On the Integration of Mobility in a Fault-Tolerant e-Health Web Information System

Florencia Balbastro[⋆†]
Department of Computer Science[⋆]
Facultad de Ciencias Exactas,
Ingeniería y Agrimensura
Av. Pellegrini 250,
(2000) Rosario, Santa Fe, ARGENTINA
florencia.balbastro@uni.lu

Alfredo Capozucca[†] Nicolas Guelfi[†]
Laboratory for Advanced Software Systems[†]
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg-Kirchberg,
LUXEMBOURG
{alfredo.capozucca,nicolas.guelfi}@uni.lu

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, dependability is a key notion. This paper presents, through a case study, how we can develop 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). In order to implement our design, we explain how to use a development framework that we have made to implement CAA, which originally was not tailored for mobile fault-tolerant applications. Thus, in this paper, we also explain how we have adapted and used CAA-DRIP for mobile devices.

1 Introduction

Dependability is the ability to deliver service that can justifiably be trusted [13]. Its main goal is to avoid service failures that are more frequent and more severe than is acceptable to the user. The attributes associated to dependability are reliability, availability, confidentiality, integrity, safety and maintainability, which, depending on the application, may be emphasised or not. The means to achieve dependability are fault prevention, fault removal, fault tolerance and fault forecasting. We focus on fault tolerance as the method for obtaining dependability. The main goal of fault tolerance is to help the software system to deliver its 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. Nevertheless, fault tolerance techniques do not provide explicit protection against faults introduced in the requirement specification phase.

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.

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. Mobile and web based e-health software systems may ensure functionalities having a certain level of safety-criticalness (but this level must be coherent with the level of dependability we want to satisfy). Thus we are facing the engineering of complex distributed systems that rely on the web and that may include mobile devices. We need to use techniques which come with special features in order to ensure dependability in complex environments (advanced frameworks). The Coordinated Atomic Action (CAA) [15] is an advanced con-

ceptual 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. This advanced conceptual framework has been already successfully used in several case studies [3, 5, 8].

In order to efficiently master the development of such complex applications providing dependability, we need to use adapted development tools. As shown in [13] a good strategy at implementation level is to make use of advanced frameworks for fault tolerance. Concerning CAA, CAA-DRIP [2] 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.

The e-health WIS studied in this paper is the Fault-Tolerant Insulin Pump Therapy, which implements the Continuous subcutaneous insulin infusion [12] (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. This e-health WIS is composed of two software subsystems; one is the FTIP administration system (FTIP-AS) and the other is the FTIP control system (FTIP-CS). 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 [1] 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/her blood sugar levels.

The present 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

2.1 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. Moreover, developers have to deal with the stricter limitations of the mobile devices, such as small processors, memories, storage, and a reduced graphical user interface. New developing tools have been already built and need to continuously evolve due to the constants changes of the new technologies.

In our case study the chosen technology was Java [10] due to several reasons: object oriented, platform independent, multithreaded aspects, deployed in billions of devices around the world, deals efficiently with security issues, and keeps on developing while new advances are coming. Java 2 Micro Edition (J2ME) [9] 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.

2.2 Coordinated Atomic Actions (CAAs)

Coordinated Atomic Action (CAA) is a fault-tolerant (FT) 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 [14] is a FT 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). Transactions [6] are used to deal with competitive concurrency on (external) objects that have been designed and implemented separately from the applications that make use of them.

A CAA starts when all its roles have been activated and they meet a pre-condition (see Figure 1). 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 [7].

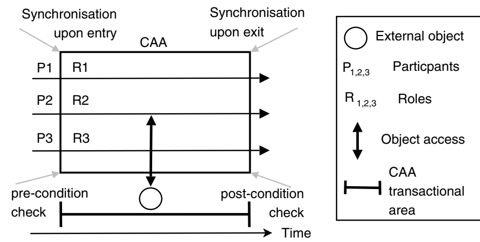

Figure 1. A simple CAA

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 [4] are different from nested CAAs in the sense that the use of composite CAAs is more flexible. 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 calling role resumes its execution according to the outcome of the composite CAA.

2.3 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 2.2. 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 include the code that the CAA has to execute to achieve its main goal (normal behaviour) or the code that is executed to recover the CAA execution from an unexpected situation (abnormal behaviour). The normal behaviour is defined by extending Role class and the abnormal behaviour 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 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 the *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 made by creating a new class that extends the *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, so that for each role, handler and compensator object, a manager object has to be defined. A CAA consists of 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.

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 possbile in term of resource consumption during execution. Since CAA-DRIP [2] 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 transfer 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.

3 The FT e-health Insulin Pump WIS

As previously mentioned, the e-health WIS under consideration is composed by the FTIP-AS and the FTIP-CS. A doctor, through the FTIP-AS, 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-, blood glucose level -BGL- safe range and insulin doses safe range).

Patients do not need to go to hospitals to be checked or received treatment because of the diabetes, neither. The FTIP-CS is embedded in a mobile device (e.g. PDA or smart phone). So that, patients can go on with their normal routine without being locked into rigid activities schedules. The FTIP-CS 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-CS, 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 insulin dose delivered by the pump are stored in a data base to allow the doctor to monitor the treatment. Both the logs and the configuration parameters of the patient's treatment are kept in the same data base (see Figure 2).

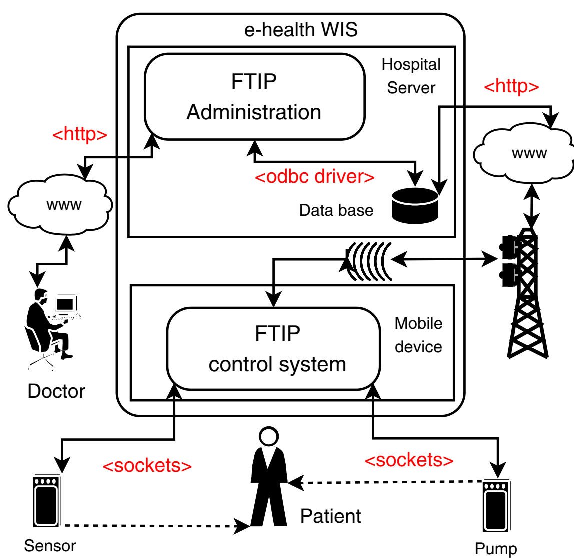

Figure 2. e-health WIS infrastructure

The safety-criticalness level of the overall software system, and in particular of the FTIP-CS is low due to the 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 it was stolen. Hence, the patient will always have time (even in the worst scenario) to restart the FTIP-CS without compromising his/her 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/her blood sugar level. However, this aspect is not part of the requirements of the FTIP-CS.

3.1 The FTIP-CS requirements

The task of the FTIP-CS is to deliver (over 24 hours per day) the right amount of insulin to keep the patient's BGL closer to the TBGL set, and inside the BGL safe range set for the patient.

The FTIP-CS uses a miniaturised sensor and pump, and a data base to check the patient's status, to decide and to administrate the dose of insulin to inject. The patient's BGL measured by the sensor is wirelessly sent to the FTIP-CS, which runs on a mobile device. Then, the FTIP-CS calculates the amount of insulin that has to be injected, taking into account the measured value of BGL, and some configuration treatment parameters (fixed by the doctor through the FTIP-AS) such as the TBGL, minimum and maximum BGL, and maximum dose of insulin. These parameters are retrieved from the remote data base. The result given by the calculation represents the units of time that the pump should work. The FTIP-CS sends wirelessly this parameter to the pump. The pump has two detectors for checking its correct behaviour. One is the plunger detector, which gets how much the plunger has been displaced. The other one is the motor detector, which corroborates if the pump's motor is working properly.

Thus, the FTIP-CS permanently executes a cycle which consists of: **(1)** retrieving from the data base the parameters set for the patient; **(2)** measuring the BGL; **(3)** calculating the pump working time (msec) based on the values taken on steps (1) and (2); **(4)** sending to the pump the working time; and **(5)** registering the data of steps (2), (3) and (4) in the data base.

When a failure occurs, the FTIP-CS must detect it and, depending on its seriousness, it will be tolerated (best case), or the delivery of insulin will be stopped and the alarm turned on (safe stop). The FTIP-CS is able to detect any of the following critical conditions: **(E1)** no values have been received from the sensor for the last *Tsensor* units of time; **(E2)** the BGL is out of safe range; **(E3)** the insulin to be delivered does not drop into the safe range programmed; **(E4)** the insulin is not being delivered properly by the pump; **(E5)** cannot establish a connection with the data base; and **(E6)** the insulin remaining in the cartridge is less than the low limit parameter.

When at least one of these critical conditions take place (except E6), the FTIP-CS stops the delivery and sounds the alarm to alert the patient about the current situation. The alarm will remain ringing until the patient switches it off. However, when the quantity of insulin in the cartridge is less than the low limit parameter (E6), the FTIP-CS rings the alarm in a warning mode for *Twarning* units of time (msec) only.

3.2 The FTIP-CS design, implementation and simulation

The FTIP-CS is designed by an outmost CAA, called CAA_Cycle, which is executed repeatedly (see Figure 3). This CAA encloses the interactions of four roles: Sensor,

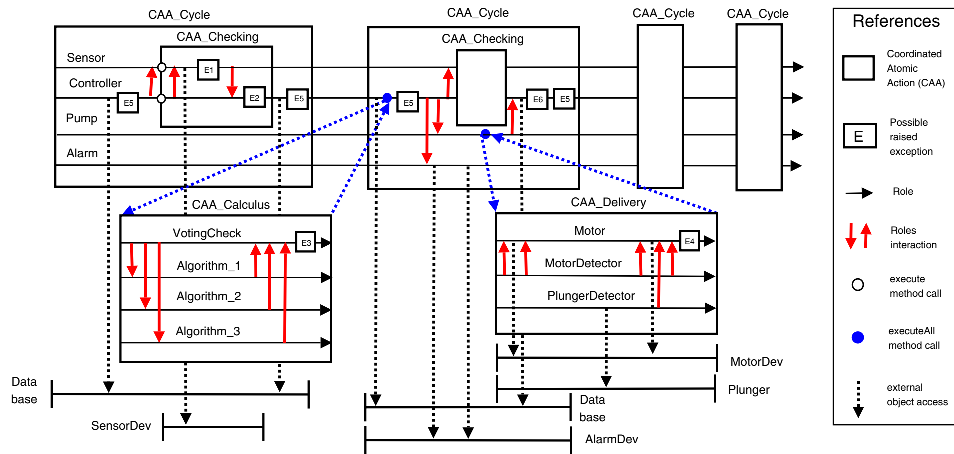

Figure 3. Design of the FTIP-CS in terms of CAAs

Controller, Pump and Alarm. The step (2) is carried out by a nested call to the CAA_Checking by the Sensor and Controller roles. Steps (3) and (4) are performed by a composed call to CAA_Calculus, by the Controller and a composed call to CAA_Delivery, by the Pump role, respectively.

To increase the dependability of FTIP-CS even more, the N-version programming technique was used to calculate the units of time that the pump has to work. Thus, CAA_Calculus has a role called VotingCheck, which acts as a supervisor. This role then: invokes three different versions of the calculation algorithm (there is a role for each of them), waits for them to complete their execution, compares the results, takes a decision by the majority and passes this to the enclosing CAA.

The last CAA taking place in the design is the CAA_Delivery. It receives the units of time that the motor of the pump has to work. This information is used by the role Motor to manipulate the device. The other two roles that act in the delivery, MotorDetector and PlungerDetector, are used to get feedback from the pump's detectors. The FTIP-CS exploits this feedback as redundant information to realise the real current pump's status (there is a problem only if both detectors have noticed a problem) and, therefore, increasing the dependability of the system avoiding false stops.

As said before, the implementation phase was carried out making use of a subset of the CAA-DRIP framework excluding non mandatory parts. As its predecessor, this reduced version of the framework allowed us to define and implement the above described CAAs easily.

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

For testing the FTIP-CS (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.), we needed to integrate a real sensor and pump. As such type of pump does not exist in the market at the writing moment (although companies are not far from it [11], and sensors are already being commercialised) we had to simulate it. So that, both the sensor and pump were simulated by small *Java* applications running on mobile devices (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).

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. We decided to use sockets because we wanted to be as low as possible in communication implementation level in order to reduce further implementation 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-CS. The pump was simulated in a similar way, representing the motor and the two detectors as three servers in charge of receiving the value of the units of time (the first of them), and giving the status of the pump (the last two servers).

The previously mentioned configuration of mobile devices was used as testbed to evaluate the dependability of the FTIP-CS system. The main aim was to check the tolerance of the FTIP-CS for the critical conditions it can face

($E_{1..6}$), including its error detection and recovery mechanisms. Notice that the detection and part of the recovery (only the policy to deal with the fault had to be implemented) mechanisms are built-in CAA-DRIP features, therefore they were brought for free to the FTIP-CS. Moreover, these mechanisms were already tested during the framework development process.

Test sets for the FTIP-CS were defined according to the critical condition that should be tolerated. Thus, we got five test sets ($TestSet_{1..5}$). $TestSet_1$ has one test case which represents the scenario where the sensor is not working (critical condition E1), while $TestSet_2$ has one test case which allows the sensor to send a BGL value which is out of the safe range (critical condition E2). $TestSet_3$ has six test cases: one for each possible voting result get by *VotingCheck* in CAA_Calculus. Three of these test cases generate the critical condition E3, while the others are useful to check the critical condition E6. $TestSet_4$ has four test cases (one for every possible pump's detectors behaviour). Only one of these test cases (when both detectors report a failure) generate the critical condition E4, while the others allow testing the critical condition E6. $TestSet_5$ has one test case, which generates the critical condition E5 (the database is unreachable). Therefore, these fourteen test cases allowed us to check the FTIP-CS behaviour facing the critical conditions $E_{1..6}$.

4 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 received 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 resource optimisation reasons. 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 (e.g. model checking) could be used and adapted to this application domain to verify some dependability properties.

Acknowledgement. We would like to thank Adriana Cultrone, Cédric Pruski and the anonymous referees for their useful comments and suggestions. This work was supported by the University of Luxembourg (MEN/IST/04/04).

References

[1] 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)*, pages 59–79, 2006.

[2] 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 2006)*, 7-10 November 2006 - Raleigh, North Carolina, USA.

[3] A. Romanovsky, P. Periorellis and A.F. Zorzo. On Structuring Integrated Web Applications for Fault Tolerance. In *Proceedings of ISADS 2003*, pages 99–106, April 2003.

[4] 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*.

[5] G. Di Marzo Serugendo, N. Guelfi, A. Romanovsky, A. Zorzo. Formal Development and Validation of Java Dependable Distributed Systems. In *Proceedings of ICECCS'99*, pages 98–108, October 1999.

[6] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. *The Morgan Kaufmann series in data management*, pages 36–37.

[7] J. Xu, A. Romanovsky and B. Randell. Concurrent Exception Handling and Resolution in Distributed Object Systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(10):1019–1032, 2000.

[8] 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):164–179, 2002.

[9] Java ME Platform. http://java.sun.com/javame/index.jsp.

[10] Java Technology. http://java.sun.com.

[11] Medtronic, Inc. http://www.minimed.com.

[12] National Institute for Health and Clinical Excellence. Guidance on the use of continuous subcutaneous insulin infusion for diabetes. (Technology Appraisal 57), February 2003.

[13] 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.

[14] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering. IEEE Press*, SE-1(2):220–232, 1975.

[15] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. Coordinated Atomic Actions: from Concept to Implementation. Technical Report 595, 1997.