The Fault-Tolerant Insulin Pump Therapy

Alfredo Capozucca, Nicolas Guelfi and Patrizio Pelliccione

Software Engineering Competence Center
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
Luxembourg, L-1359 -Luxembourg

Abstract. In this paper we describe our experience using Coordinated Atomic Actions (CAAs) to design a control system for a medical treatment, which has high reliability requirements. The "Fault-Tolerant Insulin Pump Therapy" is based on the Continuous Subcutaneous Insulin Injection technique involving different sensors and actuators in order to enable continued execution of the treatment, as well as detect faults in it. Precisely that is the challenge raised by this example, to design a control system that maintains the delivery of insulin even in the presence of a large number and variety of hardware and software failures. The implementation of this control system has been made in Java using an extension of the DRIP framework, that ensures the reliability properties of systems designed using CAAs.

1 Introduction

Software and hardware systems have become increasingly used in many sectors, such as manufacturing, aerospace, transportation, communication, energy and healthcare. Failures due to software or hardware malfunctions and to malicious intentions can have economic consequences, but can also endanger human life. In fact, if a health care system breaks down, the effect on the hospital and patients could be huge. Therefore health care systems must be available 24 hours a day, seven days a week with no exceptions (*availability*).

Different approaches have been proposed in the literature to model medical systems. The Asynchronous Transfer Mode (ATM) network provides a robust and resilient network that is able to combine high performance with the Quality of Service (QoS), which are required by advanced and mission-critical telemedicine, and clinical applications [4, 6]. *Resilience* is the ability of systems to undergo abnormal situations without loss of its essential functions. A resilient system persists for a long time despite disturbances. More precisely, resilient systems should be able to ensure their services even when some system parts have abnormal behaviors due to degradation of the components, unavailability or attack.

In this paper we focus on Coordinated Atomic Actions (CAAs) as a design structuring concept to ensure the needed requirements of reliability and availability [9] and on the framework called Dependable Remote Interacting Processes (DRIP) [10] that embodies CAAs in terms of a set of Java classes. Although the

DRIP framework supports the complete semantics of CAA, we had to change the implementation to fix some problems. Fundamentally, we have changed the notification of an exception from a composed CAA to the enclosed context, as well as the way in that each handler must be defined and linked with its corresponding manager and the internal mechanism to execute the handlers when an exception has to be handled. The change on the exception handling mechanism does not allow us to have nested handlers any more, that is a requirement for DRIP (but not for CAAs). Thus, this new framework only supports CAAs requirements. We refer to this new framework with the name CAA-DRIP and the full implementation details about it can be found in [2].

The aim of this paper is to show how CAAs and CAA-DRIP can be successfully used for medical systems which require resilience and availability. The case study that we consider concerns a diabetes control system that is aiming at correctly delivering the insulin on a patient. The doctor suitably sets the parameters and miniaturized sensors and pumps check the patient status and administer the insulin. It is of primary importance, for the patient health, that the whole application works properly 24 hours a day without interruption.

In Section 2 we give an introduction to CAA and CAA-DRIP, and in Section 3 we show the design and the implementation of the considered case study . The paper closes with conclusions and future works.

2 Background

In this section we introduce CAAs and the requirements of the CAA-DRIP framework that are used in the following of the paper.

2.1 Coordinated Atomic Actions

Backward error recovery (based on rolling system components back to the previous correct state) and forward error recovery (which involves transforming the system components into any correct state) represent the two main approaches for error recovery. The former uses either diversely-implemented software or simple retry; the latter is usually application-specific and relies on exception handling mechanisms. Distributed transactions [3] are a well-known technique that uses backward error recovery as the main fault tolerance measure in order to satisfy completely or partially the ACID (atomicity, consistency, isolation, durability) properties. Atomic actions [1] allow programmers to apply both backward and forward error recovery.

CAAs have been proposed by Xu et al. [7] in order to combine distributed transactions and atomic actions assuring consistent access to objects in the presence of concurrency and potential faults. If an exception is raised into a CAA, then an exception handler tries to recover them. In the positive case the CAA terminates normally, on the contrary, it attempts to roll-back the state of external objects. Finally, an unsuccessful roll-back causes a failure. CAAs can be nested and in this case exceptions raised by a nested CAA are propagated to the enclosing one.

2.2 CAA-DRIP

The CAA Dependable Remote Interacting Processes (CAA-DRIP) framework comprises a set of Java classes supporting CAAs.

CAA-DRIP relies on the notion of Dependable Multiparty Interaction (DMI) [8]. The main properties of a multiparty interaction are (i) using a guard to check the preconditions to execute the interaction, hence (ii) the need for having synchronization upon entry of participants; (iii) using an assertion, after that the interaction has finished, to check that a set of post-conditions has been satisfied by the execution of the interaction; (iv) and, finally, atomicity of external data to ensure that intermediate results are not passed to the outside processes before that the interaction finishes. These properties make DMIs an excellent vehicle for implementing reliable applications. Zorzo and Stroud have proposed within CAA-DRIP a general scheme for designing DMIs in a distributed object-oriented system [10]. DMIs extend the notion of multiparty interaction to include facilities for handling exceptions, which allows dealing with failures in one or more participants of the multiparty interaction, and in particular concurrent exceptions and synchronization upon exit.

CAAs can be derived from DMIs by adopting a more restricted form of exception handling with a stronger exception handling semantics. CAA-DRIP is designed to support this derivation and thus can be used to implement CAAs.

3 The Diabetes Control System

The Diabetes Control System makes use of different kinds of devices, which combine high performance, lower power consumption, and wireless communication, increasing the "intelligence" of medical sensors and actuators.

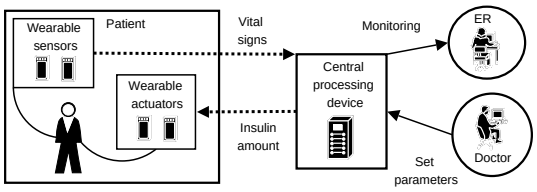

Fig. 1. The actors and their relationships.

Figure 1 shows the different actors present in our scenario. The main actor is the *patient* who is receiving the treatment and who has put on the wearable devices (sensors and actuators). The *doctor* must set the parameters for the devices to allow them to work according to the specific treatment that the patient has to receive. This information will be stored in the patient's personal record and will be consulted by the application. Moreover, the facilities for the doctor to change and consult the information about the treatment should be designed to be fault-tolerant, as well.

The last actor is the *emergency room (ER)*, where caregivers are continually monitoring the patient's vital signs. They will be the first to know if there is

a problem with the treatment that the patient is receiving. The dotted arrows represent wireless connection and show how these wearable devices are connected to the central processing device. In our representation, the doctor and the ER are connected to the network in the traditional way, but they could also be connected to the network of the hospital by wireless connection. This paper focusses on the application that controls insulin delivery to the patient.

3.1 Requirements

The fault-tolerant diabetes control system will be used to implement the technique called Continuous Subcutaneous Insulin Injection [5]. This technique uses miniaturized sensors and pumps to check the patient's status and to administrate insulin, respectively.

In this scenario, two sensors are used: one to monitor the *current blood glucose level (CBGL)* and another one to check the *heart rate (HR)*. There are also two pumps: one for injecting *long acting insulin (LAIP)* and another one for injecting *rapid acting insulin (RAIP)*.

The patient's vital signs collected by the sensors are wirelessly sent to the central processing device, which is connected to the network of the hospital. These values are used to determine the patient's status and, fundamentally, to define the *basal rate* (the amount of insulin to inject) for each pump. This is calculated by a formula that takes into account the Target Blood Glucose Level (TBGL), the Duration of Insulin Action (DIA) according to the kind of insulin used and the patient's current values measured by the sensors (CBGL and HR). The TBGL and DIA parameters must be determined by the doctor, as well as the low limit of the cartridge of each pump and the safe insulin delivery limit. These values must be defined before the treatment is launched, but if it is necessary, these values can be changed while the patient is receiving the dose, as well. More details about elements included into this control process are given in the Appendix.

The result given by the formulas represents the amount of insulin that each pump must inject to keep the blood glucose as near as possible from the patient's target blood glucose. In this way, the central processing device commands each actuator to inject the corresponding amount of insulin. The insulin will then arrive to the patient by a cannula, which is a small soft tube, inserted into the patient's body. Each actuator has also a sensor, which provides useful information about it. The application, with the help of these sensors gets information on the current status of each actuator.

When an error occurs, it must be detected and, depending on the seriousness, the control program either tries to solve it (first attempting the operation or, in second place, using the values of the previous cycle), or the control system turns on the alarm and stops the delivery of insulin. The ER personnel detects the alarm, solves the problem and then turns off the alarm.

3.2 Design

The functional requirements presented in the previous section drive the application design through the definition of seven CAAs (Figure 2). These CAAs

were designed using **nesting** and **composing**. Nesting is defined as a subset of the roles (*Params* and *Controller*) of a CAA (**CAA_Cycle**) defining a new CAA (**CAA_Checking/CAA_Executing**) inside the enclosing CAA (**CAA_Cycle**).

CAA_Cycle works like a container for the nested **CAA_Checking** and **CAA_Executing** CAAs. Its main task is to determine the amount of insulin that must be injected for each pump. These amounts of insulin are defined by the *InsulinAmount* algorithm, which is used by the *Calculus* role.

CAA_Checking and **CAA_Executing** CAAs were defined to isolate the execution of a group of tasks to determine the insulin amount as well as to deliver them on the respective pumps. **CAA_Checking** provides the input information for the algorithm used by *Calculus* and its output is passed to **CAA_Executing** which uses this information to deliver the insulin.

The "interactions" between roles are represented in Figure 2 by vertical wide solid arrows (not to be confuse with roles which are represented by horizontal thin solid arrows).

CAA_Checking has to retrieve the parameters set by the doctor for the patient and also, it has to get the values of the sensors. **CAA_Executing** sends commands to each pump and registers in a log the original commanded values and those that were really injected. Both nested CAAs use *Controller* and *Params* roles to achieve their goals (which have been explained before). The fact that the roles are embedded in two different nested CAAs, allows to hide the tasks that the roles do for the first CAA with respect to the second one.

The log is an external object, and the access to it is represented by wide slashed arrows. The patient, his personal record and the pumps are external objects, as well.

We defined four more CAAs to perform the activities corresponding to the sensors and actuators. The first one is **CAA_Sensors**, which is in direct contact with the wearable devices that have to get the patient's vital sign. The second one is **CAA_Actuators**. Both CAAs are composed.

Composed CAAs are different from the nested in the sense that the first one is an autonomous entity with its own roles and external objects. The internal structure of a composed CAA (e.g. **CAA_Sensors**), i.e. set of roles (*S_CT*, *BGC* and *HR*), accessed external objects (*Patient* and *Patient's record*) and behavior of roles, is hidden from the calling CAA (**CAA_Checking**). The *Controller* role that calls the composed **CAA_Sensors** synchronously waits for the outcome. Then, the calling role resumes its execution according to the outcome of the composed **CAA_Sensors**. If the composed **CAA_Sensors** terminates exceptionally, its calling role (which belongs to **CAA_Checking**) 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_Checking** for coordinated error recovery.

CAA_Actuators contains the composed **CAA_RAIP** and **CAA_LAIP** CAAs. Each composed CAA manages both a pump and a sensor. This sensor allows us to know the state of the pump before and after the insulin injection.

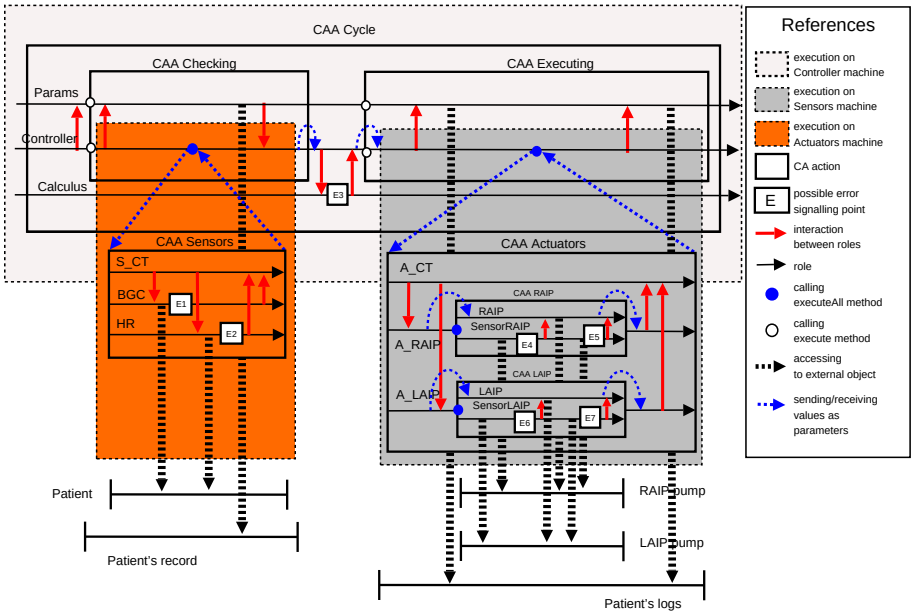

Fig. 2. The Design by CAAs and the faults that we are handling.

In **CAA_Sensors** and **CAA_Actuators** there is a special role (*S_CT* and *A_CT* respectively), who is in charge of data exchange among the roles that compose each CAA. Thus, this role receives/sends all the information from/to the enclosed/nested context. This data manipulation is done in the same way by the *RAIP* and *LAIP* roles. This way to send or receive information as parameters is represented by thin dotted arrows. As showed in Figure 2, the *Controller* machine receives information from the *Sensors* machine that commands the pumps that are running on the *Actuators* machine.

Failure definitions and analysis Before defining and analysing the different possible failures that may happen in our example, we have to state the assumptions that we have done: (i) The values of the sensors and of the actuator are always transmitted correctly, without any loss or error. (ii) Each failure on any sensor or actuator is indicated by a specific value, which shows which kind of failure happened. (iii) The alarm signalling mechanism is free of faults and does not fail.

Now, we can define and analyse various failures with respect to some elements that compose our scenario, as well as the basic requirements for handlers related to each exception that will be launched when an error is detected.

1. Sensor stops (E1 or E2): a wearable sensor could not send valid values. This failure is indicated automatically by a special value of the wearable sensor. The control system will try again getting the value to continue the cycle, but if

the problem persists the delivery will stop and the danger alarm will be turned on.
2. Delivery Limit (E3): there is an amount of insulin that should be delivered to keep the patient's target blood glucose which is dropping out of the safe range. In this case, the delivery is stopped and the danger alarm is turned on.
3. Actuator stops (E4, E6): a sensor that is monitoring an actuator has detected a problem before trying to inject the insulin. This means that the actuator is not properly working. In this case, the control program must stop the delivery of insulin and start to ring the danger alarm.
4. Delivery stops (E5, E7): a sensor that is monitoring an actuator has detected a problem after the insulin injection. It means that the actuator could not inject the required amount. The control program will try again to deliver the insulin, but if the problem goes on, the delivery of insulin will be stopped and the danger alarm will be turned on.
5. Cartridge very low (E4, E5, E6 or E7): the quantity of insulin in a cartridge is less than the low limit set in the cartridge. The basal delivery continues, but the warning alarm is turned on.
6. Cartridge empty (E4, E5, E6 or E7): a cartridge of a pump does not have any more insulin, thus the systems will be stopped and the danger alarm is turned on.

3.3 Implementation

This section describes the most important changes we made on DRIP [10] and how we used this new framework to implement our design. Due to space limitations, we just show the implementation of **CAA_Sensors** and how it is launched. Using this example, we give some ideas on the extensions made on DRIP. For more details, interested reader can refer to [2]. This CAA is composed by three roles and for each one of them we define a *Manager* (lines 2-4). Once the instantiation of these objects is done, we are able to define each *Role* object (lines 7-9) by instantiating a new class, which inherits from the *Role* class provided by the framework. We must give the name of the role, its manager and the leader manager each time that we define a new *Role* object. In this case, *mgrCT* is the leader manager and it is the responsible for the coordination of the each role when they must be executed, as well as, when an exception is raised.

Definition of CAA_Sensors

```
1  //Managers
2  mgrCT = new ManagerImpl("mgrCT","CAA_Sensors");
3  mgrCBGC = new ManagerImpl("mgrCBGC","CAA_Sensors");
4  mgrHR = new ManagerImpl("mgrHR","CAA_Sensors");
5
6  //Roles
7  roleCT = new CT("roleCT",mgrCT,mgrCT);
8  roleCBGC = new CBGC("roleCBGC",mgrCBGC,mgrCT);
9  roleHR = new HR("roleHR",mgrHR,mgrCT);
10
11 //Handlers for SensorStops exception
12 hndrSS_CT = new SensorStopsCT("hndrSS_CT",mgrCT,mgrCT);
13 hndrSS_CBGC = new SensorStopsCBGC("hndrSS_CBGC",mgrCBGC,mgrCT);
14 hndrSS_HR = new SensorStopsHR("hndrSS_HR",mgrHR,mgrCT);
15
```

```
16  //Binding between the Exception and the Handlers
17  Hashtable ehCT = new Hashtable();
18  ehCT.put(SensorStops.class,hndrSS_CT);
19  Hashtable ehCBGC = new Hashtable();
20  ehCBGC.put(SensorStops.class,hndrSS_CBGC);
21  Hashtable ehHR = new Hashtable();
22  ehHR.put(SensorStops.class,hndrSS_HR);
23
24  //Setting the binding on each Manager
25  mgrCT.setExceptionAndHandlerList(ehCT);
26  mgrCBGC.setExceptionAndHandlerList(ehCBGC);
27  mgrHR.setExceptionAndHandlerList(ehHR);
```

If there is a problem in the normal execution, we have the chance to define an alternative behaviour. The lines 11-27 show how we can define this exceptional behavior. If these lines are not present, when an exception is raised, the CAA is stopped and the problem is forwarded to the enclosed context.

The lines 12-14 correspond to the definition of the handlers that are only executed when the exception *SensorsStops* is raised. On Figure 2 the errors E1 and E2 represent the places where this exception could happen. Each handler object defined is an instance of a new class derived from *Handler* class, which belongs to the framework. The class *Handler* has been introduced in CAA-DRIP to correctly manage the information context of the CAA where the exception has been raised [2]. For each exception that we want to handle in the CAA, we have to define n handlers, where n is the number of roles defined in the CAA. Each handler must be informed of its name, its manager (which must be one of the used in the definition of the roles) and the leader manager (not necessary the same used for the roles).

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 hashtable, which is controlled by a manager (lines 17-22). Each manager (e.g. *mgrCT*) coordinates the execution of a role (e.g. *roleCT*). The role represents the normal behavior. In the case in which an exception is launched (*SensorStops*), each manager stops the execution of its associated role it starts to execute its associated handler (e.g. *hndrSS_CT*). Finally, we must set each hashtable on the corresponding handler that is managing each role and handler (lines 25-27).

The composed **CAA_Sensors** is launched from the *Controller* role. The definition of a role implies the *Role* class extension, which belongs to the framework and reimplement its *body* method. Inside this method we define the tasks that must be executed to achieve the requirements of the considered role. The following Java source code corresponds to the role *Controller* and shows how **CAA_Sensors** is called, as well as the interaction with the *Params* role and with **CAA_Sensors** CAA is achieved.

Launching CAA_Sensors

```
1
2  public void body(Object list[]) throws Exception, RemoteException {
3   try{
4          //launching the Composed CAA_Sensors
5          roleCT.executeAll(list);
6
```

```
7           //getting Composed CAA_Sensors outcomes
8           RemoteQueue rqOut = (RemoteQueue) list[0];
9           Integer bgcValue = (Integer)rqOut.get();
10          Integer hrValue = (Integer)rqOut.get();
11
12          //getting values from Params role
13          RecordPatient rp = (RecordPatient)paramsPatientQueue.get();
14
15          //passing information to CAA_Cycle
16          rqOut.put(bgcValue);
17          rqOut.put(hrValue);
18          rqOut.put(rp);
19
20   } catch (Exception e) {
21          //Local handling for Checking.Controller exception;
22          throw e;
23  }
```

The *body* method receives a list of objects as parameter (line 2), which is used to exchange information with its context. The *executeAll* method is used for the *roleCT* object (there is no difference about which CAA role is used) to launch the composed **CAA_Sensors** (line 5). This method takes an object list as parameter that is used to get the **CAA_Sensors** outcomes. Lines 8-10 show how we retrieve these outcomes from the list. Interaction among roles appears in line 13 and it represents an information flow from *Params* to *Controller*. Once the *Controller* role has all the patient's information it must send this information to the enclosing **CAA_Checking** (lines 16-18). If along the execution of these tasks an exception is raised, we have the chance to handler it locally inside the *catch* block. In this example, if an exception happens, it is directly passed to the enclosing context (line 22).

4 Conclusions and Future Work

In this experience paper we introduced a control system for a fault-tolerant insulin pump therapy. In order to ensure the needed requirements of reliability and availability, the system has been designed using the CAAs mechanism that offers approaches for error recovery. The implementation of the control system has been made in Java, using a variant of the DRIP framework, because along the implementation of this case study we found some problems in the original DRIP. These problems were fixed in a new framework which just supports CAA requirements and is called CAA-DRIP. On the future work side we plan to release CAA-DRIP explaining details on changes made.

Acknowledgments 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. Romanovsky, P. Periorellis, R. Razavi and A. Zorzo.

References

1. R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering. IEEE Press*, pages pp. 811–826, 1986.

2. Correct Web Page. http://se2c.uni.lu/tiki/tiki-index.php?page=correctdoc, 2005.
3. J. Gray and A. Reuter. Transaction processing: Concepts and techniques. *The Morgan Kaufmann series in data management*, pages pp. 36–37, 1993.
4. P. Jain, S. Widoff, and D. C. Schmidt. The design and performance of medjava. *IEE/BCS Distributed Systems Engineering Journal*, December 1998.
5. National Institute for Clinical Excellence. Guidance on the use of continuous subcutaneous insulin infusion for diabetes. $http://www.nice.org.uk/pdf/57\_insulin\_pumps\_fullguidance.pdf$. 2003.
6. G. Weiss. Welcome to the (almost) digital hospital. *IEEE Spectrum Online,* $http://www.ieeta.pt/sias/courses/imt/Resources/IEEE\_AlmostDigitalHospital\_Mar2002.pdf$, 2002.
7. 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.
8. A. Zorzo. Multiparty interactions in dependable distributed systems. *PhD Thesis, University of Newcastle upon Tyne, Newcastle upon Tyne, UK*, 1999.
9. A. Zorzo, A. Romanovsky, B. R. J. Xu, R. Stroud, and I. Welch. Using co-ordinated atomic actions to design complex safety-critical systems: The production cell case study. *Software-Practice and Experience*, pages pp. 667–697, 1999.
10. 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: "Terminology"

Basal rate: the amount of insulin delivered over 24 hours per day, providing a background of insulin at all times. The rate programmed is intended to keep the blood glucose within the user's Target Range (TR) between meals and overnight. The basal rate is measured in units per hour (u/hr).

Blood Glucose Level (BGL): the amount of glucose in the blood. BG levels average 100 mg/dl (5.5 mmol/L) for someone without diabetes. The healthcare provider help in determining the "target range" for the blood glucose level.

Heart Rate (HR): is the number of contractions of the heart in one minute. It is measured in beats per minute (bpm). When resting, the adult human heart beats at about 70 bpm (males) and 75 bpm (females), but this rate varies between people.

Duration of insulin action (DIA): a certain amount of time insulin is active and available in the body after it has been given by a subcutaneous bolus. Talking with the healthcare provider helps in determining the duration of the insulin action through blood glucose testing.

InsulinAmount algorithm: it is used to calculate the needed amount of insulin. The formula takes into account the Target Blood Glucose Level (TBGL), the Duration of Insulin Action (DIA) according to the type of insulin used, the current blood glucose (CBGL) and the current heart rate (HR). The result represents the needed amount of insulin.

$$InsulinAmount(TBGL, DIA, CBGL, HR)$$