University of Minho
School of Engineering

Rui Mário da Silva e Freitas

Formal Software Development Techniques in a Continuous Vital Signs Control System

November 2009

University of Minho
School of Engineering

Rui Mário da Silva e Freitas

Formal Software Development Techniques in a Continuous Vital Signs Control System

Thesis in Informatics Engineering

Supervisor:
Prof. Dr. João Miguel Fernandes

November 2009

É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA TESE APENAS PARA EFEITOS
DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE
COMPROMETE;

Universidade do Minho, 06/11/2009

Assinatura: Rui Mário Silva Freitas

Resumo

A área da biotecnologia encontra-se em constante crescimento. Vários produtos apareceram nos últimos anos com a promessa de melhorar as nossas vidas. Monitorizar e controlar os sinais vitais para a saúde, desporto e outros propósitos é hoje um grande mercado com um enorme impacto nas nossas vidas.

Nesta tese eu proponho um produto com a capacidade de controlar os sinais vitais de várias pessoas com o objectivo de ajudar hospitais e outro tipo de serviços de saúde a responderem melhor aos seus pacientes tanto dentro das suas instalações como fora, nas suas vidas normais.

O meu objectivo para esta tese é construir um sistema fiável usando uma técnica de análise e design de software e explicar esse processo. Pretendo, também, construir um protótipo funcional, baseado no sistema desenvolvido, que irá permitir testar esse mesmo sistema num ambiente real.

Abstract

The biotechnology area is in constant growth. Several products appeared in the last few years with the promise of improving our lives. Monitoring and controlling vital signs for health, sports and other purposes is a big market today with an enormous impact in our lives.

In this thesis I propose a product with the capability of controlling the vital signs of several persons with the aim of helping hospitals and other type of health care providers responding better to their patients inside the facility or living their lifes normally outside the facility.

My aim to this thesis is to build a reliable system using a software analysis and design technique and explain that process. I also intend to build a functional prototype, based on the system developed, that would permit to test that same system in a real environment.

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Background 2

1.2 Proposed Problem 3

1.3 Approach 4

1.4 Objectives 4

1.5 Overview 5

2 State of the Art **7**

3 Technologies **11**

3.1 VDM++ 11

3.1.1 Why use VDM++ 12

3.2 HOL 13

3.3 Automatic Proof System 15

3.4 VDMUnit 15

3.5 Web Services 16

4 Software Analysis and Design 19

4.1 Requirements Analysis 19

4.1.1 Problem Description 19

4.1.2 Introspection 20

4.1.3 Domain Analysis 20

4.1.4 Prototyping 21

4.1.5 Final Problem Description 22

4.2 System Architecture 23

4.3 Modeling using VDM++ 27

4.3.1 Classes Description 27

4.3.2 System Invariants 34

4.4 Testing 38

4.4.1 Using HOL 38

4.4.2 Using VDMUnit 39

4.5 Prototype 41

4.6 Calculating the Database Schema 44

5 Conclusions 51

5.1 Future Work 52

Bibliography 55

List of Figures

3.1 The usage of the APS tool to generate HOL code from VDM++ . . . 15

4.1 The first system scheme. 20

4.2 The second system scheme. 21

4.3 The Proof of Principle. 22

4.4 The architecture of the first system. 24

4.5 The architecture of the second system. 26

List of Tables

4.1 Differences and similarities between VDM++ and Java 42

4.2 Data types presented in the Client class for the database calculation . 45

Chapter 1

Introduction

In this document I present a system for control the vital signs continuously. Why is this important for the world we live in? Demography answers this question. According to the International Data Base[1] from the U.S. Census Bureau, there are around 6,7 billions people on earth. 97 millions have 80+ years old. The U.S. Census Bureau has a prediction for 2050 and the results say that the entire population on earth will be around 9,5 billions and the people that have 80+ years old will be 470 millions. These numbers points to a 40% growth in the entire population on earth and a 380% growth in the population with 80+ years old. So, the population on earth is getting older and we need to provide them the best medical care so they can live longer and healthier.

Now the question is: how can this system help? Imagine a person who lives completely alone. This person may have a heart failure and not be capable of asking for help. If this person had a monitoring system, the heart failure could be detected and the authorities warned automatically. This is a possible application of monitoring systems. Search for cardiac diseases and prevent them are other possible applications. Out of the cardiac area other examples are helping persons with diabetes remembering them to take their medicines.

World population is getting older and this brings the need of a monitoring system to live longer, with some quality and independence. But, would the population in general and the hospitals use a system like this? A study made by Frost &

Sullivan[2] in 2006 says that 70% of the adults and 90% of the elderly prefer home-based treatment. For the hospitals, home care treatment is less expensive because they don't need skilled nursing, inpatient rehabilitation and long term care facilities.

Vital signs monitoring systems already exist and they can be used for military purposes, sports, health care, like the ones I have been talking, and others. The system I propose in this document is for the health care area.

Health is a sensitive area, so it is necessary to build a reliable system. For that reason, strong methodologies of software analysis and design are used in this thesis and deserve my attention.

1.1 Background

Hospitals and health care facilities all over the world monitor their patients in order to check their health status. They want to ensure the well being of their patients, for example tracking down their cardiovascular system. For that reason, they have machines connected to sensors that are attached to the patient to capture vital information.

In the last century this type of machinery was very rudimentary, large machines, to many wires. For the patient to be monitored he had to give up of comfort.

With the arrival of the wireless technology to the masses and the shrinkage of the processors, many companies started to evolve their products. We began to see a revolution in the bio-technology area. What was once large, expensive and uncomfortable, it is now small, cheap and comfortable. One of those companies is Polar[3] that developed a watch that connects wirelessly with a chest strap and monitors the heart rate. It can be used for sports purposes but it is possible to find similar products working in other environments like military.

The patients in the hospital bed no longer need to give up of their comfort to be monitored. Today, it is possible to monitor patients using smaller and smarter devices working together wirelessly. These devices can send information to central

systems where authorized people can access and act according to the information received. It is already possible for physicians to use their personal computers in the office, or even in their own homes, at the same time that they are observing the vital information of their patients. Information Technology evolved into a phase where large amount of data can be easily stored and consulted remotely with access control.

1.2 Proposed Problem

Sports and health are two possible applications for a system that monitors vital signs continuously. It can be used by professional athletes that want to improve their performances. They can monitor their heart rate, respiration rate, and other vital information continuously during their training. They can store that information for further analysis with their personal trainers to improve certain aspects on their performances.

The monitoring system can be used for health purposes, where the patients are monitored for a period of time previously set. Information is captured during the monitoring and it can be accessed by an authorized physician in real-time or it can be stored for further analysis. With such systems it is possible to analyze patients in their normal day activities and perform a faster and accurate diagnosis.

Alcor, Life Extension Foundation, located at Scottsdale, Arizona, is interested in a system with specific characteristics. They need to provide a fast response to their clients if a cardiac problem is detected. They requested a system that would monitor 24 hours a day vital information. If a cardiac problem occurs with one of their clients an alert is sent back to Alcor employees so they can prepare all the necessary arrangements.

1.3 Approach

Alcor, Life Extension Foundation, made a request to me and Diogo Martins to create such system. For that reason, we moved into Scottsdale, Arizona, for a few months where we spent some time in the Alcor's facilities to understand their needs and requirements.

It was clear that they had a Software and Hardware problem. They knew that there were many available products but none of those products were good enough for their needs. Because of that me and Diogo Martins started a Requirements Analysis process. It is a cooperative process where we meet with the client and where we try to completely understand the characteristics of the system.

It was decided to extend the project scope, adding the capability of working with several Organizations/Associations, such as Hospitals, Health Companies, etc..

From the System Architecture we identified two parts of the project: mobile and control. The mobile is responsible for monitoring the patient 24 hours a day and send alerts if needed. The control part manages the alerts and the vital signs data from the users. Only authorized people can access the control for consulting the information. Diogo Martins is responsible for the mobile part[4], and I am responsible for the control part.

The control part is a Software problem only. For that reason, I need to identify the best technologies to be used. In the end I want to create a functional prototype able to run in a real environment. This is a critical system, and consequently, modeling and testing stages are very important.

1.4 Objectives

In this thesis I present a continuous vital signs control system capable of helping hospitals and other type of health care providers providing them more information about their patients. Information like vital signs, can help physicians in the process

of finding the patient problem. Some diseases can only be found with an extensive data information.

The control system along with the mobile system, permits the health care providers to receive constant updates of their patients. For example, if a patient has an heart attack and the mobile system detects it, an alert is immediately sent to the health care provider and to the contral system which can be accessed by authorized people.

A system with this characteristics must be robust. A robust system is built based on strong methodologies that makes it trusty. Those methodologies combined with a software analysis and design process lead to a consistent final prototype. The creation of that prototype capable of running in a real environment, is the main objective of this thesis.

For that purpose the following tasks are considered:

- Make an accurate analysis of the problem combining several techniques of requirements analysis to achieve a final system architecture. One of those techniques is the creation of a proof of principle where a very rudimentary prototype is created;
- Create a model based on the System Architecture using a specification language;
- Verify the model using testing methodologies. The process of verification is very important to validate the model;
- Calculate the database schema from the abstract model.

1.5 Overview

This thesis is divided into 5 Chapters being this one the *Introduction*.

The Chapter 2 is the *State of the Art*. This chapter aims to present the several existent products. Some of them were developed by companies, others by research

groups at Universities all over the world. The information gathered in this chapter is used as a starting point for the project. There is no need to create a product that already exists, for that reason it is important to have a strong knowledge of the existing ones.

To create good software it is important to choose the right tools and technologies to build it. In Chapter 3 are presented the *Technologies* that are used or referred in the following chapters. I give a brief overview over them and explain why they make part of the project.

In the Chapter 4 is presented the entire process of the *Software Analysis and Design*. It starts with the *Requirements Analysis* where the techniques applied are presented as well as the respective process of requirements analysis. A *System Architecture* is created and used to build the *Model*. In the end of this chapter the *Functional Prototype* capable of running in a real environment is created and the database schema calculation is presented.

Conclusions is the Chapter 5 and the last one. I conclude the thesis presenting my personal remarks and results. I also present my point of view and expectations for the project in the future. The project does not end with this thesis and it is important to point out the direction for the future.

Chapter 2

State of the Art

The biotechnology area has a lot of research groups around it and many projects were created in the last few years. I am only interested in those that have the purpose of monitoring and control the vital signs whether they have the interest for the health care or simply for areas like sports, military and others. I give a special focus on those with a central or control system separated from the monitoring system. In this section I enumerate, also, some companies that work in the vital signs monitoring area. This shows the focus the industrial world is giving to the area.

A system to detect the fall of the elderly persons was presented by Hansen and his team[5]. The fall detection system is equipped by 3-axis accelerometers, a GPS receiver and a embedded processor to analyze the data locally. If during the analysis a suspected fall is found the fall detection system makes a connection with the camera from the user's phone via Bluetooth and sends the data to the same phone. The system first attempts to make contact with the user through the phone speaker or the keypad, and if the user doesn't provide any response or claims for help then the system will inform the emergency service and provide them access to the data from the microphone, camera and the fall detector.

Another system I want to talk about is the AID-N system[6]. The AID-N was built to help in disaster scene environments and his main purpose is to replace the normal triage at the hospital for automatic triage. The AID-N is able to do this monitoring the patients' vital signs continuously. The patient caries with him/her

sensors and a PDA that makes the continuous monitoring and analyzes the vital signs, searching for problems with the patient's status. If, during the analysis, an abnormal reading occur, an alert is sent wirelessly to the miTag Server informing the patient's status. All the information is stored in the miTag Server which can be accessible through a web portal by medical staff, paramedics and other authorized personal to provide the best medical arrangements for the patient before he/she arrives the hospital, during the trip to the hospital and while he/she stays in the hospital.

The idea of monitoring patients during their normal day activities, instead of doing an ECG or other exams in the hospital, already exists[7]. The data from the monitoring is stored in a portable device carried by the patient for a period of time or until the next physician visit. To read the stored data on the portable device the doctor need to download the information to his/her personal computer and analyze it with a specific program. By the time the article came out, the authors were testing a new feature for the system which allowed reporting periodically the physician with the patients' normal readings. This new feature would also permit alert the physician when some problem occur with the patient.

The SMART system[8] is very similar to the AID-N system[6]. This system was built for disaster scene environment purposes and to help the hospital when the emergency department (ED) is overcrowded. It is also possible help the paramedics since the disaster scene until the hospital providing them all the necessary information to help the patient. All this is possible because every patient in the SMART system is being monitored continuously. The information from the monitoring process is stored in the SMART central, which is the main server of the SMART system. The SMART central receives all the patients' data (vital signs) and analyze it searching for abnormal readings. When an abnormal reading is found the SMART central is able to process an alert with the patient information and send it to the nearest available doctor. The SMART central is also able to process the post-triage which would help the overcrowded ED and because it is possible to check in real time the status of each patient in the system, the medical staff can help the paramedics and provide all the proper arrangements for the patient before he/she arrives the

hospital.

Paul Blair is a Calit2[9] staff researcher and he thinks that the applications for monitoring our health will be the "next killer mobile applications" on the market. He says that we have the available technology to do that but we don't use it. For Calit2[9] this is a prime area of investigation right now, according to Philip Rios. They are developing a Web Platform for Physicians and Patients, as well an entire system for monitoring the patients' vital signs. In the Web Platform the physicians have a more detailed view about the patients' vital signs, and the users can track their own vital signs. They built all the technology in their labs, like the necessary sensors to monitoring the vital signs with a Bluetooth interface. The data from the sensors is transmitted to the user's phone and than to the web.[10]

CardioNet[11] is one of the companies that work in Biotechnology area. They built a system called CardioNet MCOT[12]. This system monitors the patient 24 hours a day during 21 days. If during this monitoring an abnormality is detected, the CardioNet MCOT sends the ECG information automatically to the CardioNet monitoring center. This will allow the monitoring technicians to analyze the ECG information, respond to events, and inform the physicians. The physicians can select "the events to be monitored, and the level and timing of response by the CardioNet Center"[12]. This system was proved to be 3x more effective than LOOP event monitors in diagnosing clinically significant arrhythmias[13].

Corventis[14] is a startup company in San Jose, CA, that built a wearable and wireless technology capable of detects early signs of heart failure[15]. This technology has a sensor, built by the Corventis company, that beams the patient's data to a phone and then the phone transmits the data to servers that will process it. This servers can find many possible problems with the patient like heart failure. This technology allows the physicians to see the patients' data through the iPhone or a website from anywhere in the world. The company is studying the possibility of the sensor to diagnose sleep apnea through changes in respiration and blood oxygen levels.

Biodevices, S.A.,[16] is a spin-off from IEETA (Institute of Electronics and Telematics Engineering of Aveiro / University of Aveiro) with the mission of de-

veloping, commercialize and export biomedical engineering solutions for medical diagnosis support. They built a product called VitalJacket[17] which is a t-shirt with vital signs' sensors built in that do the monitoring continuously and a Bluetooth sensor to send the data wirelessly from the sensor to the PDA and store it to future analysis in the HWM version or for real-time analysis in the VJMobile.

Zephyr Technologies[18] is a company that builds vital signs sensors. There is a framework called Zephyr Open[19] that uses the sensors from the Zephyr Technologies company. The Zephyr Open is not a product from the Zephyr Technologies, it is an independent project leaded by Brad Zdanivsky. Using this framework it is possible to connect the sensors with a personal computer and see the monitoring in real time.

Continua, Health Alliance[20], is a partnership of more than 200 companies. They work together with the aim of design guidelines for constructing new products and establish a product certification program. The results will be better health solutions to help individuals living independently and securely.

Chapter 3

Technologies

This chapter aims to give a quick overview over the modeling and testing technologies used in this thesis. The Web Services technology is also explored and presented.

3.1 VDM++

The **VDM**[22, 23] (Vienna Development Method) is one of the most established formal methods in software construction. It started as a project in the mid of the 1970s and has been growing since then adding several group of techniques and tools. The language used is **VDM-SL** (VDM Specification Language) and it is used to present specifications in a model-oriented style. VDM has been extended to **VDM++** which is an object-oriented specification language and will be the language used in this thesis to model the problem. The VDM is also part of the **VDMTools**, a group of tools supporting the analysis of system models expressed in the formal language of the Vienna Development Method. The VDMTools provides several features that will be described later in this section.[21]

3.1.1 Why use VDM++

Using VDM++ is using a tool for modeling. So, why modeling? Software should be modeled before the implementation because a successful model allow accurate analysis and prediction of the system's behavior. Today's software industry oblige programs to be in constant growth and changing. If we have to add new features or change the system, it is better to change the model first instead of the implementation. Changing and testing the model is faster than changing and testing the implementation. A model should be an abstract approach of the system. Abstraction consists in omitting details that are not relevant to the model in order to focus in the essential system's behavior. With modeling an understanding of the system properties and structure is obtained. This offers lower cost rectifications of the software system, in case of errors or future improvements.[22, 23]

The use of VDM++ is very helpful for the design and testing processes. With VDM++, or VDM-SL if an object oriented approach is not needed, it is possible to emulate the system behavior in the design process. The ability to add restrictions to instance variables (invariants) and to operations (pre and post-conditions) raises the possibility of testing the model. With VDM++ is possible to create a mathematical model which is usually a very rigorous approach to the system.

VDM++ has also an extensive community working together to improve and create new supporting features. One of those projects is the VDMTools. VDMTools has several features to analyze VDM++ models: Syntax Checker, Type Checker, Interpreter and Debugger, Integrity Examiner, Test Facility, Automatic Code Generator, Rose-VDM++ Link, Java to VDM++ Translator.

- **Syntax Checker**: the Syntax Checker verifies the VDM++ syntax. If the syntax is incorrect the VDMTools will alert and point the problems.
- **Type Checker**: the Type Checker will check for inconsistencies in the use of variables and operators in the code.
- **Interpreter and Debugger**: the Interpreter allows the user to interact with the created model and the Debugger allows the user to debug the program

adding breaks in the middle of the code to check the values of instance variables, for example.

- **Integrity Examiner**: the Integrity Examiner allows the generation of integrity properties which can be possible sources for errors. Those properties are generated based on the invariants, pre and post-conditions, sequence bounds and map domains. This properties are very important for testing the model. They can serve as input for a theorem prover or they can be used for calculate the model by hand. It doesn't matter how you calculate it, the important thing is that those properties evaluate to true.

- **Test Facility**: the Test Facility allows you to create a test suite in order to automate the model testing.

- **Automatic Code Generator**: the Automatic Code Generator gives the possibility to generate C++ or Java code from the VDM++ specification model.

- **Rose-VDM++ Link**: the Rose-VDM++ Link creates a bi-directional translation between the UML and the VDM++.

- **Java to VDM++ Translator**: the Java to VDM++ Translator allows the creation of VDM++ models based on the Java implementation.

3.2 HOL

The basis for **HOL** (Higher-Order Logic) were first developed by Robin Milner[24]. He also designed the ML language that underlines HOL.

The **LCF** (Logic for Computable Functions) was the first attempt by Robin Milner to create a proof-checking program. **HOL** emerged from the LCF with the aim of verifying the hardware at the register transfer level with a higher-order logic formalism. The HOL system was being developed in order to become a general purpose proof assistant.

The **HOL** system was always very open and because of that many people have made several contributions to it. As a result, more than one HOL system has been built.[25, 26]

- **HOL88**: The first stable version of the **HOL** system, with various changes and enhancements. It was developed in *Cambridge* by Mike Gordon supported by *DSTO Australia* and *Hewlett Packard*. It included its own meta language implementation based on *Common Lisp*.
- **HOL90**: Was a re-implementation of the **HOL** system using the *Standard ML*. It was developed in *Calgary* and *Bell Labs*. The resulting system provided a signification performance improvement.
- **HOL98**: It uses the *Moscow ML*, which is an implementation of the *Standard ML*. It was developed in *Cambridge*, *Glasgow* and *Utah*.
- **HOL4**: It is based on the **HOL88**, is an open source project and is the only **HOL** system that is still being maintained. It can be built with either *Moscow ML* or *Poly/ML*.

Another two implementations of the **HOL** system appeared with different purposes.

- **ProofPower**: Created by *ICL* as a commercial system at the beginning, with the special target of security applications. It was designed to support the *Z notation* and it is now an open source project.
- **HOL Light**: It started as a simpler version of the **HOL** system with some changes at the design level. It included automatic provers that separate proof search from checking. It was first implemented in *Caml Light* but now uses *OCaml*. It is faster than the **HOL88**. Now it has grown into another mainstream version of HOL.

The use of the **HOL** system very useful in the model verification process. The operations in the HOL language are theorems, and they can be proved using infer-

ence rules. The HOL system can help in the automation of using those inference rules to prove if the operations are correct or not.

3.3 Automatic Proof System

This tool is being developed by Miguel Ferreira and permits the translation of the **VDM++** model to the **HOL** language automatically and the corresponding process of proving the model in the HOL system.[27]

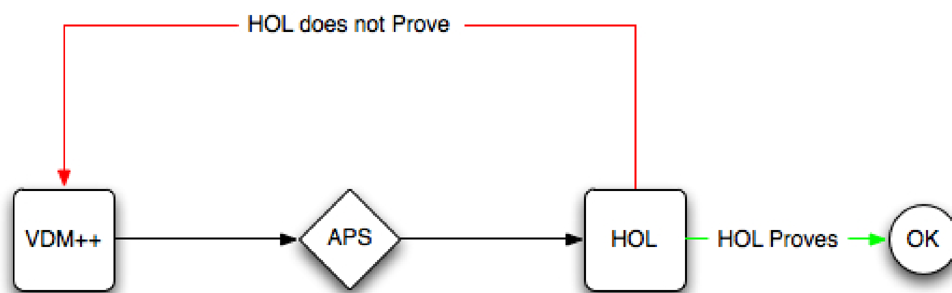

Figure 3.1: The usage of the APS tool to generate HOL code from VDM++

At the moment there are no available releases for download. The software used in this thesis was gently provided by Miguel Ferreira. The **APS** tool operates with two modes: translating and proving. The translation is not always completely accurate. Sometimes, the code produced after the translation process needs to be changed in order to create a valid **HOL** model. Although, it is a powerful and a promising tool.

3.4 VDMUnit

Unit testing is another method for software verification and validation. The idea is to create individual test cases to test the software. With this verification method we will gain confidence but no certainty on the model.

The **VDMUnit** is a framework that allows Unit testing for the **VDM++** models[28]. It is an open framework and it can be changed to be better adapted to a specific problem.

The **VDMUnit** framework is composed by four classes: *Test*, *TestSuite*, *TestCase* and *TestResult*.

- **Test**: is an abstract class with only one operation called "run" that will be used to execute the tests.
- **TestSuite**: is sub-class of **Test**. Contains the sequence of unit tests.
- **TestCase**: is sub-class of **Test**. Represents the test itself.
- **TestResult**: is responsible for maintain references to the tests that failed.

3.5 Web Services

This project is composed by two parts: Mobile and Control. Eventually, the Mobile system will work on a mobile phone. Today, there are several platforms for mobile phones. The most commonly used are:

- **Windows Mobile**[29]: The Microsoft operating system for mobile devices and smartphones.
- **Android**[30]: The Google operating system for mobile phones that runs on linux kernel and is open source.
- **iPhone OS**[31]: The Apple operating system for the iPhone and iPod Touch devices.
- **Symbian OS**[32]: Is the operating system mainly used by Nokia in its mobile phones.

The best way to connect the Mobile part with the Control part services is using a Web Service.

A Web Service is a software that permits the interoperability between two machines over a network[33]. In this case the network that will be used is the Internet.

Creating a web service is creating a public API that other software can use despite their programming language.

There are two big types of Web Services: "Big Web Services" and RESTful Web Services. The "Big Web Services" make use of a specific communication protocol called SOAP. This protocol works on top of the HTTP protocol and is used to send the services requests and responses in XML messages. RESTful web services make use of the HTTP protocol to transmit its messages. Many RESTful users claim that this is how the web services should work, because it is how the web works.[34, 35]

Chapter 4

Software Analysis and Design

In this chapter I present the process of the Software Analysis and Design for the vital signs Control system. Requirements Analysis is the first section and It is common in my and Diogo Martins' theses[4]. The Software Architecture is created here and serves as a base for the Model to be contructed. That Model is then Tested and the chapter ends with the creation of the final Prototype and the calculation of a database schema.

4.1 Requirements Analysis

The Requirements Analysis has a fundamental role in the whole system development. This is the stage where the Client and the Developer cooperate to reach an agreement about which characteristics the final product will have. In the end of the *Requirements Analysis* the developer should obtain the *System Architecture* and its functionalities and properties.

4.1.1 Problem Description

Alcor, Life Extension Foundation, is an American company working on the biotechnology area. Alcor staff is very interested in a Continuous Monitoring System able to warn them every time one of their costumers had a heart failure, causing a

serious danger of death. Alcor contacted Diogo Martins and me to create a system with these properties.

The basic goal is to send a warning to our client when a given person have a heart stop.

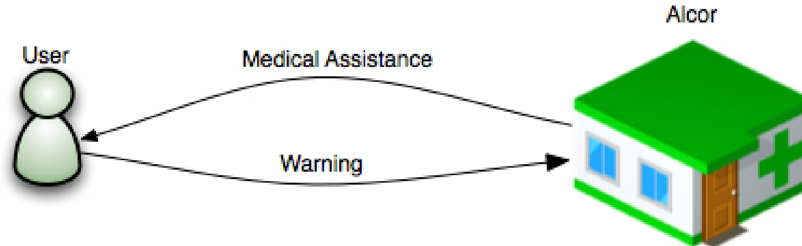

Figure 4.1: The first system scheme.

4.1.2 Introspection

To create a short overview of the problem we used an elicitation technique called Introspection. Using Introspection the analyst defines the system requirements based on what he believes the client needs are. This technique is considered to be a good starting point for other types of techniques.[36]

Based on what we knew from the Problem Description we identified the basic components of the system. Those components would be, then, a sensor receiving the heart information and transmitting it to a device capable of sending a message to Alcor. Storing the heart information of a end-user could be very useful, so the data should be stored for analysis.

4.1.3 Domain Analysis

The first technique used for the Requirements Analysis was the Domain Analysis. This technique consists in the search for documents and similar projects[36]. During the Domain Analysis we found different kinds of monitoring systems. This was important to discover which sensors could we use and to analyze the existent architectures.

About the Software development, Zephyr Open Framework[19] caught our attention. This project, leaded by Brad Zdanivsky, runs on a pc and works with the sensors from Zephyr Technology[18] company. The costumer connects the sensor via bluetooth with the computer using the Zephyr Open Framework. The data from the sensor starts being transmitted over the air from the sensor to the computer each time the both connect. The costumer is able to see it in real time and store it for further analysis.

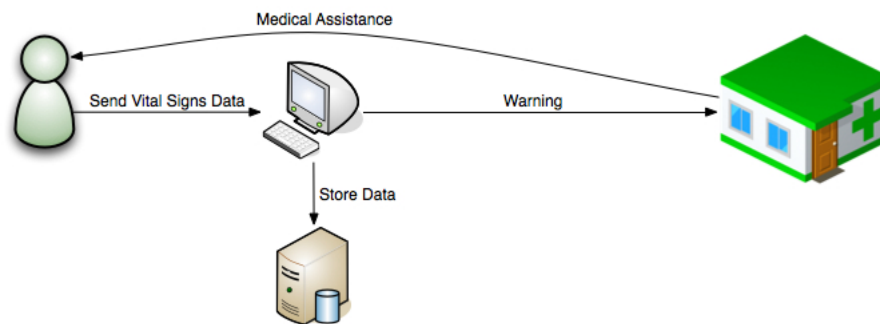

Figure 4.2: The second system scheme.

4.1.4 Prototyping

Using the framework provided by Zephyr Open, a first basic prototype was built. This is a technique known as Prototyping and consists in the creation of a rudimentary system, usually known as Proof of Principle, that implements the basic functionalities of the system. This is a good technique to receive feedback from the client and for the developer to understand if the analysis made is being the best[36]. This feedback is very important to improve the problem description and to find some details in the client wishes.

This prototype only did the monitoring and showed the results in real time, updated every second, in the computer. We decided to add a new feature that send an email if a specific value of the heart rate was observed during the monitoring. To observe the system working we couldn't use values like "0", which is a result received from a non-beating heart. Instead, we used an achievable value like "100", that is easy to get putting the costumer practicing some exercise. We did that to demonstrate that we can send emails if a value is reached, either if it is "0" or "100".

Figure 4.3: The Proof of Principle.

While presenting the Proof of Principle to the company we received some useful feedback, that allowed us to get a better problem specification. Our client suggestions were the following:

- It would be better to receive the data from the sensor in a Mobile Phone. In terms of user comfort this would be the best option;
- Before sending an alert the user should have a short period of time to cancel it. This is useful to avoid false alerts.

4.1.5 Final Problem Description

After the completion of the stages described above a final problem description was achieved. We divided the problem in two different parts: monitoring and control. Diogo Martins is responsible for the first one and I am responsible for the second one.

The monitoring part will run on a Mobile Phone or a Personal Computer and is responsible for receive the data from a vital signs sensor and send alerts to the Association each time the sensor reports an abnormal reading. Before sending the alert to the Association it is important to warn the client during a small period of time. If the client didn't cancel the warning then the alert should be sent to the Association. The client is able to observe the data received from the sensor in real time and store it in the control system.

The control part will manage the alerts and the vital signs information received from the client. This permits the further analysis of the data in better detail, which can also permit the detection of diseases or possible disasters. A history record is

also kept for each user. One of the objectives is to put the Vital Signs Information accessible to each user, or in certain cases to the Association, to consult and observe.

4.2 System Architecture

From now on I will give more attention to the control part, the main theme of this thesis.

The control system doesn't need an entity to represent the Association Alcor. Although, this does not apply to the Clients. The system can have hundreds or even thousands of Clients. This identifies the class *Client*.

Client is the class that handle all the information and operations of the client. It will store information like alerts and vital signs data. Other types of information like a name, contacts, addresses are not that relevant, because it is information that won't change a lot. Although, I consider the name of the client just as an example.

Alert and *Data* are classes as well. The *Alert* contains the type of the alert ("cardiac arrest", etc.), the GPS position for better results in attending the alert, the date, and two boolean variables indicating if the alert was attended and if the alert was false. The *Data* contains several information about the vital signs (heart rate, body temperature, etc.) and the date when the respective information was captured by the sensor.

The reason why *Alert* and *Data* are classes is to allow future system modifications easily. For example, in the future we add a new sensor to the system that allows the reading of the blood glucose, it would be much easier to add this new variable and change the functions that operate over it in the *Data* class instead of make this changes in the *Client* class. The same thing happens with the *Alert* if we decide to add a new type of alert or change a current one. This gives me the possibility to add new functionalities to the *Alert* or even *Data* classes without even touch the *Client*.

Now that the variables and the entities are identified it is time to identify the

operations. The *Client* has operations to add alerts and data. The *Alert* has operations to declare false alerts and attended alerts. At the moment, the *Data* entity doesn't need any relevant operation. The architecture is presented in the Fig. 4.4.

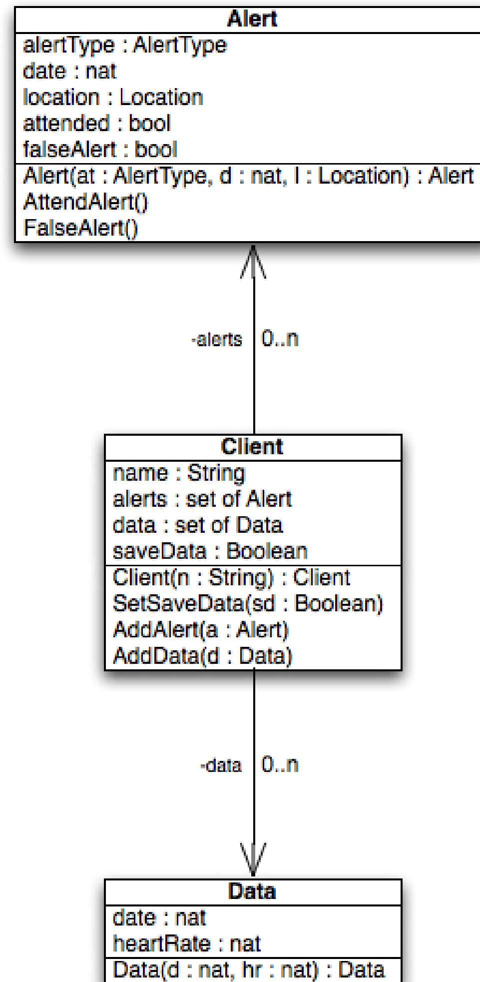

Figure 4.4: The architecture of the first system.

The system was extended to increase the project scope. The new system has more than one Association. The Client can choose several Associations and/or other Clients to receive updates and warnings. The idea is having family and friends connected and alert them if something wrong happens. The Client can have zero followers and use the system just to keep a registry of his/her vital signs and alerts.

One of the goals is having hospitals and other type of health care providers using the system. The Client can choose hospitals and health care providers within his/her

residence area.

To avoid thousands of requests made by clients to Associations, and the inability of response by those Associations, I decided that the Association would be the entity making the request to follow the client. Ideally there would be a pre signed deal before the Association makes the request.

Now that we have more than one Association we need to create a new entity *Association*. The entity *Client* will be changed in order to operate the new functionalities.

The *Client* now has to store information like which clients is he/she following, which entities are following him/her, the requests by other entities to follow him/her, the requests he/she have received to follow clients and information about the attended and unattended alerts. There are some new operations as well. The client can now request to be followed by a client, accept following a client, accept to be followed by an Association, reject following a client, reject to be followed by an Association, stop following a client, stop to be followed by an Association, attend alerts and declare false alerts.

The *Association* is the new entity and stores information about the following clients, the requests made to clients to follow them and information about the attended and unattended alerts. In terms of the operations, the Association is capable of making requests to follow Clients, stop following Clients, attend alerts and declare false alerts. The architecture is presented in the Fig. 4.5.

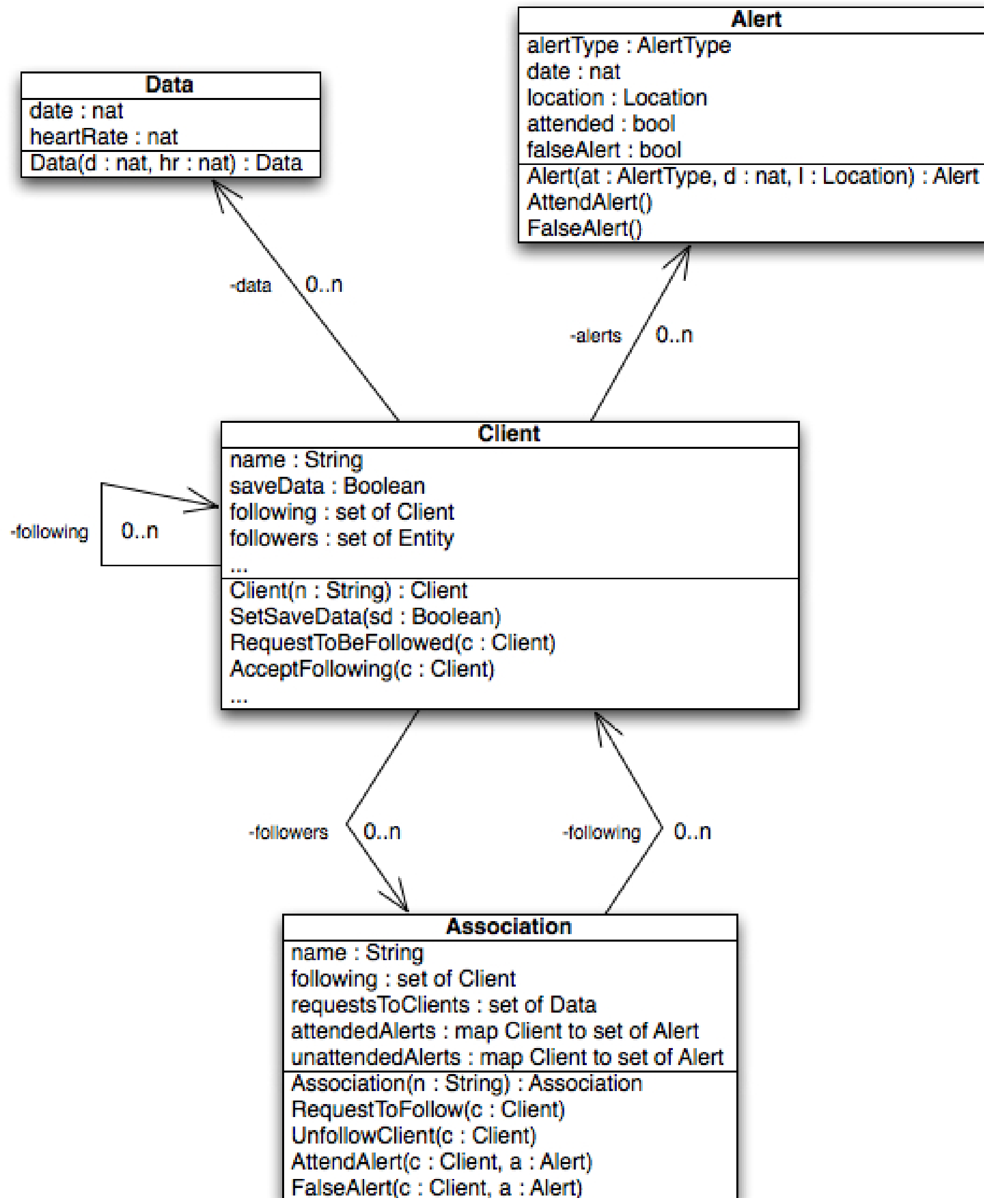

Figure 4.5: The architecture of the second system.

This new system raises a privacy problem. We don't want any entity in the system capable of reading or receiving any alert from clients they don't follow. That is called a system invariant and has to be considered in the model.

4.3 Modeling using VDM++

The model is constructed based on the information gathered in the *Requirements Analysis*. It is important to achieve a consistent model and to do that, abstraction and rigor are fundamental. Abstraction is important for the first system approach where irrelevant details should be omitted. Rigor is the quality of being valid and trustworthy. The combination between this two components is the base for every model to be successful.[28]

4.3.1 Classes Description

In this section I present the types, variables and operations of each class. The idea is to explain in better detail the structure of the model.

Client

```
types

public String = seq of char;
public Entity = Client | Association;
```

In VDM++ the String type is not defined, so I needed to define it. The Entity type was defined just to simplify, some operations may receive either a Client or an Association as a parameter.

```
instance variables

name : String := [];
```

The instance variable "*name*" stores the name of the client.

```
following : set of Client := {};
followers : set of Entity := {};
```

The client can follow other clients and that registry is kept in the "*following*" variable. For the same reason the client can be followed by several entities and that registry is kept in the "*followers*" variable.

```
requestsToClientsToFollowMe : set of Client := {};
requestsByClientsToFollowThem : set of Client := {};
requestsByAssociationsToFollowMe :
    set of Association := {};
```

Before being followed or following someone it is necessary to make and receive requests. Those requests are stored in these three variables. The variable "*request-sToClientsToFollowMe*" stores the requests that the client makes to be followed by other clients. The variable "*requestsByClientsToFollowThem*" stores the requests received to follow other clients. The variable "*requestsByAssociationsToFollowMe*" stores the requests made by Associations to follow the client.

```
alerts : map Alert to set of Entity := {|->};
data : set of Data := {};
```

The client can store alerts and vital data. The variable "*alerts*" stores the alerts sent automatically by the mobile part and the entities that attend each alert. The variable "*data*" stores the vital data also sent from the mobile part.

```
attendedAlerts : map Client to set of Alert := {|->};
unattendedAlerts : map Client to set of Alert := {|->};
```

These are the last two variables of the client class. The "*attendedAlerts*" variable stores the attended alerts from the client. The "*unattendedAlerts*" is responsible for store the alerts from the following clients that were not attended yet by the client.

```
operations

public Client: String ==> Client
```

The operation "*Client*" Is the class constructor. It is used to create instances of the Client. The parameter "*String*" is the name of the client.

```
public RequestToBeFollowed: Client ==> ()
public AcceptFollowing: Client ==> ()
public AcceptBeFollowed: Association ==> ()
public RejectFollowing: Client ==> ()
public RejectBeFollowed: Association ==> ()
public UnfollowClient: Client ==> ()
public RemoveEntity: Entity ==> ()
```

These are the operations that control the clients and Associations requests, acceptances and rejections. The operation "*RequestToBeFollowed*" is used by the client when he wants to make requests to be followed by other clients. "*Accept-Following*" is used when the client agrees to follow another client. Associations can make requests to follow the client, and to accept those requests the "*Accept-BeFollowed*" operation has to be used. The client not always agree to be followed

or following someone. When something like that happens the client can use the operation "*RejectFollowing*" to reject following a client, and use the operation "*RejectBeFollowed*" to reject be followed by an Association. Even though, the client accepts the requests from the Associations, he can decide to stop being followed by a specific Association or client using the "*RemoveEntity*" operation. For the same reason the client can decide to stop following a client using the "*UnfollowClient*" operation.

```
public SendAlert: Alert ==> ()
public AddData: Data ==> ()
public AddSetOfData : set of Data ==> ()
```

For the client to add vital information and alerts to his profile he needs to use this three operation. The "*SendAlert*" operation is used for the client to add alerts and send them to his followers. The "*AddData*" and "*AddSetOfData*" operations are used to add vital information. The first one adds only one package of data, and the second one adds several packages of data to the client profile.

```
public AttendAlert: Client * Alert ==> ()
public FalseAlert: Client * Alert ==> ()
```

When the client follows other clients he can receive alerts. To mark those alerts as attended, the operation "*AttendAlert*" is used. Sometimes alerts can be false, and when the client realizes that he can use the "*FalseAlert*" operation.

Association

```
instance variables

name : String := [];
```

Like clients, Associations have names, and the "*name*" variable stores the name of the Association.

```
following : set of Client := {};
requestsToClients : set of Client := {};
```

These two variables are responsible for keeping a registry of the requests made to clients to follow them, "*requestsToClients*", and the registry of the clients being followed by the Association, "*following*".

```
attendedAlerts : map Client to set of Alert := {|->};
unattendedAlerts : map Client to set of Alert := {|->};
```

The Association can also attend and receive alerts from his following clients. Those registries are kept in the variable "*attendedAlerts*" for the alerts attended by the association, and in the "*unattendedAlerts*" variable for the received alerts from his following clients that are not attended yet.

```
operations

public Association: String ==> ()
```

The operation "*Association*" is the class constructor. It is used to create instances of the *Association* class. The parameter "*String*" is the name of the association.

```
public RequestToFollow: Client ==> ()
public UnfollowClient: Client ==> ()
```

The association to follow clients first has to request to follow them. The operation "*RequestToFollow*" manages those requests. If, for some reason, the association decides that no longer wants to follow a specific client, then the operation "*UnfollowClient*" is used.

```
public AttendAlert: Client * Alert ==> ()
public FalseAlert: Client * Alert ==> ()
```

When the association follows clients then it will receive alerts. Those alerts can be attended using the operation "*AttendAlert*" and can be declared as false using the operation "*FalseAlert*".

Alert

```
types

public AlertType = <Dead> | <CardiacAttack>;
public Lat = real
    inv x == x >= -90 and x <= 90;
public Long = real
    inv x == x >= -180 and x <= 180;
public Location = Lat * Long;
```

The "*AlertType*" is a list of possible alerts that the control system supports. The

other three types will be explained later in the Section 4.3.2.

```
instance variables

alertType : AlertType;
date : nat;
location : Location;
attended : bool := false;
falseAlert : bool := false;
```

These are the instance variables of the *Alert* class. The "*alertType*" is the type of the alert, the "*date*" is the date when the alert occured, "*location*" stores the GPS location of the client at the time of the alert. The last two variables indicate if the alert was attended and if it was a false alert.

```
operations

public Alert: AlertType * nat * Location ==> Alert
```

The Alert operation is the class constructor. It is used to create instances of the *Alert* class.

```
public AttendAlert: () ==> ()
public FalseAlert: () ==> ()
```

This last two operations of the class *Alert* exist to declare the alert as attended, "*AttendAlert*", and to declare the alert as false, "*FalseAlert*".

Data

```
instance variables

date : nat;
heartRate : nat;
velocity : nat;
```

The *Data* class, at the moment, only permits to store information about the client heart rate and the velocity. The "date" variable stores the date when the vital information was captured by the sensor.

```
operations

public Data : nat * nat * nat ==> Data
```

"*Data*" is the only operation of the class *Data*. It is the class constructor and it is used to create instances of the *Data* class.

4.3.2 System Invariants

In the *Requirements Analysis* stage I identified a system invariant which I wrote in **VDM++** as follows:

```
inv forall c in set dom unattendedAlerts &
           c in set following
```

It is clear that one of my concerns is the privacy policy, and the invariant is the

answer to that concern. I don't want clients or associations receiving or reading others clients' alerts and data without being completely authorized.

This invariant is presented in the classes *Client* and *Association*. The variable *unattendedAlerts* is responsible for storing the references for the unattended alerts, and the variable *following* is responsible for storing the references for the clients that are being followed. The invariant says that the clients and the associations cannot have unattended alerts from clients they do not follow.

Now I have to guarantee the invariant preservation within the model. To do that I have to:

1. identify the operations that change or consult the variables presented in the invariant: *unattendedAlerts* and *following*;
2. add the necessary conditions to those operations to guarantee the invariant preservation.

Adding a new client to the *following* variable will not represent a concern, the invariant is not at risk of being unpreserved. Although, removing a client from the *following* variable might represent a privacy concern. For example, if an entity decides to stop following a client, he/she needs to be completely removed from the *unattendedAlerts* variable as well as the *following* variable. The same thing happens when a client decides to stop being followed by an entity.

So, the following items make up the list of operations that may represent a problem for the invariant, and consequently for the system integrity:

- **UnfollowClient**: present in the *Client* and *Association* classes;
- **RemoveEntity**: present in the *Client* class;
- **SendAlert**: present in the *Client* class;
- **AttendAlert**: present in the *Client* and *Association* classes;
- **FalseAlert**: present in the *Client* and *Association* classes.

UnfollowClient is the first operation. This operation removes a client from the *following* variable. To guarantee that the previous scenario doesn't happens I added the following code within the operation:

```
unattendedAlerts := {c} <-: unattendedAlerts;
```

With so, when an entity decides to stop following a client, this client is removed from the following variable and if any unattended alert from the client existed in the entity, they are also removed. The practical result is that the entity will no longer receive alerts and will no longer be able to attend alerts from that client, unless the entity starts following the client again.

RemoveEntity is a very similar operation. The only difference is that first it will remove the entity from the client profile and than remove the client from the entity profile using an operation with the same properties as the *UnfollowClient*.

Adding alerts is another operation that might represent a "problem". It is important to guarantee that only the followers receive the alerts. *SendAlert* is the operation responsible for adding the alert to the client profile and than send it to the followers. To guarantee that all the followers receive the alert I added the following code:

```
for all e in set followers do
     e.ReceiveAlert(self,a);
```

ReceiveAlert is an operation defined in the *Client* class and in the *Association* class, used to receive alerts from the following clients. To guarantee that only alerts from the following clients are received, I added a pre-condition to the *ReceiveAlert* operation:

```
pre c in set following and
    a not in set attendedAlerts(c) and
    a not in set unattendedAlerts(c);
```

c is the client that sends the alert, and *a* is the alert sent. I also added two more properties to the pre-condition to assure that alerts are received only once.

The next operation is *AttendAlert*. This operation has to guarantee that only alerts from following clients can be attended. That is assured with a pre-condition:

```
pre c in set following and
    a not in set attendedAlerts(c) and
    a in set unattendedAlerts(c);
```

Again, *c* is the client that sent the alert, and *a* is the alert sent. In the previous pre-condition there are also two more properties that guarantee that the alert is unattended by the client or by the association.

FalseAlert is the last operation. The only thing that this operation has to guarantee is that an entity that wants to declare a false alert has to be following the client that sent the alert. To do that, a pre-condition was added:

```
pre c in set following;
```

The operations related with the presented invariant are over. But I also have another invariant. The *Alert* class has a data type called *Location* which is represented by two values: latitude and longitude. The latitude assumes values between -90º and +90º, and the longitude assumes values between -180º and +180º.

```
public Lat = real
    inv x == x >= -90 and x <= 90;
public Long = real
    inv x == x >= -180 and x <= 180;
public Location = Lat * Long;
```

4.4 Testing

After modeling, it is important to validate the resultant model.

4.4.1 Using HOL

Using the theorem prover **HOL** is one of the options. If the **HOL** validates the model that means that the model is correct. If the **HOL** fails to validate the model one of two things might have happened: or the tested properties were to complex and I need to simplify them using the PF-transform[37] and repeat the process, or the model is incorrect and it needs to be corrected. The model is validated only when the theorem prover says so, until then the model might not be correct.

To validate the model using **HOL** I have to:

1. translate the model and his proof obligations from **VDM++** to **HOL**.

2. use **HOL** system over the translated model and the correspondent proof obligations to validate the model.

The translation is automatic if the *Automatic Proof System* (APS) tool is used. This tool is still in the developing process, so the translation might not be accurate. In most cases the translated code is not a valid **HOL** code, it needs some changes before running it in the **HOL** system.

My experience turned out to be much more peculiar. First of all, the **VDMTools** generated proof obligations with no expression, meaning that there was nothing to be proved. The second problem was the translator incorporated on the *APS*. This translator is not yet capable of translating models using instance variables or states. The solution that Miguel Ferreira, the developer of the *APS* tool, proposed me was to re-write the entire model to a functional style. That would be the only way of translating the model automatically to **HOL** using the *APS* tool. The other solution was to create the entire model in **HOL** from the scratch.

Both of the solutions were discarded by me. The reason why I don't want to create another model from the scratch is because I don't agree with the creation of two models for the same problem in two different languages. The first solution would be to re-write the entire model which is almost the same thing as create another model.

This type of validation using **HOL** is very important. Although, I can verify the model using other techniques like the *Unit Testing*. I will gain confidence in the model but no certainty. The easiest way to gain certainty is validating using the **HOL** system.

4.4.2 Using VDMUnit

Using the **VDMUnit** framework[28] I will test the model. I want to guarantee that the most delicate operations run properly. For that reason I created a class with several tests to be performed automatically with the help of the **VDMUnit** framework. My main concerns are:

- add connections between clients and between clients and associations;
- operations related with the alerts: adding and attending;
- remove connections between clients and between clients and associations.

Like I previously said, privacy is my main concern. Because of that, adding connections between entities must operate with the expected results. Using the

VDMUnit framework I created a test case where I simulated those operations. In my test case there are three clients and two associations: *client1*, *client2*, *client3*, *association1* and *association2*. I putted the names in here to simplify only. Next are the connections made:

- **client1** will be following: *client3*;
- **client2** will be following: *client1*;
- **client3** will be following: *client2*;
- **association1** will be following: *client1* and *client3*;
- **association2** will be following: *client2*.

Just to remember, if the *client1* follows the *client3*, then the *client1* receives the alerts from the *client3*. So, to guarantee, for example, that the *client1* doesn't receive alerts from the *client2* I have to check if during the process of creating the connections between the entities only the expected ones are created.

Running this test case I didn't found any anomaly with the model, meaning that everything had the expected result.

Now I will test the operations related with the alerts. Adding and Attending alerts could become a problem if a not authorized entity receives and attends alerts that weren't suppose to be received or attended by that same entity. For that reason I created a test case where an alert was added by the *client1*. If everything works the way it should, only the *client2*, the *association1* and the *association2* will receive and attend the alert, the *client3* won't notice the existence of that alert.

With this test case I didn't found any type of anomaly with the model. The operations worked perfectly fine.

At last I will test the connections between entities and the alerts at the same time. For example, if an alert is added by the *client2*, then the *client3* and the *association2* will receive it and, at the same time, they will be able to attend it. But, if the *client2* right after adding the alert decides that the *association2* is no

longer a good alternative for attending his alerts, then the *association2* cannot be able to attend his alerts. If the *association2* is able to attend the alert then an invariant violation is raised.

In my test case I executed the following operations by this same order:

1. *client2* adds the alert;
2. *client3* attends the alert;
3. *client2* removes the *association2* from his followers.

After this operations were executed the *association2* was no longer able to access the alert from the *client2*. This means that the test case didn't revealed any anomaly with the model and all the properties were preserved.

The model reveals some confidence and it is safe to proceed to the prototype stage.

4.5 Prototype

The **VDMTools** permit the automatic translation from **VDM++** to **Java**. In 95% of the VDM++ constructors, the code generator presented in the VDMTools produces correct and executable code.[28]

The **VDMTools Lite** is a free version of the **VDMTools** and do not have this type of functionality available. The translation from **VDM++** to **Java** has to be performed manually. Although, the VDM++ uses the object oriented paradigm, so the translation is not that hard. To easily understand some differences between VDM++ and Java consult the table 4.1.

In the model presented in the Section 4.3 there are at least two important data types: *set* and *map*.

Sets are abstract representations in VDM++, that don't have strong restrictions or properties. In Java there are many other equivalent representations like the

VDM++	**Java**
Class	Class
Operations	Methods
Instance Variables	Class Variables
Values	Constants
Types	Data Types

Table 4.1: Differences and similarities between VDM++ and Java

LinkedLists, *ArrayLists*, *HashSets* and more. In my prototype the order by which the elements are inserted into the sets matters, so the *HashSets* use is excluded. Between the other two I decided to use the *ArrayList*.

Maps in Java are known as *HashMaps*. It has the same properties as the *Map* in VDM++. So, the *Maps* translation from VDM++ to Java does not offer any sort of problem.

The Java language doesn't provide a way of declare invariants, pre and post-conditions without using external tools like **JML**[38] or **Modern Jass**[39, 40]. The only problem with the *JML* is that it is ignored by the java compiler. So, *JML* is not an alternative. Although, the Modern Jass code is compatible with the Java compiler, thanks to the API provided by the Java community that allows to create plugins for the Java compiler.

So, using the **Modern Jass** will be the solution to use the pre and post-conditions. However, it is very simple to add the properties presented in the original model without using the Modern Jass. They are translated to if-clauses in Java code. The invariant, however, disappears in the prototype because it was already tested in the model and it was proved that with the existing conditions the invariant is not violated.

For example, the following code represents the *AddData* operation in VDM++:

```
AddData(d) ==
(
```

```
    data := data union {d};
)
pre d not in set data;
```

In this operation we have a *set* and one property in the pre-condition. The *set*, as I previously said, is transformed into an *ArrayList*. The pre-condition is transformed into an If-Clause with one condition. The *ArrayList* in Java has a method called *contains* which checks if a specific element is presented in the array. That is the solution for the property in the pre-condition. So, the code in Java would be:

```
if (!(GetData().contains(d)))
{
    this.data.add(d);
}
```

If the property fails an exception is raised indicating the problem. When the mobile system uses this method or it gets the expected result or it gets an exception.

I will not present the Java code for the other methods. My idea for this section is explain how the prototype will work.

As you already know, the project is composed by two parts: mobile and control. So, the prototype will also have this two parts, but merged into one single system. However, the control part will be presented and the mobile too, but they will be part of one single program. The idea was to simplify the prototype and, at the same time, not lose the initial properties presented in the model.

Resuming, it is a local program written in the Java language that simulates the entire system. The mobile system receives the vital data from the sensor, and if wanted or necessary sends information to the control system. Since it is all the same program, there is no problem with the connections between the mobile and the control. In future versions, this two parts will be working in separate environments,

eventually the mobile part will communicate with the control part through the Internet.

To avoid create new clients every-time the system starts, the information related to clients, associations, etc., is stored. When the system starts, that information is loaded and the mobile starts sending information periodically to the control system using its methods.

My part of the prototype is independent from the mobile part. The control system doesn't need any method from the mobile to work. The control will only receive information from the mobile. However, the opposite doesn't verify. The mobile part has to use methods from the control part when necessary, for example to store vital information.

4.6 Calculating the Database Schema

In the paper *"Transforming Data by Calculation"*[37] is presented a practical method to calculate the schema of a database from an abstract model using calculus. The relational rules used to calculate the database from the model presented in the Section 4.3 are presented in the previous cited paper[37].

The first class to be calculated is the *Client*. The Client has 10 variables but it is clear that all of them represent data types, for example the variable *following* is a set of Clients and *name* is a String. So, in the Client we have: Client, Association, Alert, Data and String. These are the main data types in the class. Until now, only the String was considered to be a data type and the others were consider to be classes only. But the fact is that every time an instance of those classes is created an identification is also created to identify the instance. So, when we have a set of Clients, in fact what we have is a set of identifiers that point to the client's information.

To easily understand the calculations of the database I will simplify the notation for the data types (table 4.2).

Original	Simplified
Client	C
Association	A
Alert	Al
Data	D

Table 4.2: Data types presented in the Client class for the database calculation

Now we just need to represent each variable and start the calculations.

- **name**: is a String and I will use the letter "N" to represent it;
- **following**: $\mathcal{P}(C)$;
- **followers**: $\mathcal{P}(C + A)$;
- **requestsToClientsToFollowMe**: $\mathcal{P}(C)$;
- **requestsByClientsToFollowThem**: $\mathcal{P}(C)$;
- **requestsByAssociationsToFollowMe**: $\mathcal{P}(A)$;
- **alerts**: $Al \rightharpoonup \mathcal{P}(C + A)$;
- **data**: $\mathcal{P}(D)$;
- **attendedAlerts**: $C \rightharpoonup \mathcal{P}(Al)$;
- **unattendedAlerts**: $C \rightharpoonup \mathcal{P}(Al)$.

As I previously said, each instance of the Client class has an identifier pointing to it. So, the first line of the calculation will have a "C" pointing to all the information presented in the class. Now the calculation:

$C \rightharpoonup (N \times \mathcal{P}(C) \times \mathcal{P}(C + A) \times \mathcal{P}(C) \times \mathcal{P}(C) \times \mathcal{P}(A) \times (Al \rightharpoonup \mathcal{P}(C + A)) \times$

$\times \mathcal{P}(D) \times (C \rightharpoonup \mathcal{P}(Al)) \times (C \rightharpoonup \mathcal{P}(Al)))$

$\cong \{92\}$

$C \rightharpoonup (N \times (C \rightharpoonup 1) \times (C + A \rightharpoonup 1) \times (C \rightharpoonup 1) \times (C \rightharpoonup 1) \times (A \rightharpoonup 1) \times$

$\times (Al \rightharpoonup (C + A \rightharpoonup 1)) \times (D \rightharpoonup 1) \times (C \rightharpoonup (Al \rightharpoonup 1)) \times (C \rightharpoonup (Al \rightharpoonup 1)))$

$\cong \{107\}$

$C \rightharpoonup (N \times (C \rightharpoonup 1) \times (C \rightharpoonup 1) \times (A \rightharpoonup 1) \times (C \rightharpoonup 1) \times (C \rightharpoonup 1) \times (A \rightharpoonup 1) \times$

$\times (Al \rightharpoonup ((C \rightharpoonup 1) \times (A \rightharpoonup 1))) \times (D \rightharpoonup 1) \times (C \rightharpoonup (Al \rightharpoonup 1)) \times (C \rightharpoonup (Al \rightharpoonup 1)))$

$\leq \{112\}$

$C \rightharpoonup (N \times (C \rightharpoonup 1) \times (C \rightharpoonup 1) \times (A \rightharpoonup 1) \times (C \rightharpoonup 1) \times (C \rightharpoonup 1) \times (A \rightharpoonup 1) \times$

$\times (Al \rightharpoonup (C \rightharpoonup 1)) \times (Al \times A \rightharpoonup 1) \times (D \rightharpoonup 1) \times (C \rightharpoonup (Al \rightharpoonup 1)) \times$

$\times (C \rightharpoonup (Al \rightharpoonup 1)))$

$\leq \{78, 112, 92\}$

$C \rightharpoonup (N \times (C \rightharpoonup 1) \times (C \rightharpoonup 1) \times (A \rightharpoonup 1) \times (C \rightharpoonup 1) \times (C \rightharpoonup 1) \times (A \rightharpoonup 1) \times$

$\times (Al \rightharpoonup 1) \times (Al \times C \rightharpoonup 1) \times (Al \times A \rightharpoonup 1) \times (D \rightharpoonup 1) \times (C \rightharpoonup 1) \times$

$\times (C \times Al \rightharpoonup 1) \times (C \rightharpoonup 1) \times (C \times Al \rightharpoonup 1))$

$\cong \{86, 89\}$

$C \rightharpoonup (N \times (4 \times C \rightharpoonup 1) \times (2 \times A \rightharpoonup 1) \times (Al \rightharpoonup 1) \times (Al \times C \rightharpoonup 1)$

$\times (Al \times A \rightharpoonup 1) \times (D \rightharpoonup 1) \times (2 \times C \rightharpoonup 1) \times (2 \times C \times Al \rightharpoonup 1))$

$\leq \{112\}$

$(C \rightharpoonup N) \times (C \times 4 \times C \rightharpoonup 1) \times (C \times 2 \times A \rightharpoonup 1) \times (C \times Al \rightharpoonup 1) \times$

$\times (C \times Al \times C \rightharpoonup 1) \times (C \times Al \times A \rightharpoonup 1) \times (C \times D \rightharpoonup 1) \times$

$\times (C \times 2 \times C \rightharpoonup 1) \times (C \times 2 \times C \times Al \rightharpoonup 1))$

The following items explain the results obtained:

- $C \rightharpoonup N$: This is the table that represents the Client. It has the Client ID as a key and stores the name of the client.

- $C \times 4 \times C \rightharpoonup 1$: This table makes the relations between clients. It has 4 identifiers indicating the relation between the first and the second client: following, follower, request to follow and request to be followed.
- $C \times 2 \times A \rightharpoonup 1$: This table makes the relations between clients and associations. It has 2 identifiers indicating if the association following the client or if the association has a request to follow the client.
- $C \times Al \rightharpoonup 1$: This table makes the relation between the client and his alerts.
- $C \times Al \times C \rightharpoonup 1$: Table that relates the Client with the Alert and the Clients that attended that alert.
- $C \times Al \times A \rightharpoonup 1$: Table that relates the Client with the Alert and the Associations that attended that alert.
- $C \times D \rightharpoonup 1$: Table that relates the Client with his vital information.
- $C \times 2 \times C \times Al \rightharpoonup 1$: Table that relates the Client with the attended and unattended alerts from the other clients.

There is another table that I didn't mention. The reason why is because that table makes no sense. It has two identifiers related with alerts, and that is no need to keep that information because we already have a table that stores that information.

The second class is the Association. There is no new data types, so I will use the previous notation.

Association has 5 variables:

- **name**: is a String and I will use the letter "N" to represent it;
- **following**: $\mathcal{P}(C)$;
- **requestsToClients**: $\mathcal{P}(C)$;
- **attendedAlerts**: $C \rightharpoonup \mathcal{P}(Al)$;
- **unattendedAlerts**: $C \rightharpoonup \mathcal{P}(Al)$.

For the same reason as with the Client I will start with the Association Identifier pointing to the rest of the information.

$$A \rightharpoonup (N \times \mathcal{P}(C) \times \mathcal{P}(C) \times (C \rightharpoonup \mathcal{P}(Al)) \times (C \rightharpoonup \mathcal{P}(Al)))$$

$$\cong \{92\}$$

$$A \rightharpoonup (N \times (C \rightharpoonup 1) \times (C \rightharpoonup 1) \times (C \rightharpoonup (Al \rightharpoonup 1)) \times (C \rightharpoonup (Al \rightharpoonup 1)))$$

$$\leq \{78, 112, 92\}$$

$$A \rightharpoonup (N \times (C \rightharpoonup 1) \times (C \rightharpoonup 1) \times (C \rightharpoonup 1) \times (C \times Al \rightharpoonup 1) \times (C \rightharpoonup 1) \times$$

$$\times (C \times Al \rightharpoonup 1))$$

$$\cong \{86, 89\}$$

$$A \rightharpoonup (N \times (2 \times C \rightharpoonup 1) \times (2 \times C \rightharpoonup 1) \times (2 \times C \times Al \rightharpoonup 1))$$

$$\leq \{112\}$$

$$(A \rightharpoonup N) \times (A \times 2 \times C \rightharpoonup 1) \times (A \times 2 \times C \rightharpoonup 1) \times (A \times 2 \times C \times Al \rightharpoonup 1))$$

The following items explain the results obtained:

- $A \rightharpoonup N$: This is the main table of the Association. It stores the name for each Association.

- $A \times 2 \times C \rightharpoonup 1$: Table with 2 identifiers. It relates the Association with the Client and indicates if the association is following or if it has a request to follow the client.

- $A \times 2 \times C \times Al \rightharpoonup 1$: Table that relates the Association with the attended and unattended alerts from the clients.

The other table that I didn't mention has the same problem has the unmentioned table from the client. It has irrelevant information and it is not needed for the purpose of this problem.

It only misses two classes: Alert and Data. This classes are very simple. With them I will use the same notation presented, in the table 4.2, only for the major data types.

So, next is the Alert:

$$Al \rightharpoonup (2 \times Date \times Location \times Attended \times FalseAlert)$$

This table has the Alert ID as a key. The 2 identifiers indicate the type of the alert: "CardiacAttack" or "Dead".

And the last is Data:

$$D \rightharpoonup (Date \times HeartRate \times Velocity)$$

This last table has the Data ID as a key.

Now that all the resultant tables are identified it is necessary to identify possible repeated tables. Alert and Data class produced unique tables, so no problem was found with them. But if we look at the Association and the Client we can find repeated tables. The middle table, the one that represents the following clients and requests, is already presented in the Client resultant tables. They represent the same type of information. It is important to know how to interpret the results to avoid such "problems".

So, in the end we have 8 tables from the Client class, 2 tables from the Association class, 1 table from the Alert class and 1 table from the Data class, with a total of 12 tables for the entire model. This results are very important because they represent exactly the model structure, so if in the future when the database is implemented using this schema, the model won't need to change much because the structure of the data types is preserved.

Chapter 5

Conclusions

My main objective was to create a functional prototype of a Continuous Monitoring Control System. To create that prototype several tasks were performed. In this section my personal remarks and conclusions about the fields studied and methodologies followed are presented.

The most important phase to understand a client's needs is the Requirements Analysis. This stage was carefully considered in the developed system. Those requirements were completely identified when a Proof of Principle was created. Presenting it to the client was important to realize if the right direction was being followed.

From the Requirements Analysis, I defined the System Architecture which is the base for the Software Model construction. The modeling stage is very important to identify and define the system behavior properties. Through testing or analytical methods this properties can be verified. This verification leads to the model validation. This is important to give confidence to the developer to proceed to the implementation. VDM++ was the modeling technology chosen to model the Control system.

My previous experience with **VDM++** was practically none. I learned how to use it in my first year of the Masters Degree but I had never used VDM++ before on a project. It is a widely used modeling language and I it was very helpful in this

project. Since **HOL** wasn't used, why not to use **Java** instead of VDM++?

Even though, the Java language has tools like **JUnit**[41], which is a testing framework to create unit tests, and **Modern Jass**[39], which permits the creation of pre, post-conditions and invariants in Java, using VDM++ should not be discarded. VDM++ has a level of abstraction impossible to reproduce using Java. VDM++ allows the developer to focus only on the problem properties, using Java the developer has to think on implementation details too. Problems with implementation will distract from the essential properties.

Using the VDMTools it is possible to generate proof obligations to check by hand or using a **theorem prover**. The objective it to validate the model produced.

I tried to use a theorem prover (HOL) but I discarded this option because there is no effective automatic translation tool from VDM++ to **HOL**[26]. Although, a unit testing framework called **VDMUnit**[28] was used to verify my model. The test cases used evaluated to true. That gave me confidence to start building the prototype.

Using formal techniques the database schema can be calculated from the abstract model. These techniques are very useful to produce correct database structures.

5.1 Future Work

The project presented in this thesis is not over. The following items are my next tasks that will be completed in the near future:

- create a web service for the control system;
- develop a front-end application for the control system.

The web service will be essential for the project. Using a web service I will create an application capable of running on the internet. The mobile system will be able to access my methods and interact with the control system through the internet.

For most of the companies that is an essential point. A web service also permits the interaction between two different platforms. That is also essential for the mobile part to connect with the control system.

Developing a front-end application that access the control system will be important for the users to read and manage their profiles. This front-end application will be probably a web page. That will permit any user to access the system from anyplace and check the vital information updates from the clients they follow.

What do I expect for the project in the future? I am very happy to see companies like *Alcor, Life Extension Foundation*, believing in such technologies. But I would like to see more companies and persons working together with the aim of improving our well being and independence. The *Continua Health Alliance*[20] is trying to do so.

Detect diseases, specially the cardiac ones, in time for the affected person to be cured, preventing then a possible disaster, will be a major step for our lives. That is possible today thanks to the advances in the technology world. We cannot run in the opposite direction. I intend to build a system capable of preventing diseases and providing the best medical treatment possible. I hope that a large number of persons can take a major benefit from my system.

Bibliography

[1] "U.S. Census Bureau International Data Base with demographics data for the entire world with predictions until 2050." `http://www.census.gov/ipc/www/idb/index.html`.

[2] "Frost & sullivan." `http://www.frost.com/`.

[3] "Polar usa." `http://www.polarusa.com/us-en/`.

[4] D. Martins, "Reliable software development in a vital signs monitoring system," Master's thesis, University of Minho, November 2009. Under Revision.

[5] T. R. Hansen, J. M. Eklund, J. Sprinkle, R. Bajcsy, and S. Sastry, "Using smart sensors and a camera phone to detect and verify the fall of elderly persons," in *European Medicine, Biology and Engineering Conference*, (Prague, Czech Republic), November 2005.

[6] T. Gao, L. K. Hauenstein, A. Alm, D. Crawford, C. K. Sims, A. Husain, and D. M. White, "Vital signs monitoring and patient tracking over a Wireless network," in *Johns Hopkins APL Technical Digest*, vol. 27, pp. 66–74, 2006.

[7] T. Landolsi, A. R. Al-Ali, and Y. Al-Assaf, "Wireless stand-alone portable patient monitoring and logging system," *Journal of Communications*, vol. 2, no. 4, 2007.

[8] D. W. Curtis, E. J. Pino, J. M. Bailey, E. I. Shih, J. Waterman, S. A. Vinterbo, T. O. Stair, J. V. Guttag, R. A. Greenes, and L. Ohno-Machado, "Smart – an integrated wireless system for monitoring unattended patients," *Journal of the American Medical Informatics Association*, vol. 15, no. 1, 2008.

[9] "Calit2 : California institute for telecommunications and information technology." http://www.calit2.net/.

[10] D. Ramsey, "New wireless devices could help consumers keep track of their vital signs." Website, December 2007. http://ucsdnews.ucsd.edu/newsrel/science/12-07NewWirelessDevices.html.

[11] "Website of the cardionet company: "cardionet – get to the heart of the problem"." http://www.cardionet.com/.

[12] "Link to the cardionet mcot product." http://www.cardionet.com/patients_02.htm.

[13] S. A. ROTHMAN, J. C. LAUGHLIN, J. SELTZER, J. S. WALIA, R. I. BAMAN, S. Y. SIOUFFI, R. M. SANGRIGOLI, and P. R. KOWEY, "The diagnosis of cardiac arrhythmias: A prospective multi-center randomized study comparing mobile cardiac outpatient telemetry versus standard loop event monitoring," *Journal of Cardiovascular Electrophysiology*, vol. 18, no. 3, 2007.

[14] "Website of the corventis company." http://www.corventis.com/.

[15] ""remote monitoring of the heart – wearable, wireless technology detects early signs of heart failure" is an article by david talbot that was published in the technology review (published by mit) in april 2009." http://www.technologyreview.com/biomedicine/22519/.

[16] "Website of the biodevices, s.a., a spin-off from ieeta (institute of electronics and telematics engineering of aveiro / university of aveiro) with the mission of developing, commercialize and export biomedical engineering solutions for medical diagnosis support." http://www.biodevices.pt.

[17] "Website for the product vitaljacket made by the biodevices, s.a., company." http://www.vitaljacket.com.

[18] "Website of the company zephyr technologies." http://www.zephyr-technology.com/.

[19] "Website of the independent project zephyr open." http://code.google.com/p/zephyropen/.

[20] "Website of the continua health alliance." http://www.continuaalliance.org/.

[21] "Vdmtools: advances in support for formal modeling in vdm," *SIGPLAN Not.*, vol. 43, no. 2, pp. 3–11, 2008.

[22] C. B. Jones, *Systematic software development using VDM (2nd ed.).* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[23] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie, *Proof in VDM: a practitioner's guide.* New York, NY, USA: Springer-Verlag New York, Inc., 1994.

[24] R. Milner, M. Gordon, and C. P. Wadsworth, "Edinburgh lcf: A mechanised logic of computation," *Lecture Notes in Computer Science*, vol. 78, 1979.

[25] M. J. C. Gordon, "From lcf to hol: a short history," in *Proof, language and interaction: essays in honour of Robin Milner*, pp. 169–185, MIT Press, 2000.

[26] "Hol website." http://hol.sourceforge.net/.

[27] M. A. Ferreira, "Implementing the Overture Automatic Proof System," 2009. Submitted for publication.

[28] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, *Validated Designs For Object-oriented Systems.* Santa Clara, CA, USA: Springer-Verlag TELOS, 2005.

[29] "Windows mobile website." http://www.microsoft.com/windowsmobile/en-us/default.mspx.

[30] "Android website." http://www.android.com/.

[31] "iphone os website." http://developer.apple.com/iphone/.

[32] "Symbian os website." http://www.symbian.org/.

[33] "W3c - web services architecture." http://www.w3.org/TR/ws-arch/.

[34] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, Inc., May 2007.

[35] E. Cerami, *Web Services Essentials*. O'Reilly Media, Inc., February 2002.

[36] R. Machado, J. Fernandes, and P. Ribeiro, *Engenharia e Gestão de Software*. Colecção Engenharia de Software, Lisboa: FCA - Editora de Informática. In preparation.

[37] J. N. Oliveira, "Transforming data by calculation," in *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, (Berlin, Heidelberg), pp. 134–195, Springer-Verlag, 2008.

[38] "Website of the jml - the java modeling language." http://www.cs.ucf.edu/~leavens/JML/.

[39] "Website of the modern jass technology." http://modernjass.sourceforge.net/.

[40] J. Rieken, "Design by contract for java - revised," Master's thesis, Carl Von Ossietzky University, April 2007.

[41] "Website of the junit testing framework." http://www.junit.org/.