Service Oriented Computing and Applications manuscript No.
(will be inserted by the editor)

Analysis and Framework-based Design of a Fault-Tolerant Web Information System for m-Health

Florencia Balbastro, Alfredo Capozucca, and Nicolas Guelfi *

Laboratory for Advanced Software Systems
University of Luxembourg, Luxembourg

Received: date / Revised version: date

Abstract The e-health domain has for objective to assist and manage citizens' health. It concerns many actors like patient, doctors, hospitals and administration. Current and forthcoming generations of application will be web based and will integrate more and more mobile devices. In such application domain, called m-health, dependability is a key notion. This paper presents, through a case study, how we can analyse and design an application that controls the insulin injection and that is embedded in a mobile device belonging to an e-health Web Information System (WIS). In order to ensure the dependability of the control systems, we show how to use Coordinated Atomic Actions (CAA). CAAs provide well defined concepts for fault tolerance, error-detection and error recovery in a distributed context where competitive and cooperative concurrencies are considered. In order to implement our design, we explain how to use a development framework that we have made to implement CAA. This framework, called CAA-DRIP, originally was not tailored for mobile fault-tolerant applications. Thus, in this paper, we also propose an adaptation of CAA-DRIP for mobile devices.

Key words Dependability, Fault tolerance, Mobility, Coordinated Atomic Actions, CAA-DRIP

1 Introduction

Dependability is the ability to deliver service that can justifiably be trusted [1]. Its main goal is to avoid service failures that are more frequent and more severe than the acceptable for the user. The attributes associated to dependability are reliability, availability, confidentiality, integrity, safety and maintainability. Depending on the application, each of these attributes may be emphasized or not. The means to achieve dependability are fault prevention, fault removal, fault tolerance (FT) and fault forecasting. We focus on FT as the method for obtaining dependability. The main goal of FT is to help the software system to deliver its

* This research was supported by the Luxembourg Ministry of Higher Education and Research under the project number MEN/IST/04/04.

correct service (as defined by its specification) despite the presence of faults (i.e. defects in the software due to programmers' mistake or abnormal behaviour of the environment that surrounds the software system). Therefore, a fault tolerance technique is designed to allow a software system to tolerate faults that remain or appear in the software system after its development.

Fault tolerance is usually implemented by error detection and the subsequent system recovery. The error detection phase raises an error signal or message within the system immediately after an error is noticed. The techniques to detect errors can be concurrent, if they take place during the service delivery, or they can be preemptive, if they act when the service is suspended and thus latent errors and dormant faults can be found [1]. The recovery consists of error handling (to leave the system into a state without detected errors) and fault handling (to leave the system into a state without faults that can be activated again).

Nevertheless, FT techniques do not provide explicit protection against faults introduced in the requirement specification phase. This is the reason why the use of complementary techniques that give support at this stage, as for example "Exceptional Use Cases", is an important issue.

Software fault tolerance techniques should be used in every domain where the cost and consequences of a failure can cause serious injuries on human beings, destruction of human-made or natural systems, or business failure. This is the case for e-health. The concept of e-health has emerged between other "e-fields", like e-mail, e-commerce and e-solutions, all of which encompasses the new possibilities that the Internet is opening to each area. E-health is a field at the intersection of medical informatics, public health and business. It covers all health services and information delivered or enhanced through the Internet and related technologies [2]. Furthermore, the continuous increase in processing power and capacity of mobile devices (e.g. PDAs, smart phones) allows us to start thinking to develop e-health software systems that integrate mobility of the patients while ensuring their correct health management (e.g. remote heart monitoring of cardiac patients, food allergies that can be controlled using remote access to knowledge data bases and mobile diabetes treatment). These mobile e-health software systems allow patients to increase their quality of life and hospitals to release patients faster to avoid being overcrowded (which is very important for third world countries). Therefore, these types of software systems along with the potential of wireless technology enable users (professionals, patients, etc.) to conveniently access information and knowledge, independently of their location, and facilitate the communication between the different participants of the e-health information system.

Obviously e-health software systems need to be dependable because they are used to deal with human beings treatments and/or sensitive information. Between the attributes associated to dependability, the safeness -for the absence of catastrophic consequences- in such systems must be emphasized. Regarding the use of the web to transmit information from one part of the system to another, a delay on the transmission should not cause a failure of the service which can provoke a death or injuries to a patient. Therefore, if a high level of availability and reliability of a web information system cannot be ensured, we need to either build the subsystems without relying on the connection for their well functioning, or apply techniques for tolerating the temporarily disconnection from the network. Thus we are facing the engineering of complex distributed systems that may rely on the web and that may include mobile devices. In order to ensure the dependability of such systems, we

[1] A fault in the system is the cause of an error, but only if it is active it produces an error, otherwise it is dormant. The error that are present but not detected are said to be latent errors.

need to use techniques which come with special features to ensure dependability in complex environments (advanced frameworks). The Coordinated Atomic Action (CAA) [3] is an advanced conceptual framework which aim is to assist engineers in the design of complex distributed software systems with high availability, safety and reliability requirements through fault tolerance. More specifically, CAA focus on concurrent software systems consisting of cooperative and competitive components and it provides fault tolerance by means of cooperative exception handling. CAAs have been successfully used in several case studies [4–6] that demonstrate high usefulness and general applicability of the approach.

In order to efficiently master the development of such complex applications providing dependability, we need to use adapted development tools. As shown in [1] a good strategy at implementation level is to make use of advanced frameworks for fault tolerance. Concerning CAA, CAA-DRIP [7] is the associated implementation framework that one may use for developing dependable distributed application. However, CAA-DRIP was not thought for mobile devices and has never been used in the context of e-health web information systems (WIS) before.

Our e-health WIS studied in this paper is the Fault-Tolerant Insulin Pump Therapy, which implements the Continuous subcutaneous insulin infusion [8] (CSII or "insulin pump therapy") as an option for people with diabetes. Diabetes is a disease that occurs when the body does not make enough insulin or does not use insulin in the right way. Insulin is a hormone that helps the body use sugar (glucose) for energy. Because the body has a problem with insulin, sugar builds up in the blood (hyperglycemia). Between the different types of diabetes, type 1 (i.e. the pancreas makes little or no insulin because the islet b cells, which produce insulin, have been destroyed through an autoimmune mechanism) is usually treated with the CSII technique.

The e-health WIS is composed of two software subsystems; one is the FTIP Administration and the other is the FTIP control system. The former provides features to manage the therapy in order to improve its monitoring and increase the feedback between doctors and patients. The latter, which is an improved version of [9] and it is described in details in this paper, is the software subsystem carried by the patient in a mobile device and used to control his blood sugar levels.

The rest of the paper is structured as follows: Section 2 gives an introduction to mobility and the utilised technologies, the fault tolerance technique (CAA) and the implementation framework that supports its semantics (CAA-DRIP), in Section 3 requirements, design and implementation of the considered case study are described. The paper wraps up with conclusions and future work.

2 Background

We are interested in the engineering of dependable mobile software systems that can benefit from an SOA design. For this purpose, the concepts of Coordinated Atomic Actions, SOA and mobility are presented.

2.1 Coordinated Atomic Actions (CAAs) and COALA

Coordinated Atomic Action (CAA) is a fault-tolerant mechanism that uses concurrent exception handling to achieve dependability in distributed and concurrent systems. Thus,

systems that have to comply with their specification in spite of faults can be developed using CAA mechanism. This mechanism unifies the features of two complementary concepts: conversation and transaction.

Conversation [10] is a fault-tolerant technique for performing coordinated error recovery in a set of participants that have been designed to interact with each other to provide a specific service (cooperative concurrency). Objects that are used to achieve the cooperation among the participants are called shared objects. For each process that enters a conversation, it is established a recovery point. They enter the conversation asynchronously and they are allowed to exchange information between them but not with any outside process. To leave the conversation they need to satisfy their acceptance tests. If all tests have been passed successfully, then they leave the conversation synchronously. In other case, they are restored to their recovery points. This scheme means that if an error occurs in one of the process inside a conversation, then all of the processes that participate in the conversation have to rollback to their recovery points. In some complex contexts, it is necessary to give another possibility other than coming back to the initial state, after a long set of operations.

Transactions [11] are used to deal with competitive concurrency on objects that have been designed and implemented separately from the applications that make use of them. These kinds of objects are named external objects. Here we don't speak about shared objects, since only one thread participates in each transaction. The concept of transaction has being evolving from flat transaction, transactions with savepoints, nested transactions, and more advanced ones, in order to give it more flexibility. The problem encountered in some applications was that long-lived transactions can hold locks on shared objects for long periods of time. This is produced because of the strict property of isolation.

The concept of CAAs takes the cooperative features from conversations and the competitive ones from transactions, adding a mechanism for handling exceptions in a synchronised way.

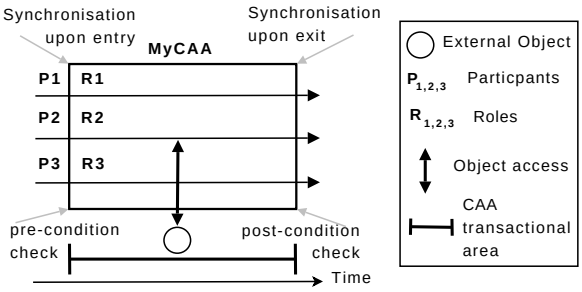

Fig. 1 A simple CAA

As it is shown on Figure 1, one CAA characterises an orchestration of operations executed by a group of roles that exchange information among themselves, and/or access to external objects (concurrently with other CAAs) to achieve a common goal. The CAA starts when all its roles have been activated and they meet a pre-condition. The CAA finishes when all of them have reached the CAA end, and a post-condition is satisfied. This behaviour returns a normal outcome to the enclosing context. If for any reason an exception has been raised in at least one of the roles belonging to the CAA, appropriate recovery measures have to be taken. Facing this situation, a CAA provides a quite general solution for fault tolerance

based on exception handling. It consists of applying both forward error recovery (FER) and backward error recovery (BER) techniques [12].

Another important characteristic of CAAs is that they can be designed in a structured way using nesting and/or composition. Nesting is defined as a subset of the participants used to carry out the roles of a CAA. These chosen participants define a new CAA inside the enclosing context defined by calling CAA. The participants of the nested CAA play different roles with respect to the roles belonging to the calling CAA. The activities carried out inside of the nested CAA are hidden for any other CAAs. The access to external object within a nested CAA is performed as in nested transactions.

Composite CAAs [13] are different from nested CAAs in the sense that the use of composite CAAs is more flexible. For example, a nested CAA with two roles can only be used inside an enclosing CAA that is played by at least two participants. Composite CAAs do not have this type of restriction. A composite CAA is an autonomous entity with its own roles and objects. The internal structure of a composite CAA (i.e. participants, accessed objects and roles) is hidden from the calling CAA. The role that calls the composite CAA, will synchronously wait for the outcome. Then, the calling role resumes its execution according to the outcome of the composite CAA. If this CAA terminates exceptionally, its calling role raises an internal exception that is, if possible, locally handled. If local handling is not possible, the exception is propagated to all the peer roles of the calling CAA for coordinated error recovery.

```
1  ;; ***********************************
2  Caa MyCAA;
3  ;; ***********************************
4  Interface
5     Use myModule;
6     Roles
7        R1, R2: myType, R3;
8
9     Body
10       Use myModule;
11
12       Role R1
13          Begin
14                ;; role R1's instructions
15          End;
16       End R1;
17
18       Role R2(myExternalObject:myType)
19          Begin
20                ;; role R2's instructions
21          End;
22       End R2;
23
24       Role R3
25          Begin
26                ;; role R3's instructions
27          End;
28       End R3;
29
30 End MyCAA;
```

Fig. 2 MyCAA written in COALA

In an attempt to clarify the CAA concepts mentioned previously, a formal notation (syntax and semantics) called COALA [14] has been defined. It not only brings simplicity

and clarity over the CAA's concepts, but also rigour since its semantics has been formally defined through CO-OPN/2 [15]. As an example to show how this notation look likes, on Figure 2 appears an sketch of the COALA program corresponding to *MyCAA* (see Figure 1).

The interpretation of such COALA example is straightforward due to the simplicity of *MyCAA*. The keyword **Caa**, follows by the name of the CAA, is used to introduce a CAA module (line 2, on Figure 2). Each CAA module consists of two parts: an *interface* and a *body*. The interface (lines 4-7) defines all the components that are visible from outside, whereas the body (lines 9-28) describes components, behaviour and particular mechanisms of the CAA. Therefore, COALA notation allows a fully description of a CAA, whereas that the notation employed on Figure 1 only allows to described the structural aspects of a CAA. That is the reason why the COALA program on Figure 2 does not include any details regarding *MyCAA*'s behaviour. However, the graphical notation is useful to get a quick overview of the design, which is not immediate looking at a large COALA program. In this manner, both notations are important since they complement to each other in the description of the design.

2.2 CAA and SOA

As it was previously mentioned, one of the attributes associated to dependability is the maintainability or the ability to undergo repairs and modifications. By designing a software system with the principles of service-orientation, this capability of evolution and maintenance is achieved.

The main features of service-orientation are the way that the system is divided into logic units, and the way that these units can be connected. Like applying the principle of separation of concerns, each of the units represents a functionality that can evolve independently from the rest. This separation allows the units to be reused and easily interchanged with other components that give the same functionality. As isolated each of them may not provide the main service of the system, they must share a common framework to achieve the communication between them. These units are called services.

The best independency that can be reached between services, while ensuring reliability, is by sharing a mechanism for publishing and discovering them. And thus, the continuity of the correct service provision can be supported by the replacement of the independent components at runtime. Clearly, the reliability of the whole system is not achieved if a needed functionality is not available at runtime. Though, the reliability gets benefit by taking care of having the necessary services always available.

The registry is the component where the services can announce themselves and let the others be able to discover them. The services publish their descriptions in the registry. The description is composed by the definition of the service (interface, operations, parameters, technology used to communicate, physical address where the service can be accessed, name) and related documents (like schemas or definition's structures, policies and legal files) [16].

Apart from the evolution and maintenance capabilities, there are other obvious benefits that service-orientation provides, like federation. Regarding the fact that the services hide the implementation of their computations, grouping them is possible even though they can use totally different technologies. The only requirement for establishing the communication and composition is that they have to share the same framework. One such framework is messaging.

Nowadays, software systems must adapt constantly to the evolution of technology, and may take advantage from similar functionalities offered by different vendors. Moreover, there has been an upward trend in service-oriented developments in the last few years, which seems that software providers will offer their products as services in the near future. This is a quite good motivation to develop a system following the service-oriented paradigm and applying the current standards for the communication framework. This reason and the above mentioned benefits, plus the requirements of dependability that an e-Health system should ensure, motivated us to combine the concepts of CAA and SOA.

For the CAA concept, the external objects can be accessed competitively by different CAAs that execute concurrently. In order to avoid information smuggling between them, the external objects must behave atomically. In other words, the changes performed on external objects by a CAA must be hidden from external agents (threads or CAAs) until the CAA finishes and commits its effects. The responsible of ensuring the integrity of an external object is the object itself.

When a system uses services to manipulate objects, in the design and code it can be found a call to a service rather than a direct call to an object's operation. But, to reflect the independency between services, a service should access the registry to extract the information of published services and then make the call to the one of its interest. For the sake of reusability, services should be stateless. That is, while they are computing a request, they might be statefull. But the time while they hold the state information should be minimized. In contrast, the registry is statefull. It must hold the information related to the published services.

For a CAA design, the access to the registry can be performed like an access to an external object. In the case that an object holds the information of the registry, there are different possible designs to manage it. One possibility is that there is one role in charge of the management of the registry, so that it receives the publish requests from the services. And the other roles only requests for the information of the published services (they do not modify the registry). Other possibility is to let more than one role to manage the registry. The first possibility includes an option to build the registry as a set of objects. Each of these objects contains the information of only one service (its description with its associated documents).

For ensuring their own integrity, objects that participate in a transactional system should provide some operations for locking and unlocking themselves. And so, the effects done on an object can be hidden to other transactions until the modifications are committed. The same concept applies for CAAs (but the CAAs concept includes another possible outcome, other than the "all or nothing"). These operations that ensure the integrity include an operation for locking that can be called by only one thread of execution at the same time (i.e. begin), an operation to unlock and save the modifications (i.e. commit), and an operation to unlock without saving any changes (i.e. abort).

If these objects must be manipulated by services, the services have to provide a mechanism to ensure the integrity of the objects. Thus, if it is necessary to lock an object to use it by a CAA, a service providing the management of the object must be able to lock it, until the CAA finishes and either commits its result or aborts.

In some systems, it is not possible or not necessary to lock some resources until a transaction or CAA finishes. In this case, the objects or the services that manipulate them provide operations to do a compensation, so as to leave the system in a state semantically equal to the previous known consistent state.

2.3 Mobile devices

Millions of wireless devices, such as personal digital assistants (PDA) and mobile phones, are used all over the world, and with them comes the usage of new applications to help the users in their daily tasks. Usually, this type of applications needs to interact with other ones through remote services in order to achieve their goals. Some of them are aware of the network, in the sense that they take advantage of the connection, but do not rely on it because of the air charges.

Wireless networks were developed as a special possibility of connection. But with the advances of technology, the intuitive use and the ease of access, they are gaining more places, and also the places where the wired networks were used. It is good to remark that this field is in the phase of development and therefore, new standards and protocols have been built in the last ten years. They are being tested for the consequent adoption, update or ruled out. Nowadays, the most used mobile devices give different possibilities for connecting to a network or to the internet. In addition to the cellular network of mobile devices, they can also use the wireless LAN based on IEEE 802.11 standards, Bluetooth and/or infrared port.

Developers have to deal with the stricter limitations of the mobile devices, such as small processors, memories, storage, and a reduced graphical user interface. However, the capabilities of the new devices continue increasing at the point of having several megabytes of memory for running applications, a fact that was unimaginable some years ago. New developing tools have been already built and need to continuously evolve due to the constant changes of the new technologies.

In our case study the chosen technology was Java [17] due to several reasons. The object oriented, distributed, platform independent and multithreaded aspects are the first qualities to mention. But other important features are that: it is deployed in billions of devices around the world, it deals efficiently with security issues, and it keeps on developing while new advances are coming. Java 2 Micro Edition (J2ME) [18] is a collection of technologies and specifications that are designed for different kinds of small devices. J2ME, therefore, is divided into configurations, profiles, and optional packages. Configurations are specifications that detail a virtual machine and a base set of APIs that can be used with a certain category of devices. The virtual machine that is specified is either a full Java Virtual Machine (JVM) or some subset of the full JVM. At the moment of writing, there are two different configurations, the Connected Device Configuration (CDC) for programming larger handheld devices like powerful PDAs, and the Connected Limited Device Configuration (CLDC) to support devices with limited resources, like mobile phones. A profile is set on a configuration and adds more specific APIs to make a complete environment for building applications. For example, the Mobile Information Device Profile (MIDP), that is widely accepted and deployed in small devices. An optional package provides functionality that may not be associated with a specific configuration or profile. Some examples of optional packages are the RMI (for the Remote Method Invocation) and Bluetooth ones. For more details on J2ME platform you can see [19].

2.4 CAA-DRIP and Mobility

CAA-DRIP is a set of *Java* classes and interfaces that allows us to implement the concepts and behaviour described in Section 2.1. This set is composed of *Manager, Role, Handler* and *Compensator* classes. There are abstract classes, like Role, Handler and Compensator which have to be extended by programmers to write the code that the CAA has to execute

to achieve its main goal (normal behaviour) and to recover it from an unexpected situation (abnormal behaviour). The normal behaviour is defined by extending Role class, whereas the abnormal one is defined by extending Handler and Compensator classes.

In the case of the normal behaviour definition, the programmer has to extend Role class and re-implement the *body* method. This method receives a list of external objects as input parameter and it does not return any value. The other methods that have also to be redefined by the programmer are *preCondition* and *postCondition*. They return a boolean value and they are used as guard and assertion of the role, respectively. The abnormal behaviour definition is achieved by creating a new class that extends from Handler class and re-implementing the *body* and *postCondition* methods. Compensation is another feature provided by the framework, which must be used in case that a specific task has to be executed to deal with one external object that needs manual recovery or in case that a composite CAA was called during the normal behaviour execution phase. A compensator is created by defining a new class that extends from Compensator class and re-implementing the *recovery* method. This method receives, as input parameter, a list with the external objects that need hand-made recovery. The recovery method has to contain the operations to leave these external objects in a consistent state.

Manager class is the controller for Role, Handler and Compensator classes. Thus, for each role, handler and compensator object, a manager object has to be defined to lead their execution. As a CAA is defined as a set of participants coming together to reach a common goal, the framework does not provide a class to define a CAA. Instead, a CAA is defined as a set of managers, roles, handlers and compensators objects linked altogether via a *leader* manager. This leader manager object is the responsible for synchronising roles upon entry and upon exit, the execution of the exception resolution algorithm and for keeping information about shared objects. Therefore, the Manager class represents the core of CAA-DRIP framework, since it provides the necessary runtime support to allow a CAA to follow its life cycle (i.e. activation, normal behaviour execution, and recovery phase if necessary).

The CAA activation process begins when each participant starts the role that it wants to play. The *execute* method (belonging to the Role class) has to be used by a participant to start playing a role. When the execute method is called, the role passes the control to its manager. Then the manager starts executing the *run* method. The first activity of the manager object to be carried out by the run method is to synchronise itself with all, other managers that are taking place in the CAA. This is done by calling the *syncBegin* method (remember that there is a leader manager that is responsible for this task). This method blocks until the leader determines that all the managers have synchronised and the CAA is ready to begin. Once the syncBegin method returns, the manager checks if the pre-condition of the role is valid. The preCondition method receives all the external objects that will be passed to the role managed by this manager as parameters. If the pre-condition is not satisfied, then a *PreConditionException* will be thrown and caught by a catch block (how an exception will be dealt with, is explained later) belonging to the run method. If the pre-condition is met, then the manager will execute the role that is under its control by calling the *bodyExecute* method of the role object.

At this point in time (and not before) the normal behaviour starts executing. It consists basically of executing the code written inside of the body method belonging to each role taken place in the respective CAA. The set of roles that compose the CAA then will exchange information among them to achieve their common goal. The normal behaviour ends once each role has finished its execution. Then each manager in charge of one role will synchronise

with all the other managers before testing its post-condition. If the post-condition is satisfied, then the CAA finishes successfully.

The steps to be executed in case that an exception is raised during the execution of any role belonging to the CAA are inside the *catch block* of the run method belonging to Manager class. These steps are the kernel of the recovery mechanism provided by CAA-DRIP framework. In the following a detailed description of the recovery mechanism is provided. Notice that the recovery phase takes place every time that an exception was raised (regardless a handler has been defined deal with it). In such situation, the role where the exception was raised notifies its manager. This manager passes the control to leader manager for interrupting all the roles that have not raised an exception (exceptions can be raised concurrently). Once all the roles have been interrupted the leader executes an exception resolution algorithm to find a common exception from those that have been raised. When such exception is found, the leader informs all managers about that exception. Then each manager activates its handler object to deal with that exception by calling to *handlerExecution* method. Once each manager completes the execution of this method and its post-condition was satisfied, then the CAA can finish. Now, if the exception resolution algorithm could not find a common exception or the handlerExecution method raised a *PostConditionException* exception, then the manager will undo all the effects of the CAA reached until this point (i.e. abort) by making use of *restoreExecution* method. Once this method has executed, the CAA finishes returning *Abort* exception. If for any reason this process could not complete its execution, the CAA will be finished returning *Failure*.

Due to the limitation of resources and performance of mobile devices, software that will run on such devices must be as small as possible and as efficient as possible in term of resource consumption during execution. Since CAA-DRIP is not a lightweight framework a first solution, before designing a specific version of the framework for Mobile device (a ME-CAA-DRIP), is to select a subset of the framework excluding non mandatory parts. In our context, the CAAs that are on the mobile device does not need to communicate with remote CAAs (which would have needed RMI), the communication with other remote components is done only using sockets (for performance and interface compliance) and HTTP for transferring information on the web. For all these reasons, we removed the RMI support from CAA-DRIP before deploying it into the mobile devices. It is interesting to note that the changes made to the framework comply with the MIDP of J2ME/CLDC, for the implementation of mobile applications. In the MIDP there is no RMI support, and this feature is widely used in the original CAA-DRIP framework. The RMI optional package can be used with the CDC configuration of J2ME, but not with our currently used MIDP/CLDC [20]

3 The e-health Fault-Tolerant Insulin Pump WIS

3.1 General overview

As previously mentioned, the e-health WIS under consideration is composed by the FTIP administration system and the FTIP control system. A doctor, through the FTIP administration system, can see how the patient's treatment runs without being physically at the hospital. The doctor only needs to have internet access to reach the web site that allows him to see the most recent levels of glucose and insulin injections of the patient and modify (if necessary) any treatment configuration parameters (e.g. target blood glucose level -TBG-, or the safe range and insulin doses safe range).

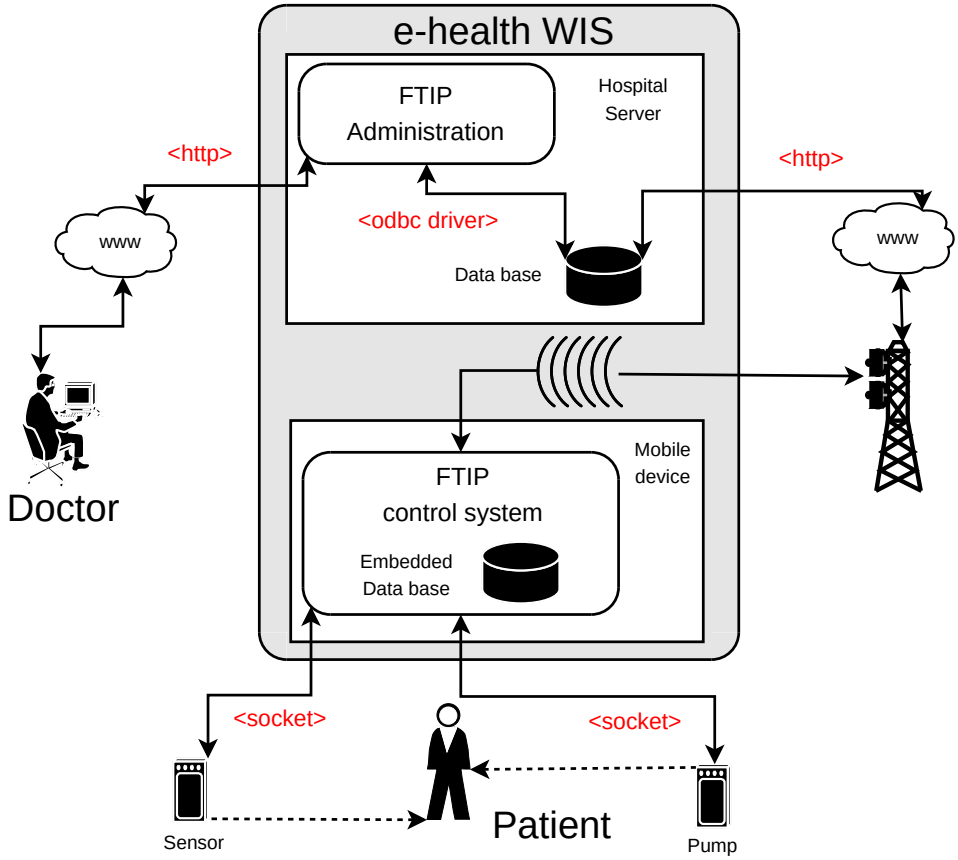

Fig. 3 e-health WIS infrastructure

Patients do not need to go to hospitals to be checked or to receive treatment because of the diabetes, neither. The FTIP control system, which implements the CSII technique to prevent acute episodes of hyperglycemia and late complications, is embedded in a mobile device (e.g. PDA or smart phone). So that, patients can go on with their normal routine without being obliged to follow rigid schedules.

The FTIP control system gets the patient's current glucose level through a miniaturised sensor that is attached to the skin with a small adhesive patch. This sensor sends continuously data to the FTIP control system, which will decide the necessary amount of insulin to be injected. The insulin is delivered by a pump which injects the insulin through a cannula that sits under the patient's skin.

Each glucose level sent by the sensor and each insulin dose delivered by the pump are stored by the FTIP control system because it is the information that allows the doctor to monitor the treatment. The control system makes use of an embedded database to store the data on the device. The data is eventually transferred to the hospital's database according to a schedule set by the patient in agreement with the doctor.

The FTIP control system also relies on the embedded data base to store the configuration parameters defined by the doctor according to the patient's profile. As these configuration parameters can change over the time, they are checked every time that a connection with

the hospital server is established. Figure 3 describes all the components involved both in the FTIP control system and in the FTIP administration system, as well as the communication protocols used by them to achieve the information exchange.

As already said, the safety-criticalness level of the overall software system, and in particular of the FTIP control system is low due to the fact that high blood sugar levels can damage the patients over time but not immediately. Therefore, it is not a problem the fact that the mobile device does not work for a while due to lack of battery power or even worse if it were stolen. Hence, the patient will always have time (even in the worst scenario) to restart the FTIP control system without compromising his health. Instead, very low level of blood sugar are potentially very dangerous in short-term. In such circumstances, it is important for the diabetic to eat something to increase his blood sugar level. However, this aspect is not part of the requirements of the FTIP control system.

We are aware that security has to be addressed by the e-health WIS since sensitive information is managed over the web, but this point is out of the scope of this paper.

3.2 The FTIP control system requirements

As mobility is one of the central issues (along with dependability) to be addressed in this paper, the description of the case study focuses on the FTIP control system only, since this is the WIS's subsystem that is embedded in the mobile device carried by the patient.

Regarding dependability, the approach consists in taking into account from the very beginning the aspects related to such concept. That is the reason why *exceptional use cases* [21] are used to perform the requirements elicitation. This approach addresses at the earliest stage of the development process the identification of all possible exceptional situations that the system may be faced with. Therefore, the fact of using this exceptional use cases enforce developers to describe not only the software system's responsibilities and its interaction with the environment to allow the user to achieve his goal (classic use case function), but also the exceptional situations that could arise in the environment or in the software system itself preventing it from providing the desired functionality to the user. For each detected potential exceptional situation, the approach also proposes to document (in a use case fashion, too) the exceptional steps that should be performed to deal with such abnormal situation. In the following, both the normal interactions and the exceptional ones of FTIP control system are documented following the above-mentioned requirements elicitation approach.

First of all, the patient has to configure the control system according to the doctor's indication. Parameters like the target blood glucose level (TBGL), blood glucose level (BGL) safe range, and the required time to change a cartridge (among others) have to be configured before starting the treatment. The use case that describes how the patient configures the control system is shown on Figure 4.

Once the configuration parameters have been set, the control system is ready to be used by the patient. The objective of the patient is to maintain his value of glucose in a healthy level (i.e. as close to the TBGL as possible) without following a strict schedule of meals or activities. This is the main functionality that provides the FTIP control system. Therefore, over 24 hours a day, the control system commands the pump to deliver certain amount of insulin according to the current patient's glucose level measured by the sensor. As it can be seen from the main success scenario of the use case shown on Figure 5, the patient only needs to launch the FTIP control system to start receiving the treatment. The next necessary steps to achieve the goal are performed by the control system itself, making use of the sensor and the pump (secondary actors).

Use Case: ConfigureTreatment
Scope: FTIP Control System
Primary actor: Patient
Intention: The intention of the Patient is to configure the FTIP control system according to the doctor's prescription.
Level: User Goal
Precondition: The sensor and the pump are attached to the patient and have been set up. The Patient's mobile device works properly.
Main Success Scenario:
1. The Patient enters TBGL value
2. The Patient enters MAX_BGL (maximum blood glucose level allowed)
3. The Patient enters MIN_BGL (minimum blood glucose level allowed)
4. The Patient enters MAX_ID (maximum insulin delivery allowed)
5. The Patient enters MIN_ID (minimum insulin delivery allowed)
6. The Patient enters the TCC (time needed to change the pump's insulin cartridge)
7. The Patient enters the LLIW (low-level insulin warning)
8. The Patient enters the DPE (delivery percent error) to be tolerated in the delivery
9. The Patient selects the sensor
10. The Patient selects the pump
11. The Patient selects the hospital

Fig. 4 Use-Case SetTreatment

The *GetGlucoseLevel* use case is shown on Figure 6. To know the current level of glucose the control system makes use of the sensor. The sensor measures the patient's blood glucose level each time that receives a request coming from the control system. The value is wirelessly exchanged between the sensor and the control system, since there is not physical connection among them (it is the same between the control system and the pump). In this use case the exceptional situations that can arise are due to either a problem in the communication between the sensor and the control system, or a measured level of glucose which drops outside the allowed range. The first exceptional situation is detected by a timeout over the control system request. The second exceptional situation is detected by checking the measured value (obtained by making the average of 200 samples) against the BGL safe range configuration parameter. It is worth saying that the fact of taking 200 samples to figure out the current patient's glucose level it is a step forward to the improvement of the FTIP control system's reliability, since this measuring technique is well-know for masking random measuring errors (which in this particular case are due to transient faults of the sensor).

The necessary steps to deal with the exceptional situation caused by a communication problem (i.e. Exception{SensorTimeout}) are enumerated in the use case shown on Figure 7. The "handler" use case consists in making a second attempt (starting from scratch) to get the glucose level. As the idea is to deliver the treatment while the patient goes on with his normal life, a lost of communication for a short period between the sensor (or pump) and the mobile device might perfectly be possible. Thus, the try-again policy is a good (and simple) solution to deal with this temporal fault.

However, if the problem persists during the second attempt, then the FTIP control system interprets it as a permanent fault. In front to this situation, the control system makes an emergency stop. As it can be seen on Figure 8, an emergency stop consists in cancelling every single task being executed and warning the patient about the situation by

Use Case: TakeTreatment
Scope: FTIP Control System
Primary actor: Patient
Intention: The intention of the Patient is to keep his glucose in a good level
Level: User Goal
Precondition: The Patient's mobile device works properly and the treatment has been configured
Main Success Scenario:
1. The Patient starts the FTIP control system
Steps 2-5 are repeated continuously
2. The FTIP control system **GetsGlucoseLevel** of the Patient
3. The FTIP control system **CalculatesInsulin** to be injected to the Patient
4. The FTIP control system **DeliversInsulin**
5. (according to certain policy) The FTIP control system connects to the hospital's server and sends the logged information regarding the treatment
Extensions:
(2-4).a (at any time) The Patient stops the FTIP control system. Use case ends in success.
(2-4).a (at any time) The Patient pauses the FTIP control system (i.e. The FTIP control system finishes its tasks in progress and then it remains waiting for a Patient's command -continue or stop).
5.a The connection cannot be established or it is interrupted. The data transmission is cancelled and use case continues in step 2.

Fig. 5 Use-Case TakeTreatment

Use Case: GetGlucoseLevel
Primary actor: N/A
Intention: FTIP control system wants to get the current patient's glucose level
Level: Subfunction
Main Success Scenario:
Steps 1-2 are repeated 200 times
1. The FTIP control system requests the Sensor to send the Patient's glucose level
2. The FTIP control system receives the Patient's glucose level from the sensor
3. The FTIP control system figures out the current Patient's glucose level, which drops inside the BGL safe range
Extensions:
2.a Exception{SensorTimeout}
3.a Exception{UnhealthyGlucoseLevel}

Fig. 6 Use-Case GetGlucoseLevel

showing a message on the mobile device's screen and ringing it as if there were an incoming call.

Regarding the other exceptional situation (Exception{UnhealthyGlucoseLevel}), it must be noticed that it can be arisen either during the first attempt to determine the glucose level or in the second one. Faced with this situation, the control system performs immediately an emergency stop as described previously.

Coming back over *TakeTreatment* use case, the next steps to be performed by the FTIP control systems correspond to decide how much insulin the patient needs (step 3), and command the pump to achieve the insulin delivery (step 4). The *CalculateInsulin* use case

Use Case: ReconnectingGlucoseSensor ≪handler≫
Context & Exception: GetGlucoseLevel{SensorTimeout}
Primary actor: N/A
Intention: FTIP control system makes a new attend to establish a connection with the sensor
Level: Subfunction
Main Success Scenario:
Steps 1-2 are repeated 200 times
1. The FTIP control system requests the Sensor to send the Patient's glucose level
2. The FTIP control system receives the Patient's glucose level from the sensor
3. The FTIP control system figures out the current Patient's glucose level, which drops inside the BGL safe range
Extensions:
2.a Exception{SensorTimeout}
The FTIP control system cannot establish a connection with the sensor.
Use case ends in failure.
3.a Exception{UnhealthyGlucoseLevel}

Fig. 7 Use-Case ReconnectingGlucoseSensor

Use Case: EmergencyStop ≪handler≫
Context & Exception: GetGlucoseLevel{UnhealthyGlucoseLevel},
ReconnectingGlucoseSensor{SensorTimeout}
ReconnectingGlucoseSensor{UnhealthyGlucoseLevel}
DeliverInsulin{PumpTimeout}, DeliverInsulin{PumpStuck},
DeliverInsulin{WrongDoseInjected}
ChangeCartridge{ChangingCartridgeTimeout}
Primary actor: N/A
Intention: FTIP control system wants to interrupt the treatment
Level: Subfunction
Main Success Scenario:
1. The FTIP control system cancels all its current tasks
2. The FTIP control system shows on the screen the reason why the treatment was interrupted
Step 3 is repeated until the Patient switches the alarm off or the mobile device runs out of battery
3. The FTIP control system rings the mobile device's alarm

Fig. 8 Use-Case EmergencyStop

is shown on Figure 9. Once the control system has got the dose that the patient needs (step 1), it checks if the amount of insulin to be delivered drops inside the allowed range (i.e. the ID safe range configuration parameter). If the control system figures out that the insulin dose is out the safe range (i.e. Exception{UnhealthyInsulinDose}), then an emergency stop is performed (see Figure 8). Otherwise, the control system requests the pump to inject the dose into the patient.

The *DeliverInsulin* use case shown on Figure 10 describes the interaction between the control system and the pump to achieve the goal (i.e. injecting the dose). Thus, the control system communicates the pump the dose to be injected (step 1). The acknowledgement sent back by the pump (step 2) means that the delivery process has been started. The next acknowledgement sent by the pump represents the end of the delivery (step 3). Finally

Use Case: CalculateInsulin
Primary actor: N/A
Intention: FTIP control system wants to know how much insulin the Patient needs to keep his glucose in a healthy level
Level: Subfunction
Main Success Scenario:
1. The FTIP control system calculates the insulin needed for the Patient
2. The FTIP control system figures out that the insulin needed drops inside the ID safe range.
Extensions:
2.a Exception{UnhealthyInsulinDose}

Fig. 9 Use-Case CalculateInsulin

the control system checks if there is any discrepancy between the dose requested and the actual injected (step 4). In case that there is a discrepancy, the control system makes an emergency stop (*Exception*{*WrongDoseInjected*}). The control system will also perform an emergency stop if: (1) it does not receive an acknowledge from the pump after certain time when a request was sent (*Exception*{*PumpTimeout*}[2]) or (2) the pump notifies the control system that it cannot deliver the dose (*Exception*{*PumpStuck*}). In case that the delivery is not possible due to a lack of insulin (*Exception*{*LackOfInsulin*}), then the control system behaves as described in *ChangeCartridge* use case (see Figure 11).

Use Case: DeliverInsulin
Primary actor: N/A
Intention: FTIP control system wants to inject insulin in the Patient
Level: Subfunction
Main Success Scenario:
1. The FTIP control system requests pump to inject certain amount of insulin
2. The FTIP control system detects pump's plunger is injecting insulin
3. The FTIP control system detects pump has stopped injecting
4. The FTIP control system figures out that pump injected the requested amount of insulin
Extensions:
1.a Exception{LackOfInsulin}
1.b Exception{PumpTimeout}
1.c Exception{PumpStuck}
4 || 1. The FTIP control system detects low level of insulin in pump's cartridge
2. The FTIP control system sends a warning notification to the Patient.
Use case ends in success.
3.a Exception{PumpTimeout}
4.a Exception{WrongDoseInjected}

Fig. 10 Use-Case DeliverInsulin

In this case, the control system will pause its execution and notify the patient about this issue. The control system will remain in that state for certain period of time, waiting for the

[2] The waiting time is long enough to cover small pump outages. Therefore, the fact of passing this waiting time without a response is interpreted as a permanent failure on the pump

patient to acknowledge the notification. The time that the control system pauses its execution to allow the patient to change the cartridge is one of the configuration parameters that must be established before launching the treatment. If the notification sent by the control system is not acknowledged by the patient, then it makes an emergency stop. Otherwise, it will remain waiting until the patient indicates to continue the treatment.

Use Case: ChangeCartridge ≪handler≫
Context & Exception: DeliverInsulin{LackOfInsulin}
Primary actor: N/A
Intention: FTIP control system requests the Patient to change the pump's cartridge
Level: Subfunction
Main Success Scenario:
1. The FTIP control system pauses its execution
2. The FTIP control requests the Patient to change the cartridge
3. The Patient acknowledges the request, changes the cartridge and indicates the FTIP control system to continue
Extensions:
3.a Exception{ChangingCartridgeTimeout}

Fig. 11 Use-Case ChangeCartridge

Now that the interactions between the control system and its environment facing both normal and exceptional situations have been clearly enumerated, the next step is to structure the control system to support these interactions. It means that some support to detect and deal with the exceptional situations must be embedded in the control system. Precisely the fact of needing to take into account exception handling for concurrent software systems was the main reason why the CAA conceptual framework was taken to guide the design of the control system. Thus, the next section provides the details about how the control system has been designed in terms of CAAs.

In addition to the functional requirements that have been elicited applying use case scenarios, there are some non-functional requirements that the software system is desired to provide. The ability to be dynamically maintained and the evolution are two of the most important features of a piece of software in the context of continuous evolving environment and technologies. In particular in our case study, these features are strongly encouraged. For example, it is very likely to happen in this long term treatment that at the time of changing sensor and pump, a different driver needs to be loaded in the system. Concerning its availability, the system should undergo this change in an acceptable period of time. And if this modification is reached without bothering the patient to find and install new components, the user-friendliness is improved.

3.3 The FTIP control system service-oriented architecture

The well known way of succeeding with the properties mentioned above, is applying an SOA. By dividing the system in loosely coupled functionalities, the independent components can be changed without affecting the remaining parts. If these components can be discoverable through a registry, the system is not attached to a particular configuration and can evolve.

The FTIP WIS was divided in two main services: the administration service and the control system service. Regarding the fact that they are services, they are autonomous and they can evolve independently. The control system service is in charge of the composition logic (see Figure 12). Its main task is to control the workflow, which is achieved by composing other services. The functionalities related to the sensor, pump, calculus of injection, management of the registry and hospital administration were identified as candidates for services.

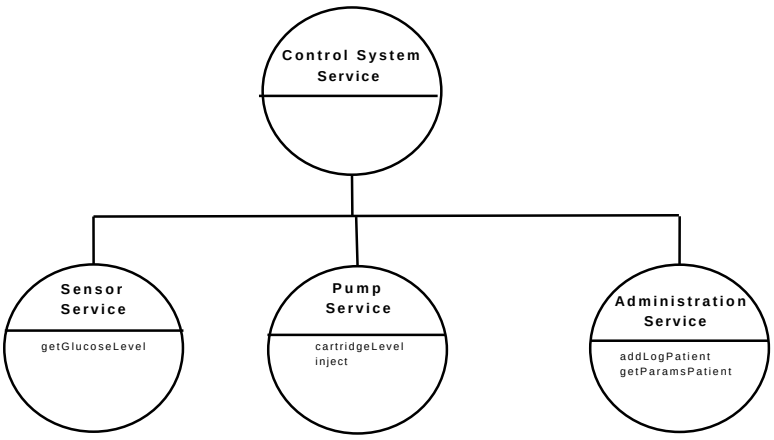

Fig. 12 The FTIP control system service composition

The *Sensor Service* has only one operation, *getGlucoseLevel*, for retrieving the value of glucose. By encapsulating this functionality with a service the control system will be able to evolve easily in case of changing devices. The control system will have to discover which is the *Sensor Service* available in the registry.

The same reasoning applies to the *Pump Service*. This service has two operation, one for injecting the dose of insulin, *inject*, and one for retrieving the level of insulin in the cartridge, *cartridgeLevel*.

The access to the hospital's database was encapsulated behind a service. This decision allows the access and manipulation of the data not only by each control system (the system will be open for many patients that might take the treatment) but also by doctors and administration personnel. Despite this service will have more operations, they are not specified here because the control system will call only the operation for adding a log to the patient's history *addLogPatient*, and the operation for retrieving the parameters set by the doctor to each patient *getParamsPatient*.

[***** Note: FB: to be extended *****]

3.4 Detailed FTIP control system design

Since a CAA encloses concepts coming from transactions and conversation, there are multiple reasons to set a CAA in the design. Obviously, it can be used as a structural unit to join participants interested in achieve the same goal, or as a transactional unit to isolate certain behaviour, but also as a fault-tolerant or synchronisation unit. Regarding fault tolerance, a

CAA defines a confinement area that avoids the error to be spread beyond the CAA's border, and moreover it provides mechanisms (e.g. forward/backward error recovery) to deal with such error. Therefore, a CAA can be used to surround certain region of the software system where a potential error can arise. Finally, as a CAA ensures that its roles will start and finish their execution at the same time, it can be used to synchronise parts of the software system. Next, the FTIP control system design is presented, detailing the aim of each involved CAA.

The design of the FTIP control system (see Figure 13) is composed of one top-level CAA called *Main*, which itself contains four nested CAAs (*Configuration*, *Reconfiguration*, *GetGlucoseLevel* and *DeliverInsulin*). *Main* is used as structural unit to delimit the logical border between the FTIP control system and its environment. *Main* is composed of five roles (*MobileDevice*, *Parameters*, *Sensor*, *Controller* and *Pump*), which are activated by external executing participants created right after the FTIP control system is started.

The role *MobileDevice* is defined to deal with the services (i.e. display, battery, connectivity, etc.) that are provided by the mobile device where the control system is embedded. Thus, this role is used as a layer in-between the control system and the patient (and the doctor, through remote access) to allow them to exchange messages. Thus, the patient interacts with the FTIP control system through a graphical user interface (GUI) following an event-based style. Each action performed by the patient over the GUI represents an event, which is caught by the *MobileDevice* role. This event then is interpreted and passed as a message to the corresponding role. The sequence is exactly the opposite when the control system wants to communicate certain information to the patient (e.g. notify that the cartridge is running out of insulin).

The configuration of the treatment is the first step to be carried out by the patient. The GUI then allows the patient to enter each required configuration parameter. Since each value must be fulfilled before lunching the treatment, the control system not only has to check that all of them have been entered, but also they have valid values. Thus, the CAA called *Configuration* is used as transactional unit to meet this requirement, since it ensures the "all or nothing" (i.e. the CAA either commits, or it aborts).

The role *Parameters* has been defined following the "separation of concerns" principle [22]. This role is exclusively in charge of performing modifications on the configuration parameters. Therefore, every time the patient (or the doctor, remotely) wants to change one of the configuration parameters this role has to be involved in the process. According to this design decision, this role comes along with *MobileDevice* role both in *Configuration* and *Reconfiguration* CAAs. In the former, the configuration is set by the patient by making use of the GUI. As explained previously, *MobileDevice* role gets the information supply through the GUI and makes it available to the rest of the control system. Therefore, in the next step *MobileDevice* passes the configuration values to *Parameters* role to make effective the modification. Regarding the latter (i.e. *Reconfiguration*), the behaviour is almost the same, except that the information in this case is provided by the doctor [3].

The roles *Sensor* and *Pump* are used to deal with the respective physical devices needed to carry out the treatment. These roles have direct access over these physical devices through their own interfaces. Therefore, as the specific instructions to get access on the sensor (pump)

[3] When a connection between the mobile device and the hospital server is established, the control system checks if the local configuration is the same with respect to the one stored on the hospital's server. Since the doctor has the possibility to monitor the treatment remotely, he could have decided to make a slightly modification on the original prescribed configuration. In that case, the doctor's changes are stored on the hospital server, and gather by the control system when a connection is established.

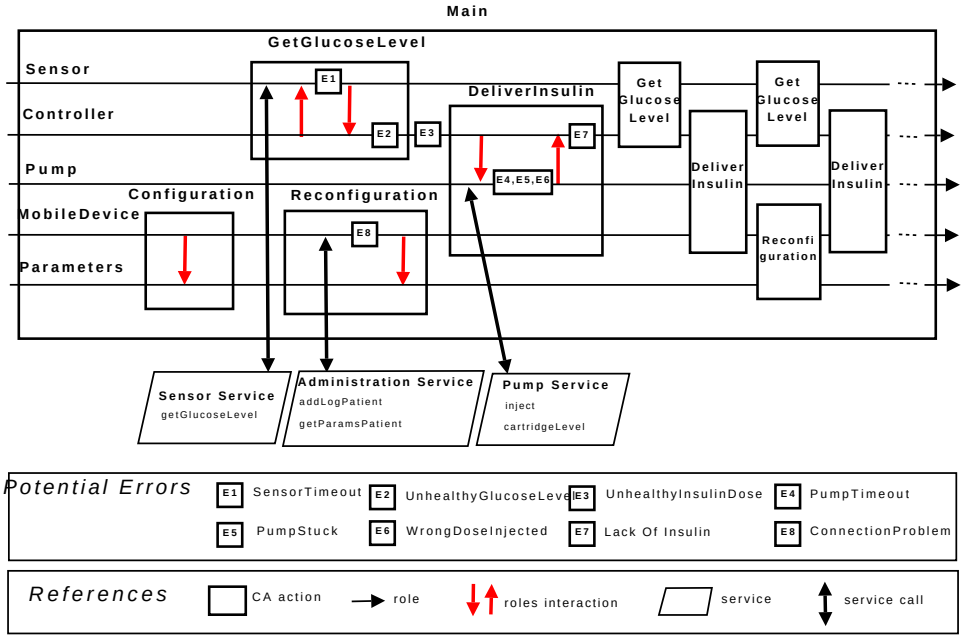

Fig. 13 Design of the FTIP control system in terms of CAAs

are encapsulated in its respective role, both the "separation of concerns" and "information hiding" [23] principles are met.

Controller role is in charge of calculating the insulin dose to keep the patient's glucose level safe. To define the dose, this role needs first to enter along with *Sensor* role in the nested CAA called *GetGlucoseLevel*. This CAA is used as synchronisation unit, since both roles must come together to perform the CAA. It is also used as fault-tolerant unit since the possible problems that can arise as consequence of malfunction in the interaction with the sensor device (or in the device itself) are handled locally (if possible) without interfering with the other functionalities of the control system like treatment configuration or connection with the hospital server. Then, once *GetGlucoseLevel* was performed, the *Controller* role has the information to calculate the dose.

The last step concerning the delivery of the treatment is to inject the dose into the patient. At this point, the *Controller* role enters along with *Pump* and *MobileDevice* roles in the nested CAA called *DeliverInsulin*. This CAA is mainly used as synchronisation unit to coordinate the tasks to be performed by the CAA's member roles upon entry and exit.

3.4.1 The COALA specification Obviously, the previous informal description is not enough detailed for developers if the goal is to get a correct and reliable implementation of the FTIP. In fact, the previous information just represents a high level overview of the design which has to be refined to avoid misunderstanding in the following development step (i.e. implementation). Therefore, a rigorous notation with formal background (i.e. logical theory which supports formal proofs about the description and their properties) has to be employed to achieve a design description clear enough. As previously mentioned, COALA is the formal

language to be taken since it was conceived to support the designing of reliable systems based on the CAA's concepts. As CAA is the chosen fault tolerant mechanism to engineer the FTIP case study, this formalism meets perfectly the needs.

```
1  ;; ***********************************
2  Caa DeliverInsulin;
3  ;; ***********************************
4     Interface
5        Use Data;
6        Roles
7          Controller: parameters,integer;
8          Pump: record;
9          MobileDevice;
10       Exceptions
11         stop:string, pumpTimeout:string, pumpStuck:string;
12         wrongDoseInjected:string, changingCartridgeTimeout:string;
13
14    Body
15       Data, Buffer, Math, Time, Services;
16       Objects
17         info: dataBuffer;
18         msg: dataBuffer;
19         ev: event;
20       Exceptions lackOfInsulin;
21       Resolution lackOfInsulin -> lackOfInsulin;
22
23       Role Controller(configParams,insulinDose)
24         Handler changeCartridge: parameters,integer;
25         Handling lackOfInsulin -> changeCartridge(configParams,insulinDose);
26         Begin
27                  ...
28         End;
29         Handler changeCartridge(configParams, insulinDose)
30                  ...
31         End changeCartridge;
32       End Controller;
33
34       Role Pump(registry);
35         Handler changeCartridge: record;
36         Handling lackOfInsulin -> changeCartridge;
37           Begin
38                  ...
39           End;
40         Handler changeCartridge(registry);
41                  ...
42         End changeCartridge;
43       End Pump;
44
45       Role MobileDevice
46         Handler changeCartridge;
47         Handling lackOfInsulin -> changeCartridge;
48           Begin
49                  ...
50           End;
51         Handler changeCartridge
52                  ...
53         End changeCartridge;
54       End MobileDevice;
55 End DeliverInsulin;
```

Fig. 14 COALA specification of DeliverInsulin

Figure 14 illustrates how the CAA *DeliverInsulin* is described in COALA (the full detailed design can be found in the appendix section). As already explained when COALA was

introduced, each CAA is defined as a module (line 2). A module is composed of an interface (lines 4-11) and a body (lines 13-51). The interface contains those elements that are visible for the context where the CAA is embedded. It is mandatory for a CAA to export its roles, so that *DeliverInsulin* exports *Controller,Pump* and *MobileDevice* roles (lines 6-8). The interface of a CAA is also used to define the signature of every role that composes it. That is the reason why *Controller* is followed by the words *parameters* and *integer* (line 7). They represent the types of the input parameters the role has to receive when it is invoked. These data type are defined in CO-OPN/2 in the "Abstract Data Type" (ADT) module called *Data* (line 5). Exceptions that can be **signal** to the enclosing context have to be defined in the interface, as well. That is the case for the exceptions *stop, pumpTimeout, pumpStuck, wrongDoseInjected* and *changingCartridgeTimeout* (lines 9-11). All of them carry a string parameter used to describe the error message.

On the other hand, the body of the CAA module contains the definition of the roles, internal exceptions and the handlers to deal with them. Information about how to resolve multiple raised exceptions is also part of the body definition. In the shown example, there is only one internal exception potentially to be raised (*lackOfInsulin* on line 16), so that the mechanism resolves to the same exception (line 18). Once the information regarding internal exceptions and how they are resolved is provided, what comes is the definition of the roles and their associated handlers. A role is defined using the reserved word **Role** followed by the name of the role and its formal parameters, which have to respect the role's signature defined in the interface. In the example, the role *Controller* has the formal parameters *configParams* and *InsulinDose* (line 20), which are of type parameters and integer, respectively. A role definition ends with the reserved word **End** followed by the role's name (line 29). Thus, everything that is in-between the reserved words **Role** and **End** defines the block role definition (lines 20-29). In this manner, the definition of the handler's header (line 21), its relationship with the internal exceptions (line 22) and its formal definition (lines 26-28) are part of the role definition, as well. It should not result strange, since it is part of the CAA's principle: a handler represents an alternative behaviour which is in accordance with the exception that has just been detected.

Therefore, a COALA specification helps designers to provide full detailed description in terms of CAA without ambiguity. A COALA design then would be the documentation to be provided to programmers to carry out the implementation phase. Next, section shows how the implementation framework bridges the gap for going from the design to the implementation.

3.5 Implementation

The use of the framework of CAA eases the implementation phase since it supports the same concepts used in the design phase. Therefore, concepts like *role* or *handler* are supported to allow to go from the design to the implementation in a more straight way. The normal behaviour of the CAA (defined as role objects) is very well separated from the abnormal behaviour (defined as handlers objects), which only takes place when an exception is raised. In this manner, the fact that CAA-DRIP supports the separation of concerns principle allows developers to tackle the implementation phase in different rounds. For instance, the first round could be to cover the functional objectives (i.e. normal behaviour of the application), whereas the second round would be to implement the exceptional cases (i.e. abnormal behaviours).

Fragments of the source code implementing the FTIP case study are shown to better explain how a CAA is defined making use of the implementation framework. For this purpose and in accordance with the example taken previously for showing the details regarding the design phase, the chosen CAA is *DeliverInsulin*. Thus, according to the COALA specification given in the previous section, the CAA *DeliverInsulin* is composed of three roles: *Controller, Pump* and *MobileDevice*.

As conceptually a CAA is defined as a set of roles that come together to achieve a common goal, at the implementation level a CAA is defined by CAA-DRIP in terms of the roles that it is composed of. Thus, to define the CAA *DeliverInsulin* three Java classes (one for each role) extending from **Role** class (which is part of CAA-DRIP implementation framework) have to be created. Figure 15 shows how *Controller* role is defined. These classes are defined inside a package with the name of the CAA the roles belong to (line 1) and the purpose of each role is implemented in the **body** method (lines).

```
1  package lu.uni.cldc_app.caa.DeliverInsulin;
2
3  public class Controller extends RoleImpl {
4
5    /* Constructor.  */
6    public Controller(String n, Manager mgr, Manager leader) throws Exception {
7          super (n, mgr, leader);
8          ...
9    }
10
11   public void body(SetOfEO setEOs) throws Exception {
12         try{
13         ...
14         } catch (Exception e) {
15         throw e;
16         }
17   }
18 }
```

Fig. 15 Definition of class Role Controller

Once each of the *Role* Java classes have been created, objects referring to these classes have to be instantiated. Due to *DeliverInsulin* is a nested CAA, the instantiation of its roles must be done at the level of the CAA where it is enclosed (i.e. CAA *Main*). Thus, it is in the constructor of each of **Role** Java classes that are entering in the nested CAA (i.e. *Controller, Pump* and *MobileDevice* roles of *Main*) where the instantiation must be done. The Java code of Figure 16 shows the instantiation of role *Controller*.

The instantiation process starts creating a *manager* object (lines 16-20), which is in charge of leading the execution of a *role* object. The role object managed by the *manager* object is instantiated by passing as arguments, an string for identification, the manager that leads its execution, and the manager object that works as leader [4] for the CAA the role belongs to (lines 18-20). In the example, *mgrDIController* is the manager leader of CAA *DeliverInsulin*, so that it has to be exported (line 21) to the other roles that compose the CAA to allow them to do their respective instantiation.

[4] A manager leader is in charge of launching/stopping the roles of the CAA by interaction with the managers that lead their execution, as well as of the execution of the resolution mechanism when multiple exceptions are raised.

The next step in the CAA definition process corresponds to instantiate the handlers for the potential exceptions the CAA has to deal with. Each defined handler is an instance of the **Handler** class, which belongs to the framework. For each exception that must be handled, a handler object must be instantiated. A handler is instantiated by providing a name, and the name of its manager (line 24-26) (the manager leader remains the same as the one given for CAA). Then, what follows is the binding between the exceptions and the handlers that deal with them. This binding is implemented by a hash-table (line 29) containing pairs (exception,handler). In the example, and according to the COALA specification shown on the previous section, the only internal exception to be dealt with is *LackOfInsulin* (line 30). Once, the binding has been defined, it has to be passed to manager that leads the role execution (line 33).

```
1  package lu.uni.cldc_app.caa.Main;
2
3  public class Controller extends RoleImpl {
4          ...
5          //Manager, role and handler to define the nested CAA DeliverInsulin
6          Manager mgrDIController;
7          Role roleDIController;
8          Handler hndChangeCartridgeController;
9          ...
10
11         /* Controller Class Constructor */
12         public Controller(String n, Manager mgr, Manager leader)
13                                                         throws Exception {
14           super (n, mgr, leader);
15           ...
16           mgrDIController = new
17                 ManagerImpl("mgrDIController","CAA_DeliverInsulin");
18           roleDIController = new
19                 Controller("roleDIController", mgrDIController,
20                       mgrDIController);
21           mgr.sharedObject("mgrDIController",mgrDIController);
22
23           //Handler for LackOfInsulin exception
24           hndChangeCartridgeController = new
25                 ChangeCartridge_Controller("hndChangeCartridgeController",
26                       mgrDIController);
27
28           //Binding between the exception and the handler
29           Hashtable ehController = new Hashtable();
30           ehController.put(LackOfInsulin.class, hndChangeCartridgeController);
31
32           //Setting the binding on the Manager
33           mgrDIController.setExceptionAndHandlerList(ehController);
34           ...
35         }
36
37         public void body(SetOfEO setEOs) throws Exception {
38           try{
39                 ...
40           } catch (Exception e) {
41                 throw e;
42           }
43         }
44 }
```

Fig. 16 Definition of the object role Controller using CAA-DRIP

The implementation framework also provides the necessary runtime support to: (1) stop the roles execution when an exception has been raised, (2) resolve what is the exception to

be handled, even when multiple exception have been raised, (3) start the handlers execution to manage the exception. Therefore, once the Java classes were created and their respective objects instantiated, the developers' tasks have finished and the software system is ready to be deployed and executed.

The connection between the FTIP control system and the remote data base server was implemented using a HTTP connection. Thus, the FTIP control system acts as a client that sends HTTP requests to the server to get the configuration parameters and log information concerning the treatment execution.

We had chosen Eclipse SDK [24] as the development platform, with the EclipseME [25] plug-in and the Sun Wireless Toolkit [26]. We want to highlight that it was a very good implementation decision since these technologies allowed us to save considerable time. The emulators that come with this toolkit helped giving a quick feedback about how the application runs and looks without need of deploy and run the application on the real mobile device (which is a very time-consuming task).

As mentioned, the CAA-DRIP framework is not lightweight. But, with the modifications introduced, the mobile version of the framework was deployed on a PDA without any problems. The whole package, composed by a .jad file (the java application descriptor file) and a .jar (which includes the control system and the framework), requires less than 100 KB of space in a secondary storage. At runtime, the application uses a larger space in main memory, in the order of the MBs. However, this is not a restriction for running the application with the new mobile technologies.

3.6 Testing

As said in [27], testing is the process of operating a system or component under specific conditions, observing or recording the results, and making an evaluation of some aspects of the system or component. Therefore, to run the testing process the system has to be fully implemented and under the same conditions as it would be when used by the stakeholder (the patient in this case). Thus, a real sensor and pump has to be connected to the FTIP control system[5] to allow the treatment to take place.

At the moment of developing the FTIP control system and writing this report, some kind of insulin pump had appeared on the market while others were still in the process of research and development ([29] [30] [31]). However neither a sensor nor a pump of the characteristics required to try the treatment were available in the market. For that reason and due also to the fact that the aim of our project focuses on developing software engineering techniques rather than selling software, both the sensor and the pump were simulated by a combination of software and hardware (both the sensor and pump were simulated by small *Java* applications running on mobile devices[6])

The communication between the three mobile devices was implemented using sockets through the wireless connections, taking advantage of the support that gives the MIDP 2.0 profile of J2ME. The decision of using sockets was made to be as low implementation level as possible regarding the communication layer in order to reduce any further implementation

[5] It was deployed on a Qtek 9000 Pocket PC, that includes a processor Intel PXA270 520 MHz, 64 MB of RAM, Windows Mobile Version 5.0 operating system and Java MIDlet Manager Runtime MIDP 2.0 Tao Group Ltd. [28].

[6] DELL Axim X30 Pocket PC, with Intel PXA270 624 MHz processor, 64 MB of RAM, Windows Mobile 2003 Second Edition operating system and Jeode runtime of J2SE [32].

Fault name	Persistence	Exception	Test Set
SensorTimeOut	Permanent	E_1	TS_1
	Transient	E_1	TS_2
UnHealthyGlucoseLevel	Permanent	E_2	TS_3
UnHealthyInsulinDose	Permanent	E_3	TS_4
PumpTimeout	Permanent	E_4	TS_5
PumpStuck	Permanent	E_5	TS_6
WrongDoseInjected	Permanent	E_6	TS_7
LackOfInsulin	Transient	E_7	TS_8
	Permanent	E_7	TS_9
ConnectionProblem	Transient	E_8	TS_{10}

Table 1 Test cases defined to the FTIP-CS

modifications on the control system at the moment of using it with real sensors and pumps. On this aim, an ad-hoc network was created between the three devices. The sensor was simulated with an application that, acting as a server in terms of sockets, gives a value corresponding to the current BGL to the client application, in this case the FTIP control system. The pump was simulated in a similar way.

Test sets for the FTIP-CS were defined according to those exceptional situations (faults) that are assumed to be tolerated. As explained previously the tolerance is provided either in terms of continuity (for those faults that are presented temporarily or can be masked) or in terms of safe stop. The persistence viewpoint then gets higher importance with respect to those used to define the *elementary fault classes* [33] like creation phase, boundary, cause, etc. Therefore, the faults along with their persistence were used as the principles to define the test sets. Following this principle, ten test sets ($TS_{1..10}$) were defined. Table 1 lists all the test sets with their respective faults they were meant to inject during the testing process.

TS_1 contains those test cases in which the sensor stops working completely, whereas TS_2 has those in which the sensor stops working just temporarily. Notice that the FTIP-CS figures out that an erroneous state has been reached (which leads towards a failure if no action is taken) when an exception is raised. Therefore, the goal of each test case is to reach an erroneous state to raise an exception and then produce the activation of the recovery mechanism belonging to CAA-DRIP framework. Thus, test cases belonging to TS_1 and TS_2 need to lead the environment[7] towards an erroneous state from where exception E_1 is raised.

Test cases belonging to TS_3 drive the sensor to send BGL values which are out of the configured safe range. These test cases then will raise exception E_2. TS_4 includes those test cases that drive the sensor to send right BGL values (i.e. they drop inside the configured safe range), but for which the insulin to be delivered is outside its configured safe range. These test cases are aimed at injecting the *UnHealthyInsulinDose* fault into the system (i.e. exception E_3 will be raised).

TS_5 includes those test cases that lead the pump to stop responding to requests sent by the FTIP-CS (i.e. *PumpTimeout* fault), whereas TS_6 includes those where the pump notifies the FTIP-CS that it cannot inject insulin (i.e. *PumpStuck* fault). Exceptions E_4 and E_5 are raised for each test case belonging to these test sets, respectively.

[7] The faults considered in this case study are all external, which means that they are originated outside the system boundary and originated into the system by interaction.

TS_7 includes test cases that lead to differences between the requested insulin amount and the actual injected by the pump (i.e. *WrongDoseInjected* fault). These test cases then are used to drive the pump to make exception E_6 occurs. The last two test sets aimed to deal with the pump are TS_8 and TS_9. Both are used to cover the case of lack of insulin on the pump (i.e. *LackOfInsulin* fault, and exception E_7 to detect it occurring). TS_8 includes those test cases simulating a successful cartridge replacement before reaching a critical level value, whereas TS_9 contains those where pump runs out of insulin to keep performing the treatment. TS_{10} includes those cases used to test the connection with the remote database either when it cannot be established or when it is lost during an information exchange (i.e. *ConnectionProblem*). These kinds of problems will produce exception E_8 to occur.

3.7 Performance analysis

As explained previously, the fact of using CAA-DRIP as implementation framework helps developers to save the gap for going from a design described in terms of CAA to the piece of software that implements such design. Obviously, the benefits of using CAA-DRIP do not come for free. This mechanism definitely adds an overhead to the application which would not be present in case of implementing the same application without relying on it. The aim of this section is to provide quantitative values of the cost of using CAA-DRIP instead of using ad-hoc approaches.

The scenario used to run the comparison consists of a very simple and unique CAA composed of a set of participants, which varies from 1 to 10000. Since the first analysis performed was to figure out the real processing time overhead introduced by CAA-DRIP to run such simple CAA, no interaction at all is carried out among the participants of the CAA. The ad-hoc approach used to implement this scenario without using the support of CAA-DRIP framework consisted of a set of threads synchronising[8] before and after the execution of their tasks. Obviously, the task to be performed for each participant of the CAA was implemented exactly in the same way both in the framework-supported solution and in the ad-hoc solution. The result of this comparison (depicted on Figure 17) shows that the processing time requested by the framework-supported solution increases proportionally to the number of participants of the CAA. Therefore, CAA-DRIP does not scale up when the number of participants of a CAA is high (remark: the FTIP case study shown previously comprises only five participants).

The second analysis was driven to figure out the processing time overhead when the participants of the CAA are highly coupled. In fact, this is the most common scenario to be found when the CAA paradigm is used, since a CAA for definition joins those participants that need to collaborate to reach a common goal. Therefore, the simple CAA used in the first analysis was adapted to allow its participants to exchange messages to each other during the execution of their tasks. The message exchanging represents the collaborative concurrency of the CAA participants to reach the common goal. The result of this analysis (depicted on Figure 18) shows that the processing time overhead introduced by using CAA-DRIP becomes insignificant respect to the ad-hoc solution.

On the basis that CAA-DRIP framework is a target-solution that mitigates the inherent complexity of coordinating the execution of multiple participants both in normal and excep-

[8] To make the comparison fairer, the synchronisation mechanism used by the ad-hoc solution was exactly the same as the one used by CAA-DRIP framework

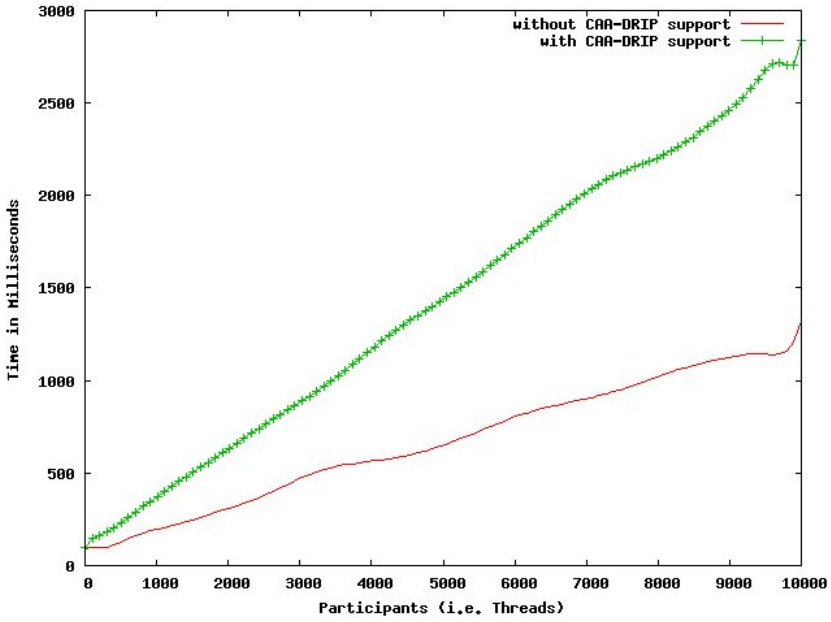

Fig. 17 CAA-DRIP framework overhead: no interaction among participants

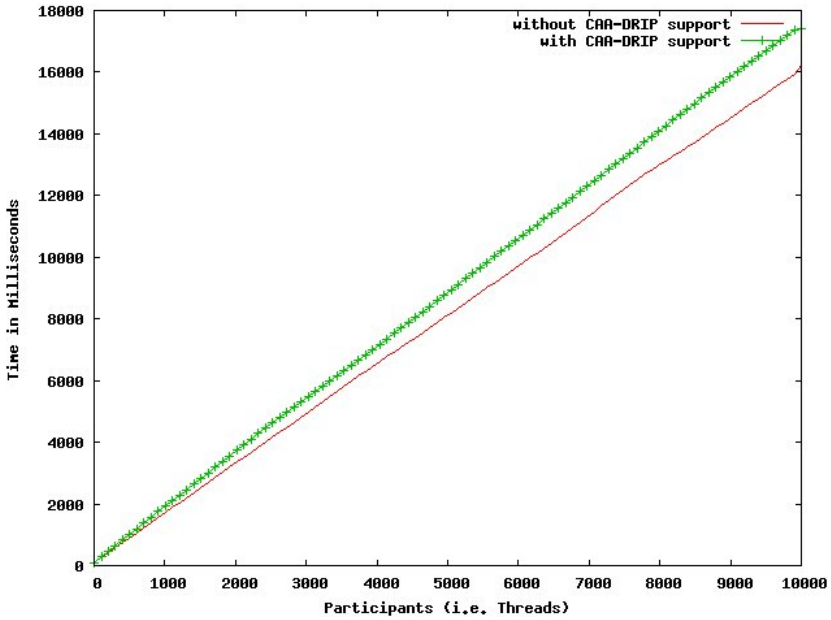

Fig. 18 CAA-DRIP framework overhead: participants highly coupled

tional situations, can be concluded that the benefits for using it are higher than processing time overhead it introduces.

The results appearing on Figure 17 and 18 were measured on a 2GHz Pentium M PC with 1Gb. of memory RAM, running Linux [9] and Java version 1.6.0_01. All the participants (i.e. threads) taking part in both applications (i.e. the one using CAA-DRIP support and the one without the framework support) used to perform the analysis were run on the same Java Virtual Machine.

4 Towards an advanced Service Oriented (SO) implementation of the FTIP control system

The increasing incorporation of the Internet as an external component in the design, implies the incorporation of related paradigms (e.g. SOA) and technologies (e.g. Web Services and UDDIs) to achieve the final product (i.e. the piece of software). Therefore, WISs are becoming the rule rather than the exception when a software product is delivered. In the same manner, techniques of fault tolerance should evolve following this tendency. Thus, such techniques should be rethought to make of openness and flexibility their main attributes in order to reach dependability. An architecture following this philosophy has been presented in [34]. In this work it is introduced a specific architectural model for building adaptable systems. This adaptation relies on the use of some key components, like dynamic resilience mechanisms (DRMs), resilience policies, metadata registry and the services for reasoning and reconfiguration.

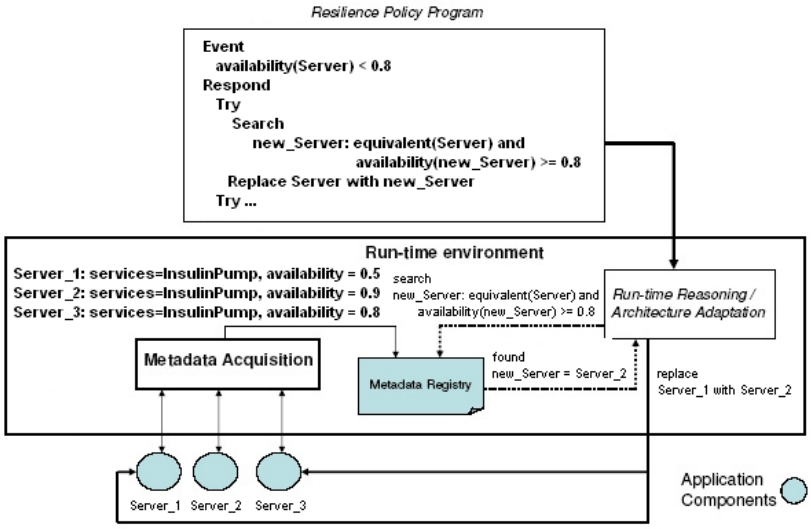

Fig. 19 A run-time environment supporting dynamic resilience

The DRMs are the generic patterns that allow the implementation of application-specific policies. The policies use metadata of the components, these are the data that describe the properties of the components. The metadata include functional properties, like specifications of component behaviour and component failure modes, and non-functional properties like

[9] The distribution used was Ubuntu 7.04. The times were taken using the Linux/Unix command **time**.

measures of availability or description of needed resources. At runtime, the metadata is stored in a registry (called the metadata registry) and the policies are implemented as services that are in charge of processing the metadata to give the desired flexibility to the system.

Figure 19 depicts how this proposed architecture would look like for the case study previously presented in this work.

5 Conclusion

The integration of mobility in the e-health domain opens the possibility of developing new software systems to improve the quality of life of patients with certain type of disease. Treatments can be remotely followed by doctors and patients can receive their therapies continuously along the day without needing to go neither to hospital nor to doctor's place. We have shown through a simple (yet realistic) case study how we can use our CAA conceptual framework and its implementation support (CAA-DRIP) to design and develop dependable WIS that include services running on mobile devices. Even if we have tailored the CAA-DRIP framework for mobile devices, we think that a complete re-design of the CAA-DRIP for mobile devices should be necessary mainly for optimising resources. This work has been done using a simulation technique for some devices (based on current physical device constraints). The next phase of our work would be to replace some of the simulation parts by real devices (as soon as they are available on the market). Finally, we should show how the forthcoming verification tools for CAA (covering testing and model checking) could be used and adapted to this application domain to verify some dependability properties.

Acknowledgement

The authors would like to give thanks to Cédric Pruski, Marcos Da Silveira and the anonymous reviewers for their comments.

References

1. P. Pelliccione, H. Muccini, N. Guelfi and A. Romanovsky: Software Engineering and Fault Tolerance. World Scientific Publishing Co. Pte. Ltd, Series on Software Engineering and Knowledge Engineering (To appear on 2007)
2. Eysenbach, G.: What is e-health? J Med Internet Res **3**(2) (2001) e20
3. Randell, B., Romanovsky, A., Stroud, R.J., Xu, J., Zorzo, A.F.: Coordinated Atomic Actions: from Concept to Implementation. Technical Report 595 (1997)
4. J. Xu, A. Romanovsky, R.J. Stroud, A.F. Zorzo, E. Canver and F. von Henke: Rigorous Development of an Embedded Fault-Tolerant System Based on Coordinated Atomic Actions. IEEE Trans. Comput. **51**(2) (2002) 164–179
5. G. Di Marzo Serugendo, N. Guelfi, A. Romanovsky and A. Zorzo: Formal Development and Validation of Java Dependable Distributed Systems. In: Proceedings of ICECCS'99. (1999) 98–108
6. A. Romanovsky, P.P., Zorzo, A.: On Structuring Integrated Web Applications for Fault Tolerance. In: Proceedings of ISADS. (2003) 99–106
7. A. Capozucca and N. Guelfi and P. Pelliccione and A. Romanovsky and A. Zorzo: CAA-DRIP: a framework for implementing Coordinated Atomic Actions. In: The 17th IEEE International Symposium on Software Reliability Engineering (ISSRE), IEEE Computer Society (2006) 385–394

8. National Institute for Health and Clinical Excellence: Guidance on the use of continuous subcutaneous insulin infusion for diabetes. www.nice.org.uk (Technology Appraisal 57) (2003)
9. A. Capozucca and N. Guelfi and P. Pelliccione: The Fault-Tolerant Insulin Pump Therapy. Rigorous Engineering of Fault-Tolerant Systems (LNCS 4157, Lecture Notes in Computer Sciences, Springer-Verlag, Berlin Heidelberg) (2006) 59–79
10. Randell, B.: System Structure for Software Fault Tolerance. IEEE Transactions on Software Engineering. IEEE Press **SE-1**(2) (1975) 220–232
11. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
12. J. Xu, A. Romanovsky and B. Randell: Concurrent Exception Handling and Resolution in Distributed Object Systems. IEEE Trans. Parallel Distrib. Syst. **11**(10) (2000) 1019–1032
13. F. Tartanoglu, N. Levy, V. Issarny and A. Romanovsky: Using the B Method for the Formalization of Coordinated Atomic Actions. (In: Proc. ICSE 2003 Workshop on Software Architectures for Dependable System)
14. Julie Vachon: COALA : a design language for reliable distributed systems. PhD thesis, Swiss Federal Institute of Technology Lausanne (Thesis no 2302) (2000)
15. Biberstein, O., Buchs, D., Guelfi, N.: CO-OPN/2: A concurrent object-oriented formalism. In: Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS), London, Chapman and Hall (1997) 57–72
16. Erl, T.: Service-Oriented Architecture, Concepts, Technology, and Design. Prentice Hall (2006)
17. Java Technology: (http://java.sun.com)
18. Java ME Platform: (http://java.sun.com/javame/)
19. Sun Developer Network: (http://developers.sun.com/techtopics/mobility/overview.html)
20. J2ME RMI Optional Package Specification v1.0: (http://java.sun.com/javame/reference/apis/)
21. Shui, A., Mustafiz, S., Kienzle, J., Dony, C.: Exceptional Use Cases. In: MoDELS. (2005) 568–583
22. Dijkstra, E.W.: On the role of scientific thought. In: Selected Writings on Computing: A Personal Perspective. Springer-Verlag (1982) 60–66
23. Parnas, D.L.: On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM **15**(12) (1972) 1053–1058
24. Eclipse SDK: (http://www.eclipse.org)
25. EclipseME: (http://www.eclipseme.org)
26. Sun Java Wireless Toolkit for CLDC: (http://java.sun.com/products/sjwtoolkit/)
27. : Ieee standard glossary of software engineering terminology. Technical report (1990)
28. Tao Group: (http://tao-group.com/)
29. University of Cambridge: http://www.admin.cam.ac.uk/news/press/dpp/2006110102 (2006)
30. CLINICIP Consortium: http://www.clinicip.org/index.php?id=161 (2007)
31. Medtronic, Inc.: (http://www.minimed.com/products/insulinpumps/)
32. Insignia Jeode Runtime Environment, Insignia Solutions: (http://www.esmertec.com)
33. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Trans. Dependable Sec. Comput. **1**(1) (2004) 11–33
34. Serugendo, G.D.M., Fitzgerald, J., Romanovsky, A., Guelfi, N.: A metadata-based architectural model for dynamically resilient systems. In: SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, New York, NY, USA, ACM (2007) 566–572

Appendix I: COALA design of the FTIP control system

```
;; ***********************************
Caa Main;
;; ***********************************
Interface
   Roles
      Sensor;
```

```
    Controller:record;
    Pump;
    MobileDevice;
    Parameters;

Body
  Use
    Data, Buffer, Heuristic, System;

Use Caa
    Configuration,
    Reconfiguration,
    GetGlucoseLevel,
    DeliverInsulin;

Caa
    Configuration["conf"];
    Reconfiguration["reconf"];
    GetGlucoseLevel["gettingGL"];
    DeliverInsulin["deliveringID"];

  Objects
    configParam: parameters;
    registry: record;
    ev: event;
    connected, configured: boolean;
    currentInsulinLevel: integer;
    pda: device;
    info: dataBuffer;


  Exception
    stop:string;
    unhealthyGlucoseLevel:string;
    unhealthyInsulinDose:string;
    pumpTimeout:string;
    pumpStuck:string;
    wrongDoseInjected:string;
    changingCartridgeTimeout:string;
    exit;


  Resolution
    stop -> stop;
    unhealthyGlucoseLevel -> stop;
    unhealthyInsulinDose -> stop;
    pumpTimeout -> stop;
    pumpStuck -> stop;
    wrongDoseInjected -> stop;
    changingCartridgeTimeout -> stop;
    exit -> exit;
    exit, stop -> stop;


  Role Sensor;
    Handler emergencyStop;
    Handling
            stop -> emergencyStop;
            exit -> exitApp;
    Begin
            Assign false to exit;
            Repeat
                    Call Sensor(registry) of GetGlucoseLevel["gettingGL"];
            Until(exit=true);
    End
    Where
      exit: boolean;
    Handler emergencyStop;
            Begin
            End
    End emergencyStop;
    Handler exitApp;
```

```
            Begin
            End
  End exitApp;
End Sensor;


Role Controller (log);
  Handler emergencyStop, exitApp;
  Handling
            stop -> emergencyStop;
            exit -> exitApp;
  Begin
    Assign false to exit;
    Repeat
    Begin
      info.put(log);
      If(ev="START" and configured=true) then
      Repeat
       Begin
         Call Controller(configParam,currentGlucoseLevel,registry)
                  of GetGlucoseLevel["gettingGL"];

         info.get(log);
         log.addGlucoseLevel(timeStamp,currentGlucoseLevel);
         info.put(log);

         Execute heuristic.calculus.setInsulinLevel(currentGlucoseLevel);
         Execute heuristic.calculus.getInsulinLevel(insulinDose);

         Execute configParams.getID_MAX(upperID);
         Execute configParams.getID_MIN(lowerID);

         If((insulinDose < lowerID) or (insulinDose > upperID)) then
            Raise unhealthyInsulinDose;

         Call Controller(configParam,insulinDose,registry)
                  of DeliverInsulin["deliveringID"];

         info.get(log);
         log.addInsulinInjected(timeStamp,insulinDose);
         info.put(log);


         If(ev="PAUSE") then
          Repeat
          Until(ev="CONTINUE");
       End;
      Until(ev="STOP");
     End
    Until(exit=true);
  End
  Where
    exit: boolean;
    upperID, lowerID, insulinDose, currentGlucoseLevel: integer;
    timeStamp: Clock;
  Handler emergencyStop;
            Begin
            End
  End emergencyStop;
  Handler exitApp;
            Begin
            End
  End exitApp;
End Controller;


Role Pump;
  Handler emergencyStop, exitApp;
  Handling
            stop -> emergencyStop;
            exit -> exitApp;
  Begin
```

```
            Assign false to exit;
            Repeat
              Call Pump(registry) of DeliverInsulin["deliveringID"];
            Until(exit=true);
  End
  Where
    exit: boolean;
  Handler emergencyStop;
            Begin
            End
  End emergencyStop;
  Handler exitApp;
            Begin
            End
  End exitApp;
End Pump;


Role MobileDevice;
  Handlers
            emergencyStop:string,
            exitApp;
  Handling
            stop(msg)-> emergencyStop(msg);
            exit -> exitApp;
  Begin
            Execute mainWindow.open();
            Assign false to exit;
            Assign false to connected;
            Repeat
            Begin
                     If(ev="CONFIG") then
                       Begin
                              Try
                                Begin
                                  Call MobileDevice of Configuration["conf"];
                                End
                              Catch(abort)
                                Begin
                                End
                        End

                     Execute pda.connection.getStatus(connected);

                     If(connected=true) then
                       Begin
                              Try
                                Begin
                                  Call MobileDevice of Reconfiguration["reconf"];
                                End
                              Catch(abort)
                                Begin
                                End
                       End

                     If(ev="EXIT") then
                              Signal exit;
            End
            Until(exit=true);
  End
  Where
    mainWindow: frame;
    exit: boolean;
  Handler emergencyStop(errorMessage);
            Begin
                     Execute alertWindow.put(errorMessage);
                     Execute alertWindow.show();
                     Repeat
                        pda.ringAlarm();
                     Until(ev="OK");
            End
            Where
```

```
                    alertWindow: alertBox;
      End emergencyStop;
      Handler exitApp;
              Begin
              End
      End exitApp;
    End MobileDevice;


    Role Parameters;
      Handler emergencyStop, exitApp;
      Handling
              stop -> emergencyStop;
              exit -> exitApp;
      Begin
        Assign false to exit;
        Assign false to configured;
        Repeat
        Begin
          If(ev="CONFIG") then
              Begin
                Try
                      Begin
                      Call Parameters(configParam,registry)
                              of Configuration["conf"];
                      Assign true to configured;
                      End
                Catch(abort)
                      Begin
                      End
              End

          If(connected=true) then
              Begin
                Try
                      Begin
                      Execute info.get(log);
                      Call Parameters(configParam,log,registry)
                              of Reconfiguration["reconf"];
                      Execute info.put(log);
                End
                Catch(abort)
                      Begin
                      End
              End
        End
        Until(exit=true);
      End
      Where
        exit: boolean;
        log: record;
      Handler emergencyStop;
              Begin
              End
      End emergencyStop;
      Handler exitApp;
              Begin
              End
      End exitApp;
    End Parameters;
End Main;




;; ***********************************
Caa Configuration;
;; ***********************************
   Interface
      Use
        Data;
```

```
    Roles
      MobileDevice;
      Parameters: parameters, record;

  Body
    Use
      Data, Buffer, Services;

    Objects
      info: dataBuffer;
      ev: event;

    Role MobileDevice;
      Begin
        Execute confWindow.open();
        Assign false to exit;
        Repeat
          Begin
              If(ev="OK") then
                Begin
                  Execute confWindow.getFieldContent("TBGL",tbgl);
                  Execute confWindow.getFieldContent("MAX_BGL",upperBGL);
                  Execute confWindow.getFieldContent("MIN_BGL",lowerBGL);
                  Execute confWindow.getFieldContent("MAX_ID",upperID);
                  Execute confWindow.getFieldContent("MIN_ID",lowerID);
                  Execute confWindow.getFieldContent("TCC",tcc);
                  Execute confWindow.getFieldContent("LLIW",lliw);
                  Execute confWindow.getFieldContent("DPE",dpe);
                  Execute confWindow.getFieldContent("SENSOR",sensorType);
                  Execute confWindow.getFieldContent("PUMP",pumpType);
                  Execute confWindow.getFieldContent("HOSPITAL",hospitalURL);

                  If(tbgl="" or upperBGL="" or lowerBGL="" or upperID="" or
                      lowerID="" or tcc="" or lliw="" or dpe="" or
                      sensorType="" or pumpType="" or hospitalURL="") then
                      Execute confWindow.putFieldContent("msg",
                              "Inclomplete form. Please fill all fields");
                  else
                      Begin
                        Execute confWindow.close();
                        Execute confValues.setTBGL(tbgl);
                        Execute confValues.setMAX_BGL(upperBGL);
                        Execute confValues.setMIN_BGL(lowerBGL);
                        Execute confValues.setMAX_ID(upperID);
                        Execute confValues.setMIN_ID(lowerID);
                        Execute confValues.setTCC(tcc);
                        Execute confValues.setLLIW(lliw);
                        Execute confValues.setDPE(dpe);
                        Execute confValues.setSENSOR(sensorType);
                        Execute confValues.setPUMP(pumpType);
                        Execute confValues.setHSP(hospitalURL);
                        Execute info.put(confValues);
                        Assign true to exit;
                      End;
                End
              else
                      If(ev="CANCEL") then
                              Signal abort;
        End
        Until(exit=true);
      End
      Where
        confWindow: form;
        exit:  boolean;
        tbgl, upperBGL, lowerBGL, upperID, lowerID, tcc, lliw, dpe: string;
        sensorType, pumpType, hospitalURL: string
        confValues: parameters;
    End MobileDevice;


    Role Parameters (configParams,registry);
```

```
        Begin
                Execute info.get(confValues);

                Execute confValues.getTBGL(tbgl);
                Execute confValues.getMAXBGL(upperBGL);
                Execute confValues.getMINBGL(lowerBGL);
                Execute confValues.getMAXID(upperID);
                Execute confValues.getMINID(lowerID);
                Execute confValues.getTCC(tcc);
                Execute confValues.getLLIW(lliw);
                Execute confValues.getDPE(dpe);
                Execute confValues.setSENSOR(sensorType);
                Execute confValues.setPUMP(pumpType);
                Execute confValues.setHSP(hospitalURL);

                Execute configParams.setTBGL(tbgl);
                Execute configParams.setMAXBGL(upperBGL);
                Execute configParams.setMINBGL(lowerBGL);
                Execute configParams.setMAXID(upperID);
                Execute configParams.setMINID(lowerID);
                Execute configParams.setTCC(tcc);
                Execute configParams.setLLIW(lliw);
                Execute configParams.setDPE(dpe);

                Assign registry.discover(Sensor) to sensorService;
                Execute sensorService.setType(sensorType);
                Execute registry.publish(sensorService, Sensor);

                Assign registry.discover(Pump) to pumpService;
                Execute pumpService.setType(pumpType);
                Execute registry.publish(pumpService, Pump);

                Assign registry.discover(Admin) to adminService;
                Execute adminService.setURL(hospitalURL);
                Execute registry.publish(adminService, Admin);

        End
        Where
          confValues: parameters;
          tbgl, upperBGL, lowerBGL, upperID, lowerID, tcc, lliw, dpe: string;
          sensorType, pumpType, hospitalURL: string
          sensorService, pumpService, adminService: service;
      End Parameters;
End Configuration;




;; ***********************************
Caa Reconfiguration;
;; ***********************************
   Interface
      Use
        Data;

      Roles
        MobileDevice;
        Parameters: parameters, record, record;

   Body
      Use
        Data, Buffer, Services;

      Objects
        info: dataBuffer;
        ev: event;

      Role MobileDevice;
        Begin
                Execute info.get(log);
                Execute info.get(registry);
```

```
            Assign registry.discover(Admin) to adminService;

            Try
              Begin
                    Execute adminService.addLogPatient(log);
                    Execute adminService.getParamsPatient(confValues);
              End
            Catch(ConnectionInterrupted)
              Begin
                    Signal abort;
              End

            Execute log.clean();
            Execute info.put(log);
            Execute info.put(confValues);
  End
  Where
    adminService: service;
    log, registry: record;
    confValues: parameters;
End MobileDevice;


Role Parameters (configParams,log,registry);
  Begin
            Execute info.put(log);
            Execute info.put(registry);

            Execute info.get(log);
            Execute info.get(confValues);

            Execute confValues.getTBGL(remoteTbgl);
            Execute confValues.getMAXBGL(remoteUpperBGL);
            Execute confValues.getMINBGL(remoteLowerBGL);
            Execute confValues.getMAXID(remoteUpperID);
            Execute confValues.getMINID(remoteLowerID);
            Execute confValues.getTCC(remoteTcc);
            Execute confValues.getLLIW(remoteLliw);
            Execute confValues.getDPE(remoteDpe);

            Execute configParams.getTBGL(tbgl);
            Execute configParams.getMAXBGL(upperBGL);
            Execute configParams.getMINBGL(lowerBGL);
            Execute configParams.getMAXID(upperID);
            Execute configParams.getMINID(lowerID);
            Execute configParams.getTCC(tcc);
            Execute configParams.getLLIW(lliw);
            Execute configParams.getDPE(dpe);

            If(remoteTbgl <> tbgl) then
                        Execute configParams.setTBGL(remoteTbgl);
            If(remoteUpperBGL <> upperBGL) then
                        Execute configParams.setMAXBGL(remoteUpperBGL);
            If(remoteLowerBGL <> lowerBGL) then
                        Execute configParams.setMINBGL(remoteLowerBGL);
            If(remoteUpperID <> upperID) then
                        Execute configParams.setMAXID(remoteUpperID);
            If(remoteLowerID <> lowerID) then
                        Execute configParams.setMINID(remoteLowerID);
            If(remoteTCC <> tcc) then
                        Execute configParams.setTCC(remoteTcc);
            If(remoteLLIW <> lliw) then
                        Execute configParams.setLLIW(remoteLliw);
            If(remoteDPE <> dpe) then
                        Execute configParams.setTCC(remoteDpe);
     End
  Where
    confValues: parameters;
    remoteTbgl, remoteUpperBGL, remoteLowerBGL, remoteUpperID: integer;
    remoteLowerID, remoteTcc, remoteLliw, remoteDpe: integer;
    tbgl, upperBGL, lowerBGL, upperID, lowerID, tcc, lliw, dpe: integer;
```

```
      End Parameters;
End Reconfiguration;




;; ***********************************
Caa GetGlucoseLevel;
;; ***********************************
   Interface
      Use
        Data;

      Roles
        Sensor: record;
        Controller:parameters, integer;

      Exceptions
        stop, unhealthyGlucoseLevel:string;


   Body
      Use
        Data, Buffer, Services;

      Objects
        info: dataBuffer;

      Exceptions
        sensorTimeout;

      Resolution
        sensorTimeout -> sensorTimeout;


      Role Sensor(registry);
        Handler reconnectingGlucoseSensor:record;

        Handling sensorTimeout -> reconnectingGlucoseSensor;
        Begin
          Try
            Begin
                Assign registry.discover(Sensor) to sensorService;
                Execute sensorService.getGlucoseLevel(glucoseLevel);
            End
          Catch(sensorTimeout)
            Begin
                Raise sensorTimeout;
            End

          Execute info.put(glucoseLevel);
        End
        Where
                glucoseLevel: integer;
                sensorService: service;
        Handler reconnectingGlucoseSensor(registry);
                Begin
                  Try
                    Begin
                        Assign registry.discover(Sensor) to sensorService;
                        Execute sensorService.getGlucoseLevel(glucoseLevel);
                    End
                  Catch(sensorTimeout)
                    Begin
                        Signal stop("Sensor does not work");
                    End

                  Execute info.put(glucoseLevel);
                End
                Where
                        glucoseLevel: integer;
```

```
                        sensorService: service;
        End reconnectingGlucoseSensor;
      End Sensor;

      Role Controller(configParams, currentGlucoseLevel);
    Handler
          reconnectingGlucoseSensor:parameters, integer;
      Handling
         sensorTimeout ->
              reconnectingGlucoseSensor(configParams,currentGlucoseLevel);
      Begin
        Assign 0 to currentGlucoseLevel;
        Execute configParams.getMAXBGL(upperBGL);
        Execute configParams.getMINBGL(lowerBGL);
        Execute info.get(currentGlucoseLevel);
        If(upperBGL <= currentGlucoseLevel)
          or (currentGlucoseLevel >= lowerBGL)) then
            Signal unhealthyGlucoseLevel("The glucose level
                      is out of the safe level");
      End
      Where
        upperBGL: integer;
        lowerBGL: integer;
      Handler
      reconnectingGlucoseSensor(configParams, currentGlucoseLevel);
          Begin
            Assign 0 to currentGlucoseLevel;
            Execute configParams.getMAXBGL(upperBGL);
            Execute configParams.getMINBGL(lowerBGL);
            Execute info.get(currentInsulinLevel);
            If(upperBGL <= currentGlucoseLevel)
              or (currentGlucoseLevel >= lowerBGL)) then
                Signal unhealthyGlucoseLevel("The glucose level
                              is out of the safe level");
          End
          Where
            upperBGL: integer;
            lowerBGL: integer;
      End reconnectingGlucoseSensor;
    End Controller;
End GetGlucoseLevel;




;; ***********************************
Caa DeliverInsulin;
;; ***********************************
   Interface
      Use
        Data;

      Roles
        Controller: parameters,integer;
        Pump: record;
        MobileDevice;

      Exceptions
        stop:string;
        pumpTimeout:string;
        pumpStuck:string;
        wrongDoseInjected:string;
        changingCartridgeTimeout:string;


   Body
      Use
        Data, Buffer, Math, Time, Services;

      Objects
        info: dataBuffer;
```

```
  msg: dataBuffer;
  ev: event;

Exceptions
  lackOfInsulin;

Resolution
  lackOfInsulin -> lackOfInsulin;


Role Controller(configParams,insulinDose);
  Handler changeCartridge: parameters,integer;
  Handling lackOfInsulin -> changeCartridge(configParams,insulinDose);
  Begin
    Execute insulinDose.getValue(insulinDoseValue);
    Execute info.put(insulinDoseValue);
    Execute info.get(actualDoseInjected);
    Execute configParams.getDPE(dpeTolerated);
    If(Abs(insulinDosevalue-actualDoseInjected) > dpeTolerated) then
      Signal wrongDoseInjected("The dose needed is
                  out of the allowed level");

    Execute info.get(remainingInsulin);
    Execute configParams.getLLIW(lliwLimit);
    If(remainingInsulin <= lliwLimit) then
      Assign "The level of insulin is low.
      Please, replace the cartridge as soon as possible";
    else
      Begin
        Assign "" to notificationMsg;
        Execute msg.put(notificationMsg);
      End;

  End
  Where
    insulinDoseValue: integer;
    actualDoseInjected: integer;
    remainingInsulin: integer;
    dpeTolerated: integer;
    lliwLimit: integer;
    notificationMsg: string;
  Handler changeCartridge(configParams, insulinDose);
    Begin
          Execute myClock.get(timeBeginning);
          Execute configParams.getTCC(tcc);
          Repeat
             Execute myClock.get(timeNow);
             If((timeNow-timeBeginning)>tcc) then
                   Signal changingCartridgeTimeout("Not enough
                             insulin in the cartridge");
          Until(ev="OK");

          Repeat
          Until(ev="CONTINUE");

          Execute insulinDose.getValue(insulinDoseValue);
          Execute info.put(insulinDoseValue);
          Execute info.get(actualDoseInjected);
          Execute configParams.getDPE(dpeTolerated);
          If(Abs(insulinDosevalue-actualDoseInjected) > dpeTolerated) then
            Signal wrongDoseInjected("The dose needed is
                        out of the allowed level");
          Execute info.get(remainingInsulin);
          Execute configParams.getLLIW(lliwLimit);
          If(remainingInsulin <= lliwLimit) then
            Assign "The level of insulin is low.
                  Please, replace the cartridge as soon as possible"
            to notificationMsg;
          else
            Begin
              Assign "" to notificationMsg;
              Execute msg.put(notificationMsg);
```

```
            End;
      End
      Where
            timeBeginning: timeStamp;
            timeNow: timeStamp;
            tcc: timeStamp;
            myClock: clock;
            insulinDoseValue: integer;
            actualDoseInjected: integer;
            remainingInsulin: integer;
            dpeTolerated: integer;
            lliwLimit: integer;
            notificationMsg: string;
  End changeCartridge;
End Controller;

Role Pump(registry);
  Handler changeCartridge: record;
  Handling lackOfInsulin -> changeCartridge;
    Begin
      Execute info.get(doseValue);

      Try
            Begin
              Assign registry.discover(Pump) to pumpService;
              Execute pumpService.cartridgeLevel(preInsulinDeliveryLevel);
              Execute pumpService.inject(doseValue);
              Execute pumpService.cartridgeLevel(postInsulinDeliveryLevel);
            End
      Catch(pumpTimeout)
            Begin
              Signal pumpTimeout("Pump does not work");
            End
      Catch(lackOfInsulin)
            Begin
              Raise lackOfInsulin;
            End
      Catch(pumpStuck)
            Begin
              Signal pumpStuck("The pump is stuck");
            End

      Assign (preInsulinDeliveryLevel - postInsulinDeliveryLevel)
                        to doseValueInjected;
      Execute info.put(doseValueInjected);
      Execute info.put(postInsulinDeliveryLevel);

    End
    Where
            doseValue: integer;
            preInsulinDeliveryLevel: integer;
            postInsulinDeliveryLevel: integer;
            pumpService: service;
            doseValueInjected: integer;
  Handler changeCartridge(registry);
    Begin

            Repeat
            Until(ev="CONTINUE");

            Execute info.get(doseValue);

            Try
            Begin
              Assign registry.discover(Pump) to pumpService;
              Execute pumpService.cartridgeLevel(preInsulinDeliveryLevel);
              Execute pumpService.inject(doseValue);
              Execute pumpService.cartridgeLevel(postInsulinDeliveryLevel);
            End
            Catch(pumpTimeout)
            Begin
              Signal pumpTimeout("Pump does not work");
```

```
                End
                Catch(lackOfInsulin)
                Begin
                  Signal stop("Not enough insulin in the cartridge");
                End
                Catch(pumpStuck)
                Begin
                  Signal pumpStuck("The pump is stuck");
                End


                Assign (preInsulinDeliveryLevel - postInsulinDeliveryLevel)
                  to doseValueInjected;
                Execute info.put(doseValueInjected);
                Execute info.put(postInsulinDeliveryLevel);

            End
            Where
                doseValue: integer;
                preInsulinDeliveryLevel: integer;
                postInsulinDeliveryLevel: integer;
                pumpService: service;
                doseValueInjected: integer;
        End changeCartridge;
      End Pump;

      Role MobileDevice;
        Handler changeCartridge;
        Handling lackOfInsulin -> changeCartridge;
          Begin
           Execute msg.get(message);
           If(message <> "") then
                Begin
                  Execute notificationWindow.put(message);
                  Execute notificationWindow.show();
                End
          End
          Where
            message: string;
            notificationWindow: warningBox;
        Handler changeCartridge;
          Begin
           Assign "Critical level of insulin.
                Please, replace the cartridge NOW"
                to message;
           Execute alertWindow.put(message);
           Execute alertWindow.show();

           Repeat
           Until(ev="OK");

           Repeat
             If(ev="STOP") then
                        Signal stop("");
           Until(ev="CONTINUE");

           Execute msg.get(message);
           If(message <> "") then
             Begin
               Execute notificationWindow.put(message);
               Execute notificationWindow.show();
             End
          End
          Where
            alertWindow: alertBox;
            message: string;
            notificationWindow: warningBox;
        End changeCartridge;
      End MobileDevice;
End DeliverInsulin;
```