UNIVERSITY OF TWENTE.

Testing real-time requirements for integrated systems

Master Thesis

R.S. Peterson

Committee:
Dr. ir. A. de Keijzer
Dr. H. Sözer
Dr. I.K. Kurtev

Enschede, August 19, 2010

Abstract

Problem Integrating two or more hard real-time devices while assuring real-time requirements is a complex task. Model-based techniques exist for assisting developers with validating real-time requirements. Hereby, a developer specifies (expected) execution timings, such as the Worst Case Execution Time (WCET). These models serve as input for model checkers which use the specified timings to validate the real-time requirements. During literature research no methods/approaches to test and assure the correctness of the specified timings were found. This research aims to close this gap.

Solution Initially, a new Domain-Specific Language (DSL) was developed to model the system. This had great resemblance with Architecture Analysis & Design Language (AADL). Because AADL is an industry standard and existing tools can be reused, AADL was chosen to model the system to be developed. AADL is extended with the ability to specify monitors, which monitor execution scenarios. The developer can implement his own monitor listeners and specify what to do on execution scenario start, finish, strict violation and deadline miss. A code generator is built which transforms an instance of the extended AADL model into source code which cooperates with the created framework. The framework provides an implementation for the created AADL extension and the essential AADL parts for modelling a case-study.

Evaluation The feasibility of the proposed solution is demonstrated by the integration of two medical devices. The goal of this integration is to combine a haptic device with a neuronavigation system. The integration has a real-time requirement; the haptic device must be updated at least 1000 times a second. Where previous ad-hoc attempts have failed to meet this requirement, using the proposed solution this requirement was met, and tested using the provided monitors. The case-study is used to measure the overhead introduced. Each monitor without considering deadlines introduces on average 2.63μs overhead per execution of a method in its execution scenario using the test set-up. Performing a deadline check takes 1μs for each deadline to be checked on the test set-up. The overhead for checking deadlines scales linear or quadratic with respect to the number of deadlines and deadline period, depending on the deadline period algorithm used.

Contents

Abstract i

Acknowledgements viii

I Setting the Context 1

1 Introduction 2
- 1.1 Context . . . 2
- 1.2 Problem statement . . . 3
- 1.3 Approach . . . 3
- 1.4 Contributions . . . 4
- 1.5 Outline . . . 5

2 Case study: Medical devices 7
- 2.1 The integrated system . . . 7
- 2.2 Neuronavigation . . . 9
  - 2.2.1 VectorVision . . . 10
- 2.3 Haptic devices . . . 11
  - 2.3.1 Passive haptic device . . . 11
  - 2.3.2 Active haptic device . . . 11
  - 2.3.3 Freedom6S . . . 12
- 2.4 Challenges . . . 12
  - 2.4.1 Brain shift . . . 12
  - 2.4.2 Missing a deadline . . . 13
  - 2.4.3 Frequency mismatch . . . 13
- 2.5 Previous work regarding the case study . . . 13

II Technological background and related work 15

3 Real-Time Safety-critical Systems 16
- 3.1 What is real time . . . 16
- 3.2 Safety-critical systems . . . 17

3.3 Performance analysis . . . 17
3.3.1 Static Analysis . . . 17
3.3.2 Dynamic Analysis . . . 18

4 Domain-specific languages 19
4.1 What is a domain specific language? . . . 19
4.2 External & Internal DSLs . . . 20
4.3 Advantages and disadvantages . . . 20
4.4 When & how to create a DSL . . . 21
4.5 Which real-time oriented DSLs exists? . . . 23
4.5.1 Functional DSLs . . . 23
4.5.2 Model based DSLs . . . 24
4.5.3 Miscellaneous . . . 26
4.6 Architecture Analysis & Design Language . . . 28
4.6.1 Software components . . . 28
4.6.2 Properties . . . 28
4.6.3 Analysis of AADL models . . . 29

III Solution design & implementation 31

5 DSL design 32
5.1 Solution technology . . . 32
5.2 Analysis of the domain . . . 33
5.2.1 Defining the problem domain . . . 33
5.2.2 Domain analysis . . . 33
5.3 Designing the DSL . . . 35
5.4 Implementing the DSL . . . 36
5.5 Comparing the DSL with AADL . . . 37
5.6 Conclusion . . . 39

6 Integration Framework 40
6.1 Framework design . . . 40
6.2 Linking the framework & AADL . . . 42
6.3 Conclusion . . . 43

7 Performance Monitors 44
7.1 Design . . . 44
7.2 Semantics . . . 46
7.2.1 Statechart design . . . 48
7.2.2 Examples . . . 49
7.3 Implementation decisions . . . 54
7.3.1 Multiple threads on a single method . . . 54
7.3.2 How to monitor deadlines . . . 54

7.3.3 Where to determine the time 55
7.4 Monitors & AADL 55
7.5 Conclusion 57

8 Case study: Integrating medical devices **58**
8.1 Design 58
8.2 Implementation 60
8.3 Results 60
8.4 Conclusion 61

IV Evaluation & Conclusions **63**

9 Evaluation **64**
9.1 Framework 64
9.1.1 Test set-up 64
9.1.2 Overhead and scalability tests 66
9.1.3 Timer jitter 71
9.1.4 Reusability of the framework 72
9.1.5 Reusability of formal analysis tools 72
9.2 Case study 73
9.2.1 Performance evaluation 73
9.2.2 Prerequisites for real-world usage 73
9.3 Conclusion 74

10 Conclusions **76**
10.1 Answers to research questions 77

11 Recommendations & future work **79**

Bibliography **80**

V Appendixes **85**

A Acronyms **86**

B Integration framework generated code **88**

C Acceleo template **89**

D Process method call pseudo code **94**

List of Figures

1.1 The overall approach for testing real-time requirements 5
1.2 The main chapters of the thesis and the recommended reading order 6

2.1 Navigation: Vector Vision 8
2.2 Haptic device: Freedom6S 8
2.3 Hardware architecture 8
2.4 Stereotaxy taxonomy 10
2.5 Passive haptic device 11
2.6 Active haptic device 11

4.1 Hume layers 24
4.2 Meta-model hierarchy 25
4.3 MARTE architecture 26
4.4 Cheddar screenshot 27

5.1 Possible representation of the case study 34
5.2 DSL Meta-model 36
5.3 DSL Data types 37
5.4 DSL properties 38

6.1 High-level integration framework 41
6.2 Detailed design integration framework 42

7.1 Class diagram of the performance monitor design 47
7.2 A prototype statechart of a monitor having an execution scenarios consisting of 1 or 2 methods 50
7.3 A prototype statechart of a monitor having an execution scenario consisting of 3 methods 51
7.4 Deadline monitor prototype 52
7.5 Execution scenario of single call to method A 52
7.6 Execution scenario of monitor tracking two consecutive executions of A 53
7.7 AADL extension to support monitor 56

8.1 Class diagram of the case study 59
8.2 Execution timings Freedom6S.getHandleLocationAndSetForce() . . . 62

8.3 Execution timings VectorVision.getPatientAndF6sLocation() 62
8.4 Execution timings of A VectorVision call followed by a Freedom6S call 62

9.1 Timings without deadline monitor 66
9.2 Execution timings with deadline periods, 33μs, 50μs, 100μs, 200μs (from top to bottom) 68
9.3 Timer jitter with deadline and without deadline monitor 71

List of Tables

4.1 Caper Jones languages 20
4.2 When to create a DSL taken from Marnik et al. [27] 22

8.1 Software configuration of case study 61
8.2 Hardware configuration of case study 61

9.1 Additional overhead introduced per monitor for different deadline periods and total monitor numbers, using 1 thread 69
9.2 Expected execution times and experienced execution times 70

Acknowledgements

This thesis has been written to describe the results of the final project I have carried out for the Master of Science degree in Computer Science at the University of Twente.

Firstly, I would like to thank my supervisors, Ander de Keijzer, Hasan Sözer and Ivan Kurtev. Our discussions and your encouragements have definitely increased the quality of the results of the research and this report.

Next, I would like to thank Michiel & Gido for their critical questions and remarks regarding the implementation and the report. Your suggestions really have improved the quality of my work.

I would also like to thank Chris & Peter for reviewing my thesis and making sure that this report is (hopefully) understandable for an English speaker.

Also I would like to thank my roommates for providing distractions when needed (and also when not needed ;)).

Last, but certainly not least, I would like to thank my girlfriend Tina, for being there and supporting me even when I was working in the evening and weekend.

Ronald

Part I

Setting the Context

Chapter 1

Introduction

This chapter starts with a short introduction into the problem domain. The chapter then continues with the context, problem statement, approach, contributions and the outline for the remainder of this thesis.

Developing real-time integrated systems and validating real-time requirements is a complex task. An integrated system consists of multiple cooperating hard real-time devices. The software for communication between and control of the devices needs to be developed and validation of the real-time requirements is essential. Developing such a system is difficult due to the fact that each device has its own real-time requirements and the integrated system might introduce new real-time requirements. Add up the ever growing size and complexity of an average integrated system and it should be clear that assuring hard real-time requirements in such a system is not an easy task.

Means to model integrated systems exist to support developers with assuring hard real-time requirements. Within such a model the developer specifies (expected) execution timings, like the Worst Case Execution Time (WCET). These models are used as input for model checkers. Model checkers use the specified execution timings to validate the real-time requirements [12, 24]. This research aims to support the developer by creating tool support for gathering empirical data to test the assumptions made in the model.

1.1 Context

This research is worked out in collaboration between the Software Engineering research group and the Medical Infolab of the University of Twente. The Medical Infolab is, among others, investigating how to improve surgeon's performance. One

of their ideas is to integrate a haptic and a neuronavigation device.

Previous ad-hoc attempts [7, 13, 20] without a systematic approach, have failed to meet the real-time requirements. The focus of this research is to creating a systematic approach for testing real-time requirements an demonstrate this with the integration of these devices.

1.2 Problem statement

There exist formal methods to validate hard real-time requirements in a model, based on execution timings specified by the developer. However, current research provides no tool-supported approach to test and assure the execution timings specified by the developer. The result is that developers are forced to predict execution timings without validating them.

It is desired that the tool-support is reusable in the context of testing real-time requirements for other devices to be integrated. This is desired because it gives developers the opportunity to test the real-time requirements using less effort and being less error-prone when compared with the ad-hoc approach.

All these facets lead to the main *problem statement*, which is defined as:

Developers have no reusable tool-supported approach for testing real-time requirements for integrated systems based on empirical evidence.

1.3 Approach

Since Domain-Specific Languages (DSLs) have been recognized as a tool/means to raise the level of abstraction [2, 27], this research aims at creating a reusable tool-supported approach for testing real-time requirements and formal validation approaches use DSLs, DSLs have the main focus. To investigate the main problem statement in a structured way the following research questions have been defined:

1. What are the requirements for a DSL aimed at the integration of devices with hard real-time requirements?
2. Is there an existing real-time oriented DSL which conforms to the requirements or is extendible to become conforming?
3. How can the resulting DSL be extended and used to test real-time requirements?

4. How much overhead does the proposed solution introduce and how does the solution scale?
5. Are there existing formal validation tools which can be used with the proposed solution?

To answer the first research question a literature review is performed on how to create a DSL. The results from this literature review are used to design a DSL. The resulting DSL contains the requirements for the integration of hard real-time devices.

To answer the second research question the DSL from research question 1 is compared with the results from a literature study which investigates existing real-time oriented DSLs. During the comparison a decision must be made to use either the created DSL or to use an existing DSL.

To answer the third research question a tool is created. This tool consists of an implementation for the DSL and a method for automatically transforming an instance of the DSL into this implementation. This tool must be able to test the real-time requirements.

To answer the fourth research question a special set-up of a case study using the proposed tool is created. The results from the performance tests are used to determine introduced overhead and the scalability.

To answer the final research question it is investigated if the proposed DSL can be used with existing formal validation tools found during the literature study into real-time oriented DSLs. Figure 1.1 presents the process of testing real-time requirements in an activity diagram.

1.4 Contributions

The main contributions of this thesis is a tool which consists of and contributes the following:

1. **DSL for specifying the integration and the real-time requirements.** The proposed DSL is an extended version of Architecture Analysis & Design Language (AADL). AADL is extended with the ability to specify monitors that monitor an execution scenario and an optional deadline for this scenario. This contribution is the result of investigation regarding research question 1 and 2.

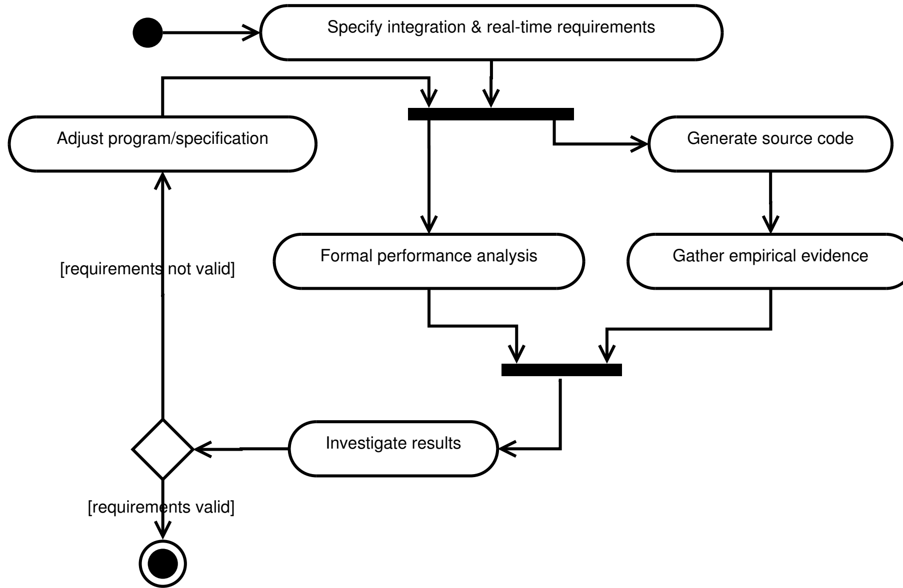


Figure 1.1: The overall approach for testing real-time requirements

2. **Framework implementing the DSL and gathering empirical evidence.** The proposed framework provides the implementation for the constructs in the DSL. This implementation consists of two parts; i) AADL constructs essential to implement the case study ii) monitors to track execution sequences. Each time when a monitor starts, finishes, misses a deadline or violates a strict restriction, the listeners specified by the developer are invoked. This contribution is the result of investigation regarding research questions 3.

3. **Code generation module to generate code based on an instance of the DSL for the framework.** To ensure consistent use of the framework and the DSL, a code generation module is created. This module is responsible for generating code for the framework based on an instance of the extended AADL model.

1.5 Outline

The thesis is organized as follows.

Part I starts with setting the context. This chapter provides the global introduction into the problem domain. Chapter 2 introduces the medical case study. The goal of the case study is to integrate a haptic and a neuronavigation device.

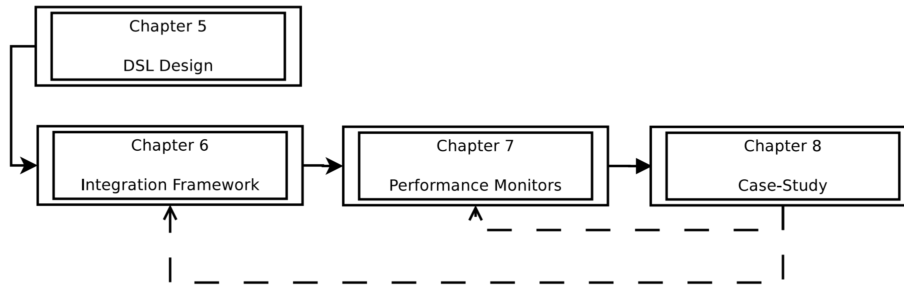


Figure 1.2: The main chapters of the thesis and the recommended reading order

Part II provides background information which is essential to understand the problem domain and the reasoning within the solution design. Chapter 3 defines what a real-time system is and explains different types of performance analysis. Chapter 4 provides an introduction into domain specific languages with a focus on real-time oriented Domain-Specific Language (DSL) and reusability.

Part III presents the proposed solution. Chapter 5 presents the design of a DSL to express the requirements for integrating hard real-time devices. This chapter also presents a comparison between the designed DSL and AADL. Chapter 6 presents the framework implementing the essential subset of the AADL model. Chapter 7 introduces the performance monitors. Part III concludes with discussing the design and implementation of the case study using the proposed tools. Figure 1.2 provides an overview of the chapters in this part. The chapters 5, 6 or 7 can be read independently, but chapter 8 depends on the previous two chapters. This is indicated by the dashed lines, the solid arrows indicate the recommended reading flow.

Part IV provides conclusions and evaluation of the proposed tools. Although the focus of this thesis is on testing real-time requirements and this research uses a case study only to demonstrate the proposed tools, the evaluation also presents prerequisites for real-world usage of the case study.

Chapter 2

Case study: Medical devices

This chapter provides an introduction to the case study. It starts with explaining why a haptic and a neuronavigation device are integrated, this part also presents a schematic overview of the whole system. The chapter continues with background information regarding both devices. Finally known challenges are presented.

2.1 The integrated system

The purpose of the case study is to aid a surgeon in performing a brain surgery, to be specific to aid with the removal of a tumour from the brain. During such a surgery it is vital to avoid damage to other parts of the brain. The difference between healthy and tumour tissue however cannot be seen by the surgeon without the use of MRI or CT scans. With these scans the surgeon faces the challenge of determining where his tools are within such a scan. To aid the surgeon with this a neuronavigation system is used, in this case called the VectorVision. Such a system shows the locations of his tools within the scans during a surgery. A picture of the VectorVision is depicted in Figure 2.1.

Using the VectorVision system introduces two important challenges for the surgeon:

1. *Focus of the surgeon* - The attention of the surgeon can be drawn away from the patient towards the navigation system. Thus the surgeon pays more attention to the screen instead of the patient.
2. *Two dimensions* - The information presented by the navigation system is two dimensional while the operation takes place in a three dimensional space.

These challenges are addressed with the introduction of a haptic device, in this

Figure 2.1: Navigation: Vector Vision

Figure 2.2: Haptic device: Freedom6S

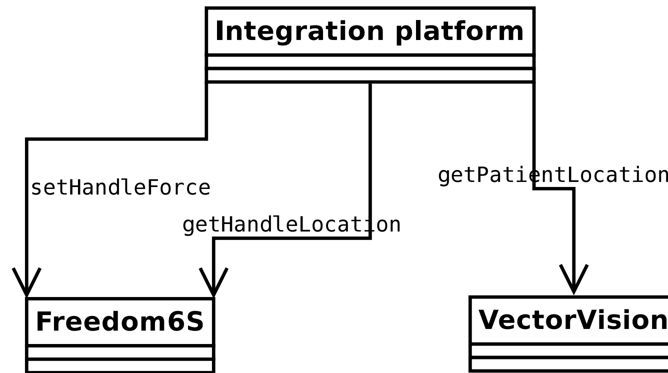


Figure 2.3: Hardware architecture

case the Freedom6S. A Haptic device is capable of producing an arbitrary force into an arbitrary direction, see Figure 2.2. The integration of an haptic device with a neuronavigation device is very interesting since the surgeon will be able to concentrate on the patient and work in a three dimensional space. While performing the sugary the haptic device provides haptic feedback regarding the boundaries of the tumour, thereby warning the surgeon when he's about to touch healthy tissue. Furthermore in the future a haptic device might be replaced with a robot which is capable of performing the same operation but with increased accuracy.

The integrated system thus consists of the VectorVision, Freedom6S and a system which links both of them, called the integration platform. A schematic overview of the relation between the devices and the integration platform is depicted in 2.3.

The integration platform requests information from both devices and applies the force on the Freedom6S. The requested informations is:

VectorVision The location of the patient and the Freedom6S is requested as often as possible (in practice 4 times per second)

Freedom6S The location of the handle is requested at least 1000 times per second

The integration platform keeps a reference picture which is being reset every time the locations are received from the VectorVision. This picture contains the locations of:

- the (marked) patient
- the Freedom6S
- the mathematical description of the (predefined) allowed area

2.2 Neuronavigation

The VectorVision is the result of several decades of research in the area of neuronavigation. The neuronavigation research area has emerged from the desire to be able to operate as accurate as possible; in this case to remove tumour tissue from the brain without damaging other parts of the brain. Stereotaxy is the methodology involved in the three-dimensional localisation of structures in the brain with respect to fixed points [28]. In order words, this means tracking the location of tools used by the surgeon during the operation. The surgeon is able to see the current location of his tool with the previously made scans.

Stereotaxy can be split into two main categories; i) the framebased category, ii) the frameless category. In both categories the patient's head is placed in a coordinate system to determine the best route. The framebased category uses a rigid frame which is attached very precisely to the patient's head. The disadvantage of this system is that the frame itself can become an obstacle.

A taxonomy of the different neuronavigation systems can be found Figure 2.4.

Frameless neuronavigation systems use two cameras to determine the locations. The cameras have a known distance between each other, this information is used to calculate the position of the tools and the head of the patient. The determination of the locations can be done with or without markers. The markerless systems use anatomical landmarks which have to be identified on the image data and on the

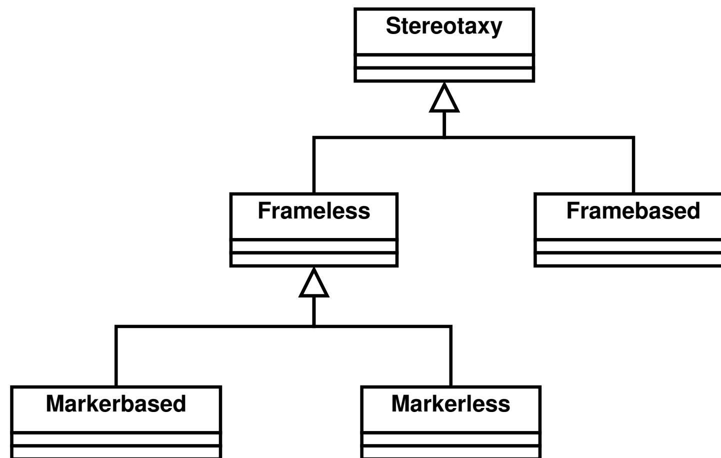


Figure 2.4: Stereotaxy taxonomy

patient. In the marker based systems these landmarks are marked and thus the location can be determined [11, 20].

For the neuronavigation systems to be useful it is essential that patient data is loaded into the system. This data can be obtained by either an MRI or a CT scan. Both these scans can be made upfront and be used to plan the operation. However, during the operation a phenomena called brain shifting can occur (Section 2.4.1). This issue is subject for medical research and can currently be addressed by the use of intraoperative scans.

2.2.1 VectorVision

The VectorVision is a frameless marker based neuronavigation system, produced by Brainlab [5, 25]. The VectorVision system consists of a computer, touch screen and 2 infra-red cameras, see Figure 2.1. The system is used within the operating room despite the fact that brain shift occurs.

The system runs Microsoft windows XP and uses a LAN interface to communicate with its environment. During some test runs the system is able to deliver the locations of the tools approximately 4 times a second.

Figure 2.5: Passive haptic device

Figure 2.6: Active haptic device

2.3 Haptic devices

Haptics refers to the sense of touch. A haptic device therefore aims at producing a certain amount of force into a certain direction. The next two sections describe the two types of haptic devices; i) passive, ii) active.

2.3.1 Passive haptic device

A passive haptic device provides interaction between the human operator and the haptic device without intelligent influence by software (see Figure 2.5). The operator works with the handle of the haptic device. The operator feels resistance as feedback (a force into a certain direction). Every time the operator makes a certain movement he or she receives the (force) feedback corresponding to that movement [20].

2.3.2 Active haptic device

An active haptic interface also provides feedback to the human operator, the difference with a passive haptic device is that with an active device the direction and amount of force is influenced by software (see Figure 2.6). The software receives the position of the handle. With this, the software calculates the force feedback that must be given to the human operator. The software provides the haptic device with the amount and the direction of the force which must be applied. An active haptic device is thus capable of producing different forces when a certain movement is made [20].

For an active haptic device it is very important to adjust the force with a frequency of 1000 times a second to ensure stable haptic interaction [23]. Stable haptic interaction means that if a force is defined that force is applied at the right time and location of the handle.

2.3.3 Freedom6S

The Freedom6S, produced by MPB technology, is an active haptic device. The device is connected to a computer using two analogue-digital converter PCI-cards. This ensures that it is possible to read the position 1000 times per second and to set the force a 1000 times per second.

The Freedom6S has 6 degrees of freedom. The handle of the haptic device can move in the x, y and z plane and next to that the handle is able to roll, pitch an yaw. The device is nearly capable of producing a virtual wall. This means that there is a virtual plane which cannot be crossed by the handle of the haptic device. When the operator moves the handle very close to the plane the haptic device produces a force which makes it difficult to move the handle through the plane. It is however impossible for the device to produce a force that cannot be overcome by the operator.

2.4 Challenges

There are known challenges regarding the devices which must be noted. These issues are; i) brain shift ii) deadline missing, and iii) frequency mismatch. These challenges are explained in the next sections.

2.4.1 Brain shift

During neurosurgery a phenomenon called brain shift can occur. Buchhols et al. [6] give the following description of the phenomenon:

> Cranial stereotactic systems which utilize preoperative computed tomography (CT) or magnetic resonance imaging (MRI) data sets to guide surgery are subject to inaccuracy introduced by the intraoperative movement of the brain (brain shift). Although these systems allow precise navigation initially during a procedure, brain shift resulting from surgical intervention can lead to progressive degradation in accuracy, with the greatest inaccuracy occurring when deep structures are manipulated.

Thus shifting of the brain can cause the used MRI and CT scans to mismatch the real-life situation. For this thesis this challenge is considered out of scope.

2.4.2 Missing a deadline

The system of course needs to meet its deadlines. However when a deadline is missed the surgeon must be notified. At the moment it is not investigated when a surgeon must be notified and when not. One can imagine that the movement speed of the surgeon during a surgery is very slow and thus missing a certain amount of deadlines within a certain time period is acceptable.

2.4.3 Frequency mismatch

The VectorVision delivers 4 frames per seconds while the force of the Freedom6S needs to be adjusted 1000 times per second. This leads to a frequency mismatch.

The Freedom6S is capable of detecting its own location 1000 times a second, this is used to adjust the force applied by the Freedom6S. The Freedom6S can only determine the location of its handle and can't determine the location of the patient without information from the VectorVision. It is assumed that this location remains static while waiting for the next update from the VectorVision.

Every time a new image is received from the VectorVision the reference picture is refreshed. The new challenge is how to cope with the changing reference picture and how to ensure the surgeon does not hurt the patient when the force is adjusted. This challenge needs to be confirmed with a real-life working case study, and is therefore, for this thesis considered out of scope.

2.5 Previous work regarding the case study

Three bachelor theses exist on integrating a haptic device and a navigation system.

Hendriks et al. [20] describe what the requirements are for such an integrated system. Burgman et al. [7] focusses on establishing the link. They require that the location of the tip of the Freedom6S is determined 1000 times a second, however in there solution they are unable to meet this requirement. Finally Geertsema et al. [13] are also able to establish the link, but the real-time requirements are not met.

Part II

Technological background and related work

Chapter 3

Real-Time Safety-critical Systems

This chapter starts with defining real-time safety-critical systems. After these definitions the chapter continues with explaining how performance measurements can be done to validate real-time requirements.

3.1 What is real time

A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period [8]. So the correct behaviour of a system is not only defined by the logical correctness of the output but also by the timing of the output. There are two categories of real-time systems:

Hard real-time Hard real-time systems are systems where it is absolutely imperative that responses occur within the required deadline.

Soft real-time Soft real-time systems are systems where deadlines are important but which will still function correctly if deadlines are occasionally missed.

Buttazzo et al. provide a slightly different definition. A hard real-time system is a system capable of executing hard-real time tasks. Buttazzo [9] et al. defines a task hard real-time *If missing its deadline may cause catastrophic consequences on the environment under control.* In the neurosurgery case missing a deadline can have severe consequences for the patient.

3.2 Safety-critical systems

A life-critical system or safety-critical system is a system whose failure or malfunction may result in:

- death or serious injury to people
- loss or severe damage to equipment
- environmental harm

The proposed integrated platform therefore clearly is a safety-critical system.

3.3 Performance analysis

There are mainly two approaches for performance analysis; i) static analysis and ii) dynamic analysis [33]. Both have their own advantages and disadvantages and their use is complementary.

3.3.1 Static Analysis

Static analysis approaches inspect the source of programs and perform the analysis without actually executing these programs. Static analysis is able to determine the dependencies between function calls. It can however not be used to determine the frequency of these calls and the execution times. This problem can be addressed by supplying a model with the static analysis. This model contains information regarding WCET, Average Case Execution Time (ACET) and execution frequencies.

Using this model schedulability and processor utilization can be determined. The main drawback of static performance analysis however is that the execution times are supplied by the developer and are not verified. Changes in hardware and/or software need to be reflected by the models. The next list presents challenges which are encountered when modelling the performance of a system:

Caching A cache stores a value temporally in a fast memory. When a value is needed it can be retrieved from this fast memory, however it cannot always be assured that the value is available in this fast memory. Using caches therefore reduces the predictability of the execution times. Ignoring all caches is also undesirable, because this leads to overestimation of the execution times.

Branch prediction CPUs perform branch prediction. This means that instructions are already fetched into the registers for conditional jumps. The CPUs keep track of if a condition is true or false and use statistics to 'guess' the outcome of the conditional jump. A correct prediction increases execution speed while a wrong guess slows execution speed.

Compilation optimizations The code that is written by the developer might be optimized by the compiler to increase performance. This may among others lead to the in-lining of functions or rewriting of loops. Although the average performance might increase, the predictability decreases.

Multi core/processor Modern trend is not to increase the CPU power by increasing the frequency but it is achieved by the addition of more cores. This addition leads to increasing complexity while validating real-time requirements.

OS The operating system can use different timer resolutions which influence the jitter in the execution of periodic threads. Also the ability of an OS to preempt a process influences the timeliness. Furthermore memory management can greatly differ from one OS to another influencing cache hits and misses.

Thread scheduling The scheduling policy must be considered because it influences the order in which the threads are executed.

3.3.2 Dynamic Analysis

Dynamic analysis makes use of data gathered from a running program. The program is executed and during the execution various data is collected that are subject to analysis. While it is unnecessary to model the underlying hardware and software there are two main drawbacks of this analysis:

Changed behaviour To be able to perform timing measurements the programming code has to be instrumented. The disadvantage is that instrumenting causes the (timing) behaviour to change.

Absence of failure cannot be proven Only the occurrence of missing deadlines can be measured, it is impossible to prove that there is no scenario in which a deadline is missed

Therefore by performing dynamic analysis it is possible to provide insights in actual timing behaviour, but it is never possible to prove real-time requirements. Furthermore, it is impossible to measure the actual execution time, because by measuring the time, the execution time is influenced. In this work, we have adopted a dynamic analysis approach.

Chapter 4

Domain-specific languages

This chapter starts with defining what a Domain-Specific Language (DSL) is. Next, the advantages and disadvantages of using a DSL are explored. The chapter continues with explaining when and how to development a DSL. Finally an overview of the real-time DSLs found during the literature study is presented in which special attention is paid to AADL.

4.1 What is a domain specific language?

Marnik et al. [27] provide the following definition for a domain specific language:

> DSLs are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with General Purpose Languages (GPLs) in their domain of application.

Deursen et al. [35] provide a slightly different definition:

> A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

There are subtle differences between the definitions but both agree on the fact that there is a certain domain to which the language is aimed. For this thesis both definitions are suitable.

Table 4.1: Caper Jones languages

Language	Domain	Level
Java	GPL	6
VHDL	hardware design	17
HTML	web page markup	20
SQL	database queries	25
Excel	spreadsheets	57

The next question that raises is are DSLs and GPLs mutually exclusive? DSLs and GPLs are not mutually exclusive [22, 27]. Instead a language can be to some degree a general purpose language and to some degree a domain specific language. Table 4.1 presents a number of languages and specifies to which degree the language is a DSL [21]. A higher level stands for more domain specific whereas a lower level means more general purpose. This is done by averaging the number of lines of source code needed to implement a function point and transforming this to the 'level' of the language (calculated by $\frac{320}{\#linesperFP} = level$). This method of classification is debatable since no clear definition exists for counting lines of code nor a method exists to define what a function point is. However the method gives a rough estimation of how general or domain specific a particular language is.

4.2 External & Internal DSLs

External DSLs are DSLs which do not have a host language to support them for type checking, compiling etc. An internal DSL is a DSL which uses a host language. Therefore the syntax of a internal DSL is limited by the syntax of the host language [10]. Another name for an internal DSL is a framework. Generally external DSLs are easier for non programmers (domain experts) when compared to internal DSLs.

4.3 Advantages and disadvantages

There are several advantages and disadvantages in using DSLs when compared with GPLs [2, 22, 27, 35]. The advantages are:

- Construct the solution in terms of the problem domain
- Enabler for reuse

- Easier to develop consistently
- Self documenting code
- Validation at domain level

The disadvantages are:

- Needs to be developed by someone with both domain knowledge & language development expertise
- Domain needs to be available
- Costs of learning
- Potential loss of performance

Research done in the area of software reuse [2, 27] concludes that adding domain knowledge to a system enhances the reuse opportunities. Biggerstaff [2] summarizes in his abstract:

> Fundamentally, the paper will make the argument that the first order term in the success equation of reuse is the amount of domain-specific content and the second order term is the specific technology chosen in which to express that content.

Thus if reusability is a requirement it clearly is a good idea to investigate the possibilities of a DSL.

4.4 When & how to create a DSL

Next to an assessment of the advantages and disadvantages Mernik et al. [27] proposes patterns which indicate that a DSL is a suitable solution, the patterns are depicted in table 4.2.

After the decision to create a DSL the following phases in creating the DSL can be identified:

1. **Domain analysis** Within literature there exists some approaches to perform a domain analysis. The general concept of these approaches is to search for the commonalities and the variabilities. Mernik et al. & Deursen et al. argue [27, 34, 35] that a feature diagram is suitable for capturing these commonalities and variabilities and such a diagram should always be constructed.

Table 4.2: When to create a DSL taken from Marnik et al. [27]

Pattern	Description
Notation	Add new or existing domain notation Important sub-patterns: • Transform visual to textual notation • Add user-friendly notation to existing API
AVOPT	domain-specific Analysis, Verification, Optimization, Parallelization and Transformation
Task automation	Eliminate repetitive tasks
Product line	Specify member of software product line
Data structure representation	Facilitate data description
Data structure traversal	Facilitate complicated traversals
System front-end	Facilitate system configuration
Interaction	Make interaction programmable
GUI construction	Facilitate GUI construction

2. **Design** When the language constructs are known, using the domain analysis, these can be used to design the DSL. This design can be based upon an existing GPL or DSL or can be a complete new language. The most important aspect to keep in mind it that a DSL is typically aimed at domain experts and they do not have to be programmers.

3. **Implementation** The next phase is implementing the DSL. This can be done by transforming the DSL into another language which can be interpreted by the software which is targeted. The targeted software can be among others a preprocessor or byte code.

4. **Deployment** The final step is to start using the created DSL, since this is a research project we consider real-life deployment out of scope. We use the designed DSL in the context of our case study though.

Detailed discussion regarding the different phases can be found in [22, 27, 35].

4.5 Which real-time oriented DSLs exists?

For the integration of the VectorVision and the Freedom6S it is very interesting to know which DSLs are available for modelling hard real-time systems. The next subsections provide an overview of the real-time DSLs which were found during literature study.

4.5.1 Functional DSLs

Hume Hume is a functional oriented programming language aimed at modelling real-time embedded systems. Hume is designed to be formally verifiable. To ensure this, Hume offers functionality using a layered approach. With each added layer the functionality of Hume increases but this is done at the costs of increased difficulty with verification of real-time requirements, see Figure 4.1.

Hume can be programmed using a box oriented approach. Each box contains has a certain amount of input and outputs, which are linked. A system exists of a composition of these boxes each again with linked output and inputs. The links are provided using automata theory [18]. The boxes itself can be programmed using the functional programming paradigm.

Although the syntax of Hume provides the ability to specify time-out exceptions, these exceptions are not implemented and thus never raised during execution. There is some work done on guaranteeing WCET [4]. The conclusion from this work is that due to the fact that they have complete access to

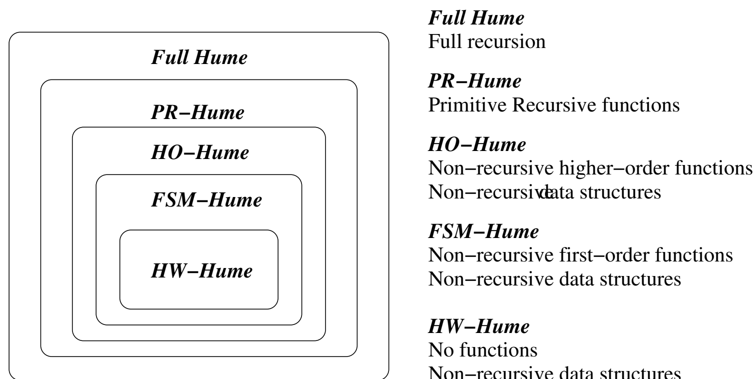


Figure 4.1: Hume layers

the abstraction machine it is possible to make reasonable estimations for the WCET. This is done for one specific embedded systems processor and they do not consider the increased complexity of modern non embedded system CPUs.

Atom Atom is a DSL aimed at hard real-time embedded systems. Atom provides the ability to do compile-time task scheduling. This very useful for determining execution and memory bounds. To provide guarantees of deterministic execution time and memory consumption, Atom places several restrictions on computation. First, Atom designs are always finite state: all variables are global and declared at compile time and dynamic memory allocation is not allowed. Second, Atom provides no function or looping constructs [1, 19].

4.5.2 Model based DSLs

Model based DSLs are based around a meta-model. A meta-model defines the rules which the instance model has to obey. The meta-model on itself also has to obey the rules of its meta-model, this model is called the meta-meta-model. Object Management Group (OMG) has proposed a Meta-Object Facility (MOF). MOF is a standard for Model Driven Engineering (MDE). The hierarchy can be found in Figure 4.2. This hierarchy is commonly used within the MDE products. The hierarchy starts on top with MOF which is a meta-meta-model and is defined by itself.

AADL Architecture Analysis & Design Language (AADL) is an architecture description language standardized by Society of Automotive Engineers (SAE). AADL was first developed in the field of avionics, and was formerly known as

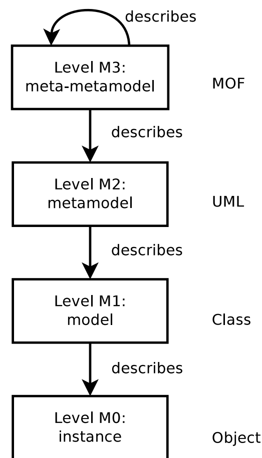


Figure 4.2: Meta-model hierarchy

the Avionics Architecture Description Language. It is derived from MetaH, an architecture description language made by Honeywell. AADL is used to model the software and hardware architecture of an embedded, real-time system. The resulting architectural model can be used for various activities such as analyses (among others schedulability and flow control) or code generation [3, 16].

AADL architectures can be made both textual and graphical. These notations can be made in a tool suite provided by AADL. AADL provides constructs which are used to design the architecture. These constructs are: Threads, Thread-Groups, Process, Port, Data, Events. Using these constructs the system can be designed and properties can be set. More in depth information can be found in section 4.5

MARTE Modelling and Analysis of Real-time and Embedded systems (MARTE) is a UML2 profile which for modelling real-time embedded applications with UML2. The standard is specified by OMG. MARTE annotates UML with the real-time properties. It can be used to model both hardware & software properties. MARTE consists out of four parts:

Foundations defining the basic concepts required to support real-time and embedded domain.

Design This profile refines the foundation by supporting detailed design of real-time software & hardware characteristics.

Analysis This profiles refines the foundations by supporting quantitative analysis of UML2 models. The profile is specially aimed at providing information needed for schedulability and performance analysis.

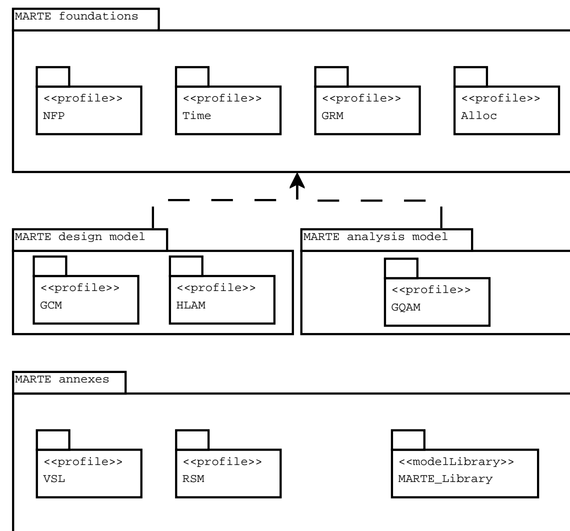


Figure 4.3: MARTE architecture

Annexes The profile gathers all the MARTE annexes such as the one defining a textual language for value specification within UML2 models.

These profiles are graphically depicted in Figure 4.3. This figure also introduces package names. Details about these packages can be found in [30].

4.5.3 Miscellaneous

REAL Requirement Enforcement Analysis Language (REAL) is DSL which can be used to define non functional requirements. REAL is linked with a AADL model as an annex to express requirements. REAL defines checks as sets and operations on these sets. The implementation of a REAL checker allows to assess if a model is compliant to the defined requirements [14].

OSATE Open-Source AADL Tool Environment (OSATE) is a tool environment using Eclipse and the Eclipse Modelling Framework (EMF) to aid with developing AADL models. The tool environment provides a front-end for creating AADL models and provides a number of tests for the AADL models. One of the tests checks the flow latency of the specified flows and presents a report on the latencies.

Cheddar Cheddar is a third-party tool used to perform schedulability analysis of AADL models. The Cheddar tool functions by assessing the execution properties specified in the AADL model and performing schedulability analysis

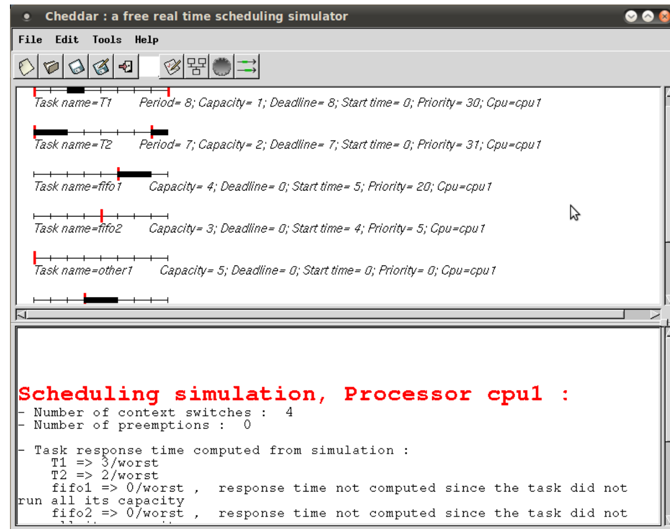


Figure 4.4: Cheddar screenshot

according to the Portable Operating System Interface [for Unix] (POSIX) standard. To be able to do this the developer has to specify the POSIX properties used within the system. The tool uses the information provided to schedule the threads, see Figure 4.4.

HOOD & Stood The Hierarchical Object Oriented Design (HOOD) method is created, in the late eighties by the European Space Agency, in order to provide a Model Driven solution for its large projects. As opposed to AADL or MARTE, HOOD is not only a modelling language, it also defines a complete development process from requirements analysis to code generation of real-time systems.

Stood is a tool suit provided to use the HOOD development method. Due to the similarity between AADL and HOOD, the Stood tool suit is suitable to model both AADL and HOOD [12].

In the previous sections several DSLs are presented. AADL seems the most promising for creating a reusable integrations. While the functional languages are stateless and might be easier to verify it is hard to perform an early model performance analysis with functional languages.

4.6 Architecture Analysis & Design Language

AADL uses a component centric model and defines the system architecture as a set of interconnected components. The standard is extensible by the use of annexes or be extending the meta-model directly. There are among others annexes available for specifying error handling and behaviour. AADL, without annexes, only provides the ability to specify non functional properties of the components. Behaviour properties or functional aspects must be defined separately.

The specification is done by specifying the type of the component and possibly one or more implementations. The next sections provide more insight into the main parts of the AADL specification. These insights are based on [24, 26, 31, 32]

4.6.1 Software components

AADL provides the ability to specify both hardware and software components. Since for this research we do not consider hardware components we present the software components:

- *Process* represents a virtual address space. Process components may contain threads, data and thread-groups.
- *Thread* represents the basic unit of execution and schedulability in AADL. Threads may contain data and subprogram subcomponents, as well as subprogram call sequences. A thread is either contained in a process or within a thread group.
- *Thread group* represents an organizational component to logically group threads contained in processes
- *Data* represents a data type or a data instance. Data components may contain other data, subprograms and subprogram-groups
- *Subprogram* is an abstraction of the procedure from imperative languages
- *Subprogram group* is an abstraction to a library of procedures from an imperative language

4.6.2 Properties

AADL allows the system designer to attach properties to any component. Properties are used to specify functional aspects of the system, for instance the period of a

thread, the computation entry point or the scheduler algorithm to be used. AADL provides a predefined set of properties. A component specified in the architecture by default inherits all the properties from its parents unless the property is defined in the components itself.

4.6.3 Analysis of AADL models

Analysis of the AADL models is done based on the provided properties. An example of a tool to perform the analysis is the Cheddar tool mentioned in the previous section. The developer can set WCET and other execution properties of a thread. These properties are used to validate the real-time requirements. This validation is done by considering the number of CPUs and validating if the different threads can be scheduled on the CPUs using the WCET properties specified in the AADL model.

Part III

Solution design & implementation

Chapter 5

DSL design

This chapter presents the design of a Domain-Specific Language (DSL). The goal of designing a DSL is to investigate the requirements for integration of devices with real-time requirements. First a domain analysis is performed, using the language constructs found during this phase the DSL is designed. The chapter continues with explaining the implementation choices. Since a number of real-time oriented DSLs have been found in literature, see Section 4.5, we continue with comparing the designed DSL and Architecture Analysis & Design Language (AADL).

5.1 Solution technology

The solution technology is closely related with the solution requirements. The requirements for the solutions have been presented in Section 1.2. To create a reusable solution which might be able to reuse existing formal validation tools the initial idea is to create a DSL, since a DSL is a enabler for reuse, Section 4.3.

A DSL can be created using two different approaches either using Model Driven Engineering (MDE) or using a classical lexer-parser approach. To quickly develop a prototype and editor for the DSL we have chosen to use MDE.

The disadvantages stated in Section 4.3 state among others that DSLs possibly cause decreased performance. This can be addressed by how the DSL is compiled and interpreted. Another general disadvantage of DSLs is that they introduce cost of learning. Within this research project this is not an issue since we design and implement the DSL, for others who decide to use the DSL this still is an issue. However, if the DSL is used in a production environment the provided validation of real-time requirements should be well worth the expense of learning the DSL. Finally to create a DSL it is essential to have access to domain experts. This research is

carried out as a joint venture between the computer science & technical medicine departments therefore the required domain knowledge and language construction knowledge is available.

The creation of a DSL consists of several phases: *decision*, *analysis*, *design*, *implementation* and *deployment*. The decision to create the DSL has been explained (enabler of reuse). The next phase is analysing the domain.

5.2 Analysis of the domain

The analysis of the domain starts with defining the problem domain. After the problem domain is defined it is analysed.

5.2.1 Defining the problem domain

The focus of the problem domain can be either on the integration of hard real-time devices or on the integration of the specific devices used within the case study. The chosen problem domain is describing the integration of hard real-time devices and the real-time properties, the reason for this is twofold.

First, currently there is no working integrated system, while three attempts have been made [7, 13, 20]. Apparently it is not trivial to create an integrated system that conforms to the real-time requirements. To create such a system it is essential that the timing of the threads is validated and assured. Assuring the timing of threads means calling the appropriate deadline miss handler in case a specified deadline is missed.

Second, the area of haptic medical navigation is a greenfield. No working integrated system with both a haptic and a medical navigation system exists that we are aware of. Therefore it is not possible to perform a domain analysis to search for common problems with integrated such devices.

5.2.2 Domain analysis

It is important that the features required by the case study are supported, since the purpose of this research is to create a tool supported approach for integrating devices with real-time requirements for gathering empirical evidence. Therefore the previous integration attempts, documentation regarding the case study is used as sources for the domain analysis. If this scope narrowing result in missing features for other integrations, those are considered out of scope.

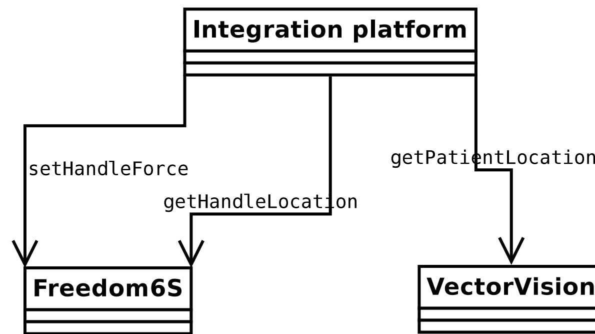


Figure 5.1: Possible representation of the case study

The goal of this analysis is to capture both the domain commonalities as well as the variabilities. An obvious commonality of systems which integrate several devices is the notion of **device**. All devices within the case study provide an **Application Programming Interface (API)** which is able to communicate with the device. The integration platform itself can also be seen as a 'device' with its own API.

The essential part of the integrated system consists of the communication between the devices and the integrated system. To control the communication between the devices, **threads** are used. Threads provide the ability to perform multiple tasks simultaneous and a-synchronised. A thread is responsible for the communication of the output of one API call to the input of another API call. Furthermore threads have properties to, among others, specify the frequency of the method calls and the deadline miss handlers. Since the case study requires to integrate hard real-time safety-critical devices, the properties of threads mostly concern specification of real-time requirements.

The **bold** marked words are the minimum concepts which must be offered by the DSL to be able to satisfy the requirements.

A possible graphical representation of the case study integration is depicted in Figure 5.1, this figure is the same as Figure 2.3. The blocks are the devices and the lines represent the threads.

During the design several assumptions are made which narrowed the scope, the next list presents the assumptions and argues why these assumptions are reasonable and do not cause the DSL to be a specific solution to this case study:

API It is assumed that all devices provide an API. This API provides an interface written in a specific language and is used to communicate with and control the device. It is very common for manufacturers to provide an API with there devices for ease of use.

Single Language Within the case study all APIs are available in C and C++. For other devices this might not be the case. To encounter this challenge Common Object Request Broker Architecture (CORBA) or another cross language calling mechanism can be used.

Deadlocks & Priority inversion It is assumed that each threads needs to lock at most one resource at a moment in time. For the integrated system this assumption is realistic because each thread only has to communicate with one resource at a time and can release this resource after the communication has finished. This is caused by the fact that the integrated system desires the latest information to be available and this information is not dependant on history. Therefore priority inversion is also not considered.

5.3 Designing the DSL

The design of the DSL is done using a MDE approach. Using this approach it is possible to use the meta-model tools provided by the Eclipse modelling framework. These models make it possible to rapidly create prototypes of the DSL. Furthermore an editor can be generated from the meta-model to create instances of the meta-model. The resulting DSL is depicted in Figure 5.2.

The top node is called DSL and is the root. The root node has left and right children. The children on the left represent one or more devices. The devices have one or more APIs which are written in a certain programming language. The root node his right children represent one or more thread groups. Thread groups are introduced to make it possible to logically group threads and assign them to a processor. Each group has one or more threads. This thread can either be periodic or aperiodic.

The threads are the core of the integration platform. Threads have several timing related properties. It is impossible to capture timing properties into primitive data types, since the typical range of an integer is not capable of capturing timing values up to nanoseconds level. Furthermore some data types might need special properties. Therefore a data type hierarchy has been designed, see Figure 5.3. Each data type must provide a compare method in order to be usable with the property hierarchy.

To provide the ability to predefine property boundaries a property hierarchy is designed. The hierarchy is depicted in Figure 5.4. The goal of this hierarchy is to facilitate the specification of initial values as well as upper and lower bounds of complex data types. It is the responsibility of the implementation to ensure the specified ranges and raise exceptions when they are violated.

The property hierarchies top node is the abstract template [1] class Property. Each

[1] Within C++ a template is very similar to a generic class within Java

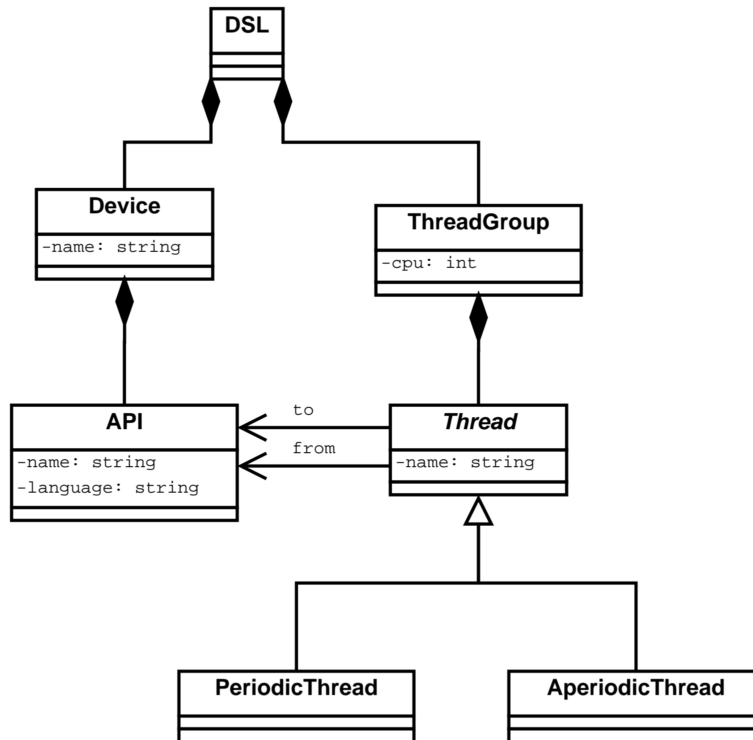


Figure 5.2: DSL Meta-model

property has a value. This value is of the same type as specified in the template. The classes which implement Property specify next to the value also if the value is bounded and if so what the allowed range is. These ranges are of the same type as the value of the Property. The specified type to the template has to be a subclass of the data type hierarchy, see 5.3.

5.4 Implementing the DSL

Section 5.1 argues that it is important for the DSL to investigate the way it is compiled and interpreted. This section explains how this is done.

Within the presented bachelor theses on this subject the integrated system is created using either 'C++ for Mathlab' or Java (no Real-Time Java). Since they were not able to meet the real-time requirements the suitability of these languages are doubted. By default Java is not appropriate for real-time applications, since the garbage collector manages memory in an unpredictable way. To overcome this real-time Java can be used. This Java version has special memory constructs which is

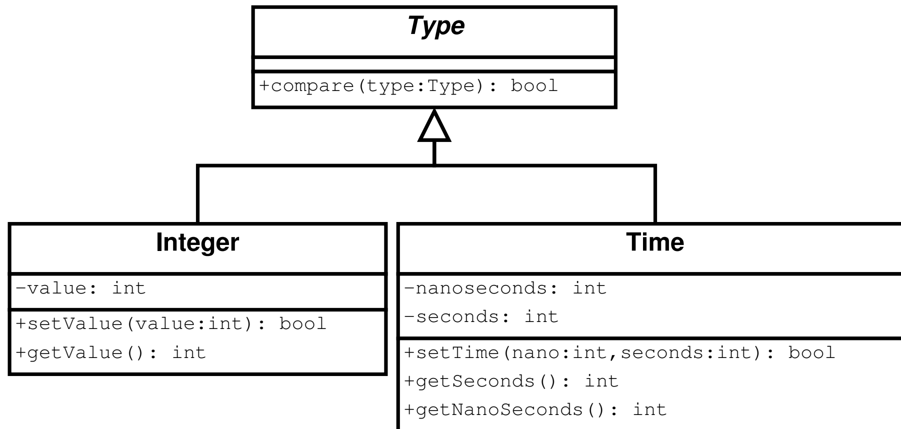


Figure 5.3: DSL Data types

not subject to the garbage collector and threads which run with a priority higher than the garbage collector.

However the use of Java is odd, because both the Freedom6S and the VectorVision have their APIs available in C and C++ and not in Java. While Java is able to perform cross language function calls to C and C++ this will introduce additional latency. Therefore the choice for a the languages are narrowed down to either C or C++. Since C++ provides a easier constructs for object-oriented programming and the DSL is developed using MDE, which is object-oriented, C++ is chosen.

5.5 Comparing the DSL with AADL

When comparing the DSL with AADL it is clear they resemble each other. The basic language constructs of the DSL are all available also in AADL. The notion of process is available in AADL and not in the DSL. This notion is not needed for the case study but might be needed for other integrations. AADL is chosen to model the integrated system and the real-time requirements of this system for the following reasons:

- AADL has great resemblance with the designed DSL
- AADL is extendible
- AADL is an industry standard
- AADL provides the opportunity to reuse of existing tools

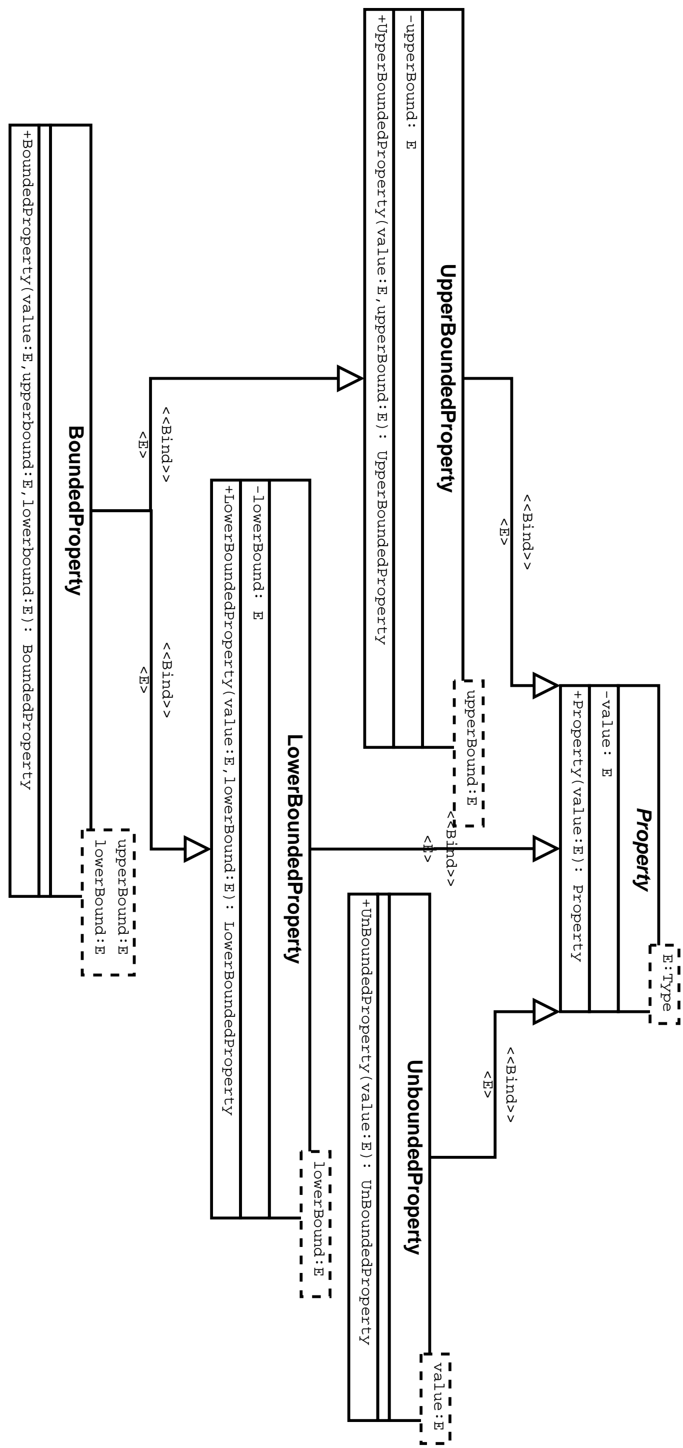


Figure 5.4: DSL properties

5.6 Conclusion

This chapter answers research question 1: *What are the requirements for a DSL aimed at the integration of devices with hard real-time requirements?* Such a DSL must consider the notion of devices, API and threads. Furthermore the chapter answers research question 2: *Is there an existing real-time oriented DSL which conforms to the requirements or is extendible to become conforming?* AADL is the most promising to be able to conform the requirements. The designed DSL is compared with AADL and AADL is chosen because of four reasons:

- AADL has great resemblance with the designed DSL
- AADL is extendible
- AADL is an industry standard
- AADL provides the opportunity to reuse of existing tools

Chapter 6

Integration Framework

This chapter describes the design and implementation of the integration framework and the code generation module which links the AADL model with the integration framework. The purpose of the integration framework is to provide an implementation for the features of AADL which are essential for dynamic performance analysis. The chapter starts with presenting the class diagram of the integration framework. The chapter continues with explaining how the AADL model and the framework are linked.

6.1 Framework design

The language in which the framework is created in C++, the argument is presented in Section 5.4. The high-level design of the integration framework is depicted in Figure 6.1. First, the responsibilities of every class are discussed, afterwards the detailed design is presented.

Integration Framework is responsible for the coordination of all the threads. The threads are created and started within the class.

Configuration is responsible for configuring the framework. The developer has to provide source code to create the objects and has to register them with the Service Locator. Furthermore the developer can specify what to do at certain events during execution.

Service Locator is responsible for providing the coupling between the user programming code and the framework. It is able to store an arbitrary object and associate it with a supplied name, this name is later used to retrieve the reference to the stored object.

Thread Manager is responsible for providing all threads with unique identifiers.

Thread Spawner is a highest priority thread which starts all the other threads and then dies. This ensures that while the thread spawner starts all threads no started threads get execution time before the thread spawner dies.

Thread represents a thread within the system.

Periodic Thread represents an AADL periodic thread in the system. Upon creation of a periodic thread the interval must be specified.

Property represents a thread property.

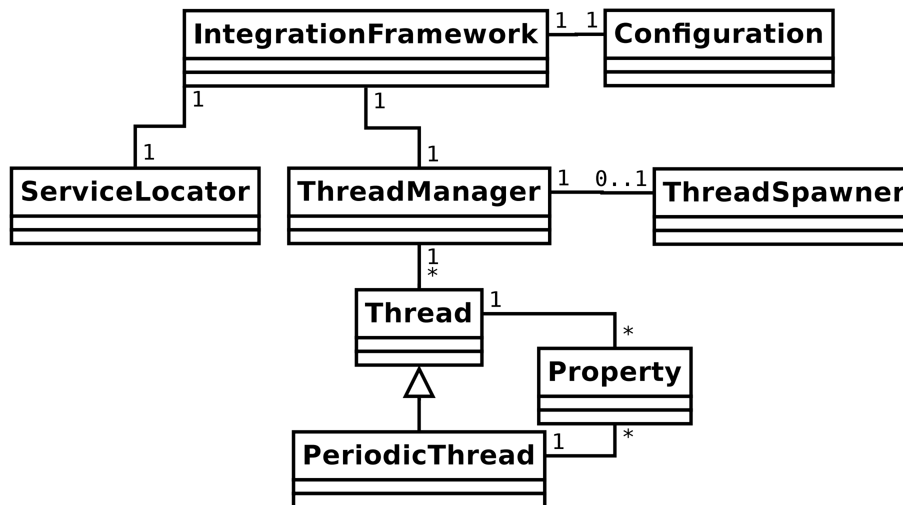


Figure 6.1: High-level integration framework

A more detailed design is depicted in Figure 6. Within this picture the constructor of the PeriodicThread expects a parameter of the type Property<Time>. The design of the property model is depicted in Figure 5.4 on page 38. The corresponding DataTypes design can be found in Figure 5.3 on page 37.

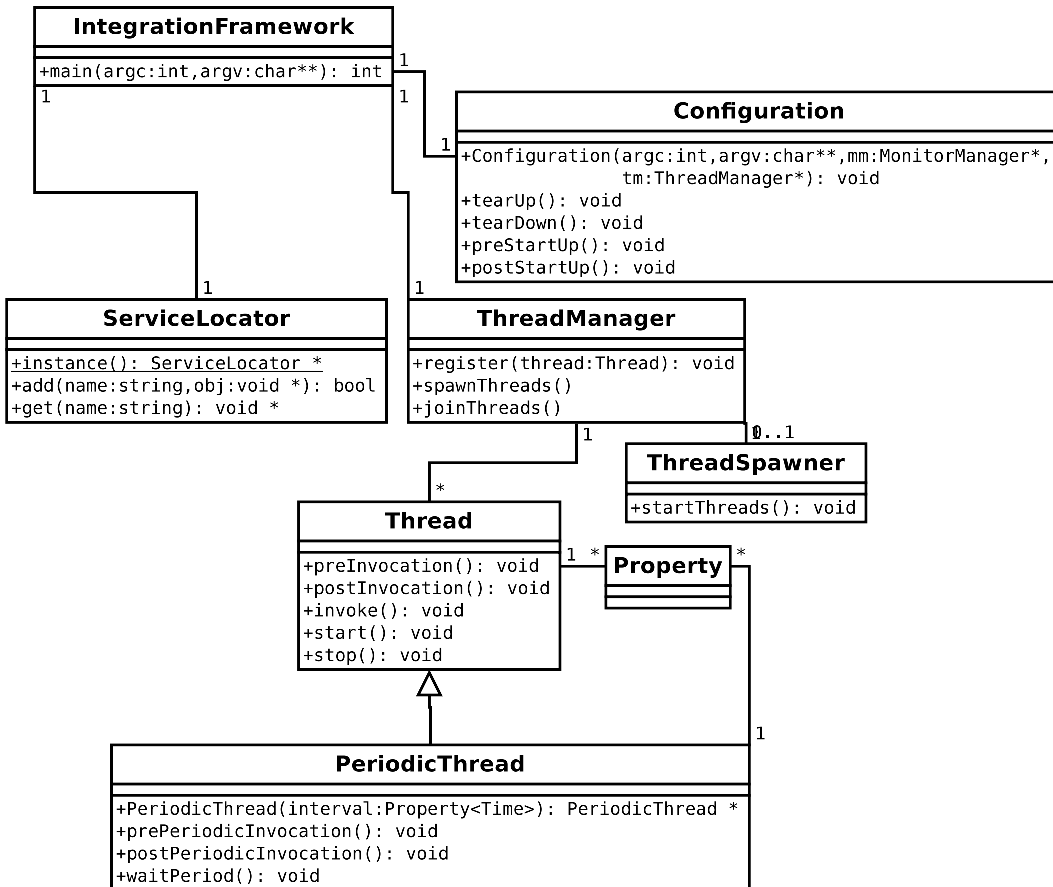


Figure 6.2: Detailed design integration framework

Within the integration framework we have chosen to use POSIX threads. POSIX threads have been used for other AADL code generation research projects [24, 36]. Furthermore the AADL property set also provides the ability to specify certain POSIX properties.

6.2 Linking the framework & AADL

The purpose of the integration framework is to cooperate with an AADL model, as explained in Chapter 5. To create a link between the framework and the AADL model two main approaches exists; i) ANother Tool for Language Recognition (ANTLR) parser generator, ii) Model Driven Engineering (MDE) approach. Since the meta-model of AADL is available and the fact that MDE is more suitable for auto gener-

ation of editors the MDE approach is chosen.

The meta-model of AADL is used together with Acceleo [29]. Acceleo is an Eclipse plug-in which can be used to define meta-model to text generators. Since the authors are familiar with the Eclipse environment, Acceleo supports the MDE approach and Acceleo is in the default Eclipse modelling framework we use Acceleo to generate the code without considering other code generators.

A generator within Acceleo consists of translation rules to translate elements of the meta-model to text. When the translation is defined, an instance of the meta-model is used to automatically generate the desired C++ code files.

As explained in Section 4.6 an AADL process contains threads (either directly or via a thread group). An AADL process has a name, this name is used to retrieve an object reference from the service locator. This reference must be supplied to the service locator by the developers and the referenced object must be able to perform the specified methods in the AADL model.

Within a process one or more threads can be defined. For each thread a class is generated. This class retrieves the object from the service locator and uses this object to call the method specified in the computation entry property of the thread in the AADL model. While creating the thread the properties specified within the AADL model are used to set the properties within the framework.

Although the AADL specification states that computation entry point must be a public method available within the global scope we have chosen to call the method by reference of the object in this implementation. This is done because we believe that this better fits the object oriented approach taken here and we are able to protect the device within the object where we find it suitable. If one however desires to change this behaviour this can be done with a small modification in the Acceleo generator module.

The detailed templates created during this research can be found in Appendix C.

6.3 Conclusion

The presented framework provides an implementation for a subset of AADL. The framework is written in C++ and implements the periodic threads constructs from the AADL specification. Furthermore thread groups are used to link objects with the threads. This framework is the first part of the answer to research question 2: *How can the resulting DSL be extended and used to test real-time requirements?* Because this framework can be used as a basis to test the real-time requirements.

Chapter 7

Performance Monitors

This chapter presents the performance monitors. The design is presented first, followed by the semantics, which are defined by statecharts. With the semantics clear, insights are provided into the implementation decisions. The chapter concludes with explaining how the AADL meta-model is extended to support the monitors.

7.1 Design

Figure 7.1 presents the class diagram; within this diagram the different entities represent the following:

MonitorManager is representing the entity which manages all the monitors. Furthermore the manager provides a deadline monitor thread which queries the monitor's time-out timers.

- *registerMonitor* is used to register a monitor
- *processMethod* is responsible for processing a method invocation by invoking the appropriate monitors.
- *getMonitors* returns all the registered monitors.
- *initDeadlineMonitor* must be invoked to initialize the deadline monitor after all the monitors are registered. The created thread has the highest priority and reads all the time-out timers of the monitors. After the timers are read the threads sleeps for the specified amount of time. The boolean parameter indicates whether or not the deadline period must include the execution of the deadline checks.

MonitorListener represents a listener which is called at certain events. The developer can implement his own listeners and associate them with the monitors.

- *deadlineMiss* is called when the associated monitor misses a deadline.
- *monitorStart* is called when the associated monitor starts. Depending on the configuration of the monitor, this method is invoked just before or just after the first method in the execution sequence.
- *monitorFinish* is called when the associated monitor has finished. Depending on the configuration of the monitor, this method is invoked just before or just after the last method in the execution sequence.
- *strictViolation* is called when the associated monitor violates a strict property. A strict property is used to define certain parts of the execution scenario as strict. Strict means that calls to observed methods other than the expected in the scenario lead to a reset of the monitor and an invocation of all the listeners. Observed methods are the methods which have the injected pre and post calls and are thus observed by the framework.

Monitor represents the monitor, which has certain properties regarding when to start and stop measuring and has an associated execution scenario. The monitor has several methods:

- *monitorStartMeasurementBegin* specifies if the timing has started just before the first method of the monitor is called or when the first method finishes. The default value true specifies that the measurement must start just before the method call is executed.
- *monitorFinishMeasurementEnd* specifies if the timing has finished just before the final method of the monitor is called or when the last method starts. The default value true specifies that the measurement must finish immediately after the method call is executed.
- *startMethod* specifies the first methodCall in the execution sequence of the monitor.
- *addListener* is used to add a MonitorListener to the monitor.
- *removeListener* is used to remove a MonitorListener from the monitor.
- *getListeners* returns all MonitorListeners.
- *processMethodCall* is responsible for tracking the state of the monitor and starting and stopping deadline timers.
- *getTimeout* returns the specified time-out.
- *getTimeoutTimer* returns the time-out timer which is checked by the deadline monitor thread.
- *getDeadlineMissed* returns the number of deadlines missed by this monitor.

- *increaseDeadlineMissed* increases the number of deadlines missed by one.

MethodCall represents a method call specified in the AADL model. A method call has several methods:

- *getMinimum* specifies the minimum amount of method executions that must be encountered. The minimum value for this property is 1.
- *getNext* specifies the optional next method in the monitor.
- *getCall* specifies the method from the AADL model.
- *getStrict* specifies if this methodCall is strict. Strict means that calls to observed methods other than the expected lead to a reset of the monitor and calling the listeners of the monitor. Observed methods are the methods which have the injected pre and post calls and are thus observed by the framework.

MonitorMethods is an enumeration used to be able to quickly match method names.

7.2 Semantics

To formalize the semantics of the monitors statecharts [37] are used. Before modelling the monitors using statecharts, we define the following:

X The capital letters within the edges refer to the method with the same name. When the same method has multiple occurrences a postfix numbering is used.

$ transition This transition matches all pre/post events input besides other outgoing pre/post edges specified at the source of an $ edge [17].

Multiple threads In between two visits of the start state the edges which are associated with the same letter (and numbering) must be performed by the same thread. This implies that if two threads execute the same method and thread 1 starts executed and gets pre-empted by thread 2 the execution of thread 2 is ignored because the statechart expects thread 1 to finish first.

Logical impossibility If an event occurs for which no arrow is defined the state chart remains in the same state.

Start() This function is responsible for the administration of starting a monitor. This administration includes recording the current time, setting the deadline (if specified) and notifying all listeners.

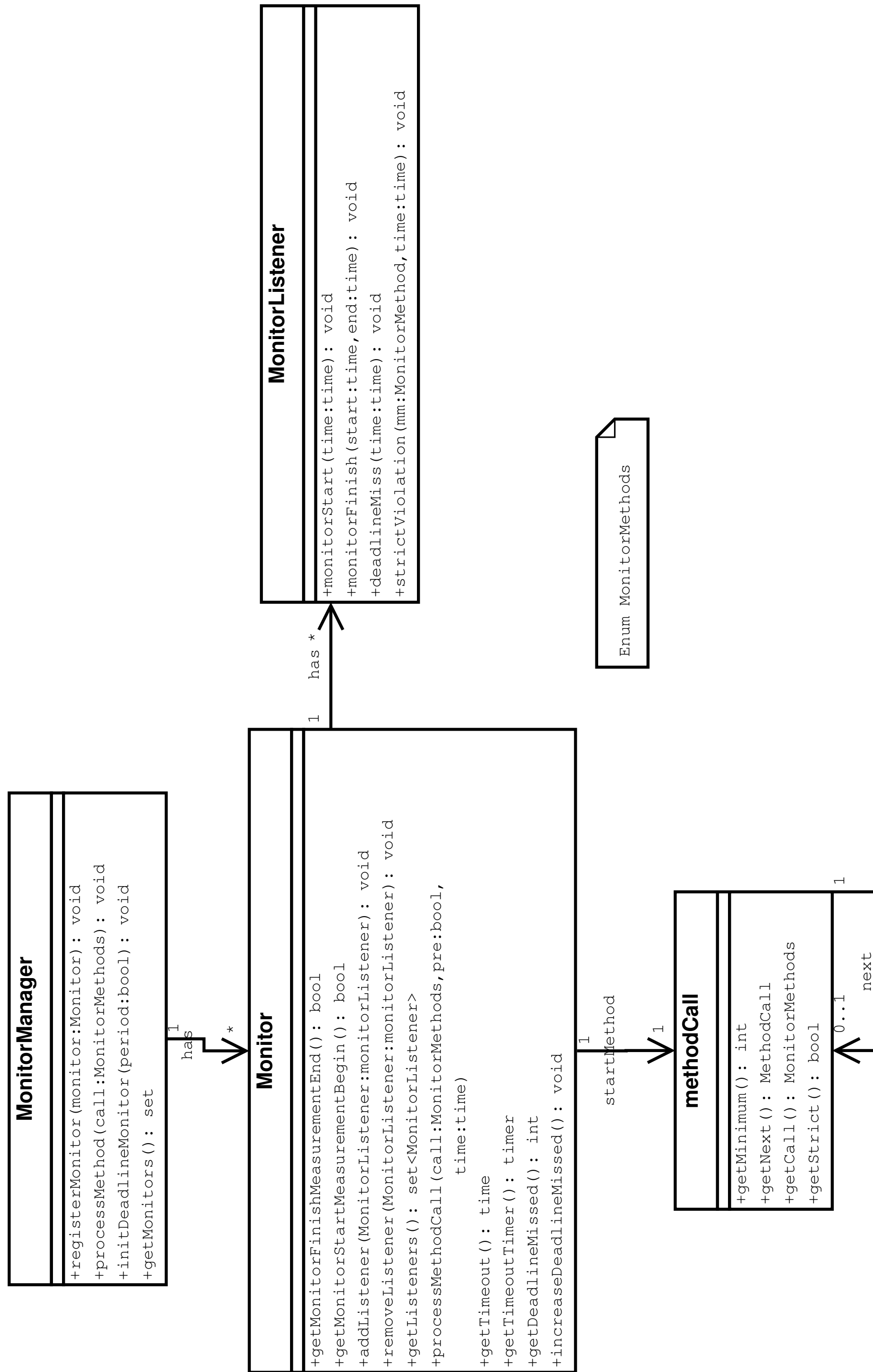


Figure 7.1: Class diagram of the performance monitor design

End() This function is responsible for the administration of finishing a monitor. This administration includes recording the current time, notifying all listeners and call the reset function.

Violation() This function is responsible for the administration of a strict violation of a monitor. This administration includes notifying all listeners and calling the reset function.

Reset() This function is responsible for the administration of resetting a monitor. This administration includes removing the deadline (if specified), resetting the start/end time, putting the monitor in its initial state.

Guards The guards refer to properties of monitor or the method call.

- **Start** This guard refers to the startMeasurementBegin property of a monitor.
- **End** This guard refers to the finishMeasurementEnd property of a monitor.
- **Strict** This guard refers to the strict property of the current state within the scenario.

The algorithm used to track the execution scenario of a monitor is given in Appendix D.

7.2.1 Statechart design

Figure 7.2 presents the statechart for a monitor which tracks an execution scenario consisting of one and two methods without considering possible deadlines. Within the statecharts all edges for a scenario consisting of tracking one or two methods are present. Which edges can be taken depends on the design of the monitor.

Statechart 7.2(a) starts in the initial state ($wait_X$) waiting for the *preX* event to occur. When this event occurs one of the 4 edges is taken depending on the boolean configuration. The statechart remains in X until a *postX* occurs on which one of the 4 edges back to the initial state is taken. If the method being monitored has the property strict set to true then at all moments in time another observed method can occur and cause the statechart to take the edge $*[strict]/violation()* causing an event to be raised to the monitors and the statechart to reset.

Statechart 7.2(b) presents tracking two methods or a sequence of two single methods. From the initial waiting state two edges are present. Depending on when the measurement must be started, one edge is taken. After the first method finishes two edges are present. If the measurement was not started on the first transition it is started by taking this edge. From the state $wait_Y$ one of the two edges is taken on the *preY* event depending on when the measurement must stop. When method

Y finishes the *postY* event occurs. As a result one of the two edges to the initial state is taken. Again, at all times a strict violation can occur when a method in the scenario has the property strict set to true and a method which does not occur in the scenario executes.

Figure 7.3 presents the general design for an execution scenario consisting of three methods. When more methods are in the execution scenario the nodes corresponding to method Y can be expanded. This expansion is realised by duplicating the states $wait_Y$ and Y and connecting them with the pre and post edges. The new nodes are inserted in the statechart.

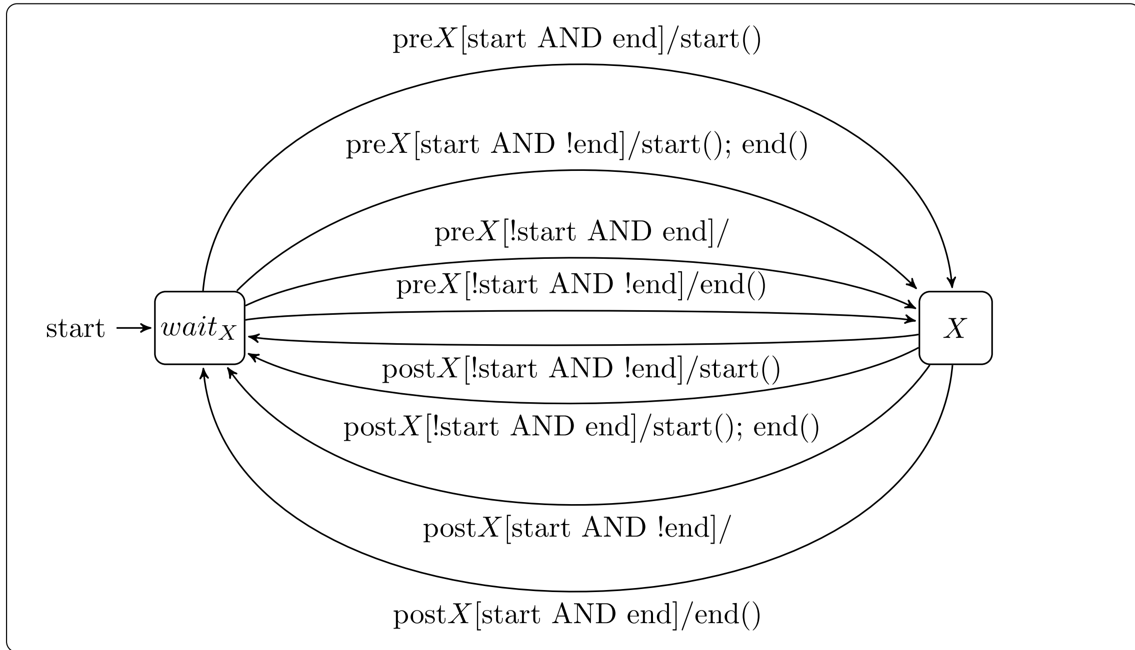
An important feature of the performance monitors is the ability to track the specified deadlines. Figure 7.4 presents the statechart for tracking the deadline of a monitor. The *after(deadline)* transition is implemented using a polling mechanism which polls the deadline timer of a monitor on a predefined interval. When a monitor has a deadline this statechart can be combined with the statechart representing the execution scenario with the statechart's 'and' operator.

When a deadline is missed and the edge labelled *after(deadline)/deadlineMiss()* is taken and all listeners of the monitor are notified. This notification is done within the deadline thread and will cause additional delay depending on the implementation of the listeners.

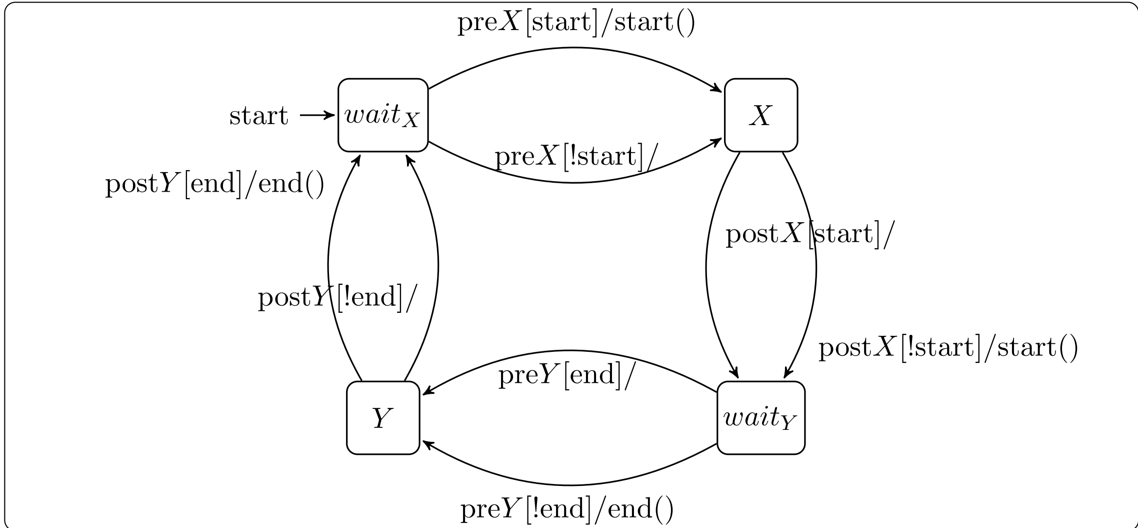
7.2.2 Examples

Figure 7.5 depicts an example model and the corresponding statechart of a monitor which tracks each execution of method A. Within this diagram the $wait_A$ state represents the waiting until a *preA* event is raised. When this event is raised the edge is taken and the start function is called. The new state is A. When the method returns, the edge *postA* is taken, the finish time is recorded and the monitor listeners are notified. The monitor now resets to the initial state to be ready to monitor the next execution.

Figure 7.6 shows the statechart for the monitor tracking the execution of two consecutive calls to method A. The method call is defined as strict, which introduces the state hierarchy with the $*[strict]/violation()* edge. The guard is still in this edge, this guard is essential because only the second method in the sequence is specified as being strict. This implies that when another tracked method is executed after the $postA_1$ call, the statechart resets. The scenario thus measures the waiting time between two consecutive calls to method A without the interference of another call after the first execution finishes.



(a) Statechart prototype for one method



(b) Statechart prototype for two methods

Figure 7.2: A prototype statechart of a monitor having an execution scenarios consisting of 1 or 2 methods

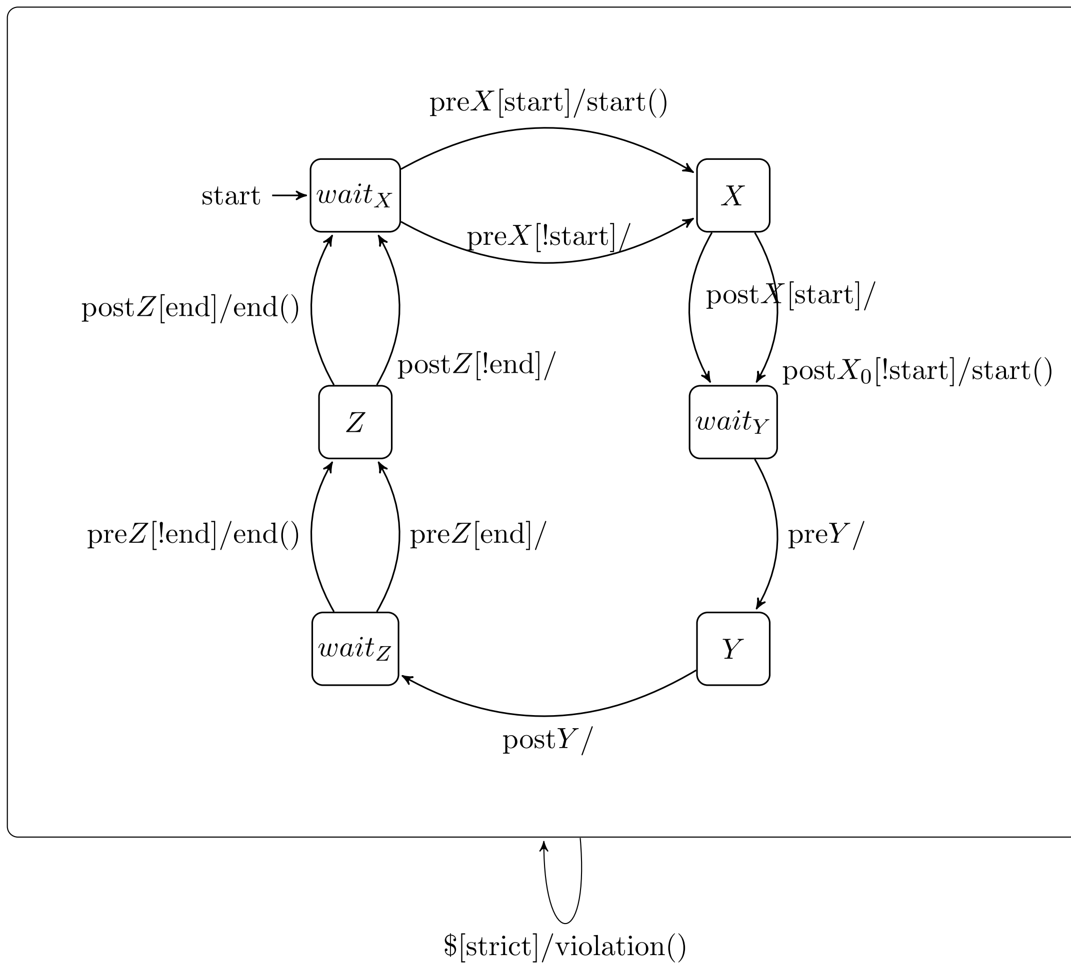


Figure 7.3: A prototype statechart of a monitor having an execution scenario consisting of 3 methods

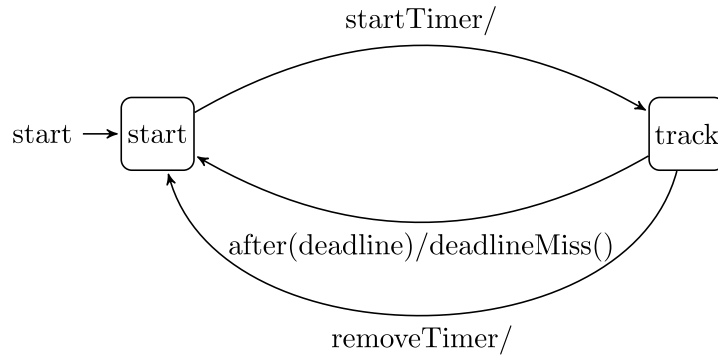
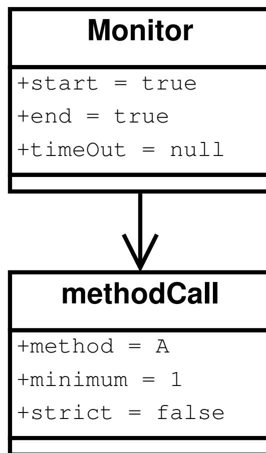
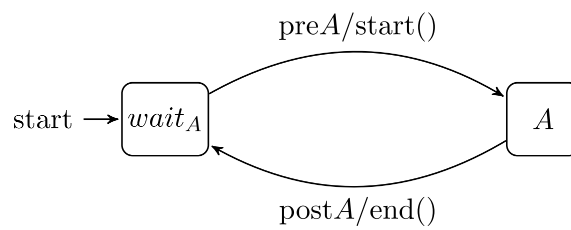


Figure 7.4: Deadline monitor prototype

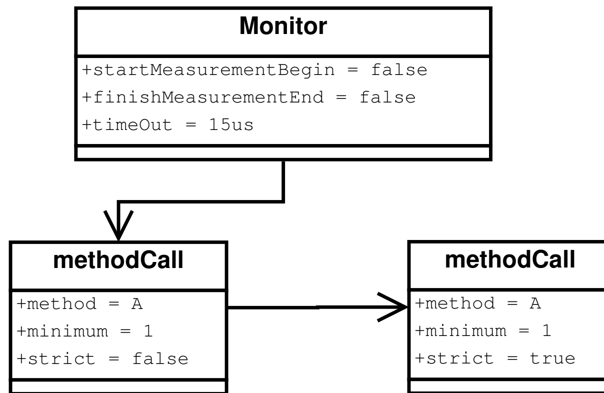


(a) Model instance

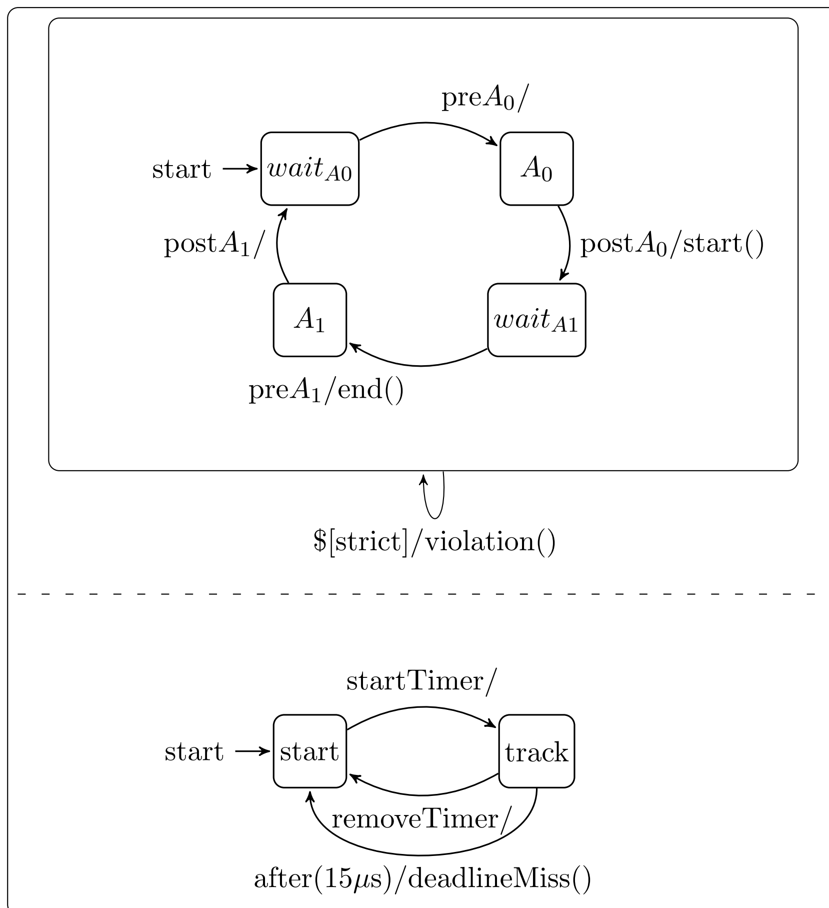


(b) Statechart with edges based on model instance

Figure 7.5: Execution scenario of single call to method A



(a) Model instance



(b) Statechart with edges based on model instance

Figure 7.6: Execution scenario of monitor tracking two consecutive executions of A

7.3 Implementation decisions

During the design and the implementation of the monitors several decisions have been made. This section provides a discussion of these decisions and the reasoning behind them.

7.3.1 Multiple threads on a single method

An interesting challenge is how to handle multiple threads executing the same method, which could cause interleaving of the threads while executing the same method. Within this thesis monitors monitor a specified execution scenario which consists of a sequence of methods. This means that calls which interleave must either be ignored or a separate instance of the monitor must be started. For this thesis we have chosen to ignore interleaved calls. To implement this each threads has a unique identifier. Whenever a thread calls one of the monitors this identifier is provided. The monitors use this identifier to match pre and post method calls.

7.3.2 How to monitor deadlines

During the implementation of the monitors several design decisions are made regarding when and how to monitor the deadlines. Monitoring deadline misses in the post call of the monitor is not feasible, because missing a deadline would not be known until the post call which occurs after the method finishes. Three other options for monitoring have been considered:

1. Create a thread, with the highest priority, every time a monitor records the starting time
2. Create one highest priority deadline monitor thread associated with each monitor which monitors the deadline of that monitor
3. Create one highest priority deadline monitor thread which queries all monitors for their deadlines

Ad-hoc tests showed that creating a thread each time the monitor is started is not feasible. Creating a thread takes about $100\mu s$ and has to be done 1000 times a second for the case study, resulting in at least 10% overhead.

Testing the second option revealed that this option is faster but influences scalability significantly, approximately a factor 2 difference with the final solution. This is caused by the fact that if 20 monitors are defined with a deadline there are also 20

highest priority threads which query the deadline timer. Therefore we have chosen for option 3, a single deadline monitor thread, managed by the MonitorManager, which tracks the deadlines of all monitors.

Option three queries the monitors regarding their deadlines on certain intervals. During the implementation of the framework two alternatives have been considered:

1. Query all monitors and when this is done do nothing for a specified amount of time

2. Within a specified amount of time query all monitors and sleep the remainder of the time

Both approaches have their advantages and disadvantages, therefore the framework provides both implementations. The main advantage of the first option is that the other threads in the system always get a specified amount of time after checking the deadlines. However it is more difficult to determine the exact interval in which the deadlines are checked (the interval is determined by the time it takes to query all monitors combined with the specified amount of time). Within option 2 this interval is specified and thus known. However the second option has the possible disadvantage that when a deadline is missed the time the deadline handler takes increases, since all the listeners have to be called. This reduces the amount of time for the program in contrast with option 1 since the interval is including this execution. This may lead to the missing of new deadlines which causes a snowball effect.

7.3.3 Where to determine the time

During the initial implementation it is perceived that the order in which the monitors are called influences the determination of the execution time. To ensure that the monitors perceive the same execution timings the time is determined at the start of the MonitorManager and the determined time is passed to the monitor. This is done before trying to obtain a mutex to ensure that we measure actual method execution time instead of the monitors execution time.

7.4 Monitors & AADL

To use the monitors together with AADL, AADL is extended and a code generation module is created. Within an AADL model the developer specifies a WCET for a thread. This WCET is used to automatically generate the code to monitor the timing

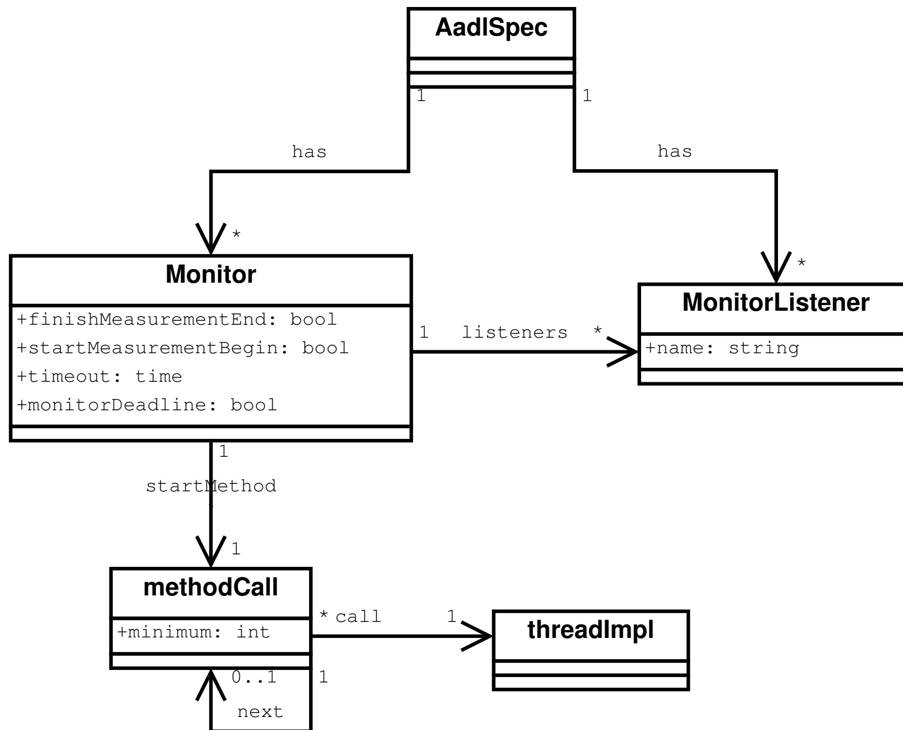


Figure 7.7: AADL extension to support monitor

of this thread. To make it possible for a developer to disable code generation of a monitor for a specific thread the AADL meta-model is extended with a property *defineDeadlineMonitor*. If this property is set to false no monitor is generated. Furthermore the AADL meta-model is extended to specify the listeners for the generated monitor.

Next to tracking the execution of the threads specified in the AADL model, a developer can define other monitors. Figure 7.7 presents the AADL meta-model extension to facilitate this. The extension is linked to the AADL meta-model on the AadlSpec EClass in the AADL meta-model. The developer can specify monitors using the thread implementations specified within the AADL model. The thread type of the thread implementation is used as a reference to the object on which the method specified in the thread must be called.

7.5 Conclusion

The primitives presented provide the means to test real-time requirements. The monitors together with the framework provide an answer to the research question 3: *How can the resulting DSL be extended and used to test real-time requirements?* The created extension for AADL is used to model the integrated system together with the desired monitors. This model is used in combination with the code generation module to create source code to instantiate the framework. The framework provides the performance monitors to track the execution timings.

Chapter 8

Case study: Integrating medical devices

The case study is about the cooperation between a neuronavigation device and a haptic device, as explained in Chapter 2. The focus of this research is on creating a reusable tool supported approach to integrate devices and gather empirical evidence. Therefore the integration of these devices has been created as a prove of concept and as a test basis for the framework.

The documents supplied with the Freedom6S, the haptic device, also contain a working example of a ball and a stick. Within this example the handle of the Freedom6S is the stick that can be moved by moving the handle in real-life. When the handle touches the ball a force is applied. For the integration this example has served as a basis. The goal of the integration is to move the ball based on the location of the head which can be determined by the neuronavigation device, the VectorVision.

8.1 Design

Figure 8.1 represents the class diagram of the case study; within this diagram the different entities represent:

StickOnBallControlView is responsible for providing the user-interface to control the application.

StickOnBall3dView is responsible for providing a three dimensional representation on the screen using openGL.

IntegrationModel is responsible for modelling the real-world situation within the program. Furthermore the entity provides methods for transforming coordinates from one coordinate system to the other.

Coordinate is responsible for representing a coordinate.

Freedom6SController is responsible for controlling the Freedom6S. The generated code calls a method from this class.

VectorVisionController is responsible for controlling the VectorVision. The generated code calls a method from this class.

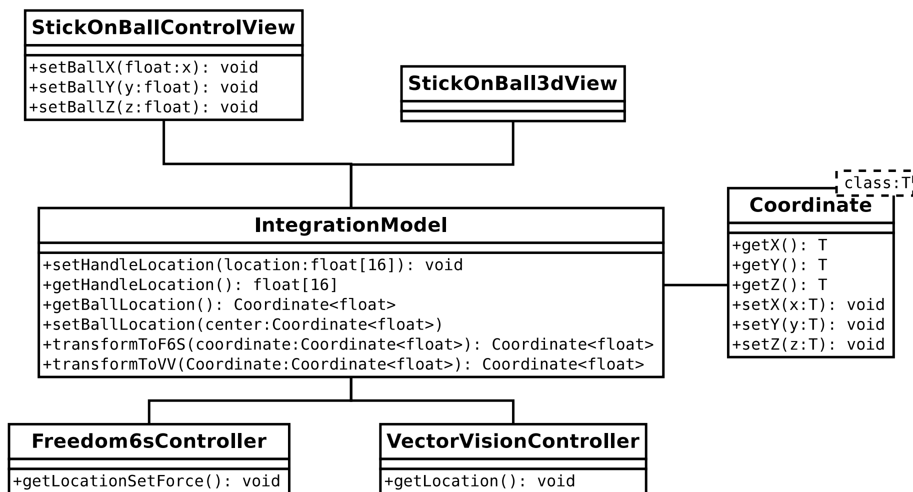


Figure 8.1: Class diagram of the case study

Next to the class diagram an extended AADL specification is designed. Listing 8.1 presents the specification. This specification is used to generate the code which ensures the frequencies specified and monitors the deadlines.

Listing 8.1: Textual version of the extended AADL model

```
thread Freedom6s
end Freedom6s;

thread implementation Freedom6s.impl
  Compute_Entrypoint => "getHandleLocationAndSetForce";
  Period => 1 Ms;
  Deadline => 1 Ms;
  listeners => ("DefaultMonitorListener");
end Freedom6s.impl;

thread VectorVision
end VectorVision;

thread implementation VectorVision.impl
  Compute_Entrypoint => "getPatientAndF6sLocation";
```

```
  Period => 200 Ms;
  Deadline => 100 Ms;
  listeners => ("DefaultMonitorListener");
end VectorVision.impl;

monitor integration
  Deadline => 101Ms;
  listeners => ("DefaultMonitorListener");
  startMethod {
    call => "VectorVision.impl.getPatientAndF6sLocation";
    next {
      call => "Freedom6s.impl.getHandleLocationAndSetForce";
    }
  }
end integration
```

8.2 Implementation

The most complicated part of implementing the integration was transforming points from one coordinate system to another. This is done by placing a VectorVision marker on a known position and direction within the Freedom6S coordinate system and use this reference marker to transform the coordinates. Furthermore during ad-hoc tests we experienced great influence of the OS on the real-time performance therefore we present some details regarding the used software and hardware, see Table 8.1 and 8.2.

8.3 Results

This section presents the results of the timing the monitors tracked. During an extensive test run of $36,000,000$ executions zero deadlines were missed. Figure 8.2 presents the execution timings of 200 updates of the Freedom6S. The figure shows very constant execution times with an average of 519.18μs. The log files showed that no deadlines were missed.

Figure 8.3 presents the execution timings of the VectorVision method. The execution times are very unpredictable and are significantly higher when compared to the Freedom6S. The VectorVision has an average of $50,671.75\mu$s (100 times higher than the Freedom6S). The log files showed that 14 deadlines were missed.

Figure 8.4 presents the execution timings of the scenario that consists of an execution of getting the patient location from the VectorVision followed by an execution of the applying the calculated force at the Freedom6S. This scenario thus tracks the time it takes for any changes in the patient location to be applied by the Freedom6S. The figure is very similar to the VectorVision figure, this is due to that the VectorVi-

Table 8.1: Software configuration of case study

What	Value
OS	Ubuntu 9.10
Kernel	2.6.32.2 #1 PREEMPT Thu Jun 17 16 : 10 : 05 CEST 2010 i686 GNU/Linux 9.10
Kernel patch	RTAI 3.8.0
Kernel options	hrtimer enabled, cpu frequency scaling disabled
Timer interface	timerfd
GCC	g++ (Ubuntu 4.4.1-4ubuntu9) 4.4.1
Compile options	-O2
Freedom6S API	2.2.0
VectorVision API	7.7

Table 8.2: Hardware configuration of case study

What	Value
CPU	Intel(R) Pentium(R) D CPU 3.00GHz
Memory	1GB DDR2 533Mhz

sion execution times is on average 100 times higher than the Freedom6S execution time and thus dominates the figure. The average of the executions in this figure is 178.31μs above the sum of both other graphs, this represents the average waiting time before the next call to the Freedom6S is performed. The log files showed that 14 deadlines were missed, which matches with the number of executions above 101ms.

8.4 Conclusion

This chapter has shown the successful integration of the case study. It has shown how the framework and the monitors are used to test the real-time requirements. This chapter demonstrates how the created tool is used to gather the empirical data. This chapter thus provides an answer to research question 3: *How can the resulting DSL be used to collect empirical evidence?*. The data shows that the real-time requirement of the Freedom6S, a maximum execution time of 1000μs, is met. Furthermore it shows that the execution time of getting the patient location from the VectorVision is on average 100 times higher less predictable when compared with the Freedom6S.

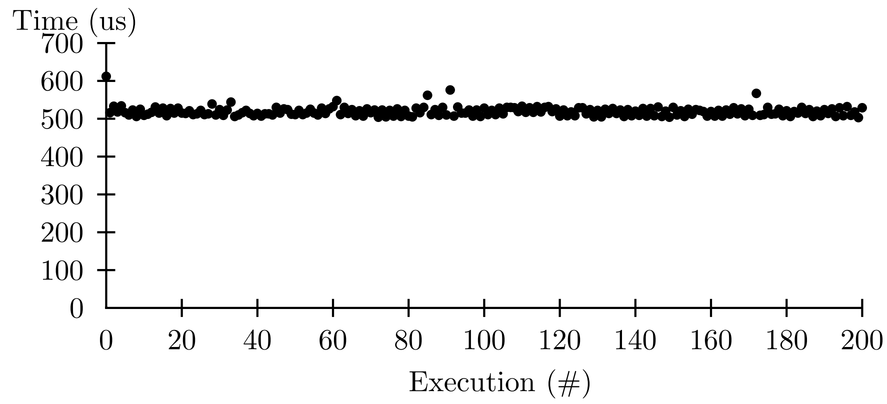


Figure 8.2: Execution timings Freedom6S.getHandleLocationAndSetForce()

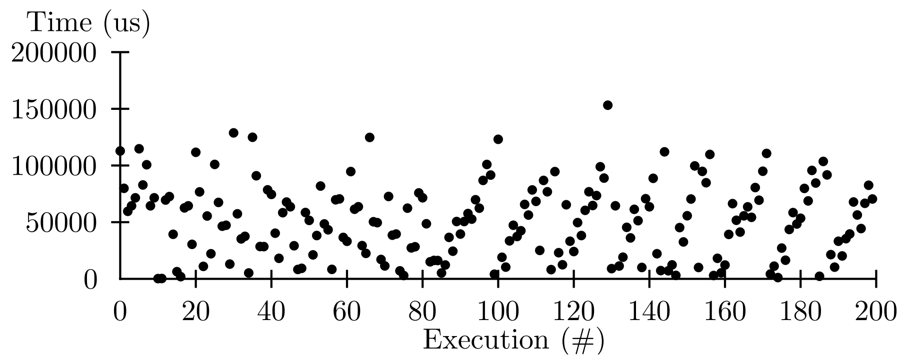


Figure 8.3: Execution timings VectorVision.getPatientAndF6sLocation()

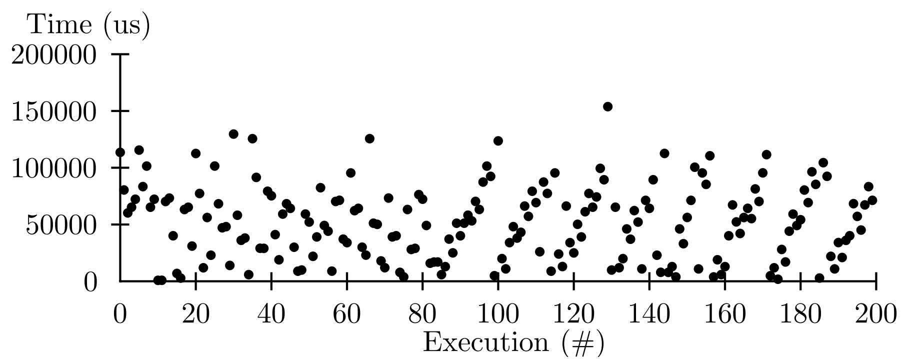


Figure 8.4: Execution timings of A VectorVision call followed by a Freedom6S call

Part IV

Evaluation & Conclusions

Chapter 9

Evaluation

This chapter presents the evaluation of the framework and the case study.

9.1 Framework

The evaluation of the framework consists of four parts: i) overhead introduced and scalability, ii) timer jitter, iii) reusability of the framework, iv) reusability of existing formal analysis tools . The section starts by explaining how the evaluation is performed.

9.1.1 Test set-up

The approach presented in Listing 9.1 is used to measure the introduced overhead. This approach is the same as a normal method call from the generated code except it has additional calls to the *EvaluationMonitor*. This monitor measures the execution time of the actual method and the *NullMonitors*. The number of *NullMonitors* can be changed to measure the influence of multiple monitors. The monitor is called *NullMonitor* because it has an empty implementation and 1 empty listener. An empty implementation is chosen because the developer has to specify the desired behaviour during the development of the integrated system. During ad-hoc tests a listener attached to a monitor without a deadline, which logs all the execution timings, caused approximately 10μs overhead per execution of a method in the execution scenario.

The first statement within the processMethod method of the MonitorManager determines the current time, which is passed to the monitors. A monitor therefore mea-

sures the time from the first statement within the *MonitorManager →methodCall(EvaluationMonitor, true, threadId)* call to the first statement within the *MonitorManager → methodCall(EvaluationMonitor, false, threadId)* call, and thus includes all *NullMonitors* calls and the first call to the *EvaluationMonitor*. This monitor has no deadline and its listener has an empty implementation for the monitorStart call; the influence of performing the measurement is therefore as small as possible.

Listing 9.1: Algorithm to measure overhead

```
MonitorManager->methodCall(EvaluationMonitor, true, threadId);
MonitorManager->methodCall(NullMonitors, true, threadId);
actual method call;
MonitorManager->methodCall(NullMonitors, false, threadId);
MonitorManager->methodCall(EvaluationMonitor, false, threadId);
```

To have a real-world scenario and an execution scenario in which the actual method call takes a stable, known amount of time, the Freedom6S is used. The VectorVision introduces an unpredictable delay which makes it not suitable to measure the time the monitors take, see Section 9.2. The Freedom6S is used because during the case study tests the execution time of the Freedom6S has proven to be stable. The hardware and software details used during the tests are presented in in the previous chapter in Table 8.2 and 8.1 (on page 61).

Within the framework different factors influence the amount of overhead. To determine the scalability of the framework it is important to determine the influence of these factors on the total overhead. The next list presents the factors that were identified and which settings are used:

- Number of monitors (varies between 0, 1, 2, 4, 8 and 16)
- Number of monitors having a deadline (varies between 0, 1, 2, 4, 8 and 16)
- The period of the deadline monitor (varies between 33μs, 50μs, 100μs and 200μs)
- Number of threads specified (varies between 1, 2, 4 and 16)
- The execution frequency of the threads (the sum of the execution frequency of all threads is fixed at 1000 times a second, for example when 2 threads are present both have a execution frequency of 500)
- Number of monitor listeners (fixed at 1)
- The deadline timing algorithm (the algorithm with the period including the deadline checks is used, the other algorithm is shortly mentioned at the end of the next section)

Each test lasts 30 minutes and consists of $1,800,000$ executions, unless stated differently. Furthermore the handle of the Freedom6S remains stable during the tests.

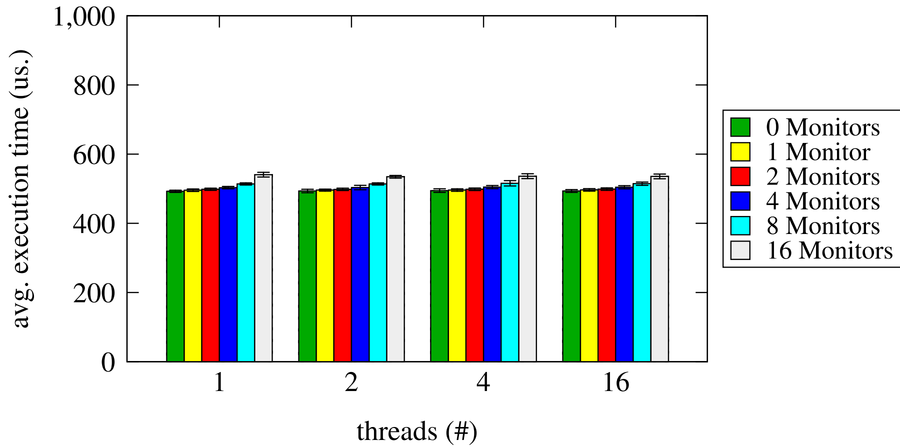


Figure 9.1: Timings without deadline monitor

9.1.2 Overhead and scalability tests

Figure 9.1 depicts the results of the average execution times of the tests without a deadline. The vertical axis represents the average execution time in microseconds and the horizontal axis represents the clusters. Each cluster represents a different number of threads and each bar within a cluster represents a different number of monitors. Furthermore the figure contains the error bars that indicate one standard deviation above and below the average.

To determine the overhead of a single monitor without a deadline, the algorithm found below is used. The resulting average amount of overhead introduced is 2.63μs for each call and each monitor which has this method in its execution scenario. This is the overhead introduced for a monitor tracking an execution sequence of one method. When the execution sequence consists of multiple methods the overhead will be lower for each called method; this is caused by not having to notify listeners and update the deadline timer on intermediate methods.

1. Take the average of all threads with 0 monitors
2. Group the execution times by the number of monitors
3. For each group calculate the average and subtract the average found in step 1
4. Divide the result from step 3 by the number of monitors in the group
5. Take the average of all the monitor groups

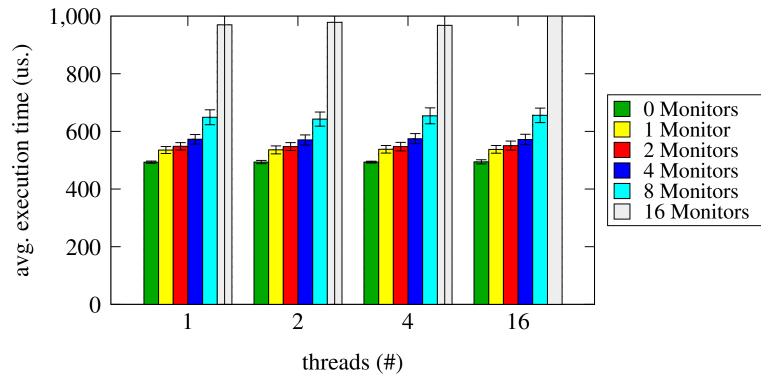
Figure 9.2 represents the average execution times of a different number of threads, monitors and deadline monitor periods. Within this figure, the bars corresponding to 0 monitors represent the executions without a deadline monitor thread. The increased execution time from 0 monitors to 1 monitor includes the introduction of the deadline monitor thread and 1 monitor.

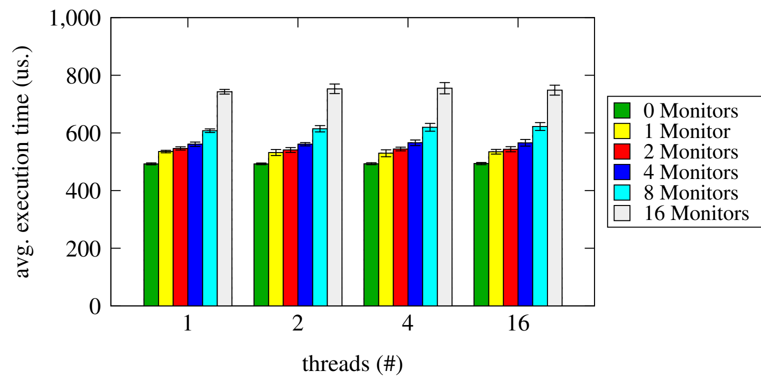
The figure shows, as expected, an increased execution time when more monitors are present or a lower deadline period is used. Furthermore a very large standard deviation increase occurs when using 16 monitors and a deadline period of 33μs. This is caused by the fact that the system cannot keep up. A low standard deviation is therefore an indicator for a system which is able to keep up with the computations.

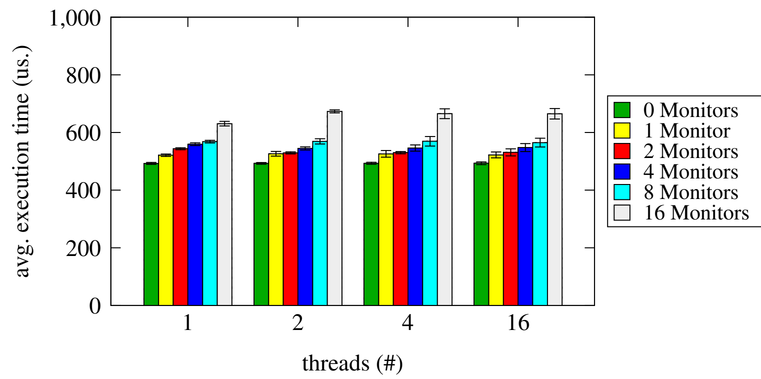
The tests to determine the overhead for each additional monitor are performed slightly different. The results are obtained by averaging 30 test runs of 1 minute. The line of reasoning for this different approach is:

1. All threads have a predefined periods.
2. At start-up a small random delay is created between starting each thread.
3. For the remainder of the program the start time is used as a reference for the period.
4. The period of the main thread is 1000μs.
5. All periods of the deadline monitor thread are a common divider of 1000μs (e.g. 200μs, 100μs etc.).
6. The previous 2 facts cause the thread execution to be interrupted by the deadline monitor thread the same number of times each execution for a certain test run.
7. The random delays cause this number to differ from test run to test run. For example an execution which takes 500μs can be interrupted 2 or 3 times by a deadline monitor with a period of 200μs. How many depends on the start-up delays and remains the same for the remainder of the test run.
8. Therefore for determining the overhead of a monitor taking the average of multiple tests is more suitable than performing a long single test.

Table 9.1 represents the results of the 30 test runs. This table contains the average additional overhead for monitoring the deadline of a single monitor for different deadline periods within the tests using 1 thread. Other threads are not considered because Figure 9.2 shows that the results from the tests using a different number of threads are very similar.







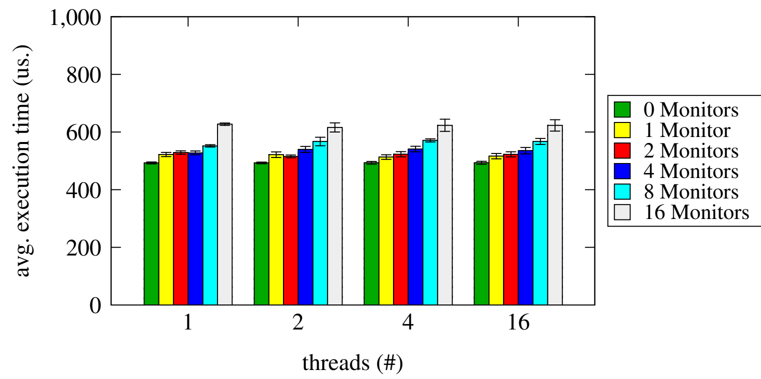


Figure 9.2: Execution timings with deadline periods, $33\mu s$, $50\mu s$, $100\mu s$, $200\mu s$ (from top to bottom)

Table 9.1: Additional overhead introduced per monitor for different deadline periods and total monitor numbers, using 1 thread

mon (#)/dp (us)	33	50	100	200
2	10.02	6.42	4.52	4.77
4	9.03	7.84	3.94	4.54
8	13.42	8.95	4.82	4.29
16	27.17	11.85	6.39	4.7
avg	14.91	8.77	4.92	4.58

The calculations are done according to Equation 9.1. Within this equation the function 'avg' takes two parameters that are used to select the average of a group of 30 test runs. The first parameter is the amount of monitors of the test and the second parameter is de deadline period of the test. The average subtracted is the average of having 1 deadline monitor. 1 deadline monitor is chosen to only include the costs of introducing the additional monitors. The 2.63μs refers to the average overhead introduced for a monitor without a deadline determined earlier.

$$for_{dp\ in\ 33,50,100,200}(for_{mon\ in\ 2,4,8,16}(\frac{avg(mon, dp) - avg(mon_1, dp)}{y-1} - 2.63)) \quad (9.1)$$

As expected table 9.1 shows that the average added overhead increases when a lower deadline monitor period is used. Since a lower period results in more deadline checking, the actual program has less time to execute. Furthermore the table shows an increase in time per monitor when more monitors are added with the same deadline (when the tests with 2 monitors are excluded). Checking more monitors within the same period of time leaves even less time for the actual program to execute. This is the reason why the overhead per monitor increases when more monitors are defined and the relation is quadratic.

The following example demonstrates the effect described above. Checking a deadline of a monitor costs 1μs and we have 2 monitors and a deadline period of 10μs. Within each 10μs, 2μs is used for checking deadlines thus 8μs remains for the program. The task to be performed takes 800μs. To perform this task while monitoring the deadline takes 100 cycles of 10μs resulting in a total execution time of 1000μs. The number of monitors is increased from 2 to 6. This results in 6μs overhead and thus 4μs computation time each 10μs. The same task now takes 200 cycles and results in a total execution time of 2000μs. The overhead introduced is gone from 200μs

Table 9.2: Expected execution times and experienced execution times

mon (#)/dp (us)	33	50	100	200
2	542.58 (549.49)	530.94 (542.48)	520.10 (529.41)	514.85 (523.16)
4	585.99 (571.83)	559.74 (564.83)	536.42 (541.96)	525.47 (537.26)
8	693.63 (649.17)	625.57 (614.49)	571.17 (574.41)	547.38 (564.16)
16	1060.89 (983.89)	803.71 (750.63)	650.62 (657.61)	594.04 (625.66)

to 1200μs while the number of monitors is increased by a factor 3. Equation 9.2 describes this relation. Within the example the equation results in the following expected execution times: i) $\frac{800}{10-(1*2))} * 10 = 1000$ and ii) $\frac{800}{10-(1*6))} * 10 = 2000$.

$$expected\ overhead = \frac{t_{execution}}{t_{dp} - (t_{mon} * n))} * t_{dp} \tag{9.2}$$

Within Equation 9.2 all units are μs, furthermore $t_{execution}$ represents the execution time without monitoring deadlines, t_{dp} represents the deadline period, t_{mon} the overhead introduced by 1 monitor and n represents the number of monitors.

Equation 9.2 is perfect for the example the usage of the framework introduces two addition parameters: i) the overhead for tracking the execution scenario ($t_{dp} + n * 2.63$) and ii) the overhead of having the monitor deadline thread (t_{thread}).

$$expected\ overhead = \frac{t_{execution} + t_{thread} + n * 2.63}{t_{dp} - (t_{mon} * n))} * t_{dp} \tag{9.3}$$

Table 9.2 depicts the expected execution times and between parentheses the experienced execution times of several test runs. To estimate the overhead of one monitor the following calculation is performed: i) the average overheads per monitor is divided by the number of interruptions for all the deadlines and ii) from these averages the average is taken. This results in an average of approximately 1μs (t_{mon}), this overhead is consistent with the overhead experienced during ad-hoc tests. The average execution time without monitoring deadlines is 494.44μs ($t_{execution}$). Furthermore an overhead of 10μs is introduced for having the deadline thread (t_{thread}), this overhead is among others caused by checking the periodic timer of the deadline thread. The conclusion is that this formula can be used as an indicator for the expected execution time.

The overhead introduced by the algorithm including checking the deadlines has a quadratic relation with the number of monitors and the deadline period. The

quadratic relation becomes a linear relation when the algorithm excluding the deadline monitor implementation is used, see Section 7.3.2. This is at the cost of an increased deadline monitor period. Tests show that having a waiting time of 100μs introduces 4.88μs overhead for each monitor. During the execution 5 interruptions occur, and the time per interruption is approximately 1μs.

9.1.3 Timer jitter

Figure 9.3 presents the jitter on the executions during the test runs using 1 thread. The jitter is calculated by:

$$\frac{\sum_{i=1}^{n} |(start_i - start_{i-1}) - 1000|}{n} \tag{9.4}$$

In this formula i represents the execution number, 0 represents the first execution, n represents the total number executions and 1000 represents the number of microseconds which is expected between each execution.

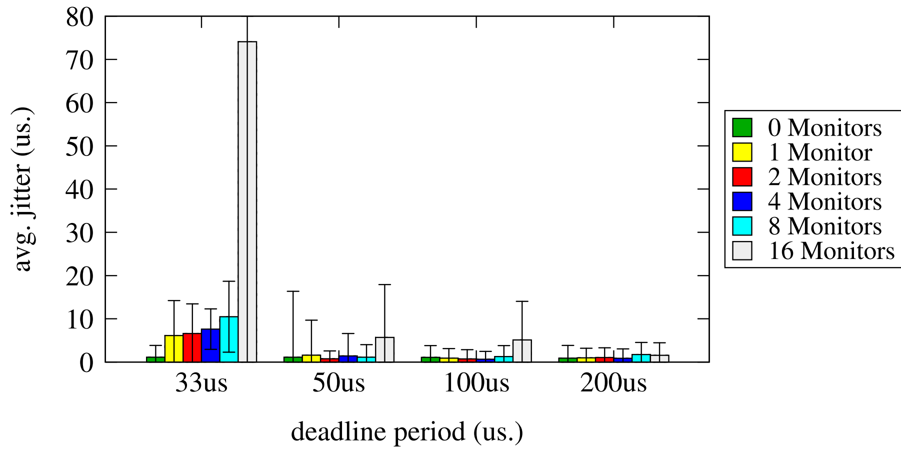


Figure 9.3: Timer jitter with deadline and without deadline monitor

The figure shows that the deadline period combined with the number of monitors with a deadline have the biggest impact on jitter. This is expected, because if the frequency of checking the deadlines is increased and the duration of performing the check increases, the chance that the check interferes with the other timers also increases.

It is also notable that the standard deviation increases when jitter increases. This is caused by the system attempting to keep up with the program. The ability of the system to keep up fluctuates and thus the standard deviation increases, as does the jitter.

9.1.4 Reusability of the framework

The framework can be reused in several ways. A developer can use the code and manually create instances of the threads and monitors. The recommended way is to create a model in the extended version of AADL. Source code is generated from this model which creates the threads and the monitors.

While generating code, the framework ensures threads use the specified execution frequency and defines monitors to track the defined execution scenarios. These monitors can be used by the developer to validate the specified real-time requirements within the AADL model and/or to perform special code in case a deadline is missed or a special scenario occurred.

9.1.5 Reusability of formal analysis tools

During literature research two tools were found which aid with formally verifying a AADL model. The next sections describe to usefulness and limitations of these tools and how the tools can be used together with the proposed solutions in this thesis.

Cheddar

The Cheddar tool is able to schedule the threads specified in the AADL model according to the POSIX standard. This fits very well with the proposed solution since the framework also uses POSIX. Cheddar uses the execution times supplied by the developer to schedule the threads. These values can be tested using the proposed solution resulting in an formal analysis which uses tested execution timings.

There are however some limitations in using Cheddar. The tool does not parse the unit which is supplied with the execution timing. Thus mixing different units within a single AADL model will result in incorrect validations. Furthermore at this moment Cheddar does not support the proposed monitor extension, thus these cannot be automatically verified.

Furthermore the tool is able to read AADL specification. However we did not encounter any documentation regarding how Cheddar interprets the AADL specifications. This causes automatic integration to be out of scope.

OSATE: Flow analysis

Open-Source AADL Tool Environment (OSATE) does not provide any tests regarding schedulability of threads. However the tool-suite provides a latency test for the flows defined in AADL. The initial idea is to use the created AADL model to automatically generate flows for each monitor which can than be analysed. However the flows are not aware of the notion of threads and the POSIX standard therefore the initial idea is not realised.

9.2 Case study

This section presents the evaluation of the case study. First an informal evaluation of the performance of the case study is provided; second a list of prerequisites is given for real-world usage.

9.2.1 Performance evaluation

As presented in Chapter 8, the case study conforms to the real-time requirements. Regarding the Freedom6S, during an extensive test-run of 10 hours consisting of $36,000,000$ executions, the highest execution time noted is 680μs and 0 deadlines are missed. The average execution time of this test is 519.18μs this time includes the overhead introduced for checking the deadlines. The average execution time of the tests without deadline monitors is 494.44μs.

While the requirements for the updates from the VectorVision are unknown, it is expected that a higher frequency is more desirable. During the tests the frequency in which the VectorVision is able to deliver the coordinates has proven to be very unpredictable.

9.2.2 Prerequisites for real-world usage

The following are prerequisites for real-world usage of the case study:

- Mechanical fixation of both coordinate systems. At the moment it is very easy to introduce errors in the transformation between the coordinate systems since the markers are placed on a ad-hoc bases. It is recommended to place markers on the handle of the Freedom6S to ease this integration and make it more reliable.

- Obtain or create an algorithm to calculate the distance from the location of the tool to a complex shape within reasonable time. Currently the only shape supported is a ball. For real-world usage this is insufficient. During literature research H-Collide [15] was encountered, which might be used.
- Investigate how much force surgeons desire at what distances and if this amount is equal for all surgeons.
- Integrate with software which is capable of defining treatment plans.
- Investigate how hard the real-time requirements are. During the research it seemed impossible to feel the difference between 900 and 1000 executions, but is this also true for surgeons and if so what are the limits.
- Investigate how to apply the force when the VectorVision provides new coordinates. At the moment this can cause a huge change in force that might be undesired during a surgery.
- Investigate the impact of brain shifting. If a surgeon relies on the force from the haptic device and a brain shift occurs the results could be severe.

9.3 Conclusion

This chapter answers research question 4 and 5. Research question 4 is: *How much overhead does the proposed solution introduce and how does the solution scale?* The introduced overhead for each monitor without considering deadlines on the test set-up is on average 2.63μs, each time a method that is in the execution scenario executes. If a deadline is specified, each deadline check takes approximately 1μs (assuming the test set-up is used).

How the solution scales depends on which deadline timing algorithm is used. The algorithm in which the period includes checking the deadlines scales quadratic and the algorithm excluding scales linear. The following equation estimates the expected execution time for the algorithm that includes checking the deadlines within the period:

$$expected\ overhead = \frac{t_{execution} + t_{thread} + n * 2.63}{t_{dp} - (t_{mon} * n))} * t_{dp} \tag{9.5}$$

Within this equation all timing units are μs and $t_{execution}$ represents the execution time without monitors, t_{thread} the overhead of having the deadline thread, t_{dp} the deadline period, t_{mon} the overhead introduced by 1 monitor, n the number of monitors and 2.63μs the overhead for tracking the execution scenario.

Tests show that timer jitter is most affected by the total load of the system. When the system approaches full load the amount of jitter increases rapidly. This also has a huge impact on the standard deviations of the execution timings. Both these numbers can be used as indicators for a system having high load.

Research question 5 is: *Are there existing formal validation tools which can be used with the proposed solution?* There is one tools which can be used: Cheddar. Cheddar is the most suitable tool since it attempts to schedule threads according to the POSIX standard and this standard is also used in the proposed framework. Furthermore the tool is able to read AADL models however details regarding which properties are used are missing, therefore an automatic integration is considered out of scope.

Chapter 10

Conclusions

This chapter presents the conclusions of this thesis, which summarizes the proposed solution. The main problem statement is: *Developers have no reusable tool-supported approach for testing real-time requirements within integrated systems based on empirical evidence.*

The solution design started with the design of a DSL that could be used to test real-time requirements using empirical evidence. The resulting DSL was compared with AADL and they showed great resemblance. Because AADL is an industry standard and existing tools can be reused, it has been decided to create a framework which supports testing of real-time requirements specified in AADL. The resulting framework can be used with an arbitrary AADL model, the framework ignores AADL features that it does not support. The resulting framework can be reused for testing real-time requirements of other systems which are modelled in AADL.

AADL is extended with the ability to specify monitors, which monitor an execution scenario with an optional deadline. This enables listeners of the monitors to be called upon monitor start, finish, strict violation and deadline miss. The developer specifies which behaviour is desired at these events by implementing a monitor listener.

The framework is demonstrated in a case study. The goal of the case study is to integrate a haptic and a neuronavigation device. The haptic device requires at least 1000 updates a second, this implies a maximum execution time of 1000μs. Using the framework, the thread that calls the update method is created, started and monitored. During a test run consisting of $36,000,000$ executions the highest encountered execution time is 680μs and no deadlines were missed.

Evaluation shows that on the test set-up a monitor without considering deadline checking introduces on average 2.63μs overhead each time a method in its execution scenario is called. Performing a deadline check takes 1μs for each deadline to be

checked on the test set-up. The overhead for checking deadlines scales linear or quadratic with respect to the number of deadlines and deadline period, depending on the deadline period algorithm used. Timer jitter within mainly depends on the deadline period and the number of monitors with a deadline. A high jitter is an indicator for a system which cannot keep up.

10.1 Answers to research questions

Question 1*: What are the requirements for a DSL aimed at the integration of devices with hard real-time requirements?*

The requirements for such a DSL are the notion of device, thread, API and the ability to specify timing properties. The research question is partially unanswered because the requirements for the DSL have been scoped to only include the essential constructs for dynamic performance analysis.

Question 2*: Is there an existing real-time oriented DSL which conforms to the requirements or is extendible to become conforming?*

There is an existing real-time oriented DSL which is extendible to become conforming with the requirements, the most suitable DSL is AADL. Since AADL is an industry standard it is assumed that all major constructs are included.

Question 3*: How can the resulting DSL be extended and used to test real-time requirements?*

The DSL is extended by extending the DSL's meta-model. AADL is extended to support the notion of monitors. Using the extended AADL model and a code generation model, code is generated to be used with the proposed framework. The monitor listeners provide the means to test the real-time requirements.

Question 4*: How much overhead does the proposed solution introduce and how does the solution scale?*

The introduced overhead for each monitor without considering deadlines is on average 2.63μs, each time a method which is in the execution scenario executes. If a deadline is specified, each deadline check takes approximately 1μs (assuming the test set-up is used).

How the solution scales depends on which deadline timing algorithm is used. The algorithm in which the period includes checking the deadlines scales quadratic and the algorithm excluding scales linear. The following equation estimates the expected

execution time for the algorithm that includes the deadlines checks in the period:

$$expected\ overhead = \frac{t_{execution} + t_{thread} + n * 2.63}{t_{dp} - (t_{mon} * n)} * t_{dp} \tag{10.1}$$

Within this equation all timing units are μs and $t_{execution}$ represents the execution time without monitors, t_{thread} the overhead of having the deadline thread, t_{dp} the deadline period, t_{mon} the overhead introduced by 1 monitor, n the number of monitors and 2.63μs the overhead for tracking the execution scenario.

Tests show that timer jitter is affected most by the total load of the system. When the system approaches full load the amount of jitter increases rapidly. This also has a huge impact on the standard deviations of the execution timings. Both these numbers can be used as indicators for a system having high load.

Question 5: Are there existing formal validation tools which can be used with the proposed solution?

There is one tool which can be used: Cheddar. Cheddar is the most suitable tool since it attempts to schedule threads according to the POSIX standard and this standard is also used in the proposed framework. Furthermore the tool is able to read AADL models, however details regarding which properties are used are missing, therefore an automatic integration is considered out of scope.

Chapter 11

Recommendations & future work

This chapter presents recommendations and future work regarding the framework and monitors. Recommendations and future work regarding the case study are presented in Section 9.2.

During the evaluation each test run consists of 1,800,000 executions. The result of the size of this test is that standard deviations are low. 1,800,000 is arbitrary picked and is not based on statistic calculations. When using this framework it is recommended to investigate the deviation of the taken measurements to determine the distribution. The distribution can be used to determine the minimal size of the experiments.

The framework uses the file descriptor timers provided by the Linux kernel. While the jitter of these timers proved to be stable when using reasonable values for the interval, the timer internally uses polling. Jitter is likely to improve when hardware timers combined with interrupt routines are used. It is therefore recommended to investigate the feasibility of using these interrupts. The main challenge using hardware timers will be the miss-match between the number of deadlines to monitor and the number of hardware timers available. This can maybe be solved by keeping a sorted list of what to do when. This list can be used in combination with one hardware timer.

It is recommended to validate proper working of the file descriptor timer which is provided by the Linux kernel. During the tests on both cores of the system the timer, on random intervals, failed to trigger for somewhere between 5 and 10ms. During the tests on the same system but only using a single core we didn't encountered any hiccups using the file descriptor timer. It might be an improvement if one core can

be dedicated to checking the deadlines to limit the influence of these checks.

The framework currently supports a subset of the features offered by AADL and the monitor extension. In the future, it is recommended to add at least the notion of processes to the framework. To investigate which features are most interesting, existing projects that are modelled using AADL can be used as a source for finding the most important features.

It is recommended to investigate if it is desirable to have two deadline threads with a different period or to change the period of the deadline monitor on the fly. This can be used to control the overhead introduced by the monitors.

The framework is currently based on a Linux kernel which has been patched with RT and PREEMPT. While this in theory provides hard real-timeness, we were unable to test if the systems truly is a hard real-time system. It is therefore recommended to investigate the real-time properties.

Bibliography

[1] Atom. Atom (programming language). [Online]. Available: http://en.wikipedia.org/wiki/Atom_(programming_language) , last visited at April 15th 2010,

[2] T. J. Biggerstaff, "A perspective of generative reuse," *Ann. Software Eng.*, vol. 5, pp. 169–226, 1998.

[3] J.-P. Bodeveix, R. Cavallero, D. Chemouil, M. Filali, and J.-F. Rolland, "A mapping from aadl to java-rtsj," in *JTRES*, ser. ACM International Conference Proceeding Series, G. Bollella, Ed. ACM, 2007, pp. 165–174.

[4] A. Bonenfant, C. Ferdinand, K. Hammond, and R. Heckmann, "Worst-case execution times for a purely functional language," in *Proc. Implementation of Functional Languages (IFL 2006)*, ser. Lecture Notes in Computer Science, vol. 4449. Springer, 2007, to appear.

[5] Brainlab. Brainlab homepage. [Online]. Available: http://www.brainlab.com , last visited at July 21th 2010,

[6] R. D. Buchholz, D. D. Yeh, J. Trobaugh, L. L. McDurmont, C. D. Sturm, C. Baumann, J. M. Henderson, A. Levy, and P. Kessman, "The correction of stereotactic inaccuracy caused by brain shift using an intraoperative ultrasound device," in *CVRMed*, ser. Lecture Notes in Computer Science, J. Troccaz, W. E. L. Grimson, and R. Mösges, Eds., vol. 1205. Springer, 1997, pp. 459–466.

[7] R. Burgman, E. Mols, J. Tijben, W. de Vries, and J. Wolfswinkel, "Koppeling freedom 6s met vectorvision," Bachelor Thesis, July 2008, dutch.

[8] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*, 4th ed. Addison Wesley, May 2009. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0321417453

[9] G. C. Buttazzo, *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Santa Clara, CA, USA: Springer-Verlag TELOS, 2004.

[10] H. C. Cunningham, “A little language for surveys: constructing an internal dsl in ruby,” in *ACM-SE 46: Proceedings of the 46th Annual Southeast Regional Conference on XX*. New York, NY, USA: ACM, 2008, pp. 282–287.

[11] G. Eggers, J. Mühling, and R. Marmulla, “Image-to-patient registration techniques in head surgery,” *International Journal of Oral and Maxillofacial Surgery*, vol. 35, no. 12, pp. 1081 – 1095, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/B6WGW-4MBBYX5-2/2/8cc0ee6af86d524950694e12ec1d3f6f

[12] Ellidiss. Hood & stood. [Online]. Available: http://www.ellidiss.fr/ , last visited at April 28th 2010,

[13] E. Geertsema, J. Jochem, M. Rots, and A. Veen, “Visual and haptic feedback in surgery of the temporal bone,” Bachelor Thesis, June 2009, dutch.

[14] O. Gilles and J. Hugues, “Validating requirements at model-level,” in *Ingénierie dirigée par les modèles (IDM'08)*, Mulhouse, France, Jun. 2008, pp. 35–49.

[15] A. Gregory, M. C. Lin, S. Gottschalk, and R. Taylor, “H-collide: A framework for fast and accurate collision detection for haptic interaction,” in *In Proceedings of Virtual Reality Conference*, 1999, pp. 38–45.

[16] S. Gui, L. Luo, Y. Li, and L. Wang, “Formal schedulability analysis and simulation for aadl,” in *ICESS '08: Proceedings of the 2008 International Conference on Embedded Software and Systems*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 429–435.

[17] G. Gülesir, “Evolvable behavior specifications using context-sensitive wildcards,” Ph.D. dissertation, University of Twente, Enschede, March 2008. [Online]. Available: http://doc.utwente.nl/58767/

[18] K. Hammond and G. Michaelson, “The design of hume: A high-level language for the real-time embedded systems domain,” in *Domain-Specific Program Generation*, ser. Lecture Notes in Computer Science, C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, Eds., vol. 3016. Springer, 2003, pp. 127–142.

[19] T. Hawkins. atom: A dsl for embedded hard realtime applications. [Online]. Available: http://hackage.haskell.org/package/atom-1.0.2 , last visited at April 14th 2010,

[20] M. Hendriks, L. Lutkenhaus, F. van Meulen, S. Steen, and W. van der Weg, “Haptic feedback and navigation during neurosurgical interventions,” Bachelor Thesis, June 2008.

[21] C. Jones, “Spr programming languages table. release 8.2,” http://www.mindspring.com/~dway/smalltalk/docs/0langtbl.pdf, March 1996. [Online]. Available: http://www.mindspring.com/~dway/smalltalk/docs/0langtbl.pdf

[22] M. Karlsch, "A model-driven framework for domain specific languages," Master's thesis, Hasso-Plattner-Institute of Software Systems Engineering, Potsdam, Germany, 2007.

[23] J. Kim, H. Kim, B. K. Tay, M. Muniyandi, M. A. Srinivasan, J. Jordan, J. Mortensen, M. Oliveira, and M. Slater, "Transatlantic touch: a study of haptic collaboration over long distance," *Presence: Teleoper. Virtual Environ.*, vol. 13, no. 3, pp. 328–337, 2004.

[24] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, "Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications," in *Ada-Europe*, ser. Lecture Notes in Computer Science, F. Kordon and Y. Kermarrec, Eds., vol. 5570. Springer, 2009, pp. 237–250.

[25] Lifehealthcare. Vectorvision. [Online]. Available: http://www.lifehealthcare.com.au/products/digitaloperatingsuitesimageguidedsystems/imageguidedsystems/brainlab-vectorvisioncompact.aspx , last visited at July 21th 2010,

[26] L. Ma, S. Gui, L. Luo, and L. Yin, "Research of automatic code generating technology based on aadl," in *ICESSSYMPOSIA '08: Proceedings of the 2008 International Conference on Embedded Software and Systems Symposia.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 136–141.

[27] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005.

[28] P. Munger, "Accuracy conciderations in mr image-guided neurosurgery," Master's thesis, McGill University, Montreal, Canada, 1994.

[29] Obeo. Acceleo. [Online]. Available: http://www.acceleo.org/pages/home/en , last visited at May 25th 2010,

[30] OMG. Marte. [Online]. Available: http://www.omgmarte.org/ , last visited at April 27th 2010,

[31] SAE, "Architecture analysis & design language (as5506)," 2004.

[32] O. Sokolsky, I. Lee, and D. Clarke, "Process-algebraic interpretation of aadl models," in *Ada-Europe '09: Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies.* Berlin, Heidelberg: Springer-Verlag, 2009, pp. 222–236.

[33] H. Sözer, "Architecting fault-tolerant software systems," Ph.D. dissertation, University of Twente, Enschede, January 2009, iPA Dissertation 2009-05. [Online]. Available: http://doc.utwente.nl/60513/

[34] A. van Deursen and P. Klint, "Domain-specific language design requires feature descriptions," *Journal of Computing and Information Technology*, vol. 10, p. 2002, 2001.

[35] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[36] R. Varona-Gomez and E. Villar, "Aadl simulation and performance analysis in systemc," *Engineering of Complex Computer Systems, IEEE International Conference on*, vol. 0, pp. 323–328, 2009.

[37] R. J. Wieringa, *Design Methods for Reactive Systems: Yourdon, Statemate and the UML*. Morgan Kaufmann Publishers, 2003.

Part V

Appendixes

Appendix A

Acronyms

AADL Architecture Analysis & Design Language

ACET Average Case Execution Time

API Application Programming Interface

ANTLR ANother Tool for Language Recognition

CBO Coupling Between Objects

CORBA Common Object Request Broker Architecture

DSL Domain-Specific Language

EMF Eclipse Modelling Framework

FSM Finite State Machine

GPL General Purpose Language

HOOD Hierarchical Object Oriented Design

OMG Object Management Group

OSATE Open-Source AADL Tool Environment

POSIX Portable Operating System Interface [for Unix]

MARTE Modelling and Analysis of Real-time and Embedded systems

MDE Model Driven Engineering

MOF Meta-Object Facility

REAL Requirement Enforcement Analysis Language

SAE Society of Automotive Engineers

UML Unified Modelling Language

WCET Worst Case Execution Time

Appendix B

Integration framework generated code

```
#ifndef FREEDOM6SGETHANDLELOCATIONANDSETFORCETHREADTHREAD_H_
#define FREEDOM6SGETHANDLELOCATIONANDSETFORCETHREADTHREAD_H_

#include "../thread/PeriodicThread.h"
#include "../util/ServiceLocator.h"
#include "../monitor/MonitorManager.h"
#include "MonitorMethods.h"

#include "../application/controller/Freedom6s.h"

class Freedom6sGetHandleLocationAndSetForceThread : public PeriodicThread {

private:
  Freedom6s * Freedom6s_p;
  MonitorManager * mm;
public:

  Freedom6sGetHandleLocationAndSetForceThread(Property<Time> *interval) :
      PeriodicThread(interval) {
    this->Freedom6s_p = static_cast<Freedom6s *>(ServiceLocator::Instance()->
        get("Freedom6s"));
    this->mm = static_cast<MonitorManager *>(ServiceLocator::Instance()->get("
        mm"));
  }

  void * invoke() {
    mm->methodCall(mm_Freedom6sGetHandleLocationAndSetForceThread, true, this
        ->getId());
    this->Freedom6s_p->getLocationSetForce();
    mm->methodCall(mm_Freedom6sGetHandleLocationAndSetForceThread, false, this
        ->getId());
    return 0;
  }
};

#endif /* FREEDOM6SGETHANDLELOCATIONANDSETFORCETHREADTHREAD_H_ */
```

Appendix C

Acceleo template

```
[module generate('http:///RT/component', 'http:///RT/core')/]

[template public name(impl : ThreadImpl)]
[impl.compType.name.concat(impl.compute_entrypoint.toUpperFirst().concat('
    Thread'))/]
[/template]

[template public generate(a : AadlSpec)]
  [comment @main /]
  [MonitorMethodsInit()/]
  [ThreadsHeaderInit()/]
  [ThreadsSourceInit()/]
  [MonitorsSourceInit()/]
  [createThreadObjectFiles(a)/]
  [MonitorsSourceFinish()/]
  [ThreadsSourceFinish()/]
  [ThreadsHeaderFinish()/]
  [MonitorMethodsFinish()/]
[/template]

[template public createThreadObjectFiles(a : AadlSpec)]
  [MonitorsHeader(a)/]
  [for (impl : ThreadImpl | a.threadImpl)]
    [MonitorMethods(impl)/]
    [Threads(impl)/]
    [ThreadsImport(impl)/]
    [if (defineDeadlineMonitor)]
      [MonitorImplicit(impl)/]
    [/if]
[file (name(impl).concat('.h'), false)]
#ifndef [name(impl).toUpper()/]THREAD_H_
#define [name(impl).toUpper()/]THREAD_H_

#include "../thread/PeriodicThread.h"
#include "../util/ServiceLocator.h"
#include "../monitor/MonitorManager.h"
#include "MonitorMethods.h"

#include "../application/controller/[impl.compType.name/].h"

class [name(impl)/] : public PeriodicThread {
```

```
private:
  [impl.compType.name/] * [impl.compType.name/]_p;
  MonitorManager * mm;
public:

  [name(impl)/](Property<Time> *interval) : PeriodicThread(interval) {
    this->[impl.compType.name/]_p = static_cast<[impl.compType.name/] *>(
        ServiceLocator::Instance()->get("[impl.compType.name/]"));
    this->mm = static_cast<MonitorManager *>(ServiceLocator::Instance()->get("
        mm"));
  }

  void * invoke() {
    mm->methodCall(mm_[name(impl)/], true, this->getId());
    this->[impl.compType.name/]_p->[impl.compute_entrypoint/]();
    mm->methodCall(mm_[name(impl)/], false, this->getId());
    return 0;
  }
};

#endif /* [name(impl).toUpper()/]THREAD_H_ */
  [/file]
  [/for]
  [for (monitor : Monitor | monitor)]
    [MonitorExplicit(monitor)/]
  [/for]
[/template]

[template MonitorMethodsInit()]
  [file ('MonitorMethods.h', false)]
#ifndef MONITORMETHODS_H_
#define MONITORMETHODS_H_

enum MonitorMethods {
  [/file]
[/template]

[template MonitorMethods(impl : ThreadImpl)]
[file ('MonitorMethods.h', true)]
  mm_[name(impl)/],
[/file]
[/template]

[template MonitorMethodsFinish()]
  [file ('MonitorMethods.h', true)]
  MonitorMethodsAmount
};

#endif /* MONITORMETHODS_H_ */
  [/file]
[/template]

[template ThreadsImport(impl : ThreadImpl)]
[file ('Threads.h', true)]
#include "[name(impl)/].h"
[/file]
[/template]

[template ThreadsHeaderInit()]
  [file ('Threads.h', false)]
#ifndef THREADS_H_
```

```
#define THREADS_H_

#include "../property/BoundedProperty.h"
#include "../property/LowerBoundedProperty.h"
#include "../property/UpperBoundedProperty.h"
#include "../thread/ThreadManager.h"

void setUpThreads(ThreadManager *tm);
  [/file]
[/template]

[template ThreadsSourceInit()]
  [file ('Threads.cpp', false)]
#include "Threads.h"

void setUpThreads(ThreadManager *tm) {
  [/file]
[/template]

[template Threads(impl : ThreadImpl)]
[file ('Threads.cpp', true)]
  Time t_[name(impl)/];
  t_[name(impl)/].setTime([impl.period/], 0);
  [name(impl)/] * thread_[name(impl)/] = new [name(impl)/](new BoundedProperty
      <Time> (t_[name(impl)/], t_[name(impl)/], t_[name(impl)/]));
  tm->registerThread(thread_[name(impl)/]);
[/file]
[/template]

[template ThreadsSourceFinish()]
  [file ('Threads.cpp', true)]
}
  [/file]
[/template]

[template ThreadsHeaderFinish()]
  [file ('Threads.h', true)]

#endif /* THREADS_H_ */
  [/file]
[/template]


[template MonitorsHeader(a : AadlSpec)]
  [file ('Monitors.h', false)]
#ifndef MONITORS_H_
#define MONITORS_H_

#include <vector>
#include "../monitor/MonitorManager.h"
#include "../monitor/MonitorListener.h"
#include "../util/Logger.h"
[for (ml : MonitorListener | a.monitorListener)]
#include "../monitor/[ml.name/].h"
[/for]

void setUpMonitors(MonitorManager *mm, std::vector<Logger * > *loggers);

#endif /* MONITORS_H_ */
  [/file]
[/template]
```

```
[template MonitorsSourceInit()]
  [file ('Monitors.cpp', false)]
#include "Monitors.h"

void setUpMonitors(MonitorManager *mm, std::vector<Logger * > *loggers) {
  [/file]
[/template]

[template MonitorImplicit(impl : ThreadImpl)]
[file ('Monitors.cpp', true)]
  MethodCall *mc_[name(impl)/] = new MethodCall(mm_[name(impl)/], 1);
  timeval *monitorTimeout_[name(impl)/] = new timeval();
  monitorTimeout_[name(impl)/]->tv_sec = 0;
  monitorTimeout_[name(impl)/]->tv_usec = [impl.deadline/];
  Monitor *m_[name(impl)/] = new Monitor(mc_[name(impl)/], true, true,
      monitorTimeout_[name(impl)/]);

  [for (ml : MonitorListener | impl.listeners)]
    ostringstream logger_file_[name(impl)/]_[i/];
    logger_file_[name(impl)/]_[i/] << "[name(impl)/]_[i/]";
    Logger *logger_[name(impl)/]_[i/] = new Logger("log", logger_file_[name(
        impl)/]_[i/].str().c_str());
    loggers->push_back(logger_[name(impl)/]_[i/]);
    [ml.name/] *ml_[name(impl)/]_[i/] = new [ml.name/](m_[name(impl)/],
        logger_[name(impl)/]_[i/]);
    m_[name(impl)/]->addListener(ml_[name(impl)/]_[i/]);
  [/for]

  mm->registerMonitor(m_[name(impl)/]);
[/file]
[/template]



[template MonitorExplicit(monitor : Monitor) post (trim())]
[file ('Monitors.cpp', true)]
  MethodCall *mc_[monitor.name/] = new MethodCall(mm_[name(monitor.startMethod
      .call)/], [monitor.startMethod.minimum/]
  [for (mc : MethodCall | monitor.startMethod.eAllContents())]
    , new MethodCall(mm_[name(mc.call)/], [mc.minimum/]
  [/for]
  [for (mc : MethodCall | monitor.startMethod.eAllContents())]
    )
  [/for]
  );
  timeval *monitorTimeout_[monitor.name/] = new timeval();
  monitorTimeout_[monitor.name/]->tv_sec = 0;
  monitorTimeout_[monitor.name/]->tv_usec = [monitor.deadline/];
  Monitor *m_[monitor.name/] = new Monitor(mc_[monitor.name/], [monitor.
      startMeasurementBegin/], [monitor.finishMeasurementEnd/],
      monitorTimeout_[monitor.name/]);

  [for (ml : MonitorListener | monitor.listeners)]
    ostringstream ml_name_[monitor.name/]_[i/];
    ml_name_[monitor.name/]_[i/] << "[monitor.name/]_[i/]";
    Logger *logger_[monitor.name/]_[i/] = new Logger("log", ml_name_[monitor.
        name/]_[i/].str().c_str());
    loggers->push_back(logger_[monitor.name/]_[i/]);
    [ml.name/] *ml_[monitor.name/]_[i/] = new [ml.name/](m_[monitor.name/],
        logger_[monitor.name/]_[i/]);
    m_[monitor.name/]->addListener(ml_[monitor.name/]_[i/]);
  [/for]
```

```
  mm->registerMonitor(m_[monitor.name/]);
[/file]
[/template]

[template MonitorsSourceFinish()]
  [file ('Monitors.cpp', true)]
}
  [/file]
[/template]
```

Appendix D

Process method call pseudo code

The process of the monitors starts with the generated code from the AADL models, see Listing D.1. Within this code the monitor manager is called before and after the actual method call.

The next step in the process is presented in Listing D.2. The MonitorManager first determines the exact time the method was called. Next a mutex must be obtained. This is done to ensure that all the monitors perceive the execution of methods in the same order. When the mutex is obtained all the monitors are called providing them with: i) which method, ii)before or after iii) at what moment in time it is called, iv) which thread.

Listing D.3 presents the algorithm of the monitor. The algorithm keeps track of the time the monitor entered and when it finished as well as the progress in the execution scenario.

Listing D.1: The generated code based on the AADL model

```
//threadId is the id of this thread

MonitorManager->methodCall(A, pre, threadId);
actual method call;
MonitorManager->methodCall(A, !pre, threadId);
```

Listing D.2: The process monitor manager methodCall(method, when, threadId) Algorithm

```
//obtain time so all monitors perceive the same time
time = currentTime();
//obtain mutex lock so all monitors perceive same behaviour
mutex_lock();
for(monitor which has this method) {
```

```
  processMethodCall(method, when, time, threadId);
}
mutex_unlock();
```

Listing D.3: The processMethodCall(methodCalled, when, time, threadId) Algorithm

```
//pre is true when the methodCall is before the actual method otherwise it is
    false
//post is true when the methodCall is after the actual method otherwise it is
    false
//preThreadId is the id from the thread which called pre
//expectPost is true when a pre is received

if(currentState == methodCalled) {
  //timer administration
  if(currentState == startState && amount == 0) {
    if(startTime == 0 &&
      (pre && monitorStartMeasurementBegin) ||
      (post && !monitorStartMeasurementBegin)) {
      startTime = time;

      if(hasDeadline) {
        setTimeout();
      }

      notifyAllListeners(monitorStart);
    }
  }

  if(!nextState && (amount+1) >= minimum) {
    if((pre && !monitorFinishMeasurementEnd) ||
      (post && monitorFinishMeasurementEnd  && expectPost && preThreadId ==
          threadId)) {
      endTime = time;

      if(this->timeout != 0) {
        clearTimeout();
      }

      notifyAllListeners(monitorFinish);

    }
  }

  //state transition
  if(post) {
    if(expectPost && preThreadId == threadId) {
      expectPost = false;
      preThreadId = -1;
      amount++;
      if(amount >= minimum) {
        if(endState) {
          //reset to initial state
          amount = 0;
          startTime = 0;
          endTime = 0;
          currentState = startState;
        } else {
          currentState = nextState;
        }
```

```
        }
      }
    } else {
      if(!expectPost) {
        expectPost = true;
        preThreadId = threadId;
      }
    }
} else {
  //if monitor started and the current method is strict
  if(this->current != 0 && current->getStrict()) {
    if(this->timeout != 0) {
      clearTimeout();
    }

    amount = 0;
    startTime = 0;
    endTime = 0;
    currentState = startState;

    notifyAllListeners(strictViolation);
  }
}
```