all optimizations to the JIT compiler. In hard real-time systems, such a compiler is not available, and static bytecode optimizations need to be made by the bytecode generator.

16.2.1.2 Libraries

One of the advantages of Java is the large set of library functions available. Unfortunately, when developing real-time embedded systems and predictability is crucial, standard Java library cannot be used. Instead, a predictable set of library functions should be provided, which is both analyzable and thoroughly documented such that the developer can choose the functionality based on knowledge about how the actual parameters influence the execution time. The library should be analyzable, which in many contexts means that the source code needs to be available, for correct annotations. An example of a real-time library for Java is the Canteen library [15].

16.2.2 Missing Features

The following sections describe some features which could improve Java in regards to being a better real-time language.

16.2.2.1 Memory Features

In Java, non-primitive types are allocated on the heap. This may waste resources in systems with limited resources, since these objects must later be garbage collected. In situations where objects are only created in order to aggregate or abstract over information passed to a method, e.g. Relative time, as proposed in Section 4.2.2, there is no need to allocate this on the heap, since the object is not referenced outside the scope of this method. Alternatively, the `RelativeTime` object could be replaced by the primitive stack allocated type `long` representing relative time, at the loss of abstraction; also, this would not work if relative time contained more than one primitive type.

Stack allocated objects are required in order to achieve the abstraction along with the ability to control how memory is allocated i.e. when to use stack and when to use heap. This is done in languages like C#, by introducing `struct`, which has the same characteristics as a class except that it is stack allocated and thus a value type, not a reference type. The high-performance language X10, developed by IBM, has value classes, which can be used to the same effect as stack allocated structures [12]. When variables of value types are assigned, the data is copied as opposed to copying only the reference as done in the assignment of reference types. This means that structs are de-allocated when the scope they exist in is destroyed, as primitive types.

Introducing structs in the Java languages should pose no problem as long as structs have constant size, since the stack height must be known statically.

16.2.2.2 Annotations

Annotations were added to the Java language in Java 5.0 [35]. Using annotations the developer can attach meta-data to the program being developed; data that is compiled into the class files and can even be retrieved at runtime using reflection. Standard annotations such as `@Override` or `@Deprecated` are available, but also custom annotations can be created by the developer. Java allows annotation of declarations, such as *constructors* and *fields*, but not *statements*, such as while loops.

Several analysis tools require annotation of loop statements in the program, namely the upper bound of loop iterations. Since annotations provided by the Java language cannot be applied to statements, these annotations are written as comments, which are ignored by the compiler, requiring

the annotated source code while analyzing the program.

Allowing annotations on loops would ease the development of tools, such as SARTS, giving them access to loop annotations without requiring the source code.

16.2.3 Summary

In this section, the use of Java for real-time embedded systems has been discussed along with some improvements which would ease the development of tools. The need for real-time libraries with predictable properties is also required to further ease the development of such systems in Java. Although many programmers are introduced to Java as their first and maybe only programming language, the transition into developing real-time embedded systems does not happen without further education. The development of profiles like SCJ2 is an attempt to make Java more suitable for real-time development, but it is still limited by the language. Features like structs and fine grained annotations cannot be added through a profile, but need to be a part of the language.

Chapter 17

Conclusion

An automated model-based schedulability analysis of hard real-time systems has been presented and implemented in this project.

This has resulted in a prototype application, SARTS, which automatically translates a real-time system implemented in Java to an abstract time preserving UPPAAL model. Verification can then be performed on this model, in order to determine whether the system is schedulable. This translation is an abstraction of the Java code, including the actual bytecode, in order to determine the WCET of the implementation, which is used in the schedulability analysis.

The automatic translation from Java to UPPAAL ensures the correspondence between the actual implementation, and the model being verified. However, further work needs to be conducted in order to prove that this translation is correct. This automatic translation also allows the developer to abstract away from the actual verification process and no knowledge of model checking is required. However, the developer still has to annotate loop bounds, which is a source of errors.

The state space explosion when performing model checking is a known problem, and limits the ability to use model checking for complex systems. The scalability of SARTS has been examined, and it does not scale very well; even small changes in the implementation can cause dramatic change in verification time. However, several optimizations have been implemented, and it has been shown how it significantly reduces the verification process. Further improvements have been discussed in order to increase the accuracy and scalability of SARTS.

Several experiments have been conducted, in order to compare SARTS to existing tools, and the actual execution on JOP. It has been shown that SARTS is also capable of performing WCET analysis comparable with tools such as WCA. Furthermore it has been shown how the use of a model-based approach can lead to a more accurate result than traditional approaches to schedulability analysis; achieving the goal of our thesis.

SARTS, as a proof of concept application has been developed and proves as an interesting direction for schedulability analysis. For this to be a complete tool, much work still needs to be done to make the analysis sound and to improve performance. Furthermore, the usability of SARTS can greatly be increased by incorporating it into a development environement, e.g. as a plug-in to Eclipse, in order to provide a natural link between the development and the verification process.

Chapter 18

Future Work

This chapter discusses some of the future work directions of SARTS, including a lot of different research areas. Some of the functionality which can improve the accuracy and the performance of SARTS have been discussed in Chapter 15.

18.1 Multicore

Research is currently being conducted in order to develop a multicore version of JOP [26]. This multicore version is still intended for hard real-time systems, and predictability is therefore necessary. The functionality of SARTS must be extended to support the additional features of a multicore hardware architecture. If the underlying hardware is predictable, it should be possible to incorporate this feature in SARTS. This could probably be done by introducing additional schedulers, representing the used scheduling strategy.

18.2 Additional Scheduling Strategies

The current implementation of SARTS assumes a deadline monotonic scheduling strategy is used. It would be desirable to be able to switch the scheduling strategy, e.g. to a EDF or VBS strategy.

18.3 Hardware Interrupts

It is currently only possible to invoke sporadic tasks through periodic tasks. Hardware interrupts can therefore only be handled by having a periodic task poll for an interrupt and then fire a sporadic event. This is also a limitation on JOP where a polling technique must be used. However, it should be possible to model hardware interrupts as sporadic tasks, provided a minimum inter-arrival time is present and obeyed.

18.4 Change Underlying Hardware

It should be possible to change the underlying hardware, so the analysis is not bound to JOP. This should be a rather straight forward procedure, providing the new hardware is predictable regarding execution time. The WCET for each bytecode must be known and updated. Additional JOP specific features must be disabled, e.g. the method cache. If the new hardware has any performance increasing features, this must also be added to the translation. JOP can be clocked at different frequencies. The default is 60 Mhz, and the maximum is 100 Mhz, changing clock frequency requires only small modifications of SARTS.

18.5 User Feedback

The feedback from SARTS is very limited, e.g. when performing a schedulability analysis, it returns yes or no. If the system is schedulable, it would be desirable to know the lower limit of processor clock frequency while still being schedulable, in order to reduce production cost. If the system is not schedulable, it would be desirable to know what the clock frequency must be, in order to determine whether to buy a faster CPU or reduce the complexity of the system. It would also be desirable to know exactly which tasks miss their deadline, beneficial when performing optimizations of the system, before a new analysis is performed. When performing a WCET analysis of a specific method, it is only possible to verify whether the WCET is higher than X and lower than Y. It would be desirable to get the WCET value of each method in order to reduce execution time in case a system is deemed not schedulable.

A plug-in to the IDE, e.g. Eclipse, would also increase the usability of SARTS, where the actual translation and model checking is hidden for the user, and only the result is returned. The IDE could highlight the worst case path in the system, furthermore it should be possible to see the WCET for each statement in the IDE. Furthermore adding collapsing functionality into the IDE would provide the developer with a tool to make the code more deterministic.

18.6 Theoretical Work

The decidability of SARTS is discussed in Section 11.6.1, where it is shown how the model can be translated to a timed automata without stopwatches, thus the problem is decidable. However, further work must be conducted in order to prove that the translation from Java to Uppaal is correct. The current implementation is not sound, because not all paths are examined, i.e. when a loop is exited before its bound is met. This can result in moving

the occurrence of a blocking region, resulting in an optimistic analysis. This problem is described in Section 16.1. These issues must be solved before SARTS can be used to provide a reliable verification of industrial products.

18.7 Scalability Improvements

Scalability is a well known problem when using model checking, and SARTS is no exception. However, several possible improvements have been discussed, and further research needs to be conducted in order to determine how these affect scalability. An interesting approach is to improve the determinism in the implementation as discussed in Section 15.7.

An important note is to use the optimizations already available in UPPAAL, such as convex-hull approximation; if the system is said to be schedulable using convex-hull approximation it is guaranteed to be schedulable. Verification time is also reduced by using more standard optimizations provided in UPPAAL, such as aggressive state space reduction.

Chapter 19

Bibliography

[1] Yasmina Abdeddaim and Oded Maler. Preemptive job-shop scheduling using stopwatch automata. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 113–126, 2002.

[2] aJile Systems. *aJile Systems, Inc. - Home*. `http://www.ajile.com/`, 2007. Online: 29/10-2007.

[3] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. *Times: a tool for schedulability analysis and code generation of real-time systems*. `citeseer.ist.psu.edu/amnell03times.html`, 2003.

[4] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[5] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal*. `http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf`, 2004. Online: 6/12-2007.

[6] Thomas Bøgholm, Henrik Kragh-Hansen, and Petur Olsen. *Real-Time Java*. `https://services.cs.aau.dk/public/tools/library/details.php?id=1199704154`, 2008. Online: 10/1-2008.

[7] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley Longmain, 2001.

[8] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[9] Greg Bollella et al. *The Real-Time Specification for Java*. `https://rtsj.dev.java.net/rtsj-V1.0.pdf`, 2000. Online: 19/9-2007.

[10] Patrick Lam et al. *Soot: a Java Optimization Framework*. `http://www.sable.mcgill.ca/soot/`, 2007. Online: 23/10-2007.

[11] Thomas H. Cormen et al. *Introduction to Algorithms*. McGraw-Hill, 2001.

[12] Vijays Araswat et al. *Report on the Experimental Language X10*. `http://x10.sourceforge.net/docs/x10-101.pdf`, 2006. Online: 6/6-2008.

[13] The Eclipse Foundation. *The AspectJ Project*. `http://www.eclipse.org/aspectj/`, 2008. Online: 2/5-2008.

[14] Trevor Harmon. *Volta*. `http://volta.sourceforge.net/`, 2007. Online: 12/12-2007.

[15] Trevor Harmon, Martin Schoeberl, Raimund Kirner, and Raymond Klefstad. Toward libraries for real-time java. *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 458–462, May 2008.

[16] Laurie Hendren. *Uses of the Soot Framework*. `http://www.sable.mcgill.ca/%7Ehendren/sootusers.pdf`, 2006. Online: 12/5-2008.

[17] James J. Hunt, Fridtjof B. Siebert, Peter H. Schmitt, and Isabel Tonin. Provably correct loops bounds for realtime java programs. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 162–169, New York, NY, USA, 2006. ACM.

[18] IRISA. *Signal/Polychrony*. `http://www.irisa.fr/espresso/Polychrony/`. Online: 14/4-2008.

[19] java.net. *Source Code Analysis Using Java 6 APIs*. `http://today.java.net/pub/a/today/2008/04/10/source-code-analysis-using-java-6-compiler-apis.html`, 2008. Online: 10/4-2008.

[20] Jagun Kwon, Andy Wellings, and Steve King. *Ravenscar-Java: a high integrity profile for real-time Java*. `http://www.cs.york.ac.uk/ftpdir/reports/YCS-2002-342.pdf`, 2002. Online: 30/10-2007.

[21] Santos Laboratory. *About Bandera*. `http://bandera.projects.cis.ksu.edu/`, 2005. Online: 23/10-2007.

[22] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[23] Peter Mehlitz, Dimitra Giannakopoulou, Corina Pasareanu, and Masoud Mansouri-Samani. *Java PathFinder*. `http://javapathfinder.sourceforge.net/`. Online: 3/12-2007.

[24] University of Maryland. *FindBugs*. `http://findbugs.sourceforge.net/`, 2008. Online: 2/5-2008.

[25] E.-R. Olderog and H. Dierks. Moby/rt: A tool for specification and verification of real-time systems. *j-jucs*, 9(2):88–105, 2003. `http://www.jucs.org/jucs_9_2/mobyrt_a_tool_for`.

[26] Christof Pitter and Martin Schoeberl. Towards a java multiprocessor. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 144–151, New York, NY, USA, 2007. ACM.

[27] The Java Community Process(SM) Program. *JSR 302: Safety Critical Java^{TM} Technology*. `http://jcp.org/en/jsr/detail?id=302`, 2005. Online: 7/11-2007.

[28] RTCA. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*. `http://www.rtca.org/downloads/ListofAvailableDocs_WEB_OCT%202007.htm#_Toc180489517`, 1992. Online: 7/11-2007.

[29] M. Schoeberl. Real-time garbage collection for java. *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, pages 9 pp.–, 24-26 April 2006.

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

[31] Martin Schoeberl. *JOP: A Tiny Java Processor Core for FPGA*. `http://jopdesign.com/`, 2007. Online: 29/10-2007.

[32] Martin Schoeberl. *JOP Reference Handbook*. `http://jopdesign.com/doc/handbook.pdf`, 2007. Online: 21/2-2008.

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

[34] Martin Schoeberl, Hans Søndergaard, Bent Thomsen, and Anders P. Ravn. A profile for safety critical java. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 94–101, Washington, DC, USA, 2007. IEEE Computer Society.

[35] Peter Sestoft. *Java Precisely*. MIT Press, Cambridge, MA, USA, 2002.

[36] Peter Sestoft. *Java Performance. Reducing time and space consumption.* `http://www.itu.dk/people/sestoft/papers/performance.pdf`, 2005. Online: 28/5-2008.

[37] DARTS IT Dept CS Uppsala University Sweden. *TimesTool*. `http://timestool.com/`, 2007. Online: 22/10-2007.

[38] Jean-Pierre Talpin, Abdoulaye Gamati´e, David Berner, Bruno Le Dez, and Paul Le Guernic. *Hard real-time implementation of embedded software in JAVA*. `http://www.irisa.fr/prive/talpin/papers/fidji03.pdf`, 2003. Online: 14/4-2008.

[39] Uppsala University and Aalborg University. *UPPAAL*. `http://www.uppaal.com/`, 2006. Online: 28/5-2007.

[40] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. *Lecture Notes in Computer Science*, 2072:99–117, 2001.

[41] Andy Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, Ltd, 2004.

Part IV

Appendices

Appendix A

Acronyms

BCEL	Byte Code Engineering Library.
CFG	control flow graph.
EDF	Earliest Deadline First.
FPS	Fixed-Priority Scheduling.
FSM	finite state machine.
JML	Java Modeling Language.
JOP	Java Optimized Processor.
JVM	Java Virtual Machine.
LCM	least common multipe.
RTSJ	Real-Time Specification for Java.
RTSM	Real-Time Sorting Machine.
SARTS	Schedulability Analyzer for Real-Time Systems.
SCJ	Safety Critical Java.
SIR	SARTS Intermediate Representation.
VBS	Value-Based Scheduling.
WCA	WCET Analyzer.
WCET	Worst Case Execution Time.
WCRT	Worst Case Response Time.

Appendix B

UPPAAL

UPPAAL is a verification tool developed in co-operation between Uppsala University and Aalborg University [39]. This appendix provides an introduction to parts of the syntax used in UPPAAL. For a more throughout description see [5]. This is only the most basic of UPPAAL in order to ease the understanding of the models presented in this report.

A small example is used to describe some of the terminology used in UPPAAL, when creating a model. The example illustrates a simple buffer, which allows add and remove instructions, and a couple of clients requesting to add or remove from the buffer. The example is depicted in Figure B.1, containing both the client and buffer model.

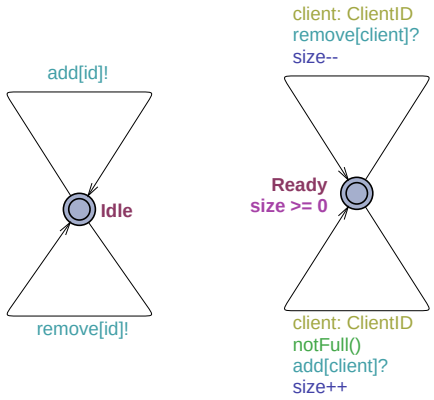


Figure B.1: (a) A simple client communicating with (b) a simple buffer.

In general a UPPAAL model contains a set of finite state machines (FSMs), called *templates*. The buffer and the client are templates. These templates represent parts of the system, and consist of two basic elements, states and

transitions. A state is represented by a circle, and a transition is a directed edge between two states. Each state can have a unique name attached, e.g. the buffer has a state called `Ready` and the client a state called `Idle`. The name must only be unique for the given template. Each state can have an *invariant*, a Boolean expression which must always evaluate to true when the FSM is in this state. For example `size >= 0`, while the buffer is in the `Ready` state, this ensures that the buffer can't be dequeued when it is empty.

Transitions connect states, and they can loop to the same state. A transition can have four attributes:

- **Select:** Non-deterministically selects a value from a type and assigns it to a variable. The syntax is: `<Variable Name> : <Type>`, e.g. the buffer uses `client: ClientID` to represent the client requesting an add or remove. `ClientID` is defined as an integer with a specific range representing the number of clients.
- **Guard:** A Boolean expression which must evaluate to true for the FSM to be able to follow the transition, e.g. `notFull()` prevents the clients from adding more elements if the buffer is full. Here `notFull()` is a custom developed method returning *true* if the buffer is not full, and *false* otherwise. This could also be guaranteed using an invariant stating `size <= MaxSize`. Note even though the guard is fulfilled, the transaction cannot be performed if it violates the invariant of the target node.
- **Sync:** Sends or receives on a channel, which is shared among the machines. An "!" represents send/signal and a "?" represents receive/listen, e.g. `remove[client]?` receives a remove request from a specific client.
- **Update:** A comma separated list of variable assignments, using normal assignment operators, e.g. `size++` increments the size of the buffer.

To ease the readability of the model, the attributes are color-coded by type and attributes should be placed near the corresponding state or transition.

The states in the template can be normal, urgent, or committed. A simple example depicted in Figure B.2 is used to describe the different states. Three different templates are shown, called `P0`, `P1`, and `P2` representing three isolated processes, each containing a local clock `x`. Time is represented by clocks in UPPAAL, which are constantly increasing and represented by natural numbers. A template must contain exactly one initial state, which is the starting point, represented by a small circle inside the state. Each of the processes has the initial state set to `S0`, the clock `x` is then reset on the transition to state `S1`, which is normal in `P0`, urgent in `P1`, and committed

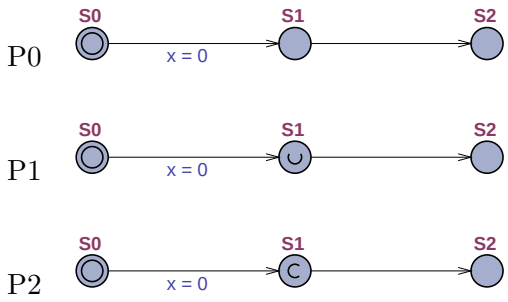


Figure B.2: Three different automata with a local clock [5].

in `P2`. An urgent state is represented by a small "U" and committed by a small "C" inside the state. The following describes how the difference in the `S1` state affects the different processes.

- **Normal:** Time can elapse in this state, and the value of the clock `x` in state `S2` cannot be ensured.
- **Urgent:** The time is frozen during an urgent state. The value of the clock `x` is therefore ensured to also be 0 when entering `S2`. An urgent state is semantically equivalent to resetting a designated clock, `y`, on all incoming edges and add the invariant `y<=0` to the state, ensuring it to leave before time elapses.
- **Committed:** Similar to urgent states, time is frozen. Furthermore, if a template is in a committed state, the next transaction must leave this state, i.e. when `P2` is in `S1` the only possible transition is to `S2`, independent of other states in the system. Note, if `P1` is in `S1`, then `P0` is able to take a transition, but if `P2` is in `S1`, then `P0` cannot take a transition.

At first glance the difference between urgent and committed might not seem that obvious, but the strength of committed is the ability to create an atomic sequence. When different templates need to exchange values, this must be done through public variables, e.g. the buffer from the first example might be extended to return the dequeued value to the client. If several clients can communicate with several buffers, and the public variable is shared between the buffers, can result in a race condition, the committed state ensure that this is not possible. This concludes the description of Uppaal.

Appendix C

Limitations

The current implementation of SARTS has the following limitations.

Packages. A package must be defined for the system being verified for SARTS to translate the system correctly.

Anonymous classes. It is not possible to use anonymous classes, e.g. implement a periodic thread in the main method.

Release parameters. When instantiating release parameters, the values must be literals, i.e. it is not possible to use variables. Also, the instantiation of release parameters must be performed as a parameter to the task constructor, i.e.
`new SporadicThreadImpl(new SporadicParameters(1,4,2));`

Sporadic fire. When using `RealtimeSystem.fire(int)` the value must be a literal, similar to release parameters.

Switch case. Switch statements are not supported, and will cause errors.

Recursion. Recursion is not allowed in SCJ2, but it is not detected by SARTS. If recursion is used, the current implementation always determines the system to not be schedulable.

Java libraries. Java libraries are not allowed in SCJ2, using these will cause errors.

Interfaces. Interfaces are not supported, using interfaces will cause errors.