Journal of Applied Computing Research, 2(1):11-21
January-June 2012

CraftContext: A Test Platform for Context-Aware Applications

Caroline Rizzi Raymundo, Patrícia Dockhorn Costa
Universidade Federal do Espírito Santo. Av. Fernando Ferrari, 514, Goiabeiras, 29075-910,Vitória, ES, Brazil
carol.rizziray@gmail.com, pdcosta@inf.ufes.br

Abstract. This paper presents a tool, called *CraftContext*, capable of leveraging the test phase of context-aware application development. *CraftContext* offers a 3D simulation environment, which is rich in details and resources, and is accessed by a robust and portable CORBA-based API. *CraftContext* excels most current existing testing tools due to its adaptability to different domains.

Key words: context-awareness, test tool, CraftContext, JacORB, CORBA, minecraft.

Introduction

The increasing popularity of smartphones boosted the search for Context-Aware Application development techniques (e.g., Dey, 2001; Costa, 2007). Such applications (called here CA applications) are capable of autonomously providing services and information based on the user's *context*. A simple example would be an application that autonomously controls the air conditioner of a house based on the user's location (e.g., if the user is approaching the home, the air conditioner is automatically turned on).

In order to gather context information, these applications usually make use of sensors, such as the ones found on modern mobiles (GPS, accelerometer, luminosity, etc.). A frequent problem faced by CA application developers is the lack of a robust and thorough simulation environment. Shah *et al*. (2010) discuss several points about the challenges of testing context-aware applications, especially with respect to the hard task of using physical context sources (e.g., sensors) in a controlled and reproducible way. For instance, consider an application that monitors the velocity of a vehicle and the environment around it. The idea is detecting risky situations, such as the ones in which cars are approaching sharp curves or paths of intense pedestrian traffic. When the application detects that the vehicle is at a high speed in a risky situation, an alert is sent to the driver. In order to test such an application by means of real sensors, the test engineer needs to place himself or herself in that risky situation. A way of overcoming such problem is the virtual simulation of the user's context.

Context simulation tools, such as those presented in Bylund and Espinoza (2002), Broens and van Halteren (2006) and Martin and Nurmi (2006), aim at generating context information data and performing the role of sensors, providing CA applications with the generated data. These tools usually provide an environment that simulates the real world. They depend on the existence of virtual agents to generate context, based on their actions in the virtual world. Since these simulation tools are usually of general purpose, they tend to provide support for common sensors only (such as position, velocity, temperature, luminosity, etc.), which are easily found in smartphones and other mobile devices. This limitation is noticeable and problematic when the CA application requires context information of a specific domain, which goes beyond the resources of most common sensors. Some examples of unusual sensors are the ones that detect vital signs (heart beats, respiratory rate, etc.), weight, depressurization, etc.

This paper proposes a context simulation tool that offers a large set of simulated sensors. This tool, called *CraftContext*, uses the game called Minecraft (Mojang, 2012a) to simulate the real world and generate context information data. Due to the wide variety of events and resources offered by Minecraft, several real world situations and scenarios can be simulated or mapped for particular features of the game. For example, a sensor inside a vending machine, which aims at detecting when the machine is empty (an example of a very specific domain), could be simulated by a game component called "chest". When there are no items left, the chest triggers an "empty" event notification. The main idea is that CA application developers can creatively use the extensive range of possibilities in the game to simulate any sensors they may need in the test phase.

CraftContext also allows callback functions that can be invoked to modify Minecraft's world, retrieve information or send messages to players. The commands and requests sent by the players can also be listened to by CA applications.

In the next section we present related work, which critically discusses solutions that are similar to the one described here. Then we present the proposed tool, *CraftContext*, as well as the Minecraft game. In the fourth section we discuss a test scenario implementation in order to demonstrate the tool in use. Finally, the final section presents concluding remarks.

Related Work

As recognized by related work (e.g., Dey *et al*., 2001; Martin and Nurmi, 2006; Broens and van Halteren, 2006; Bylund and Espinoza, 2002), testing CA application is a challenging task. Sama *et al*. (2008) point out intrinsic features of CA applications that contribute to increasing the complexity of the validation phase. According to Sama *et al*. (2008) context information may be classified as physical context (real world context), sensed context (data coming from sensors), inferred context (premises inferred from sensory context) and presumed context (conclusion based on the inferred data). Any glitch in the data flow between the various context levels may cause transient inconsistency in the system, leading to responses that are in disagreement with reality. In order to avoid this and other possible complications passing unnoticed through the validation step, it is important to use a robust environment that supports adequate evaluation.

As mentioned previously, using real context sources (i.e., sensor's data) to perform tests is hard. This difficulty occurs mainly due to the lack of control over real world events. Such events usually generate ever-changing data, reducing the chances of accurately reproducing a specific test scenario. In addition, other issues that may impose challenges when using real devices are (i) financial issues (the purchase of expensive devices or in large quantities), (ii) dealing with risky situations (when the test engineer needs to place himself or herself in a dangerous situation), (iii) logistics issues (when it is necessary to travel long distances, to wait for long periods of time, etc.).

Dey *et al*. (2001) propose a toolkit that works like an intermediary between sensors and context-aware applications. This toolkit consists of context widgets, which are software components structured in a distributed architecture. Each component communicates with a specific sensor, aiming at hiding details about context information acquisition from the CA application. The toolkit proposed in Dey *et al*. (2001) eases the communication complexity between application and sensor. However, by using real sensors in the test phase, this tool does not solve the problem of the lack of a controlled environment where the test might be easily reproduced.

In order to overcome this problem, other authors (e.g., Martin and Nurmi, 2006; Broens and van Halteren, 2006; Bylund and Espinoza, 2002) have proposed using virtual simulation of sensors to generate context information data. In this way, context information data can be obtained by context simulating applications, which can be configured to provide information according to the test engineer's interests and needs. Therefore, physical and/or financial issues would no longer be a limitation.

Martin and Nurmi (2006) propose a tool for simulating a semi-real environment. This environment is built based on models that follow a well-defined pattern. In order to use this tool, the developer needs to provide an agents' model, a world model and a context model, which must be previously defined. In this approach, both the agents' model and the world's models need to be preprogrammed. However, there is no specific tool for this end, which delegates this hard task to the test engineer.

The tool proposed by Broens and van Halteren (2006), called *SimuContext*, only requires

the pre-configuration of context values and their related entities. Differently from the approach proposed by Martin and Nurmi (2006), SimuContext offers an appropriate tool for pre-configuring context values. The context possibilities are vast, since SimuContext does not restrict kinds or forms of context, leaving the programmer in charge. However, SimuContext does not support the simulation of a virtual world, which reduces the tool to a pre-programmed context data generator.

A tool similar to the one proposed here has been presented in Bylund and Espinoza (2002). This tool, called *QuakeSim*, is based on the *Quake III Arena* game to simulate the real world. The players may establish connections to the same server and interact with each other, producing context information data. This data is gathered by QuakeSim, which sends them to the client CA application, performing the role of sensors. The richness of the game, however, is narrowly explored, so only common sensors (position, temperature, velocity, etc.) are simulated by the tool.

CraftContext differs from the other tools presented in this section mainly due to the richness of the game it has been built upon. In addition, since *CraftContext* is a first person game style, it allows test engineers to put themselves in the user's view, giving a real feel to the simulation. Additionally, *CraftContext* allows CA applications to be implemented in almost any programming language and platform, as presented in the next section.

CraftContext

We have developed *CraftContext* using the Minecraft game to simulate the world and generate context information. In this section we discuss our definition of context, the main features of Minecraft and the architectural design of *CraftContext*.

Context

CraftContext works as a context source, i.e., it is capable of providing *context*. In the literature, context appears under many definitions, and Dey (2001) presents the most cited one. According to Dey, "context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves".

CraftContext follows a similar definition of context, in which the entities' roles described by Dey are performed by the players, environment, objects and all the creatures inside Minecraft's world. Therefore, in *CraftContext*, context is any event of interest around the players (including their actions), characterizing their current situation.

The Minecraft Game

Minecraft (Mojang, 2012a) is a game played in first person, developed by Mojang AB (Mojang, 2012b). It simulates a 3D virtual world composed basically by blocks, in which nature elements are represented by cubes of one square meter, creating landscapes that vaguely resemble the *LEGO* toy. These blocks may be removed and replaced almost anywhere, and can also be used as raw material for constructions.

The game's sandbox style provides the players with freedom of action and movement inside the world, with no obligation of reaching a specific goal. Minecraft also allows interaction between several players inside the same environment (multiplayer game), enriching the variety of context possibilities. The game has a vast territory, equivalent to eight times the length of the earth surface, and offers many ecosystems, such as jungles, deserts, rivers and seas.

We have chosen Minecraft as the base of *CraftContext* mainly due to its large variety of environments, resources and possibilities, many of which faithfully represent the real world. Furthermore, we have also taken into account that Minecraft offers total freedom to the players' actions and movements, which is a feature that is hardly found in other *Role Playing Games*. Finally, a decisive factor for choosing Minecraft is that it makes it easier to develop custom extensions to the game.

The Architctural Design of *CraftContext*

We have developed *CraftContext* as an extension plugin of Minecraft, which is capable of interacting with Minecraft's world to either change it or listen to its events. Since the official Minecraft server does not support plugin creation, we have used the *Bukkit* server (Bukkit Team, 2012), which is the most famous modified version of Minecraft's original server.

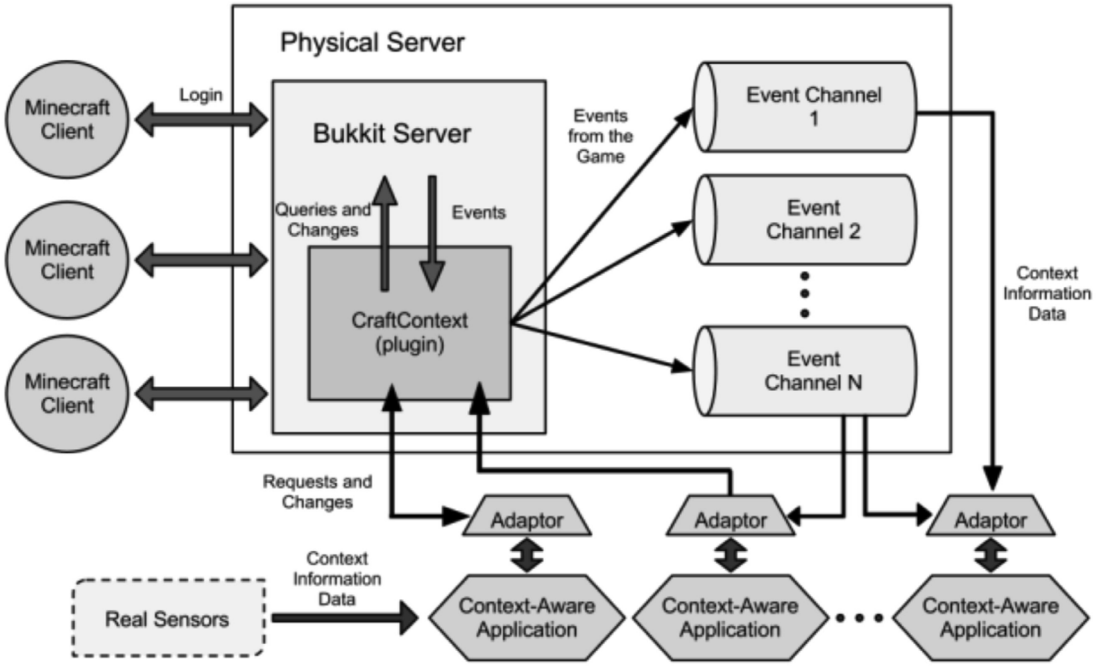

Figure 1. The *CraftContext* architecture.

CraftContext has been implemented in Java and all the remote communication between client applications and the plugin is performed with JacORB (JacORb, 2012), which is an open-source Java implementation of CORBA (Common Object Request Broker Architecture) (OMG, 2012). CORBA is a standard for developing heterogeneous distributed systems and provides, among other services, remote communication transparency, allowing the programmer to perform remote calls as if they were local. CORBA also offers many levels of resource independence, enabling client applications to be developed in almost any operating system and programming language. This feature was fundamental for choosing CORBA as opposed to other potential technologies, such as Java RMI, which would require the client applications to be implemented in Java. In addition, the remote call services provided by Java RMI are limited if compared to CORBA. CORBA offers a robust set of communication options through event channels, which have been fundamental for implementing the proposed tool.

Three kinds of interaction models can be established between a client application and *CraftContext* (all of them implemented with JacORB): *message passing*, *request-response* and *publish-subscribe* (Costa, 2007). Figure 1 depicts in detail the proposed architecture. The plugin *CraftContext* works as an internal module for the Bukkit server. The player needs a Minecraft client and an access account to connect to the server. As mentioned earlier, it is possible to simultaneously connect several players to the same server, in which the players can share the same environment and possibly interact with each other. The plugin catches the occurrences inside the game and publishes them in specific event channels, according to types of events (like in a *publish-subscribe* model). These channels work as event mediators, hiding from the client the server's location and availability. Similarly, the server is unaware of how many subscribed clients there are and who they are.

The CA application is prepared to receive context information data from real sensors, but for simulation purposes they are provided by the *CraftContext* plugin. In order to solve this problem and to encapsulate the simulation from the real application, an adaptor should be implemented. This adaptor works as an interface between the *CraftContext* plugin and the client application, which allows the application's code to remain unchanged when tested with *CraftContext*.

The adaptor has to subscribe itself to the event channels in order to receive events of in-

terest. The *CraftContext* plugin uses the event channels to provide context events, which are events of interest in the virtual world, as well as to provide events that represent requests from the players. Besides the event channel, the client application (or its adaptors) may still request services directly from the plugin, such as modifications to the world, information about the world's current state (like in a *request-response* model) and sending messages directly to the players (like in a *message passing* model).

CraftContext is responsible for starting event channels and registering event listeners on the Bukkit server. After completing these tasks, *CraftContext* is ready to send event notifications to applications' adaptors. The sequence diagram in Figure 2 shows an example of event notification message flow.

At first, the adaptor should subscribe to the event channels of interest. Event subscription can be performed at any moment while the application is running. After event subscription, the application is able to receive notifications from the Bukkit server (by means of the event channel).

On the other side, the Bukkit server is triggering events asynchronously. These events are sent by the server to the registered listeners. The listeners receive these notifications and publish them in the appropriate event channels. Finally, the channels push the event notification to the subscribed adaptors, using their callback functions.

In the next section we discuss a case study in which a context-aware application is implemented and tested with the *CraftContext* plugin.

Case Study

In this section we present a test scenario in which a CA application and its adaptor are implemented. Later, we discuss the test results.

Scenario

Consider a CA application for smartphones capable of communicating with a monitoring device (working as a sensor) via Bluetooth. This monitoring device continuously gathers a patient's glucose values and sends them to the CA application.

Suppose that at a certain point in time the sensor detected a glucose rate lower than the normal value, which characterizes a mild episode of hypoglycemia. In this case, the application should alert the patient with a suggestion to revert the situation. The glucose rate may keep falling, worsening the hypoglycemia level to a moderate episode. In this case, the application should locate the patient (using GPS location information) and find the nearest hospital at that moment. A message is then sent to the patient, reporting the current situation and the address of the nearest hospital.

If the patient's glucose rate reaches critical levels (severe episode of hypoglycemia), which may characterize risk of fainting, the application should search for the patient's nearest friends and send them a message. This message would inform the friends about the patient's condition and current location. The application should notify the patient when he

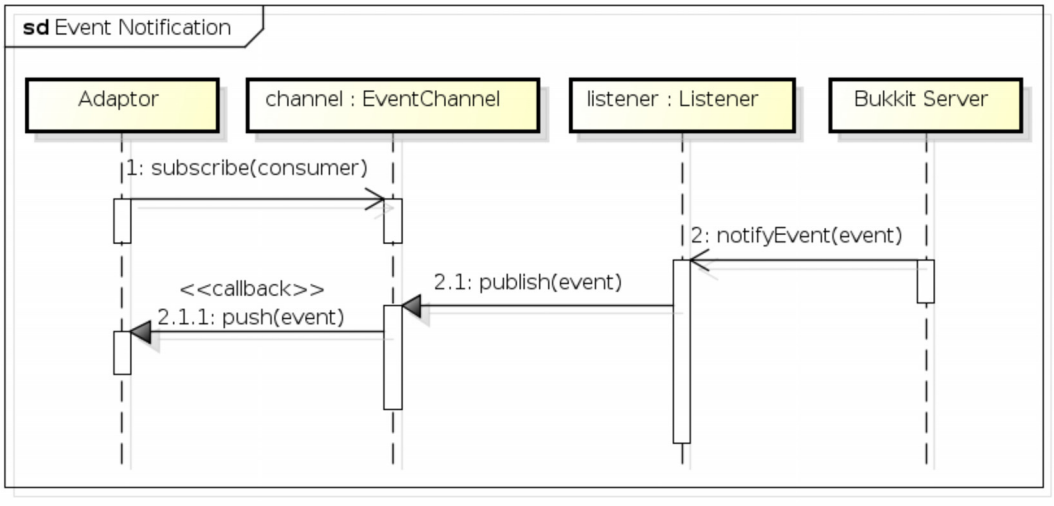

Figure 2. Sequence diagram of an event notification example.

or she becomes stable, i.e., when the glucose rate returns to normal.

Figure 3 depicts a screenshot of such a scenario in the game. In this picture we can observe a player, playing the role of a patient, in front of a building, representing a hospital.

The Context-Aware Application

We have implemented the proposed CA application using a rule-based approach. For this end, we have used *JBoss Drools* (JBoss Community, 2012), which provides a Java rule-based programming environment. In addition to its powerful rule engine, Drools also provides its own language for rule specification, called DRL (Drools Rule Language).

At the configuration phase, the CA application requires information to register the patient to be monitored. The user of the application can use additional commands at any moment during the application's life cycle to register friends who could help the patient in an emergency situation. With this information, the application is capable of monitoring the patient.

A glucose sensor provides glucose rate information for the CA application, at fixed time intervals. Upon receiving information from the sensor, the application inserts these data into the working memory of Drools' rule engine. The inserted information data (also known as facts) may trigger rules, leading the system to take conclusions or actions. Drools' rules are split in four portions: (i) the rule's name, (ii) optional attributes of the rule, (iii) the conditions for activation and (iv) inferences and actions to be taken if the conditions are met.

Figure 4 shows one of the rules we have implemented in the application. This rule defines that an event called *MildHypoglycemia* (responsible for flagging the patient's situation) should be inserted into the working memory in case the condition is met. The condition is a conjunction between two sub-conditions, in which the first checks whether the patient is having a mild hypoglycaemia episode and the second checks for the nonexistence of an event called *MildHypoglycemia*.

Figure 3. Screenshot of the game.

The Adaptor

The CA application expects data provided by sensors, i.e., the glucose monitor and the GPS. Both sensors have their own APIs for communicating with the application. In order to use *CraftContext* as context source (as opposed to real sensors) we have to adapt its information data to a format that the application understands. Aiming to encapsulate the communication with *CraftContext*, we suggest developing an adaptor that (i) transforms information data coming from *CraftContext* and (ii) is also capable of handling the communication between *CraftContext* and the CA application.

The simplest way to develop an adaptor is by replacing the sensors' API libraries with others that have the same name, but functionally different. For example, imagine that the GPS's API provides a *request-response* method, called *getLocation()*, to retrieve the patient's current position (latitude, longitude and altitude). So the adaptor should also define a method called *getLocation()*, which provides the same information. This information, however, is retrieved from the game, and not from the GPS.

The information gathered from the game has to be converted into a format understandable by the application. For example, the application works with a data structure called *Location* that does not exist in *CraftContext*. On the other hand, the tool works with a corresponding structure, called *Position*. The adaptor's job, in this case, is to receive a *Position* data structure from the game, to turn this data into a *Location* data structure, which is then

```
rule "Situation: Mild Hypoglycemia"
    when
        Glucose($rate:rate <= Hypoglycemia.MAX_MILD
                && > Hypoglycemia.MAX_MODERATE)
        not (exists MildHypoglycemia())
    then
        insert(new MildHypoglycemia($rate));
end
```

Figure 4. Example of a Drools inference rule that detects mild hypoglycemia episodes.

sent to the CA application. Figure 5 depicts this example.

The application developer needs to distinguish the parts of the application that should be adapted and the parts that remain the same. Figure 6 depicts a screenshot of the eclipse project tree organization of the proposed application, which we call *HypoSupervisor*. The original application uses two different libraries (which are fictitious in our example): *cellphone* and *sensor*. The *cellphone* library provides all the means to access and use the cellphone's services, such as GPS, messenger, maps, etc. The *sensor* library provides the necessary API to communicate with the glucose rate sensor. The application itself has been developed in a package called *HypoSupervisor*. This is the part of the application that should be kept unchanged.

In order to test the *HypoSupervisor* application using *CraftContext*, we have made a copy of the original project, which we called *HypoSupervisorTest*. In this test project we have created two additional libraries, which received the same names of the libraries used by the application: *cellphone* and *sensor*. These two libraries are now the adaptors, i.e., internally they are prepared to receive information data from *CraftContext*, but externally they have the same interface as the original libraries. Meanwhile, the *HypoSupervisor* package remains exactly the same as the one in the original project.

The implementation of the adaptor's methods has to respect all features of the original methods. For example, the *getLocation* method, depicted in Figure 5, is part of the *GPS* class, which is implemented as part of the package called *gps* (seen in Figure 6). The original method provides an object called *Location* and returns a *GPSSignalNotFound* exception if the GPS device is unable to reach the satellite's signal. In order to simulate the original class's behaviour, the adaptor's class requests the player's position to *CraftContext* and transforms the returned value into a *Location* object. If the player is not logged in, the adaptor throws the *GPSSignalNotFound* exception.

Besides *request-response* communications, the adaptor must also be able to manage event notifications. Figure 7 depicts a sequence diagram that exemplifies the adaptor's role during a severe hypoglycemia episode. In order to receive event notifications, the application should first subscribe an event consumer to the event channel. The first step in Figure 7 shows the adapter transforming the sensor's listener into an event consumer, which is subscribed to the event channel responsible for player notifications.

Upon the occurrence of an event, the channel pushes the event notification to the adaptor by means of a callback. To simulate the glucose rate sensor, we use the player's starvation measurer, which is a well-known parameter of the Minecraft game. Thus, a starving player simulates a patient in a critical situation of hypoglycemia, while a fed player represents a stable patient. So, if the received event notification is indicating a change in the player's food level, the adaptor sends this information to *HypoSupervisor* as a *glucoseRate* object. Then the *HypoSupervisor* analyses the hypoglycemia severity. If it is critical, the application has to find the patient's friends and warn them about the patient's urgent condition.

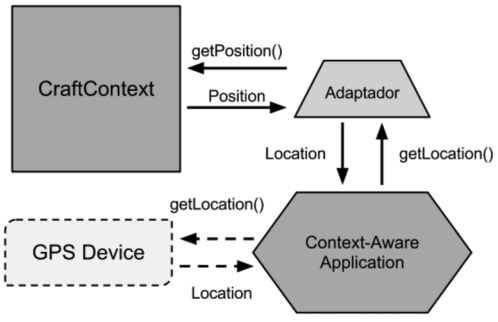

Figure 5. Example of a situation in which the adaptor works in a request-response interaction.

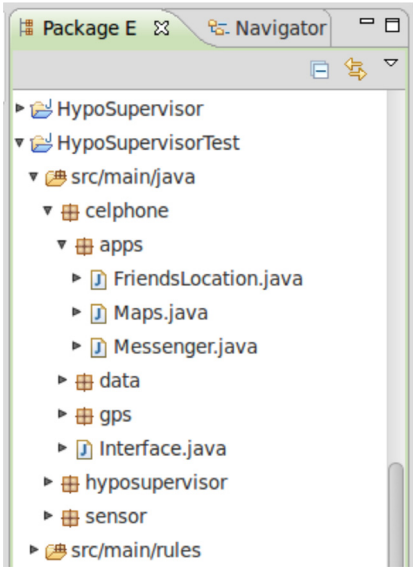

Figure 6. Screenshot of the eclipse project tree organization of the CA application.

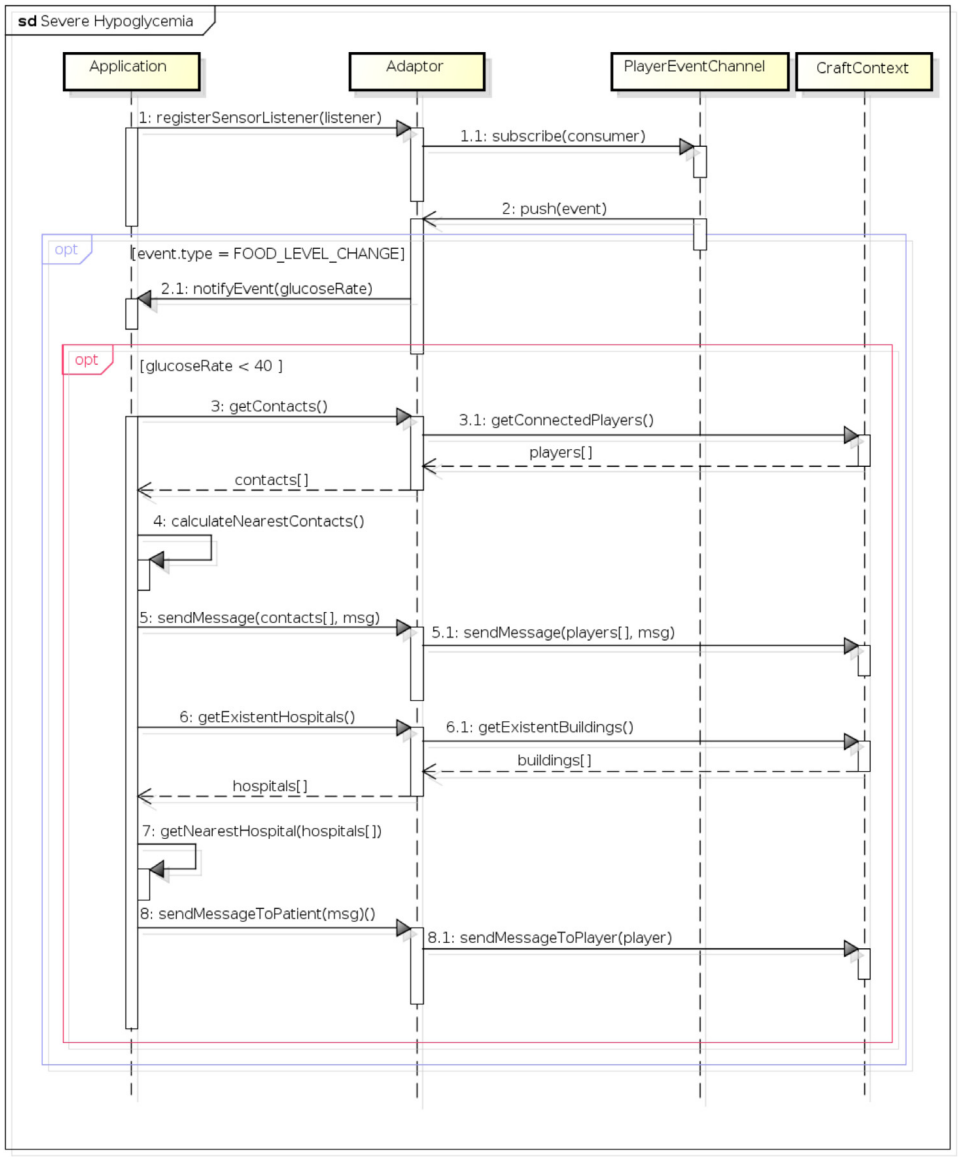

Figure 7. Sequence diagram of a severe hypoglycemia detection scenario.

Then, the *HypoSupervisor* requests the friends' contact list from the adaptor, which in turn requests *CraftContex* to return a list of connected players. The returned players are transformed into contacts and delivered to the application. Then the *HypoSupervisor* calculates the nearest contacts and sends them a message about the patient's situation.

The *HypoSupervisor* should also discover the nearest hospital at that moment. So, it requests the existing hospitals from the adaptor, which performs a conversion similar to the one previously described. The application uses the retrieved hospitals to calculate the nearest one. Finally, the *HypoSupervisor* sends a message to the patient informing about his or her critical situation and providing the address of the nearest hospital.

Tests

We have performed basic black-box tests, based on functional requirements of the application described in the section "The Context-Aware Application". These tests have included four players, remotely connected to the same server, among which one played the role of the patient, two were registered in the application as the patient's friends and the last one represented just a stranger to the patient. In the simulated world, we have built three buildings: two hospitals and a restaurant.

Firstly, we tried to simulate situations that would lead the *HypoSupervisor* to throw exceptions. For instance, we have simulated the situation in which the glucose sensor device is turned off. In that case, the application should be able to detect the patient's absence and show a message on the screen asking the patient to turn on the device. This behaviour was successfully simulated in the game by starting the application without logging in the player that performs the role of patient. The *PlayerNotFound* exception thrown by *CraftContext* was then converted by the adaptor into a command to call the application's method that prints out a warning message.

Another similar case happens when the application fails to find the position of a patient's friend. In this situation, *CraftContext* should throw a *GPSSignalNotFound* exception. We have simulated that behaviour in *CraftContext* by forcing a hypoglycemia episode when there were no registered friends logged in to the game. In such a case, the adaptor converted the *PlayerNotFound* exception into a *GPSSignalNotFound* exception.

Next, we have performed tests with all the players logged in to the game. The first step consisted of analysing whether the variations in the player's starvation level (which is how we have simulated the patient's glucose rate changes in the game) were being correctly sensed and sent to the application. The latter, in turn, should send warning messages to the player, according to his or her current hypoglycemia phase.

For this test we simply gradually changed the level of the player's starvation. Each starvation level changing event is sent to an event channel, where the *HypoSupervisor* application is subscribed to (by means of the publish-subscribe model presented in the section "The Architectural Design of CraftContext"). When the application's adaptor is notified about the new player's starvation level, it calculates the analogue value in real glucose rate and inserts it into the application's working memory. Then, the application's logic kernel infers the patient's condition and uses the adaptor to send him a message informing about his or her current health situation. Depending on the severity of the current hypoglycemia status, the application may suggest a hospital to the patient and/or alert a friend. The adaptor is in charge of adjusting those messages in order to send them to the respective players (patient and friends).

Given the communication model described above, we have achieved the expected behaviour.

The final test concerns the analysis of whether the geographical position of players and buildings is being correctly sensed and notified to the *HypoSupervisor*. In order to describe those tests, we have used decision tables, which are useful artefacts to allow visualization of all possible situations of interest.

Table 1 describes the situations in which players should receive alert messages (warning them about the patient's severe hypoglycemia condition). The criterion used here is the distance between the player that plays the role of a friend and the patient. Since we have already tested the application's behaviour when the patient and/or friends are offline, this table considers that all players are already logged in to the game. So, for instance, if we have the first scenario configuration in Table 2, i.e., *Friend 1* is closer to the patient than the others players, *Stranger* is farther away than the others players and *Friend 2* is neither closer to nor farther from the patient in relation to the others players, then the addressee of the warning message would be the player called *Friend 1*. The last two scenario configurations in Table 1 give us an interesting analysis, which is that even when the nearest player to the patient is the player called *Stranger*, he or she does not receive the warning message. Instead, the second nearest player, who is a patient's friend, receives the message. This behaviour occurs because *Stranger* is not registered in the application as a patient's friend, so he or she is not able to receive messages from the application.

Table 2 shows how the application decides which building should be suggested to the patient in case of moderate or severe hypoglycemia episodes (considering their proximity to the patient). In addition to testing geographical position information, this test also checks whether *CraftContext* is correctly allowing specification of building types. In our scenario, we have two buildings (*Hospital 1* and *Hospital 2*), to which we have assigned the *hospital* type, and another building (*Restaurant*), to which we have assigned the *restaurant* type. *CraftContext* allows applications to search for buildings using building types as the search parameter. Therefore, the *HypoSupervisor* can use this functionality to simulate the search for the hospitals which are near the patient. Note that in the last two scenario configurations in Table 2, even when the building *Restaurant* is the nearest place for the patient, it is not suggested to him or her (since it is not of type hospital). Instead, the suggested place is the second nearest building, which is a hospital.

We have simulated all the scenario configurations presented in both Table 1 and 2. The application has exhibited the expected results for all the tests we have performed. Performing these experiments took us a little over one hour after configuring the tests' parameters and implementing the adaptor. It is possible that if we had used real location and glucose rate sensors probably we would have spent more time to finishthe complete set of test. Context simulation tools help to save time and resources, since they do not require the acquisition of real sensors for the testing phase.

In addition to the validation phase, tests should also be performed during the development phase. Such tests are often required, so they should be simple and fast to avoid delaying the application's development. Field tests are time-consuming, thus impracticable during the development phase. This is another important reason for using simulation tools such as *CraftContext* when testing CA applications.

Concluding remarks

This paper has addressed the recurrent problem of testing context-aware applications. This problem results from the difficulty of using real sensors to gather context information in a controlled and reproducible way, which may lead to time-consuming and unreliable tests. To the best of our knowledge, the solutions proposed in the literature to overcome these problems are usually limited with respect to their applicability, offering support to a limited number of CA applications.

This paper proposes a tool, called *CraftContext*, capable of expanding the set of solvable domains due to its foundation, the Minecraft game, which is rich in detail and resource diversity. Therefore, *CraftContext* is capable of providing a large range of context information types, giving support for specific domain applications, such as the ones that require unusual sensors.

We have implemented a CA application that required specific sensors, which are not easily found in popular devices (e.g., smartphones and tablets), such as the glucose measurer. We have successfully simulated these sensors using *CraftContext* and our test sets have been performed as if we had real sensors.

CraftContext's current version provides means to simulate GPS information, connection status (by notifying when the player logs in or out) and persons' health and starvation status. The current version is also capable of creating new buildings and notifying the players' presence inside them. For the next versions, we intend to increment the range of simulated sensors by adding sensing related to other Minecraft actions, such as opening/closing doors, pushing rail cars, changing chests' items, etc. In addition, *CraftContext* is open source[1] and can be modified at any moment by its users in order to meet their needs for sensor simulation.

We intend to continue improving *CraftContext*, especially with respect to the *quality of context*. CA applications strongly depend on the availability and quality of sensors, which may not be accurate, since they introduce noise, delays and imperfections into the information being sensed. In order to reflect such imperfections, the next versions of *CraftContext* should consider the quality parameters of context information, such as precision, reliability and latency (Broens and van Halteren, 2006).

Acknowledgment

Caroline Rizzi is funded by the Brazilian Research Funding Agency CNPq under grant number 558243/2010-0.

References

BROENS, T.; VAN HALTEREN, A. 2006. Simucontext: Simply simulate context. *In:* INTERNATIONAL CONFERENCE ON AUTONOMIC AND AUTONOMOUS SYSTEMS, 2, Los Alamitos, 2006. *Proceedings*… Los Alamitos, p. 1-6.

BUKKIT TEAM. 2012. Bukkit Forums. Available at: http://Bukkit.org. Accessed on: 01/08/2012.

BYLUND, M.; ESPINOZA, F. 2002. Testing and demonstrating context-aware services with Quake III Arena. *Communications of the ACM*, **45**(1):46-48. http://dx.doi.org/10.1145/502269.502294

COSTA, P.D. 2007. *Architectural support for context-aware applications: from context models to services platforms*. Enschede, Overissel. Ph.D. Thesis. University of Twente, 306 p.

DEY, A.K. 2001. Understanding and using context. *Personal and Ubiquitous Computing*, **5**(1):4-7. http://dx.doi.org/10.1007/s007790170019

DEY, A.K.; ABOWD, G.D.; SALBER, D. 2001. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, **16**(2):97-166. http://dx.doi.org/10.1207/S15327051HCI16234_02

[1] http://github.com/carolrizzi/CraftContext

JACORB. 2012. JacORB. Available at: http://www.jacorb.org/. Accessed on: 01/08/2012.

JBOSS COMMUNITY. 2012. Drools: Jboss Community. Available at: http://www.jboss.org/drools/. Accessed on: 01/08/2012.

MARTIN, M.; NURMI, P. 2006. A generic large scale simulator fou Ubiquitous computing. *In:* ANNUAL INTERNATIONAL CONFERENCE ON MOBILE AND UBIQUITOUS SYSTEMS, 3, Los Alamitos, 2006. *Proceedings*… Los Alamitos, p. 1-3. http://dx.doi.org/10.1109/MOBIQW.2006.361721

MOJANG. 2012a. Minecraft. Available at: http://www.minecraft.net. Accessed on: 01/08/2012.

MOJANG. 2012b. Mojang: Creators of Minecraft. Available at: http://www.mojang.com. Accessed on: 01/08/2012.

OMG. 2012. OMG's CORBA Website. Available at: http://www.corba.org. Accessed on: 01/08/2012.

SAMA, M.; ROSENBLUM, D.S.; WANG, Z.; ELBAUM, S. 2008. Multi-layer faults in the architectures of mobile, context-aware adaptive applications: a position paper. *In:* INTERNATIONAL WORKSHOP ON SOFTWARE ARCHITECTURES AND MOBILITY, 1, New York, 2008. *Proceedings*… New York, p. 47-49. http://dx.doi.org/10.1145/1370888.1370901

SHAH, S.; ILYAS, M.; MOUFTAH, H. 2010. *Pervasive communications handbook*. London, Taylor and Francis, 500 p.

Submitted on August 20, 2012.

Accepted on December 11, 2012.