The Fault-Tolerant Insulin Pump Therapy

Alfredo Capozucca, Nicolas Guelfi, and Patrizio Pelliccione

Laboratory for Advanced Software Systems
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
Luxembourg, L-1359 -Luxembourg
{alfredo.capozucca, nicolas.guelfi, patrizio.pelliccione}@uni.lu

Abstract. The "Fault-Tolerant Insulin Pump Therapy" is based on the Continuous Subcutaneous Insulin Injection technique which combines devices (a sensor and a pump) and software in order to make glucose sensing and insulin delivery automatic. These devices are not physically connected together and they come with the necessary features to detect malfunctions which they may have.
As the patient's health is the most important, the therapy has to be able to work despite the fact that hardware and/or software faults have or may ocurr.
This paper presents the development cycle for the Insulin Pump Therapy Control System case study, starting from requirements and reaching the implementation following a top-down approach. It will show how the Coordinated Atomic Actions (CAAs) structuring mechanism can be used for modelling Faul-Tolerant (FT) systems and how CAA-DRIP development environment is used to implement it.

1 Introduction

Software and hardware systems are being used increasingly in many sectors, such as manufacturing, aerospace, transportation, communication, energy and healthcare. Failures due to software or hardware malfunctions and malicious intentions can not only have economic consequences, but can also endanger human life. In fact, if a health care system breaks down, the consequences for hospitals and patients could be huge. Dependability is vital in health care systems which must be available around the clock without exception.

One of the techiques used to achieve dependable systems is fault tolerance. This technique seeks to preserve the delivery of correct service even in the presence of active faults. It is implemented by error detection and subsequent system recovery. In the context of fault tolerance and distributed systems, a promising technique emerged in recent years named Coordinated Atomic Actions (CAAs) [14] is used to coordinate complex concurrent activities and support error recovery between multiple interacting objects. By using CAAs for designing and structuring these kinds of systems the necessary requirements of reliability and availability are met.

The aim of this paper is to show how CAAs can be used successfully to design medical software with reliability and availability requirements. The code generation is delegated to the CAA-DRIP framework [4], which embodies CAAs in terms of a set of Java classes. The case study concerns a diabetes control system which aims at delivering insulin to patients in the correct manner. A tiny sensor checks the patient's status and a pump administers the insulin. Doctors have to set some parameters on the pump which are used by the software embedded on it. It is of primary importance, for the patient's health, that the whole application works properly 24 hours a day without interruption.

The case study is described following the *WRSPM* reference model [9]. It is a general methodology wich introduces the use of formal languages (no one in particular) early in the development software process. In particular, this methodology advocates a careful and explicit description of the environment that is independent of the presence and operations of the system to be constructed. Therefore *W*, the *world* or *domain knowledge*, has to provide presumed facts about the operational domain where the system to be built will be embedded. The other parts composing the methodology are: *requirements (R)*, which indicate what users need; *specifications (S)* which provide enough information for a programmer to build a system to satisfy the requirements, and *program (P)* which implements the specifications on the *Machine (M)*. *W, R* and *S* are described using formal languages.

The paper is organized as follows. In Section 2, some background information on the methodology, followed in order to reach the system specification, and details on the CAAs structuring mechanism are given. In Section 3, the CAA-DRIP framework will be presented, which provides support to implement specifications described in terms of CAAs. Section 4 provides a detailed description of the domain application (introducing basic terminology), requirements, specification, design and implementation of the considered case study. Finally, Section 5 will provide some conclusions and ongoing work.

2 WRSPM methodology and Coordinated Atomic Actions (CAAs)

Software engineering methodology is aimed at guiding software system developers from user-level requirements until the code generation and execution. The WRSPM methodology does exactly that by putting special emphasis on the use of formal languages in order to capture the requirements as well as to describe the assumptions about the application domain.

The WRSPM methodology consists of five artifacts. *W*, the world, represents the presumed facts about the environment (domain knowledge) where the program *P* will be embedded. This program *P* will be running on a programming platform *M* and it has to satisfy the requirements *R*. *S* is the specification that provides enough information to programmers to build the program *P*.

To describe the application domain a primitive vocabulary, called *designate terminology*, needs to be available. It provides the terms used to described *W*,

R and S, and an informal explanation of their meaning to clarify the role that these terms may play. The designated terminology of the case study presented here can be found in the *Appendix* at the end of the paper.

The specification S is the intersection between the *system* (S,P,M) and its *environment* (W,R,S). The overlapping of these groups means that S lies in the common vocabulary of the environment and the system but, thanks to the domain knowledge W, S still has enough information to meet the requirements R. Since WRSPM does not impose a specific formal notation, statecharts [6] are used (with semantics as implemented in the STATEMATE system [10]) as the formal language for writing the descriptions.

In this paper, the purpose of using WRSPM methodology is to reach a specification (S) which, under the assumptions and constraints imposed by the "world" (or "domain knowledge") (W), is able to meet the requirements (R). Therefore, WRSPM is followed until the specification S is defined. Once S is determined, a new description of the system in terms of CAAs will be provided in order to meet the requirements of dependability, in particular, reliabilty, in spite of faults coming from the environment (external faults) and internal dormant faults which may be activated.

Coordinated Atomic Actions (CAAs) is a fault tolerance mechanism that uses concurrent exception handling to achieve dependability in distributed and concurrent systems. Thus, by using CAAs, systems that comply with their specification in spite of faults can be developed.

This mechanism unifies the features of two complementary concepts: *conversation* and *transaction*. Conversation [13] is a fault tolerance technique for performing coordinated recovery in a set of participants that have been designed to interact with each other in order to provide a specific service (cooperative concurrency). Objects that are used to achieve the cooperation among the participants are called **shared** objects. Transactions are used in order 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 referred to as **external** objects.

One CAA characterises an orchestration of actions executed by a group of roles that exchange information among them, and/or access to external objects (concurrently with others CAAs) to achieve a common goal. The CAA starts when all its roles have been activated and a pre-condition has been met. The CAA finishes when all of them have reached the CAA end and a post-condition is met. 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. In that regard, a CAA provides quite a general solution for fault tolerance based on exception handling. It consists of applying both forward error recovery (FER) and backward error recovery (BER) techniques.

Basically, the CAA exception handling semantics says that once an exception has been raised the FER mechanism has to be started. At that point, the CAA can finish normally if the FER can meet the original request (**normal** outcome)

or exceptionally if the original request is partially satisfied (**exceptional** outcome). Otherwise, if the same or another exception is raised during the FER, the FER mechanism is stopped and the BER is started. The BER's main task is to recover every external object to its last consistent state (roll back). If the BER is successful, the CAA returns the **abort** outcome. If for any reason the BER cannot be completed, then the CAA has failed and the **failure** outcome is signaled to the enclosing context.

Every external object that is accessed in a CAA must be able to be restored to its last consistent state (if BER is activated) and provide its own error recovery mechanism [15]. Therefore, when the BER takes place, it restores these external objects using their own recovery mechanisms. However, sometimes the designer/programmer might want to use an external object that does not provide any recovery mechanism (due to reasons of cost or physical constraints [2]). Therefore it would be necessary to allow designers/programmers to specify/implement a hand-made roll back inside the CAA. This can be achieved by refining the classic BER to deal with external objects that are restored using their own mechanism (called *AutoRecoverable* external objects) and also to deal with those that have to be restored by a hand-made roll back (called *ManuallyRecoverable*)

3 The CAA-DRIP Implementation Framework

A set of *Java* [1] classes and interfaces called CAA-DRIP has been defined using the DRIP framework [16] as a starting point. CAA-DRIP allows us to implement the CAAs concepts and behaviour described in Section 2 in a straightforward manner.

The core of this implementation framework is composed of *Manager, Role, Handler* and *Compensator* classes. The *Manager* class is the controller for *Role, Handler* and *Compensator* classes. Then, each role, handler and compensator object created is managed by a manager object. There is not a class to represent a CAA. It consists of a set of managers, roles, handlers and compensators linked together via a *leader* manager. This *leader* manager is one of the manager objects used to manage a role, handler and compensator. The *leader* manager is the responsible for synchronising roles upon entry and upon exit as well as handlers and compensators, in case the recovery process must take place.

Extending the CAA-DRIP framework classes

Some of the classes that are provided by CAA-DRIP implementation framework have to be extended by programmers. That is the case for *Role, Handler* and *Compensator* classes.

The definition of a role is made by creating a new class that extends the *Role* class. The programmer has to re-implement the *body* method. This method receives a list of external objects as input parameter and it does not return any value. The defined operations inside this method are executed by the participant which activated the role.

The other methods that have also to be redefined by the programmer are *preCondition* and *postCondition*. They return a *boolean* value and are used as guard and assertion of the role, respectively.

Once role classes have been created, it is necessary to define a handler for each CAA role in order to deal with exceptions (in this context, an exception means that a fault has been activated). This step is made by creating a new class that extends the *Handler* class. The programmer has to re-implement the *body* and the *postCondition* methods.

If the raised exception cannot be handled by FER, then the CAA has to undo all its effects on the external objects. This task is done by BER, which is composed of two phases, *Roll back* and *Compensation*. Compensation is used to execute specific tasks, in case there is at least one *external* object that needs manual recovery (see Section 2).

Compensation is achieved by defining a compensator for each CAA role. A compensator is made by creating a new class that extends the *Compensator* class. The *recovery* method has to be re-implemented by the programmer. This method receives, as input parameter, a list with the external objects that need hand-made recovery. The method has to contain the operations to leave these external objects in a consistent state.

Instantiating a CAA

Once roles, handlers and compensators classes have been created, the programmer has to make use of them to create the CAA itself. As said, a CAA consists of a set of managers, roles, handlers and compensators linked together via a *leader* manager. Thus, *Manager* framework class is used to create the manager objects.

When a *Manager* object is created it has to be informed of its name and the name of the CAA. Once the managers have been created it is necessary to create the role objects. Each role upon creation is informed of its name, which manager will be its controller and the manager that will act as the *leader*.

Each handler upon creation needs to know its name and the manager that will control this handler. The link between the exception to handle and each handler is implemented by a hash-table which has as key the exception and as value the handler object. The method *setExceptionAndHandlerList* is used to inform the manager about the relationships *exception-handler* that have been set before.

If the CAA has to handle manually recoverable objects (see Section 2), a compensator has to be created also. Then, analogously to a handler creation, each compensator upon creation is informed of its name and the manager that will drive its execution.

Executing a CAA

How classes are instantiated to create a CAA has been just shown. Now how these objects behave when the CAA is activated has to be explained. 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.

The first activity a manager executes 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 which 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 the *catch(Exception e)* block. 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.

After the role has finished its execution, the manager synchronises with all the other managers before testing its post-condition. If the post-conditions are satisfied, then the manager will synchronise with all the other managers and the CAA will finish successfully.

A *catch* block will be executed if an exception takes place during the execution of any role belonging to the CAA. In such situation, the role where the exception was raised notifies its manager. This manager passes the control to *leader* manager for interrupting[1] 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[2] from those that have been raised.

When such an exception is found, the *leader* informs all managers about that exception and FER (for the found exception) is activated. If every handler completes its execution and the CAA post-condition has been satisfied, then the CAA can finish.

Now, if the exception resolution algorithm could not find a common exception, or other exceptions were raised in the FER (even *PostConditionException* exception could be possible if FER did not satisfy the post-condition), then the BER mechanism will be started.

BER calls *restoreExecution* method. This method uses the compensators objects, if any, to undo the CAA effects. Once this method has executed, the CAA finishs returnig *Abort*. If for any reason the BER process could not complete its execution, CAA will be finished returning *Failure*.

More details about CAA-DRIP implementation framework can be found in [4].

[1] Notice that a role will be interrupted in our framework only if the role is ready to be interrupted, i.e. the role is in a state that it can be interrupted.

[2] In the worst case, the common exception is *Exception*.

4 The Fault-Tolerant Insulin Pump (FTIP) Control System case study

In this section WRSPM methodology is used to describe the case study in details. Once the FTIP control system specification is reached, a new specification in terms of CAAs is introduced to improve the reliability of the control system.

4.1 The world (W): Regulation of Blood Glucose in a non diabetic and a diabetic person

According to WRSPM methodology it is essential to provide a careful and explicit description of the domain, which in this case corresponds to medical software systems for treatment of diabetes.

Normally, blood glucose (blood sugar) is maintained in a narrow range [8]. The hormones which assure this are insulin and glucagons. Both of them are secreted by the pancreas. If the blood glucose level is too low then the pancreas secretes glucagons. If the blood glucose level is too high then the pancreas secretes insulin.

Diabetes Mellitus [12,7] is an illness whereby the level of glucose in the blood is abnormally high. This can be caused by either an absolute deficiency of insulin secretion, or as a result of reduced effectiveness of insulin, or both. Therefore, our first approach in the domain is to understand how the body of a non diabetic person regulates increases in the glucose level. Figure 1 describes how the body reduces the glucose level when it is higher than normal.

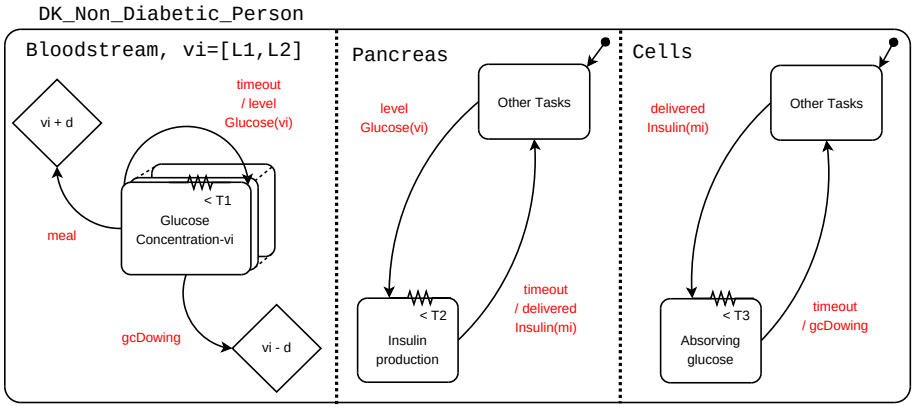

Fig. 1. Regulation of Blood Glucose in a non diabetic person

In a diabetic person this process does not succeed because of the lack of insulin secretion by the pancreas (event *deliverdInsulin*(m_i) is not present). Therefore, a diabetic person must make use of a insulin therapy to fulfil the process as in a normal person.

4.2 The requirement (R)

Another artifact which has to be provided under WRSPM is *Requirements*, which is indication of what users need. The range of glucose in the blood of a person should be between **70 mg/dl and 110 mg/dl** (mg/dl means milligrams of glucose in 100 milliliters of blood). Therefore, the functional requirement which needs to be met for any diabetes therapy is that the glucose level be kept below of 110 mg/dl. Figure 2 describes this requirement formally (**T1** represents the time that the body takes to update the glucose concentrations in the bloodstream.)

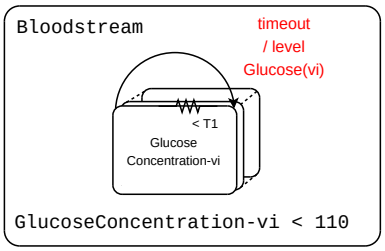

Fig. 2. The main requirement

4.3 The world (W): the devices

The Continuous subcutaneous insulin infusion is typically called **insulin pump therapy** [12]. Insulin pump therapy is a way of continuously delivering insulin to the body at a controlled rate. For example, more insulin can be delivered when it is needed at meal times.

The FTIP control system uses two devices, a sensor and a pump, to comply with the requirements. The sensor is a piece of hardware that communicates the patient's glucose level to the pump. The insulin pump is a small device that pumps insulin into the body through a cannula (small, thin tube) or a very thin needle inserted under the skin.

The proposal in this case study is to join and improve the characteristics of sensors and pumps that currently can be found in the market [7,11] to achieve a **close loop insulin delivering** without any participation of the patient. Therefore, the sensor sends the actual patient's glucose level to the pump. This pump has embedded software that is able to maintain the patient's glucose in a safe level by delivering necessary doses of insulin (basal rate) day and night. Obviously, this basal rate, can be easily increased or decreased to match the actual patient's needs that can change because of physical activities, illness or, simply as a meals has been taken.

Any interruption in insulin delivery (loss of insulin potency, or sensor/pump malfunction) may result in *hyperglycemia* (high blood glucose). Therefore, both the hardware (sensors and pumps) and specially the software which take place in this therapy have to be built with special techniques to achieve highly reliable and safety operation.

As these devices are part of the "world" where the software system is embedded, according to WRSPM, their behaviour have to be described precisely.

The sensor

The sensor is an adaptation of one provided by [11] for glucose monitoring. The tiny sensor used in FTIP has an integrated small transmitter which wirelessly and continually communicates the patient's glucose level to the pump. Every $T_{SensorValue}$ ($T_{SensorValueTimeout} = T_{s2} + T_{s1}$) units of time the sensor sends an updated glucose value. This sensor behaviour is formally described in Figure 3 and it is part of the "world" W where the FTIP takes place.

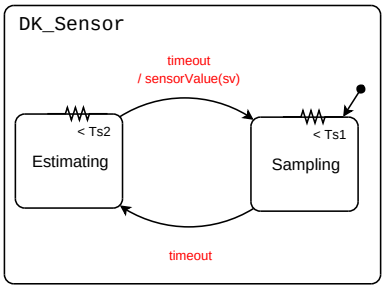

Fig. 3. Sensor behaviour description

The pump

The pump used by the FTIP control system is a fusion of two different pumps which can currently be found on the market [7,11], plus some special characteristics which have been added to allow it to work cooperatively with the sensor.

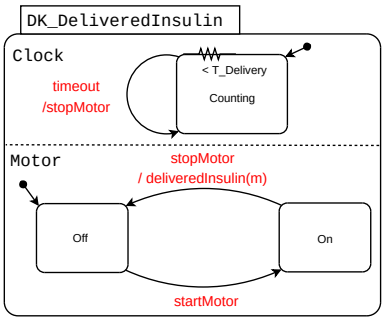

Fig. 4. Domain knowledge of the insulin delivering

The pump has an internal clock which provides the current time at any given moment. The FTIP control system starts the motor which is kept working for $T_{Delivery}$ units of time. The $T_{Delivery}$ value is defined by the control system

according to the current patient's glucose value, which is sent by the sensor, and the individual settings defined by a doctor. The pump also contains a cartridge with fast-acting insulin which is supplied to the patient's body by a cannula that lies under the skin. The motor is connected to a piston rod which sends forward a plunger in order to deliver the insulin to the body.

Therefore, the necessary amount of insulin to keep the glucose at a safe level (**deliveredInsulin(m)** on Figure 1) is delivered as a result of the work of the motor. The relationship between the times at which the motor is in operation and the amount of insulin delivered, which is part of the domain knowledge that is accepted as true to develop the FTIP control system, is formally described in Figure 4.

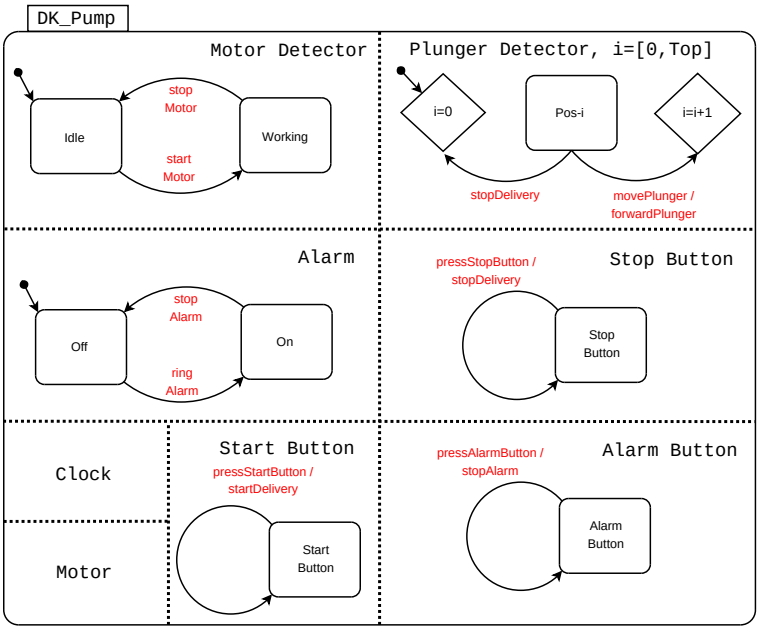

Fig. 5. Pump's elements behaviour

The plunger is built in such a way that it does not move unintentionally as a result of changes in atmospheric pressure. It moves only to its initial position (by pressing the **Stop** button) when the patient needs to fill the cartridge with fresh insulin.

The pump has also *infrared detectors* to check the correct movement of the plunger and the status of the motor. Both detectors help to determine whether the pump is delivering the insulin properly.

An alarm to alert the patient of abnormal situations is built-in to the pump. A button (**Alarm**) to allow the patient to switch it off when it rings and a button for starting the insulin delivery (**Start**) are also features of the pump.

Figure 5 shows how the physical elements which compose the pump behave. Their behaviour must be taken into consideration to specify the FTIP control system.

4.4 The specification (S)

According to WRSPM methodology aspects introduced so far, a specification S has to be provide such as, S supplemented by the world W, must satisfy the requirements R. Formally, it means:

$$W \| S \Rightarrow R \text{ , where}$$
$$W = DK\_Sensor \| DK\_Pump \| DK\_Delived \| DK\_Diabetic\_Person$$

Meeting the requirements

To start receiving the insulin doses, the patient has to press *Start* button. Once this button has been pressed, FTIP control system starts its execution, which consists of performing repetitively a set of operations. This operations are spread over three processes running in parallel and cooperating each other. These processes are *Checking, Controller* and *Delivery*.

Checking is in charge of getting values from sensor device and to provide them to *Controller* process. Then, *Controller* uses these values to define how long the pump's motor has to work. The period of time in which the motor is working, according to the domain knowledge, represents the quantity of insulin that the patient needs. Once *Controller* process has defined how long the motor has to work, it sends this information to *Delivery* process. *Delivery* process uses this value to start and control the activity of the motor.

This sequence of steps defines a cyle that is executed repetitively to keep the patient's glucose level below 110 mg/dl (the requirement). While *Delivery* process is working with the pump's motor, *Checking* process is getting fresh values from the sensor to be used in the next loop of the cycle.

Safety

Because the insulin is delivered almost continuously, any interruption in the supplying may result in serious problems to the patient. Then, it is reasonable to make checks to ensure that the devices are working properly.

FTIP control system checks that each amount of insulin that has to be delivered does not drop out of the safe range programmed. The pump's detectors and the internal clock are used by the control system to check the correct movement of the plunger, to check the status of the motor and to check if the sensor is sending the values on time. Therefore, the control system is able to detect any of the following critical conditions:

- no values have been received from the sensor for the last T_{Sensor} units of time (**E1**),
- the current patient's glucose level is out of the safe range (**E2**),

– the insulin that has to be delivered to keep the glucose in a safe level does not drop into the safe range programmed (**E3**),
– the insulin is not being delivered properly by the pump (**E4**).

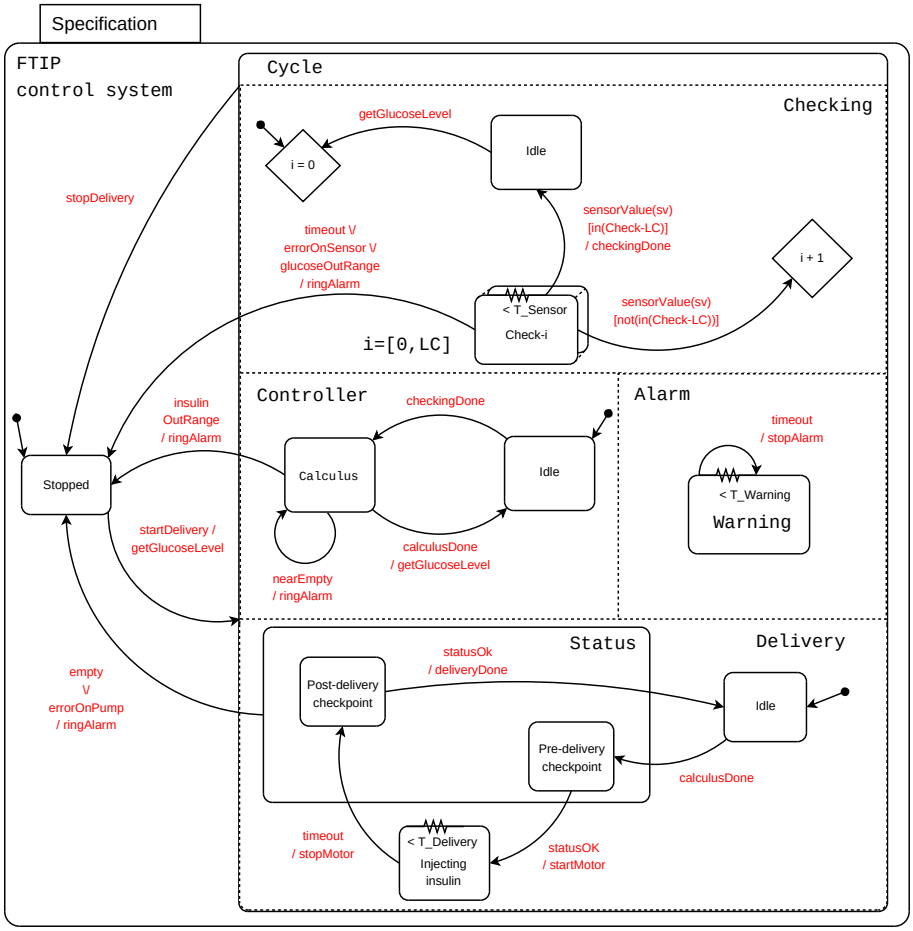

Fig. 6. FTIP control system specification.

When, at least one of these critical conditions takes place, the control system full stops the cycle execution and sounds the alarm to alert the patient about the current situation. The alarm will remain ringing till the patient switchs it off. Instead, when the quantity of insulin in the cartridge is less than the *low limit* parameter, the control system keeps executing the cycle and it rings the alarm, as a warning, for only $T_{Warning}$ units of time.

Figure 6 shows the formal description of the aspects introduced before concerning the functional requirement and safety.

For the moment nothing has been said about reliability, which is other of the atributes used to define dependability. Reliability is the ability of the system to

keep on working in spite of unexpected circumstances (e.g. **E1,E2, E3, E4**). The next Section shows a new specification of the system in terms of CAAs in order to improve the reliabity of FTIP control system.

4.5 The specification in terms of CAAs

Both the specification presented in the previous section and the CAA conceptual framework, are used to reach a new specification to satisfy the requirements despite of the presence of potential faults. The reliabilty improvements concern *E1, E3* and *E4* critical conditions, which have been introduced in the previous section.

Instead of directly stop the control system execution if one of this critical conditions takes place, error recovery and redundancy have been added to tolerate *E1, E3* and *E4* critical conditions. In the following, details of how reliabilty has been met are shown.

- A possible reason why control system is not receiving values from the sensor (*E1*), could be because of it is being affected by an electronic "noise" that surrounds the patient. Usually, this type of problem represents a transient fault for its limited duration, so there is not need to stop the system. When the control system faces a situation as described before, a numerical method based on old received values is used to make an aproximation of the current glucose level. This alterative is used for the period that the sensor is not responding (forward error recovery). Nevertheless, if the sensor does not responde beyond of a such time limit, for safety reasons, the control system must stop its execution.
- Instead of rely on only one algorithm to define $T_{Delivery}$ (units of time that the motor has to work), **N-version programming** approach is used [3]. According to [2], *the semantics of N-version programming are as might be expected N-versions of a program (N > 1) which have been independently designed to satisfy a common specification is executed and their results compared by some form of replication check. Based on a majority vote, this check can eliminate erroneous results (**E3**) (i.e. the minority) and pass on the presumably correct results calculated by the majority to the rest of the system.*
- The infrared detectors are used in a combinated way to avoid false alarms. If really there is a problem on the plunger or the motor, then both detectors must report the abnormal situation (**E4**), since both devices are physically connected. Thus, the control system does not stop if only one detector reports a problem.

The new FTIP control system specification is described as a set of CAAs (Figure 7) that interact cooperatively among them to satisfy the functional requirement as well as those concenrnig safety and reliability.

The Cycle
CAA_Cycle is the outmost CAA. It is composed of four roles, *Sensor, Controller, Pump* and *Alarm* and its main task is to perform repetitively a set of opera-

tions: (1) getting current glucose level, (2) calculate $T_{Delivery}$, and (3) insulin delivering, which were described in the previous section.

The first step is carry out by *CAA_Checking*, the second step is done by *CAA_Calculus*. The last step is performed by *CAA_Delivery*.

Sensor, Pump and *Alarm* are the roles used to manage the access to the *SensorDev, MotorDev* and *AlarmDev* devices, respectively. The *Controller* role coordinates the others roles of *CAA_Cycle* to keep on executing these steps.

Once the *Controller* role has received the information from *CAA_Calculus*, the *CAA_Checking* and *CAA_Delivery* can be performed in parallel to improve the system performance. The *Controller* role synchronizes these CAAs in a way that it is always able to have the necessary information to perform the previous described steps.

The CAAs enclosed by *CAA_Cycle* were designed using **nesting** and **composing**.

Nesting

Nesting is defined as a subset of the roles that belong to the enclosing CAA (*CAA_Cycle*). These roles (*Sensor* and *Controller*) are the same roles that have been defined for *CAA_Cycle*, but they are used to define a new CAA (*CAA_Checking*). The operations that they are doing inside *CAA_Checking* are hidden for the other roles (*Pump* and *Alarm*) as well as for the others nested or composed CAAs that belong to *CAA_Cycle*. As said, *CAA_Checking* is in charge to carry out the first step and to handle **E1** and **E2** critical conditions.

If **E1** takes place, forward error recovery is applied to try to keep the control system running (reliability). In case that **E2** arrives or forward error recovery does not succeed for handling **E1**, the delivery of insulin has to be stopped (safety). It is achieved stopping the roles belonging to *CAA_Checking* and signaling an exception to the enclosing context (which is *CAA_Cycle*). When *CAA_Cycle* detects the exception, the control system is stopped and the alarm is rung to let the patient know about the abnormal situation.

Composing

Composing is different from nesting in the sense that CAAs can be used in other contexts. A composed CAA (*CAA_Calculus* and *CAA_Delivery*) is an autonomous entity with its own roles. The internal structure of a composed CAA (i.e., set of roles, accessed external objects and behaviour of roles) is hidden from the calling CAA (*CAA_Cycle*). For instance, when the role *Controller* which belongs to *CAA_Cycle* calls to the composed *CAA_Calculus* it synchronously waits for the outcome. The calling role, *Controller* in this case, resumes its execution according to the outcome of the composed *CAA_Calculus*. If the composed CAA (*CAA_Calculus*) terminates exceptionally, the calling role (*Controller*) raises an internal exception which is, if possible, locally handled. If local handling is not possible, the exception is propagated to all the peer roles of *CAA_Cycle* for coordinated error recovery.

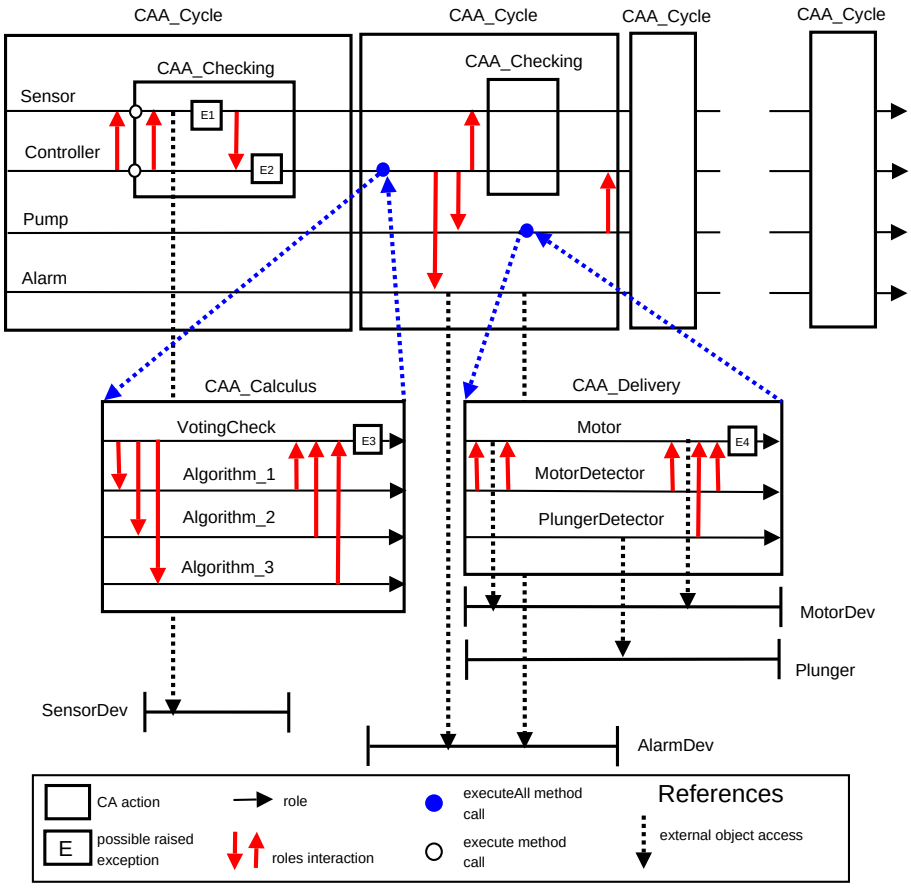

Fig. 7. CAA Design.

N-version programming

CAA_Calculus has been specially defined to calculate $T_{Delivery}$ (step two). using N-version programming. The implementation of N-version programming requires a supervisor program (known as **driver program**) that is in charge for invoking each of the versions, waiting for the versions to complete their execution and comparing and taking a decision according to the N-results received. Even if the CAA technique was not thought to implement N-version programming, its features allow implementing this programming technique (N-version) easily. In this case, the *driver program* as well as each version of the algorithm used to calculate $T_{Delivery}$ has been implemented as roles belonging to *CAA_Calculus*. These roles are *VotingCheck*, which implements the driver program, and *Algorithm_1*, *Algorithm_2*, and *Algorithm_3*. Once the result has been defined by the majority, *VotingCheck* passes it to the enclosing CAA or, if the **E3** critical condition takes place, it will signal an exception.

Delivering the insulin
The last CAA taking place in the design is *CAA_Delivery*. This CAA receives as input value the units of time that the motor of the pump has to work. This information is used by *Motor* (the role) to manipulate the motor (the device). In order to avoid confusion between the motor itself and the role that handles it, the device is named *MotorDev*. The other roles (*MotorDetector* and *PlungerDetector*) are used to check the behaviour of *MotorDev* and the movement of the *Plunger* respectively. *PlungerDetector* gets how much the *Plunger* has been displaced. This value is useful to corroborate if the motor is working properly.

If the insulin delivering is not working properly, which means that both detectors have noticed a problem on the pump (**E4** critical conditions), *CAA_Delivery* will full stop the delivery and it will signal an exception to the enclosing CAA.

4.6 Detailled Design and Implementation

This section shows how the FTIP control system is implemented using CAA-DRIP framework. Due to space limitations, the implementation of only two CAAs are shown. The full details can be found in [5].

CAA_Calculus is composed of four roles and for each of them a *Manager* (lines 2-5 on Figure 8) is defined. Once the instantiation of these objects is done, each *Role* object (lines 8-15) can be defined by instantiating a new class, which inherits from the *Role* class provided by the framework.

The name of the role, its manager and the leader manager must be given each time a new *Role* object is defined. In this case, *mgrVotingCheck* is the leader manager and it is the responsible for the coordination of the CAA.

If there is a problem in the normal execution, an alternative behaviour can be defined in order to deal with the problem. The lines 18-37 show how the exceptional behaviour can be defined. If these lines are not present, when an exception is raised, the CAA is stopped and the problem is forwarded to the enclosing context. It means that the exception is signaled to the enclosing context.

The lines 18-21 correspond to the definition of the handlers that are only executed when the exception *E3* is raised. On Figure 7, error E3 represents the places where this exception could happen. Each defined handler object is an instance of a new class derived from *Handler* class, which belongs to the framework. For each exception that should be handled by the CAA, n handlers have to be defined, where n is the number of roles defined in the CAA. Each handler must be informed of its name and its manager.

The next step is the explicit definition of the binding between the considered exception, and the handlers that have been defined to manage it. Each binding is represented by a hash-table, which is controlled by a manager (lines 24-31). Each hash-table has to be set on the manager, which is controlling the handler (lines 34-37).

Each manager (e.g. *mgrVotingCheck*) coordinates the execution of a role (e.g. *roleVotingCheck*). The role represents the normal behaviour. In the case in which an exception is launched (*E3*), each manager stops the execution of its associated role and then it starts to execute its associated handler (e.g. *hndrE3_VC*).

CAA_Cycle is executed repeatedly until the patient stops manually the delivery (by pressing the *Stop* button) or a critical condition took place and it could not be handled. The *Controller* role works as a coordinator of the tasks that have to be carried out in *CAA_Cycle*. One of these tasks is to launch the composed *CAA_Calculus* that was described earlier.

```
1  //Managers
2  mgrVotingCheck = new ManagerImpl("mgrVotingCheck","CAA_Calculus");
3  mgrAlgorithm_1 = new ManagerImpl("mgrAlgorithm_1","CAA_Calculus");
4  mgrAlgorithm_2 = new ManagerImpl("mgrAlgorithm_2","CAA_Calculus");
5  mgrAlgorithm_3 = new ManagerImpl("mgrAlgorithm_3","CAA_Calculus");
6
7  //Roles
8  roleVotingCheck =
9      new VotingCheck("roleVotingCheck",mgrVotingCheck,mgrVotingCheck);
10 roleAlgorithm_1 =
11     new Algorithm_1("roleAlgorithm_1",mgrAlgorithm_1,mgrVotingCheck);
12 roleAlgorithm_2 =
13     new Algorithm_2("roleAlgorithm_2",mgrAlgorithm_2,mgrVotingCheck);
14 roleAlgorithm_3 =
15     new Algorithm_3("roleAlgorithm_3",mgrAlgorithm_3,mgrVotingCheck);
16
17 //Handlers for E3 exception
18 hndrE3_VC = new E3_VC("hndrE3_VC",mgrVotingCheck);
19 hndrE3_A1 = new E3_A1("hndrE3_A1",mgrAlgorithm_1);
20 hndrE3_A2 = new E3_A2("hndrE3_A2",mgrAlgorithm_2);
21 hndrE3_A3 = new E3_A3("hndrE3_A3",mgrAlgorithm_3);
22
23 //Binding between the Exception and the Handlers
24 Hashtable ehVC = new Hashtable();
25 ehVC.put(E3.class,hndrE3_VC);
26 Hashtable ehA1 = new Hashtable();
27 ehA1.put(E3.class,hndrE3_A1);
28 Hashtable ehA2 = new Hashtable();
29 ehA2.put(E3.class,hndrE3_A2);
30 Hashtable ehA3 = new Hashtable();
31 ehA3.put(E3.class,hndrE3_A3);
32
33 //Setting the binding on each Manager
34 mgrVotingCheck.setExceptionAndHandlerList(ehVC);
35 mgrAlgorithm_1.setExceptionAndHandlerList(ehA1);
36 mgrAlgorithm_2.setExceptionAndHandlerList(ehA2);
37 mgrAlgorithm_3.setExceptionAndHandlerList(ehA3);
```

Fig. 8. Definition of CAA_Calculus

The *Java* code in Figure 9 shows how the *body* method of the *Controller* role is implemented for the *CAA_Cycle*. The *Controller* role works as a coordinator of the tasks to be carried out in *CAA_Cycle*. One of these tasks is to launch the composed *CAA_Calculus* that was described earlier.

The first time that *CAA_Cycle* is called (lines 7-15) the *Controller* role starts to execute *CAA_Checking* (line 10) in order to get the information provided by the sensor. Once the role has got the information, it returns the value to the enclosing context (line 15). After *CAA_Cycle* has been executed once, the enclosing context is able to provide the *sv* value, which has been taken in the previous execution of *CAA_Cycle*. Thus, *Controller* role gets the *sv* value (line 18-19) and then passes it as an input parameter (line 23) to *CAA_Calculus*. The

CAA_Calculus execution (line 24) returns the period of time (*tDelivery* value) that the motor has to be working (line 26-27).

When the *tDelivery* value is known, it has to be passed to the *Pump* role (line 29). Next, the *Pump* role receives *tDelivery* and call *CAA_Delivery* to delivery the insulin. While *CAA_Delivery* is executing, *Controller* role launches *CAA_Checking* (line 33) to get information from the sensor that will be be used in the next iteration of *CAA_Cycle*.

When the information comming from *CAA_Checking* and *Pump* roles has been received (line 35-36 and 38 respectively), and passed to the enclosing context (lines 40 and 41), *Controller* role can finish its execution and pass the control to the enclosing context where *CAA_Cycle* is embedded.

```
1  public void body(ExternalObjects eos)
2   throws Exception, RemoteException {
3    try{
4      //Getting information from the enclosing context
5      Loop loop = (Loop)eos.getExternalObject("loop");
6      if(loop.isfirst()){
7       //launching nested CAA_Checking
8       ExternalObjects checking =
9            new ExternalObjects("checking");
10      roleControllerChecking.execute(checking);
11      //getting outcome from CAA_Checking
12      SensorValue sv =
13           (SensorValue)checking.getExternalObject("sv");
14      //Sending information to the enclosing context
15      eos.setExternalObject("sv",sv);
16     }else{
17       //getting sensor value
18       SensorValue sv =
19            (SensorValue)eos.getExternalObject("sv");
20       //launching composed CAA_Calculus
21       ExternalObject calculusREOs =
22            new ExternalObjects("calculus");
23       calculus.setExternalObject("sv",sv);
24       roleVotingCheck.executeAll(calculus);
25       //getting outcome from CAA_Calculus
26       Time tDelivery =
27            (Time)calculus.getExternalObject("tDelivery");
28       //passing information to Pump role
29       pumpQueue.put(tDelivery);
30       //launching nested CAA_Checking
31       ExternalObject checking =
32            new ExternalObject("checking");
33       roleControllerChecking.execute(checking);
34       //getting outcome from CAA_Checking
35       SensorValue sv =
36            (SensorValue)checking.getExternalObject("sv");
37       //getting values from CAA_Delivery by Pump role
38       Status st = (Status)pumpQueue.get();
39       //Sending information to the enclosing context
40       eos.setExternalObject("sv",sv);
41       eos.setExternalObject("st",st);
42     }
43   }catch (Exception e) {
44     throw e; //Local handling for Controller exception;
45   }
46 }
```

Fig. 9. Body method of Controller class

5 Conclusions and Ongoing Work

In this paper a control system for a fault-tolerant insulin pump therapy has been described. In order to ensure the needed requirements of reliability, the system has been designed using the CAAs mechanism, which offers approaches for error recovery. The implementation of the control system has been made in Java, using an implementation framework called CAA-DRIP which fully supports the CAAs semantics. This work is part of the ongoing CORRECT project [5]. On the future work side, the plan is to apply the full CORRECT methodology, improving the design part and trying to automatically generate a skeleton code from the CAA design model, via transformation rules. Since it is usually impossible to generate a complete implementation, the expected result is to generate an implementation schema, with a set of classes and their methods and exceptions declaration. The role of a programmer will be then to write the body of the application methods, while the exception detection, resolution and propagation will be automatically managed by the other parts of the schema.

Acknowledgements
This work has benefited from a funding by the Luxembourg Ministry of Higher Education and Research under the project number MEN/IST/04/04. The authors gratefully acknowledge help from A. Campéas, B. Gallina, P. Periorellis, R. Razavi, A. Romanovsky and A. Zorzo.

References

1. Java 2 Platform, Standard Edition (J2SE). http://java.sun.com.
2. T. Anderson and P. Lee. Fault-tolerance: Principles and practice. *Prentice Hall*, 1981.
3. A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Sofi. Eng*, pages pp. 1491–1501, 1985.
4. A. Capozucca, N. Guelfi, P. Pelliccione, A. Romanovsky, and A. Zorzo. CAA-DRIP: a framework for implementing Coordinated Atomic Actions. *Laboratory for Advanced Software Systems Technical Report nr. TR-LASSY-06-05*, 2006.
5. Correct Web Page. http://lassy.uni.lu/correct, 2006.
6. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
7. DISETRONIC, A member of the Roche Group. www.disetronic.com.
8. Endocrine Disorders & Endocrine Surgery. *www.endocrineweb.com/insulin.html*.
9. C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Softw.*, 17(3):37–43, 2000.
10. D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.
11. Medtronic. www.minimed.com.
12. 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), February 2003.

13. B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering. IEEE Press*, SE-1(2):220–232, 1975.
14. J. Xu, B. Randell, A. Romanovsky, C. M. Rubira, R. J. stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. *Proceedings of the 25 International Symposium on Fault-Tolerant Computing*, pages 499–508, 1995.
15. J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Symposium on Fault-Tolerant Computing*, pages 499–508, 1995.
16. A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 435–446. ACM Press, 1999.

Appendix: "Designated Terminology"

The World (W)

- ***The body***
  - **L1** ≈ Lowest blood glucose concentration (in mg/dl) tolerated by a person (mg/dl means milligrams of glucose in 100 milliliters of blood) (**EC/U**).
  - **L2** ≈ Highest blood glucose concentration (in mg/dl) tolerated by a person (**EC/U**).
  - **d** ≈ positive value that represents the displacement of blood glucose concentration (**EC/U**).
  - **Meal** ≈ Carbohydrates (or sugars) which are absorbed from the intestines into the bloodstream after a meal (**EC/U**).
  - **gcDowing** ≈ the blood glucose level decreases (**EC/U**).
  - **T2** ≈ units of time required by the pancreas to produce insulin in order to decrease the detected blood glucose concentration (**EC/U**).
  - **deliveredInsulin(**m_i**)** ≈ **m** is the amount of insulin that has to be delivered to reach a normal level of glucose (**EC/U**).
  - **T3** ≈ units of time that the body's cells are absorbing glucose (**EC/U**).
- ***The Sensor***
  - **sensorValue(sv)** ≈ **sv** is the blood glucose concentration detected by the sensor (**EC/S**).
  - **Ts1** ≈ units of time that the sensor is taking blood from the body (**EC/U**).
  - **Ts2** ≈ units of time required by the sensor to define the current (**sv**) blood glucose concentration (**EC/U**).
- ***The Pump***
  - **startMotor** ≈ the motor is started (**MC/S**).
  - **stopMotor** ≈ the motor is stopped (**MC/S**).
  - **TOP** ≈ the plunger has reached its last position. It means that there is not insulin in the cartridge any more (**EC/S**).
  - **movePlunger** ≈ the plunger has moved one position forward (**EC/U**).

- **forwardPlunger** ≈ the *plunger detector* has detected that the plunger moved one position forward (**EC/S**).
- **ringAlarm** ≈ sounds the alarm (**MC/S**).
- **stopAlarm** ≈ the alarm is stopped (**MC/S**).
- **pressStartButton** ≈ the patient presses the *Start* button (**EC/U**).
- **pressStopButton** ≈ the patient presses the *Stop* button (**EC/U**).
- **pressAlarmButton** ≈ the patient presses the *Alarm* button (**EC/U**).
- **startDelivery** ≈ the patient has pressed the *Start* button (**EC/S**).
- **stopDelivery** ≈ the patient has pressed the *Stop* button (**EC/S**).
- **stopAlarm** ≈ the patient has pressed the *Alarm* button (**EC/S**).

Requirements (R)

- **T1** ≈ units of time required by the body to update the blood glucose concentration (**EC/U**).
- **levelGlucose(**v_i**)** ≈ the blood glucose concentration in the body is **vi** (**EC/U**).

Specification (S)

- **LC** ≈ number of samples to take of the current patient's glucose level (**MC/U**).
- **getGlucoseLevel** ≈ the system asks to the sensor for number of samples to take of the current patient's glucose level (**MC/U**).
- $T_{Delivery}$ ≈ units of time that the motor will be working (**MC/S**).
- T_{Sensor} ≈ waiting limit time to get the current patient's glucose concentration from the sensor (**MC/U**).
- $T_{Warning}$ ≈ units of time that the alarm will be ringing to warn the patient about low insulin level in the cartridge (**MC/S**).
- **calculusDone** ≈ the FTIP system finds out the period of time that the motor has to work in order to supply the insulin into the patient (**MC/U**).
- **statusOk** ≈ the motor and the plunger have passed the checks made by the the FTIP system (**MC/U**).
- **errorOnSensor** ≈ malfunction detection on the sensor (**EC/S**).
- **errorOnPump** ≈ malfunction detection on the pump (**EC/S**).
- **glucoseOutRange** ≈ level of glucose out of safe range (**EC/S**).
- **insulinOutRange** ≈ dose of insulin to be delivered drops out of the safe range (**MC/S**).
- **nearEmpty** ≈ cartridge almost empty (**MC/S**).
- **empty** ≈ cartridge empty (**MC/S**).