Design and implementation of a real-time embedded application

Thomas Baron Philippe Jean Gaël Mercier

January 8, 2007

Departement of Computer Science, Aalborg University

TITLE:
Design and implementation of a real-time embedded application

PROJECT PERIOD:
SSE3, September 4th 2006 - January 9th 2007

GROUP MEMBERS:
Thomas Baron
Philippe Jean
Gaël Mercier

SUPERVISOR:
Anders Peter Ravn

CUSTOMER REPRESENTATIVE:
Hans Søndergaard

NUMBER OF COPIES: 2

Abstract

This report states the steps of the development of a real-time embedded application using the Object-Oriented Analysis & Design method. It implements an industrial case that illustrates the use of an implementation of the Ravenscar-Java Profile.

Contents

1 Introduction **1**

2 Problem Domain Analysis **3**
- 2.1 Context . . . 3
- 2.2 System definition . . . 5
- 2.3 The FACTOR criteria . . . 5
- 2.4 Problem domain model . . . 5

3 Application Domain Analysis **9**
- 3.1 Use cases . . . 9
- 3.2 Functionalities . . . 10
  - 3.2.1 Turn On/Off . . . 10
  - 3.2.2 Acquirement . . . 10
  - 3.2.3 Temperature regulation . . . 11
  - 3.2.4 Monitoring . . . 12
- 3.3 Interfaces . . . 12
  - 3.3.1 `MirrorEngine` . . . 13
  - 3.3.2 `InfraRed` . . . 13
  - 3.3.3 `InfraRedDetector` . . . 13
  - 3.3.4 `TemperatureSensor` . . . 13
  - 3.3.5 `Thermostat` . . . 13
  - 3.3.6 `ExternalCommunicationDevice` . . . 14
- 3.4 Summary . . . 15

4 Design **16**
- 4.1 Design criteria . . . 16
- 4.2 Architecture . . . 17
- 4.3 Detailed Design . . . 19
  - 4.3.1 Device component . . . 19
  - 4.3.2 Logic component . . . 21
  - 4.3.3 Check component . . . 29
  - 4.3.4 Scheduling the Function Layer (Logic & Check) . . . 30
  - 4.3.5 Data component . . . 34
  - 4.3.6 Communication component . . . 36

5 Implementation & Testing **37**
- 5.1 Implementation . . . 37
  - 5.1.1 Software usage . . . 37
  - 5.1.2 Coding conventions . . . 37
- 5.2 Testing . . . 38
  - 5.2.1 Unit Testing . . . 38
  - 5.2.2 Integration Testing . . . 40
  - 5.2.3 Results of the schedulability . . . 41
  - 5.2.4 Acceptance Testing . . . 42
  - 5.2.5 Environment Model . . . 43

6 Conclusion **45**
6.1 Summary . . . 45
6.2 Discussion . . . 46
6.3 Further work . . . 46

Bibliography **47**

A CD-ROM enclosed **49**

List of Figures

2.1 Analysis instrument . . . 3
2.2 FTIR system rich picture . . . 3
2.3 Interferometer principle . . . 4
2.4 FTIR physical module. . . . 4
2.5 Overview of the classes in the system . . . 6
2.6 The TemperatureSensor Statechart Diagram . . . 6
2.7 The Thermostat Statechart Diagram . . . 6
2.8 The Interferometer Statechart Diagram . . . 7
2.9 The InfraRedDetector Statechart Diagram . . . 7
2.10 The MirrorEngine Statechart Diagram . . . 8
2.11 The InfraRed Statechart Diagram . . . 8
2.12 The ExternalCommunicationDevice Statechart Diagram . . . 8

3.1 Use cases . . . 9
3.2 System states . . . 11
3.3 Acquire . . . 12

4.1 The layered Architecture . . . 18
4.2 Class Diagram of the temperature sensor . . . 19
4.3 Class Diagram of the thermostat . . . 19
4.4 Class Diagram of the mirror engine . . . 19
4.5 Class Diagram of the infrared . . . 20
4.6 Class Diagram of the infrared detector . . . 20
4.7 Class Diagram of the external communication device . . . 20
4.8 Class Diagram of the temperature reading . . . 21
4.9 TemperatureReading Activity Diagram . . . 21
4.10 Class Diagram of the temperature regulation . . . 22
4.11 TemperatureRegulation Activity Diagram . . . 22
4.12 Class Diagram of the monitoring . . . 23
4.13 Monitoring Activity Diagram . . . 23
4.14 Class Diagram of the acquirement . . . 24
4.15 Acquirement Activity Diagram . . . 24
4.16 Scanning process Activity Diagram . . . 25
4.17 Finalisation process Activity Diagram . . . 25
4.18 Interrupting process Activity Diagram . . . 26
4.19 Class Diagram of the input communication . . . 27
4.20 InputCommunication Activity Diagram . . . 27
4.21 Class Diagram of the output communication . . . 28
4.22 OutputCommunication Activity Diagram . . . 28
4.23 Class Diagram of the check layer . . . 29
4.24 Watchdog Activity Diagram . . . 29
4.25 Class Diagram of the acquirement register . . . 34
4.26 Class Diagram of the acquirement mode . . . 34
4.27 Class Diagram of the interferogram . . . 35
4.28 Class Diagram of the logger and log . . . 35
4.29 Class Diagram of the temperature buffer . . . 35

4.30 Class Diagram of the PID 36
4.31 Class Diagram of the monitor register 36
4.32 Class Diagram of the communication layer 36

5.1 Classes Logger and LoggerTest 39
5.2 Example testLogger 39
5.3 LoggerTest standard output 39
5.4 Scenario Temperature Regulation 40
5.5 class TemperatureRegulationTest 41
5.6 Trace command line 41
5.7 TemperatureRegulationTest standard output 42

List of Tables

3.1 MirrorEngine operations . . . 13
3.2 InfraRed operations . . . 13
3.3 InfraRedDetector operations . . . 13
3.4 TemperatureSensor operations . . . 13
3.5 Thermostat operations . . . 14
3.6 ExternalCommunicationDevice operations . . . 14

4.1 Priority of design criteria . . . 16
4.2 Table of priorities using the DMS. . . . 31
4.3 Worst-case execution time for each shared resource . . . 32
4.4 Blocking time for processes . . . 33
4.5 Worst case response time (R) of the processes . . . 34

5.1 Thread capacities . . . 42

Chapter 1

Introduction

The work discussed in this report is part of the Ravenscar-Java Development project [4]. The aim of this project is to analyse whether the Ravenscar-Java profile is a realistic Java profile for industrial-time systems, to show how Real-Time UML is used as a design tool and to compare a Ravenscar-Java solution with a C++ solution.

The main aim of this project is to design an application to manage a part of an embedded system for the hardware provided by FOSS Analytical and to assess if it is suitable for it. It has to use new technologies as Java for Real-Time and modern methods of design oriented object.

In order to redesign and implement the solution, the Ravenscar-Java Profile will be used. The Ravenscar-Java Profile is a subset of Java language features designed for hard real-time systems. It is made as a subset that simplifies the RTSJ[1] [9], and has a lot of interesting properties for designing real-time systems, as for example predictability or analysable memory utilisation.

The first part of the Ravenscar-Java Development project was to implement the Ravenscar-Java profile in the way that it can be suitable for industrial cases. In order to meet with the project goal, the Ravenscar-Java profile was implemented on a Java processor, the aJ-100 [28]. This processor is developed by aJile Systems. The aJ-100 processor from aJile Systems is a 100 MHz direct execution Java processor. It is a pure Java micro controller that uses Java bytecode as its native instruction set. The implementation of the Ravenscar-Java profile only uses one of two JVM units.

However, a Real-Time Platform does not say how to design, implement and test an application. We need to follow a analysis and development method, adapting the HRT HOOD[2] [13] one.

We have to answer in the work to :

1. How to design a system with an object oriented method ?
2. How to use the Ravenscar-Java Profile in this industrial case ?
3. Is the implementation suitable for the case (testing results. . .) ?

Related work

Our work is established on some fundamental documents about the Java Ravenscar profile [16], [22]. They are themselves based on the RTSJ (Real-Time Specification for Java) [10] and the Ada Ravenscar profile [7], [12]. We can also find articles about the Real Time Object Oriented Design itself [13], [19]. All these articles are the theoretical background of our subject.

One of our lecture was about a comparison of Ada and Java for Real-Time [24] and we could had some comment on it. Ada is used for a long time in embedded real time systems requiring a high level of reliability and security, but its complexity is one of its problem. Java is widely used for its many advantages and has got a large community. Many researches are made to put Java in the Real-Time world [21], [11].

We use an implementation of the Ravenscar Java profile [28], [26] on the aJile-100, to develop our application. Applications of Real Time Java are more and more made but this field is relatively

[1] Real Time Specification for Java
[2] Hard Real-Time Hierarchical Object Oriented Design

new, regarding to Ada use for instance. A bachelor's thesis [25] talks about Real Time Java, throught the development of a real time library using the Real-Time Linux KURT, and a small application on it.

Real industrial applications seem to emerge slowly. For example, this implementation [29], developed by a company, differs from ours because it is focused on communication protocol and not on an embedded device, nevertheless the same type of real-time constraints is applied to each of them.

Contribution

This project proposes an adaptation of OOAD[3] [19] to HRT HOOD. We also give an implemented and tested prototype on the aJ-100 processor platform, on a Jstick board [5].

Overview

This report is structured as followed : in chapter 2, we talk about the Problem Domain Analysis: we explain the system's purpose and the parts that system should help administrate, monitor, or control. It is a a model of the real world. We also put the project in its context. The chapter 3 belongs to the Application Domain Analysis. We define all the criteria related the user organisation (use cases). In this part we state all the functionalities and the interfaces of the system. Design of the solution is dedicated to chapter 4, where is specified and detailed a structure in layers : it is the architecture of the application. Moreover, we focus on the schedulability theory in the function layer part to insure that all deadlines could be met. In chapter 5, we give an explanation about our implementation and the testing, following an incremental process. Finally we conclude in chapter 6, making an assessment of the work done.

[3]Object-Oriented Analysis and Design

Chapter 2

Problem Domain Analysis

In this section, we give the details of the activity involved at the first stage of the project. This activity is analysis in which some items are taken apart and described. Basically, there are two kinds of analysis named **problem domain analysis** and **application domain analysis**.

In this chapter, we focus on problem domain analysis which identify and model the problem domain. The problem domain is a part of a context that is administrated, monitored or controlled by a system. Our analysis is concentrated on what the system should deal with and is described in the following sections.

2.1 Context

The FTIR (Fourier Transform InfraRed [2]) analyser is controlled by an embedded software. It is an instrument for analysing and controlling the quality of agricultural production, food, pharmaceutical and chemical products. Figure 2.1 is an example of this kind of instrument.

Figure 2.1: Analysis instrument

The FTIR system falls into three following modules (Figure 2.2):

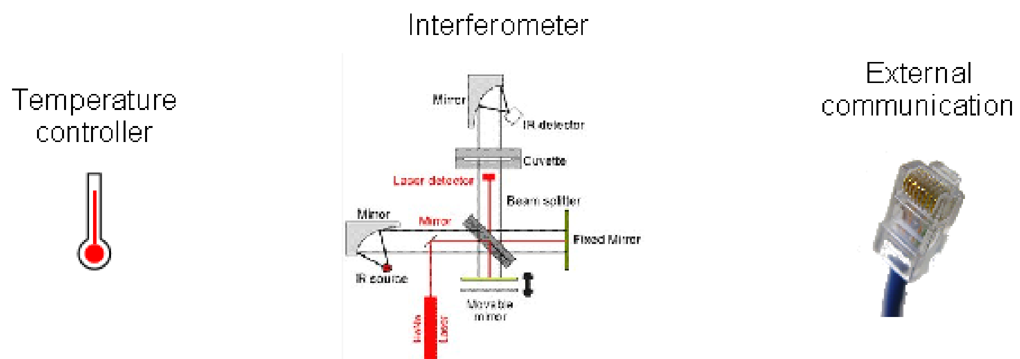

Figure 2.2: FTIR system rich picture

1. The interferometer module (Figure 2.3): In order to measure a sample, a **beam of infrared** (IR) light is passed through a liquid sample and the interferometer produces a signal called an interferogram. In the liquid sample some energy for specific frequencies is absorbed, which gives a unique characteristic of the sample. The measurement is obtained by a **detector**. Because the interferogram is composed of all infrared frequencies, it needs to process a number of scans of different lengths. Thus the interferometer has a **moveable mirror** to move to the next position in order to take a new measurement. These results are averaged into an resulting interferogram, by the future application.

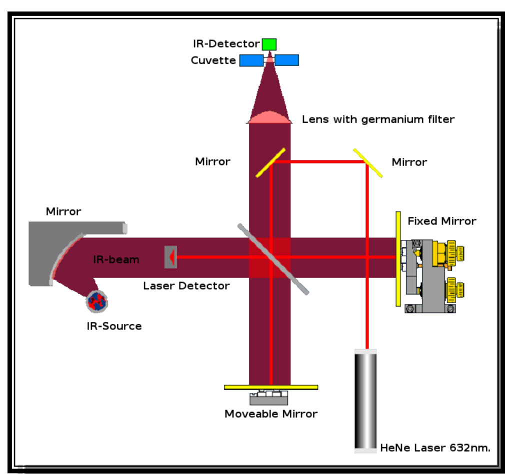

Figure 2.3: Interferometer principle

2. The temperature controller module: The FTIR instrument is enclosed in an isolated box (Figure 2.4), a thermobox, in order to regulate the temperature of the instrument. The aim of this thermobox is to keep a constant temperature around the interferometer. The temperature inside the box is controlled at the following places: **the thermobox**, **the IR-source** (InfraRed source), the **interferometer** and the **sample cuvette**. Each place has a **temperature feeler** and a **thermostat** element for individual regulation. The software must implement a PID regulator for this regulation.

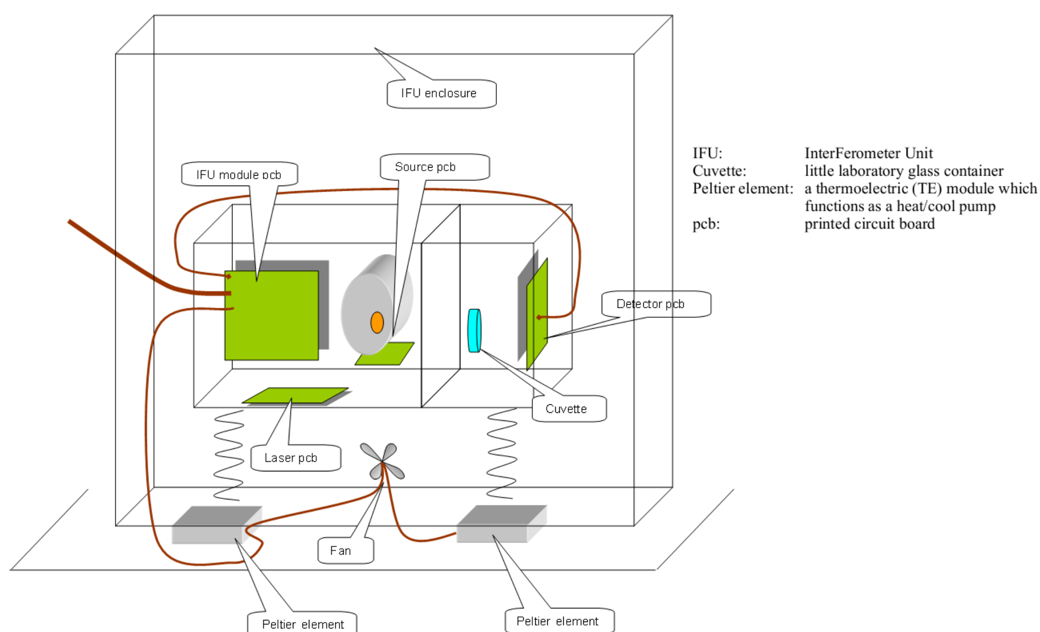

Figure 2.4: FTIR physical module.

3. The external communication module: We know only that a measurement is started from outside. When a measurement is finished the result is sent back along with some other information. The communication follows a protocol which is not specified at the moment but the access medium is a RJ45 cable.

2.2 System definition

The instrument in the production version is also called FTIR system. It is an instrument measuring the infrared spectrum of liquid samples. This system has real-time constraints and it is aimed at measuring the spectrum of a sample. The different parts of the system such as the interferometer, the temperature controller and the external communication should be controlled. Thus the instrument should be able to launch a test on a sample and transmit the result to an external computer.

2.3 The FACTOR criteria

The FACTOR [19] criteria helped us to validate our system definition:

FUNCTIONALITY for end use: The first goal of the system is to measure the IR spectrum for a sample. The system should be able to produce an interferogram or report error conditions that prohibits it.

APPLICATION DOMAIN of end use: It is to launch the sampling of the spectrum and collect results from the interferometer to transmit them. The software have also to report errors occurring in the system and to monitor the correct behaviour of the system.

CONDITIONS for success: The system must be as dependable as the one currently produced. Furthemore the prototype must be designed for easy maintenance and adaptation to new hardware, software and features.

TECHNOLOGY to be used: The aJ-100 real-time processor [6]. It uses Java as its native language. The software shall be developed according to the Ravenscar-Java Profile [28](a subset of the Real-Time Specification for Java [9]) and its implementation on the processor. Tools Software used for this project are *Eclipse*, *JEM Builder* and *Charade*. *Eclipse* and its plugins enable to design of real-time embedded system using Object-Oriented techniques. *JEM Builder* and *Charade* are used to configure, load and start JVM[1] on the aJ-100 processor. We have not chosen tools to test the application but we think of using the UML 2.0 testing profile.

OBJECT SYSTEM: Temperature control, Interferometer, external communication, data logging modules.

RESPONSIBILITY: To implement the functionalities ensuring the same behaviour as the existing solution but with a Java real-time processor and a clear architecture.

2.4 Problem domain model

After reading Hans Søndergaard's technical report [2], we have made out the models present in this section. Models are characterized by the following objects which described the system. Statechart diagrams are also shown to extend our class definitions by adding descriptions of their behaviour.

[1]Java Virtual Machine

FTIRSystem class

The FTIRSystem class models the entire instrument (Figure 2.5). It is composed of the Interferometer, TemperatureSensor, Heater, Cooler and ExternalCommunicationDevice.

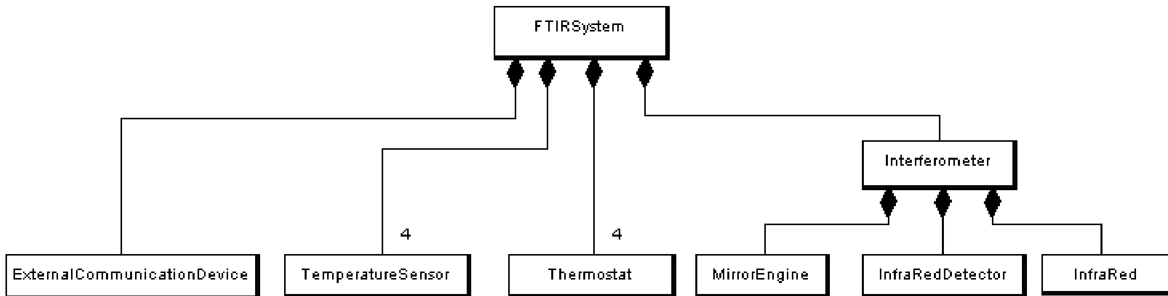

Figure 2.5: Overview of the classes in the system

TemperatureSensor class

The TemperatureSensor class models the working of the temperature feeler. It just consists of reading the temperature.

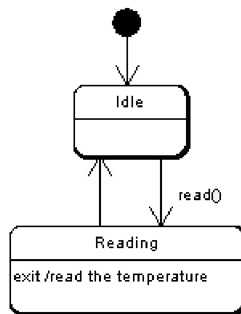

Figure 2.6: The TemperatureSensor Statechart Diagram

Thermostat class

The Thermostat class describes how some thermoelectric modules works. It consists of cooling or heating the place where the module is.

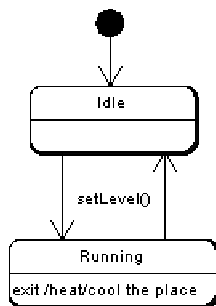

Figure 2.7: The Thermostat Statechart Diagram

Interferometer class

The Interferometer class models the working of the interferometer. It is composed of the MirrorEngine, InfraRed and InfraRedDetector. The behaviour of these classes are described further. The interferometer allows to scan a sample and can be interrupted.

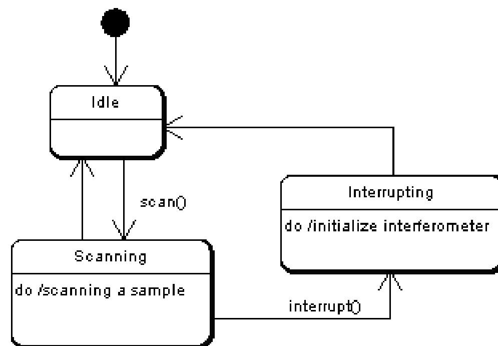

Figure 2.8: The Interferometer Statechart Diagram

InfraRedDetector class

The InfraRedDetector class models the working of the InfraRed Detector. It consists of turning on/off the detector and taking the measurements.

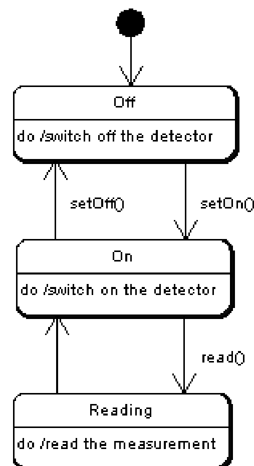

Figure 2.9: The InfraRedDetector Statechart Diagram

MirrorEngine class

The MirrorEngine class models the working of mirror. It consists of controlling the move of mirror.

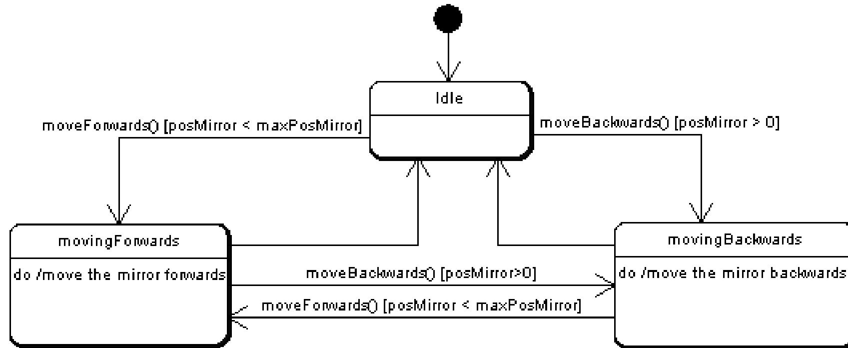

Figure 2.10: The MirrorEngine Statechart Diagram

InfraRed class

The InfraRed class models the working of the InfraRed. It consists of trurning on/off the beam.

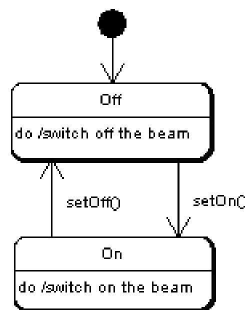

Figure 2.11: The InfraRed Statechart Diagram

ExternalCommunicationDevice class

The ExternalCommunicationDevice class models the working of the communication with external devices. It consists of sending messages about interferogram results and information relating to the monitoring of the system and receiving messages from outside.

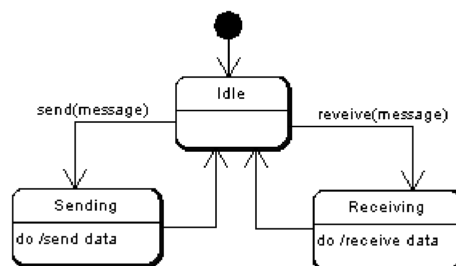

Figure 2.12: The ExternalCommunicationDevice Statechart Diagram

Chapter 3

Application Domain Analysis

This part discusses Application Domain Analysis. The purpose of this part is to determine a system's usage requirements.

3.1 Use cases

The purpose of this section is to determine how actors interact with the future application. An actor is an abstraction of users or other systems that interact with the target application. A use case is defined as a pattern for interaction between the application and actors in the application domain. As expected results, we will get a description for each use case and actor.

The system has only one actor which interacts with it. Behind this actor – *external system* –, we assume there is an operator or somebody who monitors and controls the entire measurement and analysis process including the FTIR system. First of all, the system just observes the power switch. Then, as its action, the *external system* can acquire an interferogram. As a result the *external system* receives data from the measurement like analog values combined with their bounds, or errors that occur during the measurement. Figure 3.1 shows the application use cases.

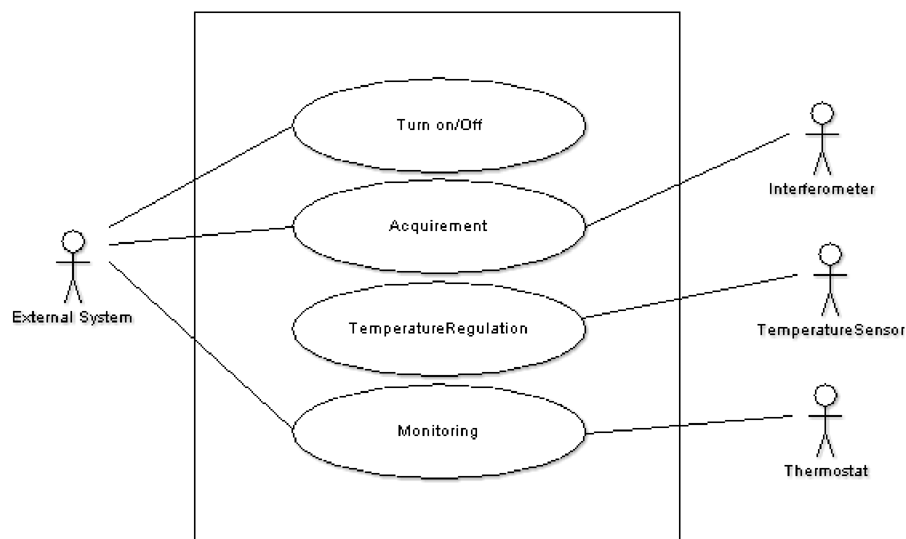

Figure 3.1: Use cases

External System The *external system* is the abstraction of a system (or an operator) who controls the entire measurement and analysis process including the FTIR system.

Interferometer The *interferometer* is an element composed by the `MirrorEngine`, the `InfraRedDetector` and the `InfraRed`.

TemperatureSensor The *temperature sensor* is the device from which temperatures are read.

Thermostat The *thermostat* is the device which is used to control either the heater and the cooler.

Turn On/Off The *turn on/off* use case is the action of turning on/off the system.

Acquirement The *acquirement* use case is the action of starting an entire measurement of a liquid sample.

TemperatureRegulation The *temperature regulation* use case is the action of regulating the temperature at the four defined places.

Monitoring The *monitoring* use case is the action of gathering all defined and useful information and sending them to the *external system*.

3.2 Functionalities

The purpose of this section is to determine the system's information processing capabilities. As expected results, we will get a complete list of functions with specification of complex functions.

3.2.1 Turn On/Off

When the *external system* – or certainly the person behind it – turns the system on, it observes the power switch and starts regulating the temperature. When all the temperatures are within bounds, the system is ready to acquire an interferogram. If at any time, whether the system is acquiring an interferogram or not, a physical module is not at the ideal temperature, then the current acquirement — if there is one – will be stated as failed and the system will go back to the state where it waits for all the temperatures to be within bounds. Figure 3.2 shows the system states.

3.2.2 Acquirement

The *acquirement* use case is the only one which is directly initiated by the *external system* actor. It is composed of two functions. The first one is the *acquire* function. It is used to start an acquirement of an interferogram. The second one is the *interrupt* function. It is used to interrupt an acquirement.

Acquire

The *acquire* function is the main one of the system. We can notice that a complete measurement of a sample is composed of 32 scans where each scan has up to 3200 measurements. When the system is ready – all physical modules are at the right temperature – an interferogram can be acquired. The prerequisite is that the system is fully initialized. The function returns a table of the interferometer results to the *external system* actor. Figure 3.3 shows the sequencing of the *acquire* activity.

Interrupt

The *interrupt* function is available when the system is proceeding a measurement. It basically stops the interferometer during its acquisition process and reinitialize all the system. As the *acquire* function, this one is initiated by the *external system* actor. After the *interrupt* function has been activated, no result is produced. It represents a change of state in the model, thus it is an "update" function.

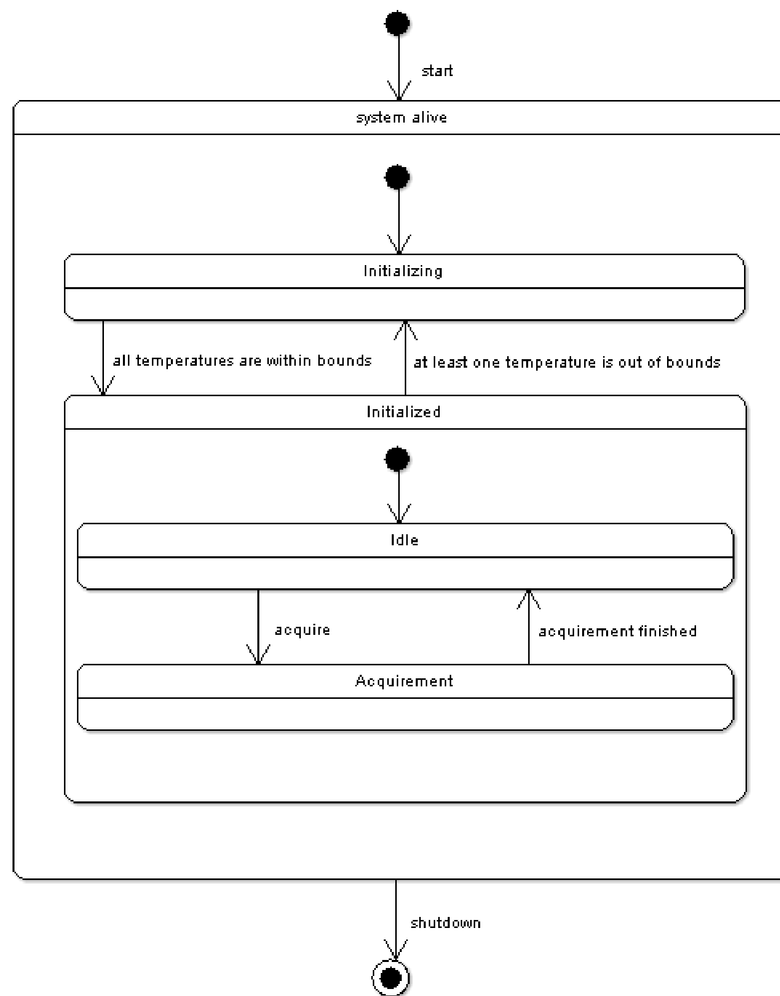

Figure 3.2: System states

3.2.3 Temperature regulation

The *temperature regulation* use case is a parallel one of the *acquirement* use case. It is initiated when the system is turned on. It is composed of two functions. The first one is the *temperature read* function. It is aimed at reading from thermometers. The second one is the *temperature regulate* function. It is used to regulate the temperature with a *Thermostat* device according to the values read by the *temperature read* function.

Temperature read

The aim of the *temperature read* function is to take measurements periodically of four parts in the system: The thermobox, the InfraRed source, the interferometer and the sample cuvette.

Temperature regulate

The aim of the *temperature regulate* function is to control the heating and the cooling of the physical modules of the system. This periodic routine use an average of five temperatures to regulate the temperature in each place. The type of this function is "compute": it manipulates the heater and the cooler when a temperature is different (c.f. PID regulation) from its reference. It results in a direct intervention in the problem domain: the need is to maintain constants temperatures (34°C in the thermobox, 30°C at the infrared place and 42°C inside the interferometer and at the cuvette) for the interferogram acquirement.

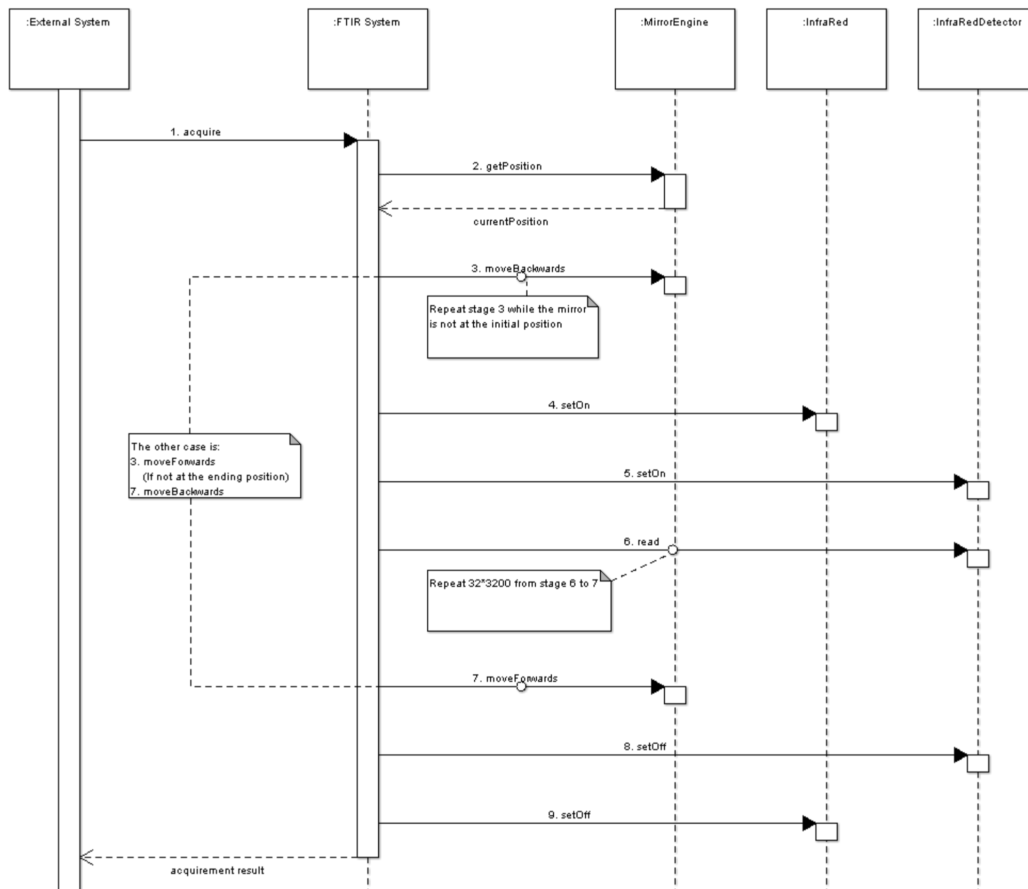

Figure 3.3: Acquire

3.2.4 Monitoring

As the *temperature regulation* use case, this one is initiated when the system is turned on. It is aimed at checking the state of the system in a general manner. It is composed of two functions: the *monitor* function which checks that the system is in the normal conditions of working and the *function aliveness check* function which checks that all processes of the system are running.

Monitor

The *monitor* function is used to save all the analog values such as temperatures and mirror speed in order to send them to the *external system*. It checks that each temperature value is between its bounds. If it is not the case, an error is saved which will be sent to the *external system*. The speed of the mirror is processed in a similar way. The *monitor* function also saves how many measurements have been taken and how many have failed. The type of this function is “compute”: it results in data which will be sent to the *external system* actor.

Function aliveness check

It is the watchdog that checks the aliveness of functions. It controls that all the system functions have registered themselves to it. If it is not the case, it saves an error which will be sent to the *external system* actor.

3.3 Interfaces

The purpose of this section is to determine a system’s interfaces. As expected results, we will get list of operations for each external device and protocols for interaction with other systems.

Every error that occurs when calling the methods below are registered. Such errors are saved by the functions that caught them in order send them to the *external system*.

3.3.1 MirrorEngine

We assume that the mirror engine is a synchronous stepper motor [3] of which the displacement length between two steps is negligible compared the system time constraints. Table 3.1 shows `MirrorEngine` operations.

`GetPosition`	Used to get the current position of the mirror. The range is between 0 and 3199.
`MoveForwards`	Used to move forwards the mirror to its next position.
`MoveBackwards`	Used to move backwards the mirror to its next position.

Table 3.1: `MirrorEngine` operations

3.3.2 InfraRed

Table 3.2 shows `InfraRed` operations.

`SetOn`	Used to turn the `InfraRed` on.
`SetOff`	Used to turn the `InfraRed` off.

Table 3.2: `InfraRed` operations

3.3.3 InfraRedDetector

Table 3.3 shows `InfraRedDetector` operations.

`SetOn`	Used to turn the `InfraRedDetector` on.
`SetOff`	Used to turn the `InfraRedDetector` off.
`Read`	Used to get the value of the beam caught by the detector. The returned value is a 16 bits integer.

Table 3.3: `InfraRedDetector` operations

3.3.4 TemperatureSensor

Table 3.4 shows `TemperatureSensor` operations.

`Read`	Used to read the value from the thermometer. The returned value is a 32 bits floating point value depicting a temperature in Celsius degrees.

Table 3.4: `TemperatureSensor` operations

3.3.5 Thermostat

Table 3.5 shows `Thermostat` operations.

SetLevel	Used to set the level of the Thermostat. The range depends on the implementation of the Thermostat. However, the range must be between 0 and n, with n greater than 0.

Table 3.5: Thermostat operations

3.3.6 ExternalCommunicationDevice

We assume that the protocol between the *External System* and the system itself is CLDC[1] and the data format is UTF-8. Table 3.6 shows ExternalCommunicationDevice operations. The incoming protocol is straightforward as there are only two expected messages. Thus, the messages are coded on one byte. The expected values are 00 for the *acquire* message and 01 for the *interrupt* message. The outgoing protocol is just to send data to the outside. There are four kinds of outgoing messages — three for the logs and one for the interferogram — that are susceptible to be sent. However, they are all using the same pattern. A message can be divided into three parts. The first part is the type of the message. It is a byte; 00 for an ERROR message, 01 for a WARNING message, 02 for an INFORMATION message and 03 for an interferogram. The second part is the count of bytes of the third part; it is coded on two bytes. The third part is the message. It can be either a log message coded in UTF-8 or a raw part of an interferogram.

Receive	Used to receive data from the *external system*.
Send	Used to send data to the *external system*.

Table 3.6: ExternalCommunicationDevice operations

Incoming communication

The only two incoming messages are *acquire* and *interrupt*.

Outgoing communication

The outgoing messages are:

- errors
- speed and its bounds
- temperature regulation informations:
  - temperatures and their bounds
  - difference with the reference temperature
  - result of PID regulation and details of its computation
- interferometer:
  - how many time the FTIR module has been on
  - how many measurements has been taken
  - how many measurements failed
- watchdog
- interferogram

[1] Connected Limited Device Configuration

3.4 Summary

In this section, we have seen what the use cases of the application are, its functionalities and the interfaces of the components it interacts with. The elaboration of the application domain was quite straightforward without significant uncertainties. For instance, we do not know yet the length of an outgoing message, but it does not matter as we had defined a variable message length. This part will be used as an entry point of the design part.

Chapter 4

Design

The purpose of the design is to specify a solution which can be easily converted into programs and tests. The process design can be divided into three categories: criteria, architectural and detailed design. *Criteria* selection specifies criteria for the specific application. *Architectural design* details the largest software structures, such as subsystems, packages and tasks. *Detailed design* specifies attributes and methods within individual classes for each functionality of our application.

4.1 Design criteria

Object Oriented Analysis & Design [19] emphasizes three general criteria which are usability, flexibility, comprehensibility. There are also many quality criteria proposed by researchers over the past years (usable, secure, efficient, correct, reliable, maintainable, testable, ...). Of course, we do not use all these criteria because criteria are often in conflict or not relevant to the specific project. Design have to accentuate essential criteria for the application.

Having considered the general object-oriented criteria, we can give a priority for each criteria:

Criterion	*Very important*	*Important*	*Less important*	*Irrelevant*	*Easily fulfilled*
Usable		✖			
Secure			✖		
Efficient			✖		
Correct	✖				
Reliable				✖	
Maintainable		✖			
Testable		✖			
Flexible			✖		
Comprehensible		✖			
Reusable			✖		
Portable				✖	
Interoperable			✖		
Temporal correctness	✖				

Table 4.1: Priority of design criteria

Table 4.1 shows the priority of design criteria. We placed special emphasis on correctness and temporal correctness, because without these characteristics, the system cannot be used at all; these aspects without which the system is considered incorrect or incomplete. Moreover, temporal correcteness is essential to real-time systems. In order to verify whether the system violates its specified timing constraints, we will make a schedulability test. Of course dynamic testing is also a method to verify temporal behavior of the software. We also put some emphasis on usability, maintainability, testability and comprehensibility because these characteristics are much-valued in Object Oriented programming. These criteria allow the system's adaptability, a

low cost to locate and fix system defects, an easiness to ensure that the deployed system performs its intended function and an easy understanding of the system. We gave all other characteristics lower priority, except for reliability and portability which are irrelevant because we design a real-time application bound to a specific technical platform.

4.2 Architecture

Software architecture is a process of designing the global organization of a software system. The system should model the problem domain and implement all functional requirements. The aim of this part is to create a comprehensible and flexible system structure with connections between each component. A component architecture is a structural system view that makes the system easier to understand and organize the design work.

Architecture design

The component architecture defines the overall system structure and models relationships between components as dependencies. These dependencies can later be realized in different ways (aggregation, specialization or dynamic method calls). We defined a layered architecture with five components as described below (Figure 4.1).

Device (System interface): This layer takes care of the control of the different devices.

Logic & Check (Function): This layer takes care of the control of the system in order to implement all functional requirements (temperature regulation, interferometer measurement, monitoring, ...).

Data (Model): This layer stores the state of the problem domain, used by and generated by the functional layer.

Communication (Technical platform): This layer implements the communication stack to communicate with other network devices.

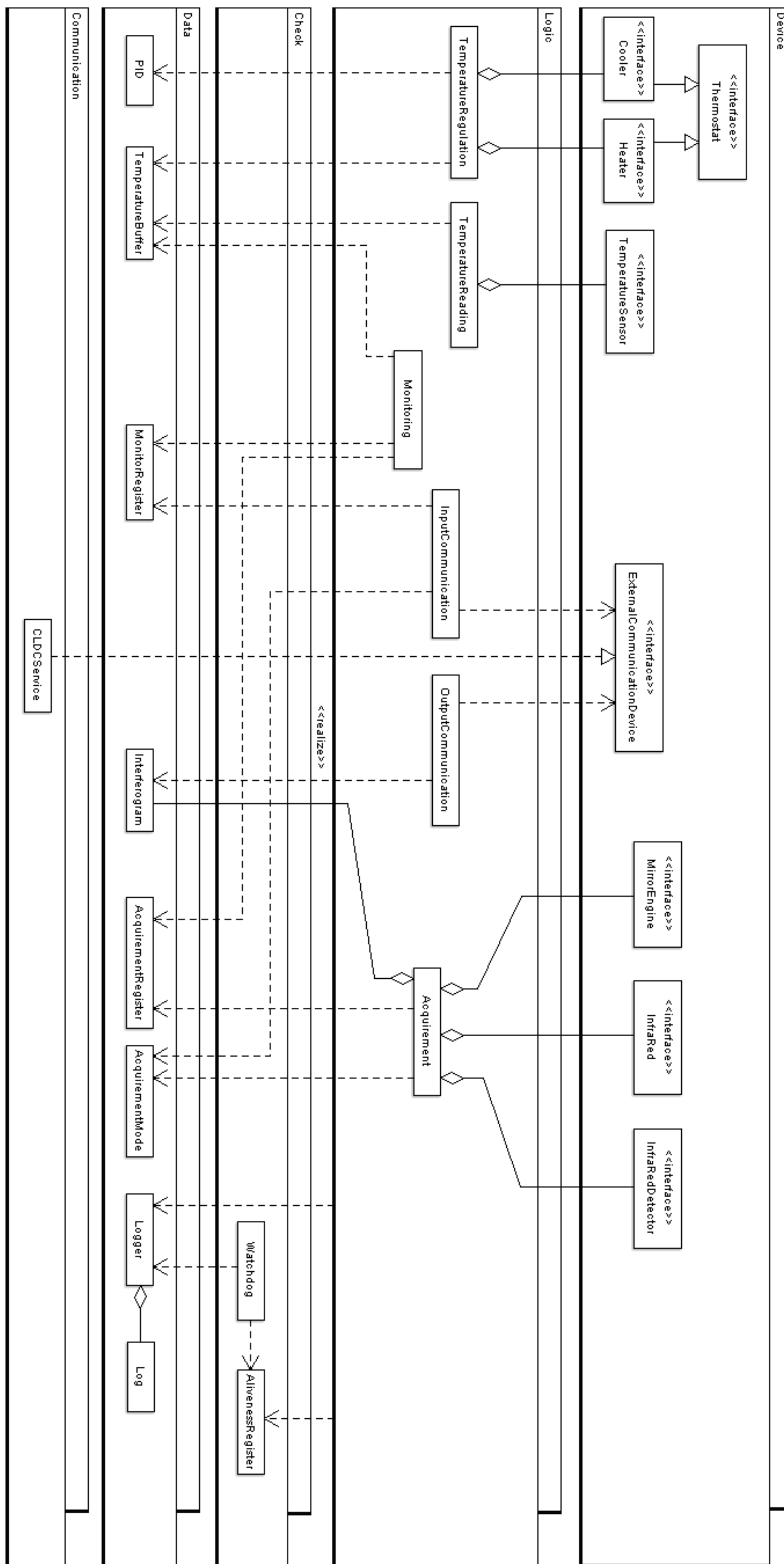

Figure 4.1: The layered Architecture

4.3 Detailed Design

After designing the very high level structure of the system, we can work down to detailed decisions. A component is any piece of software or hardware which has a clear role in the system. It can be isolated, allowing to replace it with a different component that has equivalent functionality. In figures below, we show the details of the different classes with their specifications and interfaces. The specification of classes that contains essential attributes and non-trivial operations is as below.

4.3.1 Device component

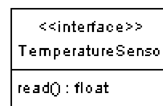

Figure 4.2: Class Diagram of the temperature sensor

TemperatureSensor
– Purpose: Control the temperature sensor
– Operations: Read the temperature

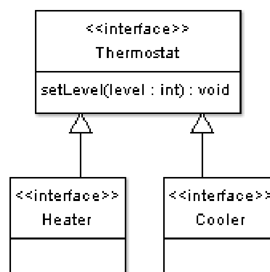

Figure 4.3: Class Diagram of the thermostat

Thermostat
– Purpose: Control a thermostat to heat/cool
– Operations: Update the level of the thermostat

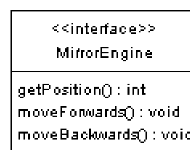

Figure 4.4: Class Diagram of the mirror engine

Mirror Engine
– Purpose: Control the motion of the mirror
– Operations: Update/read the position of the mirror

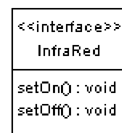

Figure 4.5: Class Diagram of the infrared

InfraRed
- Purpose: Control the infrared
- Operations: Turn on/off infrared

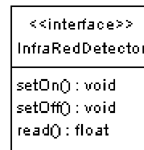

Figure 4.6: Class Diagram of the infrared detector

InfraRedDetector
- Purpose: Control the infrared detector
- Operations: Turn on/off infrared detector, read the measurement

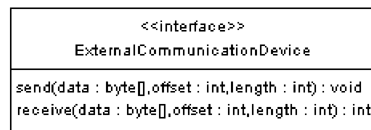

Figure 4.7: Class Diagram of the external communication device

ExternalCommunicationDevice
- Purpose: Control the communication device for all external communications
- Operations: Send/receive data

4.3.2 Logic component

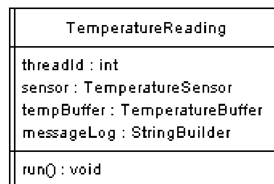

Figure 4.8: Class Diagram of the temperature reading

TemperatureReading
– Purpose: Read and store the temperature
– Attributes: Thread id, temperature sensor, temperature buffer, message.

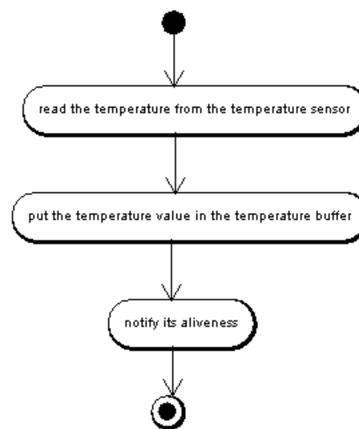

Figure 4.9: TemperatureReading Activity Diagram

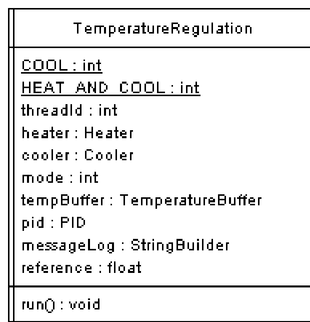

Figure 4.10: Class Diagram of the temperature regulation

TemperatureRegulation

– Purpose: Regulate the temperature according to the stored temperatures

– Attributes: Thread id, heater, cooler, mode, temperature buffer, temperature reference, pid to compute the new temperature, message sent to the logger

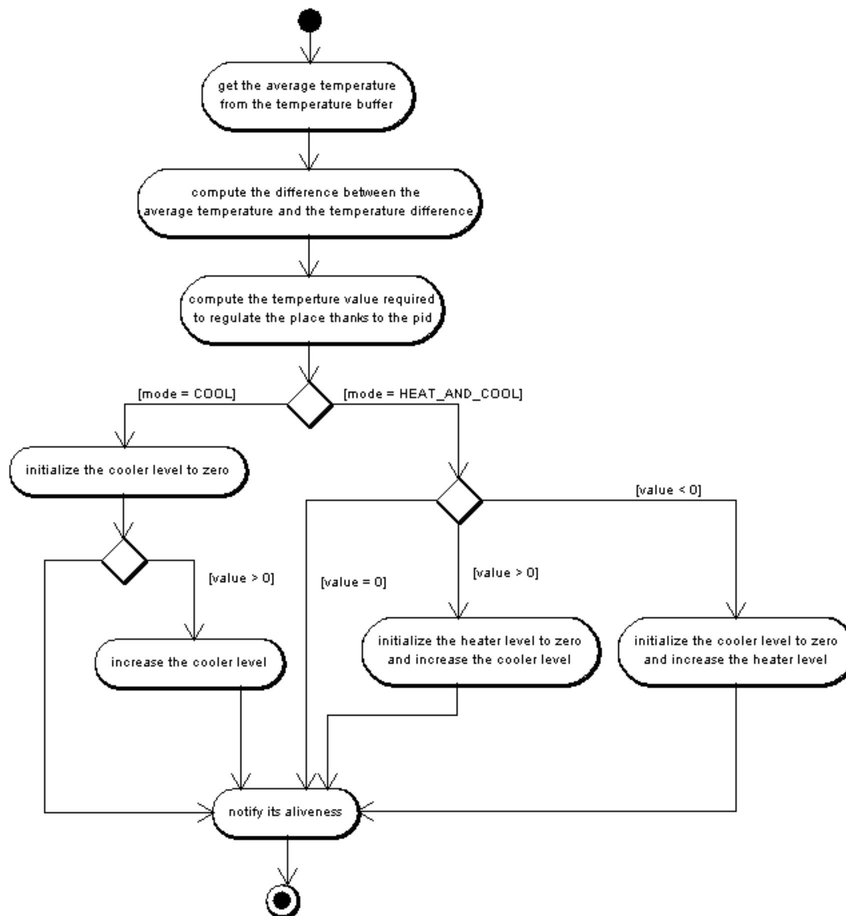

Figure 4.11: TemperatureRegulation Activity Diagram

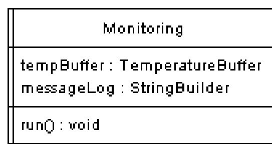

Figure 4.12: Class Diagram of the monitoring

Monitoring

– Purpose: Check temperatures and mirror speed in order to send errors if values are not within bounds

– Attributes: Temperature buffer, message

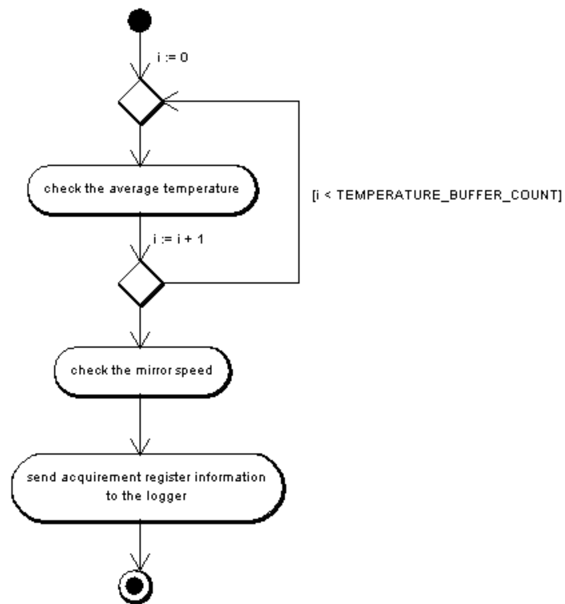

Figure 4.13: Monitoring Activity Diagram

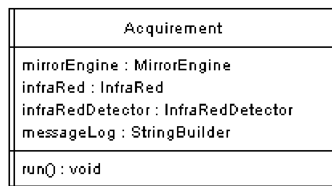

Figure 4.14: Class Diagram of the acquirement

Acquirement
– Purpose: Start an acquirement of an interferogram
– Attributes: Mirror engine, infrared, infrared detector, message

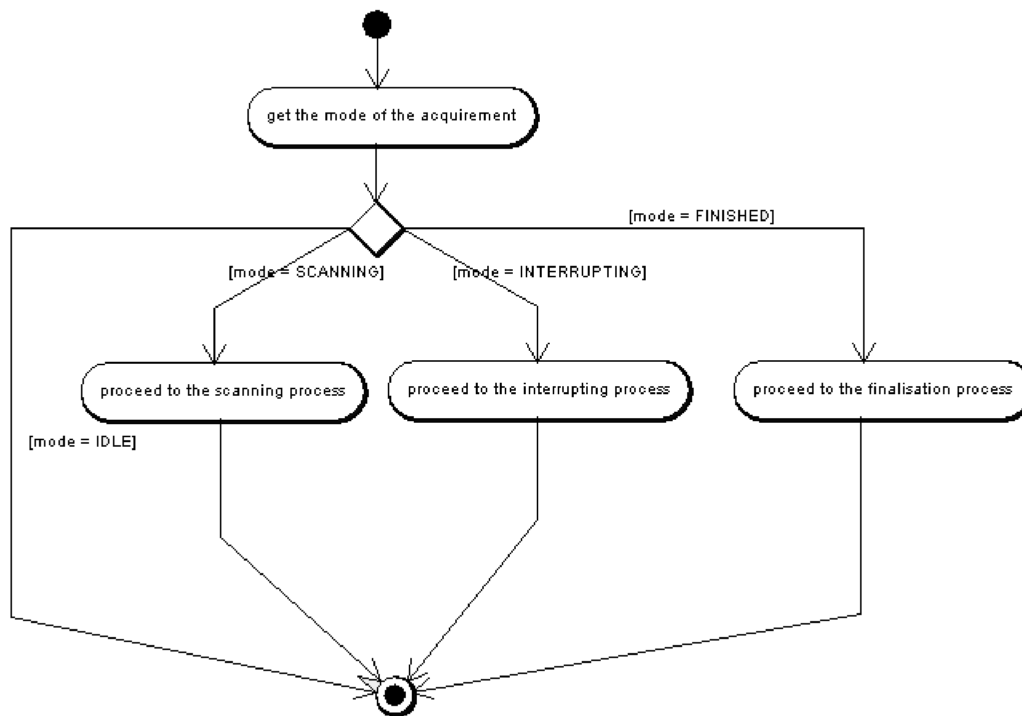

Figure 4.15: Acquirement Activity Diagram

Figure 4.16 shows the scanning process:

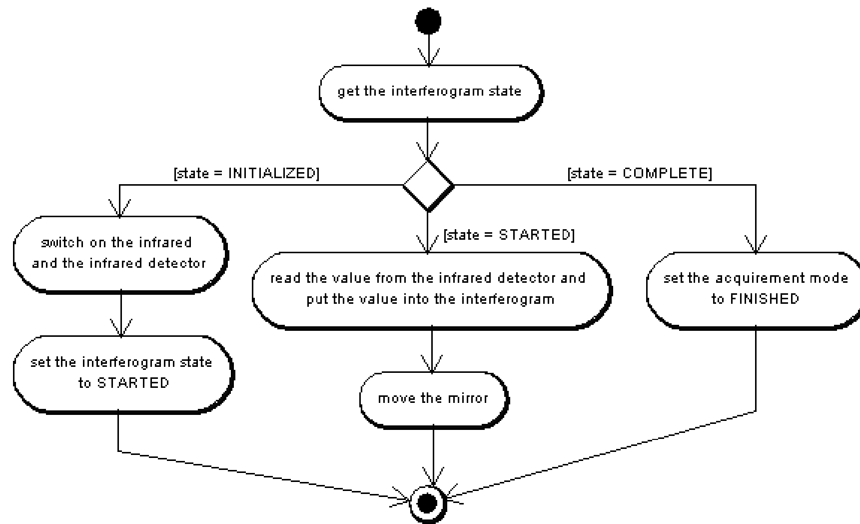

Figure 4.16: Scanning process Activity Diagram

Figure 4.17 shows the finalisation process:

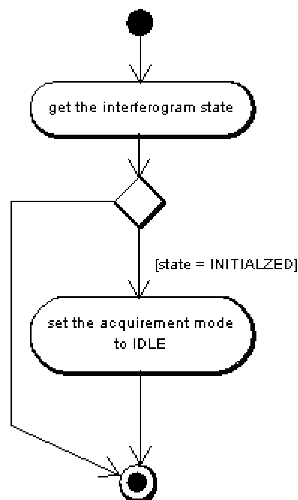

Figure 4.17: Finalisation process Activity Diagram

Figure 4.18 shows the interrupting process:

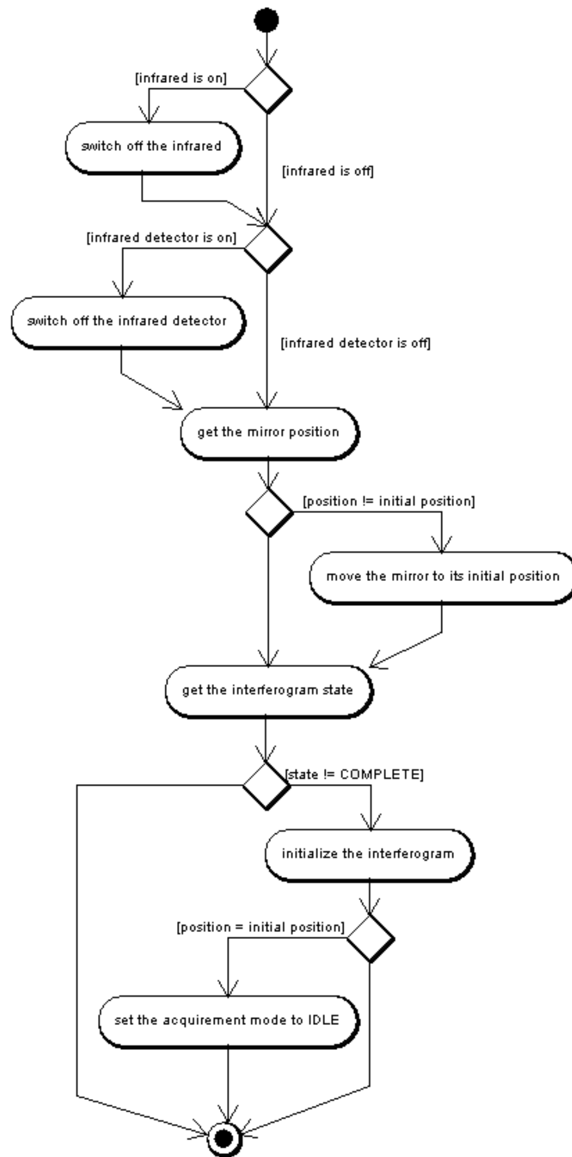

Figure 4.18: Interrupting process Activity Diagram

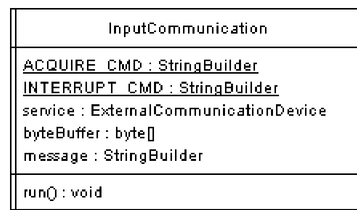

Figure 4.19: Class Diagram of the input communication

InputCommunication

– Purpose: Receive message from the external communication device

– Attributes: Service which provides the communication, byte buffer to receive data, message sent to the logger or command received

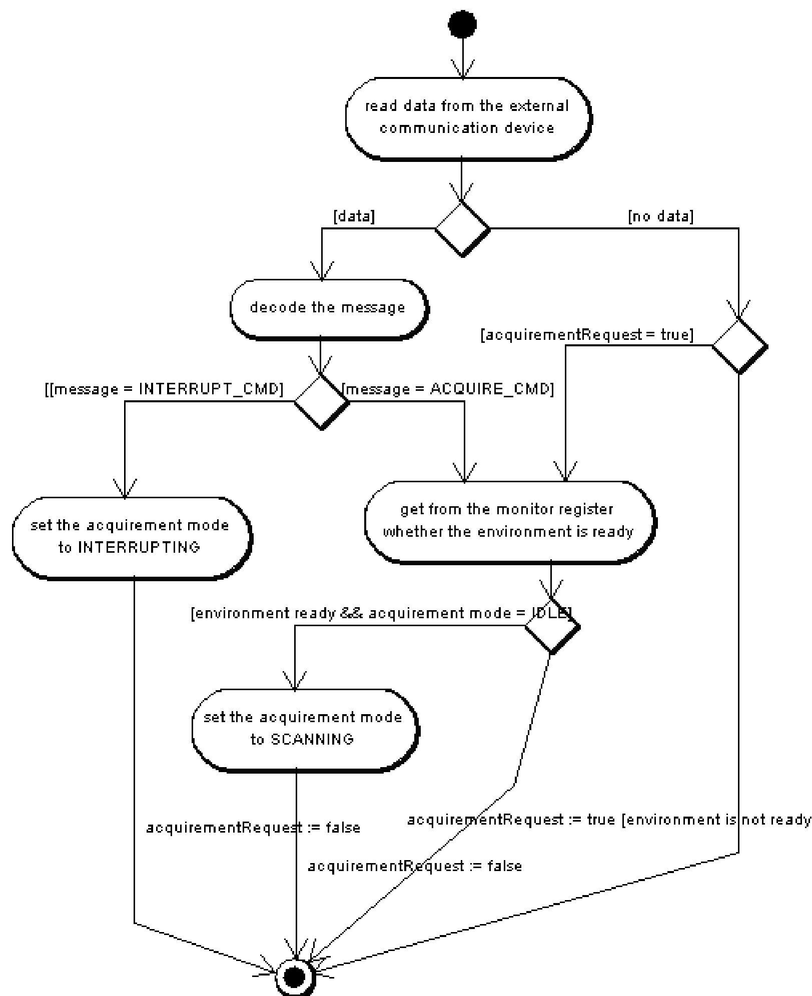

Figure 4.20: InputCommunication Activity Diagram

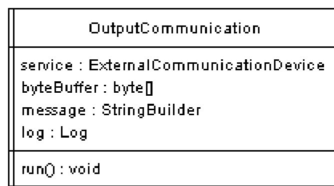

Figure 4.21: Class Diagram of the output communication

OutputCommunication
– Purpose: Send message (error, warning, information or interferogram) to the external communication device
– Attributes: Service, byte buffer to send data, log to send, message

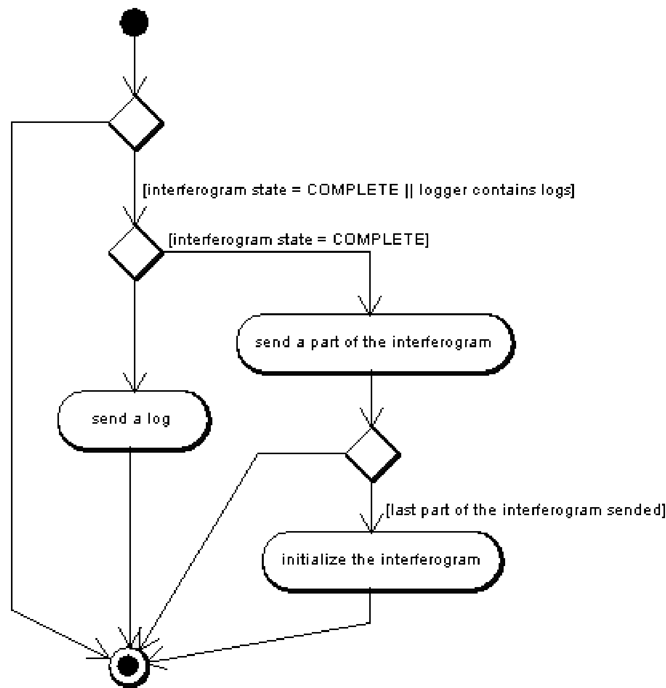

Figure 4.22: OutputCommunication Activity Diagram

4.3.3 Check component

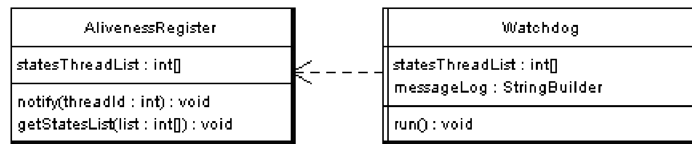

Figure 4.23: Class Diagram of the check layer

Watchdog
– Purpose: Check if all threads are always alive, have not missed their deadlines and log an error whether a thread did not notify its aliveness

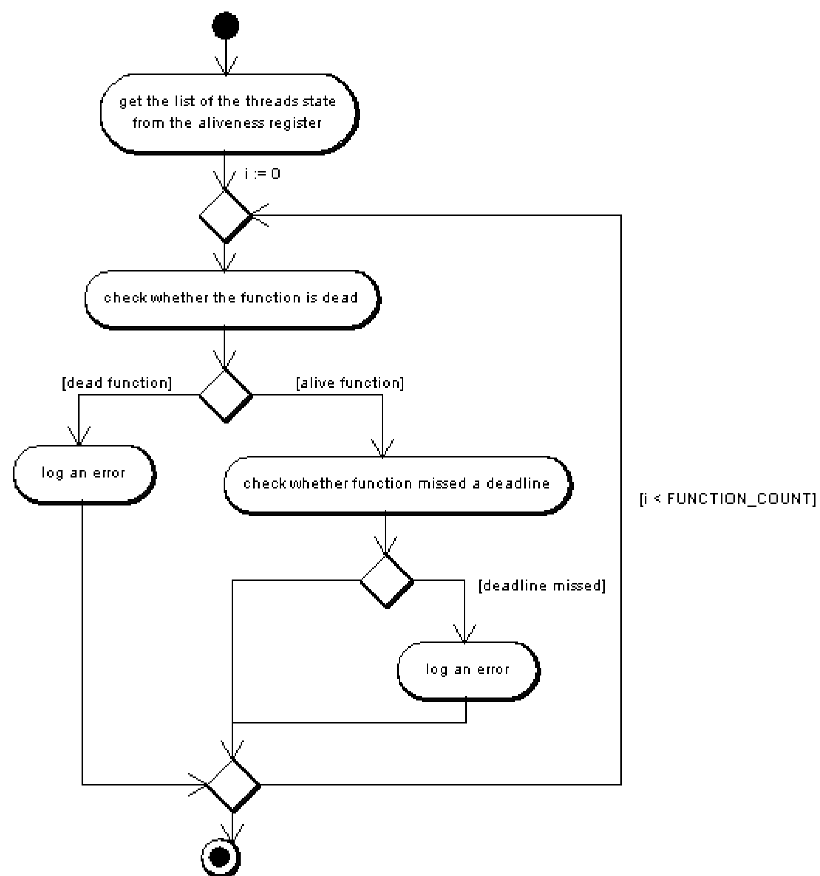

Figure 4.24: Watchdog Activity Diagram

AlivenessRegister
– Purpose: Register information about the aliveness of all threads
– Attributes: List which thread has notified its aliveness
– Operations: Update the state of a thread, read the list of thread states

4.3.4 Scheduling the Function Layer (Logic & Check)

The schedulability analysis is also a part of the logic and check components, because the development of an embedded real-time application supposes that the processor and other shared components need to be scheduled. First, we know that scheduling is based on three concepts:

- an algorithm for allocating the resources *(scheduling mechanism)*
- an algorithm for ordering access to resources *(scheduling policy)*
- a mean of predicting the worst-case behaviour of the system when the policy and mechanism are applied *(schedulability analysis)*

Scheduling mechanism

We will use the Fixed Priority Scheduling (FPS) mechanism in a preemptive scheme for this purpose, a common way to schedule real-time systems: it is used in Ravenscar-Java profile. A static-priority algorithm assigns all priorities at design time, and those priorities remain constant for the lifetime of the task : we are in this case [18].

Scheduling policy

Theoretical approach We have to follow five assumptions [15] in order to make the schedulability analysis possible. These assumptions are as follows [18], p. 48 ; we can underline that "Not all of these assumptions are absolutely necessary, and the effects of relaxing them will be discussed in a later section."

1. The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.
2. Deadlines consist of run-ability constraints only – i.e. each task must be completed before the next request for it occurs.
3. The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks.
4. Run-time for each task is a constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.
5. Any aperiodic tasks in the system are special; they are initialization or failure recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines.

The deadline monotonic conditions There exists some scheduling algorithms, as Rate Monotonic (RM) policy. We can try ([14] p.474) to use the Deadline Monotonic policy [8]. We cannot use RM for instance because it needs deterministic deadlines exactly equal to periods, and we have some tasks with a deadline different from the period. Both (RM and DM) could be supported on the implementation of the Ravenscar Java Profile as said in section 6.1 of [28]. We chose to use a Wikipedia article as a start to verify some conditions of the Deadline monotonic scheduling. From [1] it is written:

1. All tasks have deadlines less than or equal to their minimum inter-arrival times (or periods). *This condition is satisfied, according to the next section.*
2. All tasks have worst case execution times that are less than or equal to their deadlines. *We can assume this condition is satisfied.*
3. All tasks are independent and so do not block each others execution (for example by accessing mutually exclusive shared resources). *We have to consider this condition in our Response Time Analysis by using a "Blocking" term in the equations.*

4. No task voluntarily suspends itself. *We do not have this kind of action in the system. This condition is satisfied.*

5. There is some point in time, referred to as a critical instant, where all of the tasks become ready to execute simultaneously. *Yes, we have this critical point in the system, at the start. This condition is satisfied.*

6. Scheduling overheads (switching from one task to another) are zero. *According to the manual of aJ-100 [6], Thread to thread yield in less than 1 µs. In our system, the smallest deadline is 200 µs. We can assume that the Scheduling overhead is negligible, so it can be ignored. This condition is satisfied.*

7. All tasks have zero release jitter (the time from the task arriving to it becoming ready to execute). *We can assume this condition is satisfied.*

We will use DM [14], p.484 (Scheduling) to give priorities to our tasks, because when $D < T$, Deadline Monotonic priority ordering is optimal. The general principle is the fixed priority is decided implies the following way [17] :

$$D_i < D_j \Rightarrow P_i > P_j$$

Priorities of processes Note : If there is no deadline specified for a task we assumed that it is equal to its period (for a periodic task) or minimal inter-arrival time (for a sporadic task) according to [27]. Moreover we decide when deadlines are equals between tasks to assign a arbitrary priority.

Task	Type	T (ms)	D (ms)	C (ms)	P	U
Acquirement	sporadic	0.333	0.2	0.04	13	0.2
Temperature regulation (×4)	periodic	1000	100	2.5	9..12	0.025×4
Temperature reading (×4)	periodic	200	200	1	5..8	0.01×4
Monitoring	periodic	333	333	30	4	0.09
Output communication	periodic	333	333	15	3	0.045
Input communication	periodic	500	500	25	2	0.045
Watchdog	periodic	8000	8000	80	1	0.01

Table 4.2: Table of priorities using the DMS. T is the period or inter-arrival minimal time, D the deadline, C the worst case computation time, P the priority and U the Utilization.

Schedulability analysis

These are the shared resources we have : *TemperatureBuffer(1 to 4), Logger, AlivenessRegister, AcquirementRegister, Interferogram, AcquirementMode*. So, we need to compute B_i, which is the maximum blocking time that process i can suffer. K is the number of critical sections (resources).

$$B_i = \sum_{k=1}^{K} usage(k, i)C(k)$$

See Table 4.3 for $C(k)$.

The worst blocking time (B_i) for each process is in Table 4.4. We followed the basic model to compute B_i proposed in [14], p.489.

Name of the shared resource	C (worst-case execution time)
TemperatureBuffer4	0.1 ms
TemperatureBuffer3	0.1 ms
TemperatureBuffer2	0.1 ms
TemperatureBuffer1	0.1 ms
Logger	0.1 ms
AlivenessRegister	0.1 ms
Interferogram	0.01 ms
AquirementRegister	0.04 ms
AquirementMode	0.01 ms

Table 4.3: Worst-case execution time for each shared resource

We will use this formula to compute the response time of each tasks (we have to consider the blocking B_i):

$$R_i = C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Applied to our example we can have the results (all results are in milliseconds) in Table 4.5.

Conclusion

All the results from the assessments (Response time analysis) prove that the system is schedulable, or at least, seems to be *a priori*. The worst response time of each process is always less or equal to its deadline, so that it indicates us that it is impossible to miss a deadline, with all the values we have defined. We considered that we have chosen a quite pessimistic computation time for each process; in the real case we expect to find lower values. Anyway, the response time analysis will be also done *a posteriori* in the implementing and testing process (see 5.2.3).

Process	Shared resources	Worst-case blocking time B_i
Acquirement	AcquirementMode AcquirementResister AlivenessRegister Interferogram	0,16 ms
Temperature regulation #4	AlivenessRegister TemperatureBuffer#4	0.2 ms
Temperature regulation #3	AlivenessRegister TemperatureBuffer#3	0.2 ms
Temperature regulation #2	AlivenessRegister TemperatureBuffer#2	0.2 ms
Temperature regulation #1	AlivenessRegister TemperatureBuffer#1	0.2 ms
Temperature reading #4	AlivenessRegister TemperatureBuffer#4	0.2 ms
Temperature reading #3	AlivenessRegister TemperatureBuffer#3	0.2 ms
Temperature reading #2	AlivenessRegister TemperatureBuffer#2	0.2 ms
Temperature reading #1	AlivenessRegister TemperatureBuffer#1	0.2 ms
Monitoring	AcquirementMode AcquirementRegsiter AlivenessRegister Logger TemperatureBuffer#1 TemperatureBuffer#2 TemperatureBuffer#3 TemperatureBuffer#4	0.25 ms
Output Communication	AlivenessRegister Interferogram Logger	0.2 ms
Input Communication	AcquirementMode AlivenessRegister	0.1 ms
Watchdog	AlivenessRegister Logger	0 ms

Table 4.4: Blocking time for processes

Task	Priority	T	C	**D**	**R**	B	I
Watchdog	1	8000	80	**8000**	**188.1**	0	108.1
Input communication	2	500	25	**500**	**97.32**	0.1	72.22
Output communication	3	333	15	**333**	**69.02**	0.2	53.82
Monitoring	4	333	30	**333**	**52.03**	0.25	21.74
Temperature reading #1	5	200	1	**200**	**17.86**	0.2	15.74
Temperature reading #2	6	200	1	**200**	**16.74**	0.2	15.54
Temperature reading #3	7	200	1	**200**	**15.58**	0.2	14.38
Temperature reading #4	8	200	1	**200**	**12.76**	0.2	11.56
Temperature regulation #1	9	1000	2,5	**100**	**11.6**	0.2	8.9
Temperature regulation #2	10	1000	2,5	**100**	**8.78**	0.2	6.36
Temperature regulation #3	11	1000	2,5	**100**	**5.92**	0.2	3.22
Temperature regulation #4	12	1000	2,5	**100**	**3.1**	0.2	0.4
Acquirement	13	0.333	0,04	**0.2**	**0.2**	0.16	0

Table 4.5: Worst case response time (R) of the processes, according to an use and a modification of an excel sheet [23]

4.3.5 Data component

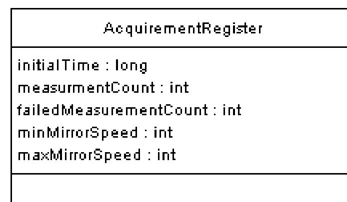

Figure 4.25: Class Diagram of the acquirement register

AcquirementRegister
– Purpose: Register information about the acquirement (elapsed time, number of measurements, min and max of the mirror speed, ...)
– Attributes: Initial time, measurement count, failed measurement count, minimal and maximum mirror speed
– Operations: Read elapsed time, update/read measurements count, update/read min and max of the mirror speed

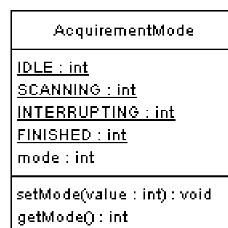

Figure 4.26: Class Diagram of the acquirement mode

AcquirementMode
– Purpose: Register the mode of the acquirement
– Attributes: Mode
– Operations: Update/read the mode of the acquirement

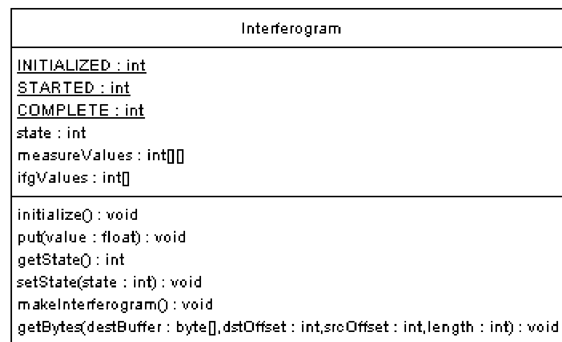

Figure 4.27: Class Diagram of the interferogram

Interferogram

– Purpose: Register all measurements taken by the acquirement and compute the final interferogram
– Attributes: State, table of all measurements, table of the interferogram's values
– Operations: Update the interferogram's state, compute the interferogram, add a measurement, initialize the interferogram, give the interferogram values in byte

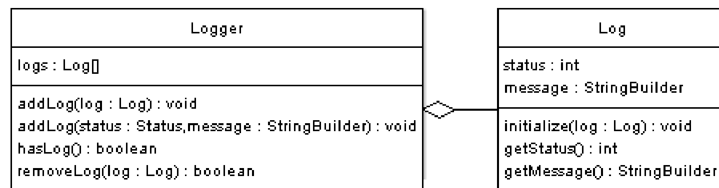

Figure 4.28: Class Diagram of the logger and log

Logger

– Purpose: Register all information which have to be sent by the external communication device
– Attributes: A table of all logs
– Operations: Add a new log, read and remove the next log to send

Log

– Purpose: Register any information
– Attributes: Status of the message (error, warning, information,...), message
– Operations: Initialize a log, read status/message

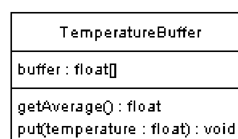

Figure 4.29: Class Diagram of the temperature buffer

TemperatureBuffer

– Purpose: Register temperature values read by the sensor
– Attributes: Buffer to store temperature values
– Operations: Add a new temperature value, read the average of the temperature values

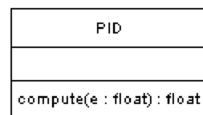

Figure 4.30: Class Diagram of the PID

PID
– Purpose: Compute the Proportional Integral Derivative

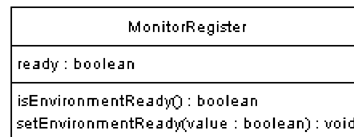

Figure 4.31: Class Diagram of the monitor register

MonitorRegister
– Purpose: Register the environment state
– Attributes: Ready (true/false according to environment ready/not ready)
– Operations: Read/update the environment state

4.3.6 Communication component

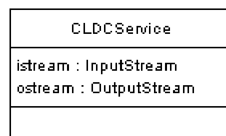

Figure 4.32: Class Diagram of the communication layer

CDLCService
– Purpose: Send and receive data on a socket
– Attributes: Input and output stream to send data
– Operations: Send/receive data

Chapter 5

Implementation & Testing

This part discusses implementation & testing. The purpose of this part is to show how we have developed the application and how we have tested it. Each part of the realisation has followed an iterative process: implementation then testing, etc.

5.1 Implementation

This section discusses implementation. The section shows what kind of tools we used and the coding conventions that we have chosen.

5.1.1 Software usage

We used various applications in order to run the project. Some of them are dependent of the aJ-100 processor.

Eclipse *Eclipse* is an IDE[1] which allows computer application development. We used *Eclipse* to write the Java source code and this report with a *SVN*[2] server.

JEMBuilder *JEMBuilder* is an application provided by aJile Systems in order to compile Java classes into a binary file which can be executed on the aJ-100 processor.

Charade *Charade* is a software provided by aJile Systems to handle the loading and the execution of a program on the aJ-100 processor.

5.1.2 Coding conventions

Every class of our application follows the same coding conventions. All the documentation is written with the *javadoc* standard. Each class file begins by its name, the *package* and the *import* declarations and finally by the class documentation:

```
/*
 * Logger.java
 */

package org.ftir.data;


/**
 * Defines a <code>Logger</code>.
 *
```

[1]Integrated Development Environment
[2]Subversion

```
 * @version 1.0.0 26/10/2006
 * @author Thomas Baron
 * @author Gael Mercier
 * @author Philippe Jean
 */
```

The following is the standard Java file with every field and method documented.

5.2 Testing

This section discusses the testing phase. Testing is the process used to check the correctness, completeness, security and quality of developed application. In our case, we have specified a set of tests in order to verify the robustness of the embedded real-time application.

There are two major approaches to testing. The first one is called *black box testing*[3] and the second one is called *white box testing*[4].

Black box testing Black box testing alludes to tests that are conducted at the software interface. Test cases demonstrate that software functions are operational, that input is properly accepted and output is correctly produced. Black box testing requires the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.

White box testing White box testing alludes to tests that are conducted with knowledge of the program structure and it exercises every logical path of the software. Using white box testing methods, the software engineer produces test cases that guarantee that all independent paths within a module have been exercised at least once, exercise all logical decisions on their true and false sides, execute all loops at their boundaries and within their operational bounds and exercise internal data structures to assure their validity. All these tests should give expected results.

The unit testing and integration testing are white box testing. The acceptance testing is black box testing.

5.2.1 Unit Testing

Unit testing aims at finding defects in individual classes. Unit test is procedure used to validate that a particular module of source code is working properly. Ideally, each test case is independent from the others; auxiliary objects can be used to assist in testing a module in isolation. Besides, unit testing provides a sort of *living document*. Clients and other developers looking to learn how to use a module can look at its unit test to determine how to use it to fit their needs and gain a basic understanding of the interface.

Procedure

For our application, unit tests were developed for every relevant module. Unit testing were not conducted for the functional part[5] because that is covered by the first level of integration testing. For the rest of the application, unit testing were conducted using the library `JUnit`. For each tested class, another is created named in the same way with the suffix `Test`. Each test class can be run independently from the others.

Example

Class `Logger` was tested as well as other classes of package `org.ftir.data`. First of all, a class called `LoggerTest` was created. This class derives class `TestCase` of library `JUnit`. Then, three test methods were developed in order to validate the whole class by passing through all logical

[3] Also called functional testing
[4] Also called structural testing
[5] Package `org.ftir.logic` and class `org.ftir.check.Watchdog`

paths. Figure 5.1 shows classes `Logger` and `LoggerTest`. Figure 5.2 shows the source code of the first test of `LoggerTest` called `testLogger`. This test aims at testing the constructors of class `Logger`. Figure 5.3 shows the execution trace of class `LoggerTest`. In that picture, the three dots mean that there are three successful tests. If a test failed, the dot is replaced by a '`F`'.

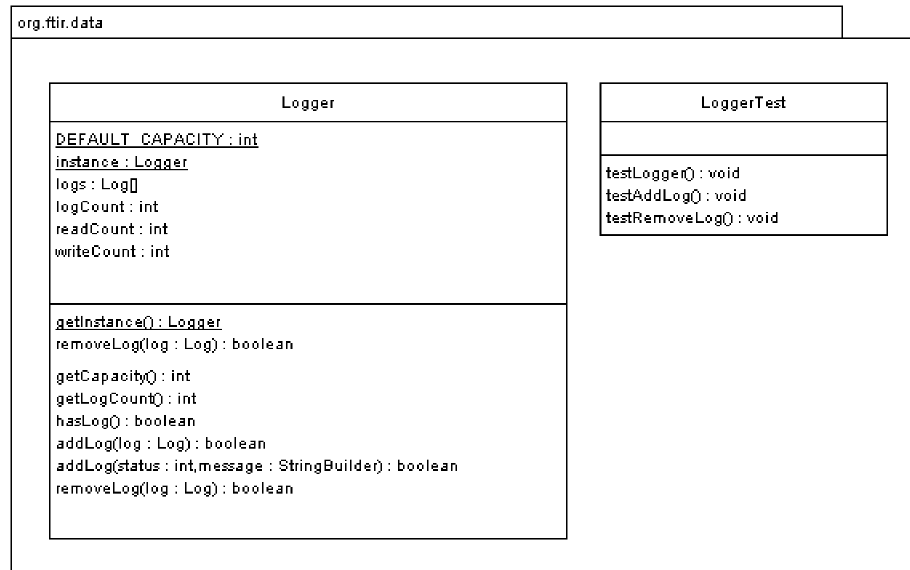

Figure 5.1: Classes `Logger` and `LoggerTest`

```
public void testLogger() {

        // constructs a Logger and tests whether the capacity // and the log
        count are properly initialized Logger lgr1 = new Logger();
        Assert.assertTrue(lgr1.getCapacity() == Logger.DEFAULT_CAPACITY);
        Assert.assertTrue(lgr1.getLogCount() == 0);
        Assert.assertFalse(lgr1.hasLog());

        // the same as the previous one but while specifying the capacity int
        capacity = 16; Logger lgr2 = new Logger(capacity);
        Assert.assertTrue(lgr2.getCapacity() == capacity);
        Assert.assertTrue(lgr2.getLogCount() == 0);
        Assert.assertFalse(lgr2.hasLog());

    }
```

Figure 5.2: Example `testLogger`

```
... Time: 0,11

OK (3 tests)
```

Figure 5.3: `LoggerTest` standard output

5.2.2 Integration Testing

Integration testing is the phase in which individual software modules are combined and tested as a group.

Procedure

In our application, integration testing were conducted in the functional layer[6]. Unit testing has already covered the model layer. First of all, in order to test a function, a sequence diagram were drawn to describe expected method calls between the function and other classes. Then, a test class is created to activate the function. Finally, the test class is run with a trace library to get the sequence of method calls. To validate the test, the sequence diagram and the method call trace have to be compared manually. No automatic test validation is used for integration testing.

Example

Figure 5.4 shows the sequence diagram of the function `TemperatureRegulation`. Figure 5.5 shows the test class of the function `TemperatureRegulation`. Figure 5.6 shows the command line to get the method call trace. Figure 5.7 shows the trace of method calls. That is relevant at the level of method `run`. Indeed, we can see expected messages: `getAverage`, `compute`, `setLevel`, `setLevel` and `notify`.

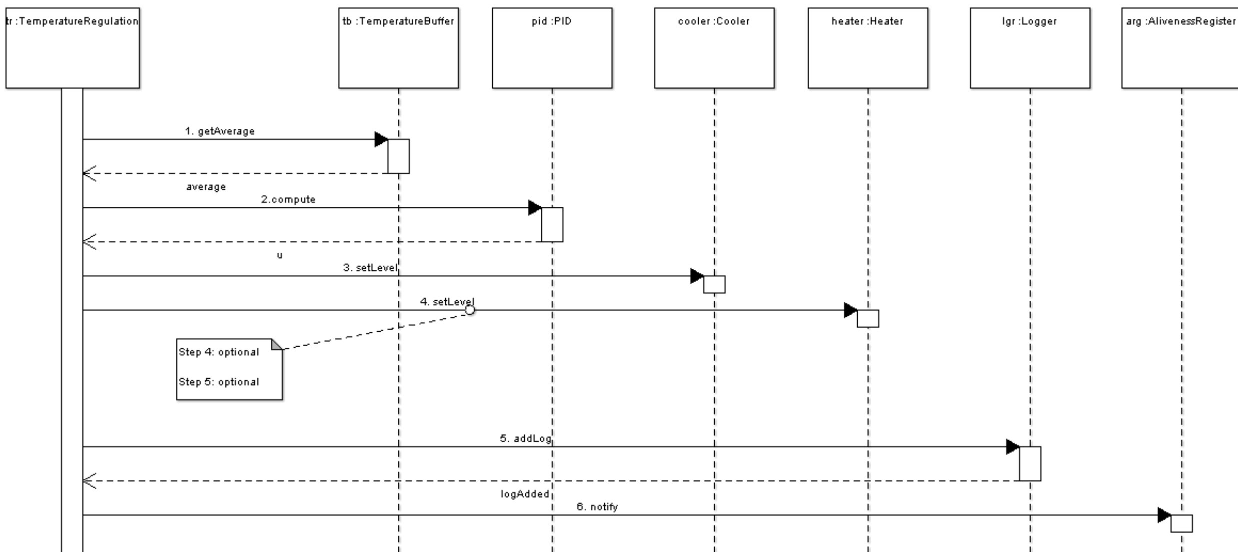

Figure 5.4: Scenario `Temperature Regulation`

[6]Package `org.ftir.logic` and class `org.ftir.check.Watchdog`

```
/*
 * TemperatureRegulationTest.java
 */

package org.ftir.logic;

import org.ftir.check.AlivenessRegister;
import org.ftir.data.FunctionIdentifiers;
import org.ftir.data.Logger;
import org.ftir.data.Reference;
import org.ftir.data.TemperatureBuffer;
import org.ftir.device.Cooler;
import org.ftir.device.DeviceException;
import org.ftir.device.Heater;


/**
 * Defines the test used as integration test of the function
 * <code>TemperatureRegulation</code>.
 *
 * @version  1.0.0 21/11/2006
 * @author   Thomas Baron
 * @author   Gael Mercier
 * @author   Philippe Jean
 */
public class TemperatureRegulationTest implements FunctionIdentifiers {

    static public void main(String[] args) {
        new AlivenessRegister();
        new Logger();
        TemperatureBuffer buffer = new TemperatureBuffer();
        TemperatureRegulation regulation = new TemperatureRegulation(
                TEMPERATURE_REGULATION_1,buffer,Reference.TEMPERATURE_1,
                new HeaterCoolerImpl(),new HeaterCoolerImpl());
        regulation.run();
    }

    static private class HeaterCoolerImpl implements Heater,Cooler {
        public void setLevel(int level) throws DeviceException {}
    }

} // class ------------------------- TemperatureRegulationTest
```

Figure 5.5: class `TemperatureRegulationTest`

```
java -jar ..\lib\trace.jar -classpath ..\classes -noID -exclude StringBuilder
-exclude DeviceException org.ftir.logic.TemperatureRegulationTest
```

Figure 5.6: Trace command line

5.2.3 Results of the schedulability

In order to check the schedulability analysis and its evaluation (see 4.3.4), we have a program for each thread that compute its capacity time (the worst time capacity to be precise), to know if it is in the good bound.

For instance, the worst case computation time we allowed for the `Acquirement` was 40 μs.

```
|org.ftir.logic.TemperatureRegulationTest.main([Ljava.lang.String;)
||org.ftir.data.Reference.<clinit>() ||org.ftir.data.Reference.<clinit>
||getValue(org.ftir.data.Reference) ||getValue=34.0
||run(org.ftir.logic.TemperatureRegulation)
|||getAverage(org.ftir.data.TemperatureBuffer) |||getAverage=0.0
|||compute(org.ftir.data.PID, -34.0) |||compute=-170.0
|||setLevel(org.ftir.logic.TemperatureRegulationTest$HeaterCoolerImpl, 0)
|||setLevel
|||setLevel(org.ftir.logic.TemperatureRegulationTest$HeaterCoolerImpl, 5)
|||setLevel
|||org.ftir.check.AlivenessRegister.getInstance()
|||org.ftir.check.AlivenessRegister.getInstance=org.ftir.check.AlivenessRegister
|||notify(org.ftir.check.AlivenessRegister, 8)
|||notify
||run
|org.ftir.logic.TemperatureRegulationTest.main
```

Figure 5.7: `TemperatureRegulationTest` standard output

Thread	Capacity
Watchdog	35 μs
Acquirement	35 μs
Monitoring	389 μs
TemperatureReading	28 μs
TemperatureRegulation	129 μs

Table 5.1: Thread capacities

We have tested each "branch" of the `Acquirement` thread and the worst case computation time is 35 μs, which is less than 40 μs. For the other results, the experience give us results under 10 or 100 times under the maximum we estimated (the `Monitoring` uses 389 μs of computation time, but in our theorical approach it could use 30.000 μs).

We also had to make some modifications (e.g. priorities) to make our design fit with the Ravenscar implementation [28].

5.2.4 Acceptance Testing

After the programmers have made the unit testing and the integration testing, the realisation of a software product ends up by acceptance testing. It is the last test done, by the end user or the customer. It is a suite of black box tests that demonstrates confirmity with requirements.

Customer's point of view: It is the only test where the customer is responsible for. The tests have to be done in the environment of the end user. This question is a bit complicated in our case, because we have made a simulation, and the system is not exactly suitable to real instruments of FOSS company.

Defining acceptance criteria: Before starting the acceptance test, we should know what criteria (we can make a link with the design criteria we have defined in 4.1) will be used to decide whether the system is acceptable or not. Does the system have to be perfect? No, but the acceptance criteria should define how the decision will be made. We can assume that the customer can accept a type of small errors related to a low level criteria, but there some other levels of errors that will make the system unacceptable. The customer can also want to validate the other steps of testing [20]: "Part of the acceptance criteria may be to revalidate some of the other system tests. For instance, the customer may want to thoroughly test security, response times, functionality, etc. even if some of these tests were done as a part of system testing."

5.2.5 Environment Model

In order to test our application on the aJile processor, we implemented a mock environment that allows us to simulate temperature variation at the different parts of an FTIR. This environment includes formulas to calculate the temperature variation. Each of the four places has its own formula more or less different from the others.

Thermobox The thermobox is the place which contains the three others. The formula of the temperature variation is the following:

$$\frac{dX_1}{dt} = [C_1 \cdot U_1] - [D_1 \cdot V_1 + DD_1] + \left[E_1 \cdot \left(X_{avg} - X_1\right)\right]$$

$C_1 = 0.12$, Heating Coefficient
$U_1 = 0..n$, Heating Level
$D_1 = 0.12$, Cooling Coefficient
$V_1 = 0..n$, Cooling Level
$DD_1 = 0.30$, External Diffusion Constant
$E_1 = 0.06$, Internal Diffusion Coefficient
X_{avg}, Current Temperature Average
X_1, Current Temperature

Cuvette The cuvette is the place where the sample is located. The formula of the temperature variation is the following:

$$\frac{dX_2}{dt} = [C_2 \cdot U_2] - [D_2 \cdot V_2] + \left[E_2 \cdot \left(X_{avg} - X_2\right)\right]$$

$C_2 = 0.12$, Heating Coefficient
$U_2 = 0..n$, Heating Level
$D_2 = 0.12$, Cooling Coefficient
$V_2 = 0..n$, Cooling Level
$E_2 = 0.06$, Internal Diffusion Coefficient
X_{avg}, Current Temperature Average
X_2, Current Temperature

Interferometer The interferometer is the place where the mirror is located. The formula of the temperature variation is the following:

$$\frac{dX_3}{dt} = [C_3 \cdot U_3] - [D_3 \cdot V_3] + \left[E_3 \cdot \left(X_{avg} - X_3\right)\right]$$

$C_3 = 0.12$, Heating Coefficient
$U_3 = 0..n$, Heating Level
$D_3 = 0.12$, Cooling Coefficient
$V_3 = 0..n$, Cooling Level
$E_3 = 0.06$, Internal Diffusion Coefficient
X_{avg}, Current Temperature Average
X_3, Current Temperature

Infrared The infrared is the place where the infrared is located. The formula of the temperature variation is the following:

$$\frac{dX_4}{dt} = [C_4 \cdot U_4] + [CC_4 \cdot W_4] - [D_4 \cdot V_4] + \left[E_4 \cdot \left(X_{avg} - X_4\right)\right]$$

$C_4 = 0.12$, Heating Coefficient
$U_4 = 0..n$, Heating Level
$CC_4 = 0.25$, Infrared Heating Coefficient
$W_4 = 0..1$, Infrared Activation (On/Off)
$D_4 = 0.12$, Cooling Coefficient

$V_4 = 0..n$, Cooling Level
$E_4 = 0.06$, Internal Diffusion Coefficient
X_{avg}, Current Temperature Average
X_4, Current Temperature

Chapter 6

Conclusion

This part concludes this report and gives a perspective on our work on designing and implementing an embedded real-time application.

6.1 Summary

At the beginning of this report, we have exposed the goal of our work through three questions. After presenting to you the different aspects of our work, we are able to answer these questions.

How to design a system with an object-oriented method? We used an object-oriented method [19] to design an embedded real-time system. This method provides a method to get an analysis document and a design document. To develop an application during a project, it is very important to get such a method and to follow it. In our case, chapters concerning the problem domain analysis, the application domain analysis and the design were elaborated with this method. It allowed not to be lost during the analysis and the design. Consequently, we can say that the *Object-Oriented Analysis & Design* method was an asset for the realisation of this project.

How to use the Ravenscar-Java Profile in this industrial case? Since there is not much feedback about developing industrial application using the Ravenscar-Java profile, our work was like adding one of the first bricks to a wall. However, the profile appeared to be very straightforward to understand and thus to use. Besides, the aim of the profile was to be more simple than the *Real-Time Specification for Java*. Concretely, only one class is dependent of the profile. The rest of the application is completely independent of the profile. This is due to the two phases of the profile [28]. Indeed, there are the *initialization phase* and the *mission phase*. The class which is dependent of the profile belongs to the *initialization phase* and the rest of the application belongs to *mission phase*. The way to design this application based on the profile phases makes it usable, maintainable, testable and comprehensible. In conclusion, we can say that using the Ravenscar-Java profile in industrial case is very intuitive and efficient.

Is the implementation suitable for the case? Another purpose of our work was to know whether the Ravenscar-Java profile implementation on the aJ-100 processor is suitable for industrial cases. According to the results we get from our tests, we can say that the implementation is suitable for our case. However, this implementation presented some restrictions opposite the original Ravenscar-Java profile. Indeed, the implementation does not allow every memory management that was originally defined in the Ravenscar-Java profile. Indeed, the implementation only allows *immortal memory* management [28]. Consequently, we had to write Java code in a way that we were not used to. However, this restriction was only a little constraint. Except the matter of memory allocation, the Ravenscar-Java profile implementation was a straightforward API[1] to design an embedded real-time application. Indeed, we did not have to deal with the specific aJ-100 API. Thanks to this implementation of the Ravenscar-Java profile, we can say that our application meets the temporal correctness criterion.

[1] Application Programming Interface

6.2 Discussion

Analysis The analysis was conceived thanks to information bring by the custumer representative. However, we do not get all necessary information about devices used by the application like the mirror or the thermostat. Consequently, we had to make a choice during the analysis phase in order to finish it and to move further on. Except the matter of device behavior, we has followed the information given by the custumer representative, as much as possible.

Environment model In order to test and to run our application, we had to develop an environment model including the temperature simulation environment. This model has been developed only to run the application and cannot be interpreted as a fair representation of a real environment. Thus, the weak point of the application is that it was developed with an environment model and has been never tested with a real environement.

6.3 Further work

There are two further steps after the end of the work depicted in this report. They can be handled at any time since our application is running.

The first one is to inquire FOSS about our work to get some feedback. A feedback from FOSS can allow to validate or not our choices concerning the devices used by the application. Indeed, it misses some details about the behavior of those devices. To handle this, it requires to modify the analysis part, then the design part and finally the implementation part.

The second step is to test the application with the real environment. Indeed, we have only tested the application with a simulated environment. It should be very interesting to see how the application react with a real environment. The aim of this step is to validate or not our choices regarding the temperature regulation. So, a consequence of this could be the modification of the temperature regulation part in order to work with a real environment.

Bibliography

[1] http://en.wikipedia.org/wiki/Deadline-monotonic_scheduling.

[2] http://en.wikipedia.org/wiki/Fourier_transform_spectroscopy.

[3] http://en.wikipedia.org/wiki/Stepper_motor.

[4] http://www.cs.aau.dk/ravenscar.

[5] http://www.jstik.com. Systronix homepage: http://www.systronix.com/.

[6] Inc. aJile Systems. *aJ-100 Reference Manual*, December 2001. Version 2.1. Web page : http://www.ajile.com/downloads/aJ-100ReferenceManual.pdf.

[7] P. Amey and B. Dobbing. High integrity Ravenscar. In *8th International Conference on Reliable Software Technologies – Ada-Europe 2003 (AE03), Toulouse, France*, 2003.

[8] N. Audsley. Deadline monotonic scheduling (department of computer science, University of York), 1990.

[9] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, and R. Belliardi. JSR 1: Real-time specification for Java. *Java Community Process*, 11/12/2001.

[10] G. Bollella and J. Gosling. The Real-Time Specification for Java. *Computer*, 33(6):47–54, 2000.

[11] B. M. Brosgol. A comparison of the concurrency features of Ada 95 and Java. *ACM SIGADA Ada Letters*, 18(6):175–192, November/December 1998.

[12] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar profile in high integrity systems, 2003.

[13] A. Burns and A. J. Wellings. HRT-HOOD: A structured design method for hard real-time systems. *Real-Time Systems*, 6(1):73–114, 1994.

[14] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 1997.

[15] P. S. Heidmann. Rate monotonic analysis, an overview. Paul S. Heidmann is also the writer of "A Statistical Model for Designers of Rate Monotonic System" Homepage: http://www.heidmann.com/paul/.

[16] J. Kwon, A. J. Wellings, and S. King. Ravenscar-Java: A high integrity profile for real-time Java, 2002.

[17] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation North Holland*, 2:237–250, 1982.

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

[19] L. Mathiassen, A. Munk-Madsen, P. A. Nielsen, and J. Stage. *Object Oriented Analysis and Design*. Marko Publishing ApS, Aalborg, Denmark, 2000.

[20] T. Mochal. Acceptance testing: The customer is the ultimate judge. `http://builder.com.com/`, 2001.

[21] L. M. Pinho and F. Vasques. To Ada or not to Ada: Adaing vs. Javaing in real-time systems. *ACM SIGADA Ada Letters*, 19(4):37–43, December 1999.

[22] P. P. Puschner and A. J. Wellings. A profile for high-integrity real-time Java programs. In *Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 15–22. IEEE Computer Society, 2001.

[23] A. P. Ravn. Response time calculation, `http://www.cs.aau.dk/~apr/RTS/responsetime.xls`.

[24] B. I. Sandén. Real-time programming safety in Java and Ada. *Ada Lett.*, XXIII(2):32–46, 2003.

[25] D. Selvarajan. Implementation of real time Java using KURT (report of Bachelor of engineering, Bharathiar University, Coimbatore, India), 2000.

[26] H. Søndergaard. Periodic threads on aJ-100. Technical report, University College - Vitus Bering Denmark, 2004.

[27] H. Søndergaard. FOSS Case : FTIR. Technical report, University College - Vitus Bering Denmark, 2006. version 1.4.

[28] H. Søndergaard, B. Thomsen, and A. P. Ravn. A Ravenscar-Java profile implementation. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 38–47, New York, NY, USA, 2006. ACM Press.

[29] V. S. Warrier. Java based implementation of communication protocol stacks for utility industry, `http:www.kalkitech.com/downindex/JavaStackWP1.0.pdf`.

Appendix A

CD-ROM enclosed

The attached CD-ROM contains the following:

- Source code
- Test batch files
- UML diagrams
- *Javadoc*
- Readme
- *build.xml*
- PDF files of references
- Latex files of this report