Implementing Self-Adaptability in Context-Aware Systems

Boris Mejías[1] and Jorge Vallejos[2]

[1] Université catholique de Louvain, Louvain-la-Neuve, Belgium
boris.mejias@uclouvain.be
[2] Vrije Universiteit Brussel, Brussels, Belgium
jvallejo@vub.ac.be

1 Introduction

Context-awareness is the property that defines the ability of a computing system to dynamically adapt to its context of use [1]. Systems that feature this property should be able to monitor their context, to reason about the changes in this context and to perform a corresponding adaptation. Programming these three activities can become cumbersome as they are tangled and scattered all over in the system programs.

We propose to model context-aware systems using feedback loops [2]. A feedback loop is an element of system theory that has been previously proposed for modelling self-managing systems. A context-aware system modelled as a feedback loop ensures that the activities of monitoring, reasoning and adapting to the context are modularised in independent components. In this work, we take advantage of such modularisation to explore different programming paradigms for each component of the loop. We believe that this model can be applied to other kind of applications where the use of different programming paradigms in one system is a straight forward solution.

The paper is organised as follows. Section 2 gives a brief introduction to feedback loops. Section 3 uses feedback loops to model our case study as a self-adaptable context-aware system. Our case study corresponds to a user interacting with a intelligent house through her/his mobile device. Details about the architecture and implementation of a first prototype are given in section 4. Conclusions are presented in section 5.

2 Feedback Loops for Self-Managing Systems

The use of *feedback loops* for modelling self-managing software is the result of abstracting observations taken from existing self-managing systems. Such systems can be found in automated systems and also in nature, as it is shown in [2], where feedback loops are introduced for modeling software. A very related example taken from industrial control systems are *proportional-integrated-derivative controllers* (PID-controller), which also work as a feedback loop. Such mechanism can be used for controlling temperature, pressure and other parameters. For instance, a desired temperature (setpoint) is set for a room. There are thermometers that constantly measure the temperature giving this information to the PID-controller, which computes the error between the desired temperature and the current one. According to the error, an action is triggered to cool

down or heat up the room. After the action is triggered, a new measurement is taken in order to observe if the feedback was positive or negative. A negative feedback makes the parameter closer to the setpoint, because it reacts in the opposite direction of the error. These kind of systems present a continuous evolution of the monitored parameter, oscillating around the setpoint until it converges to it.

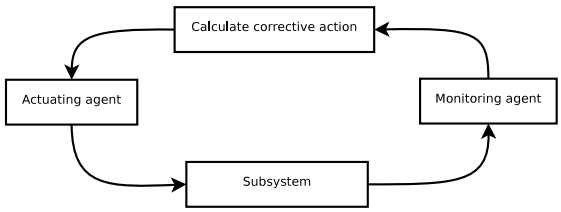

Fig. 1. Basic structure of a feedback loop (taken from [2])

The idea of a feedback loop can be applied to the construction of self-managing software taking into the account the differences with respect to hardware-based automated systems. First of all, the evolution of a running software is discrete, having every loop triggered by events. It is also not evident to define one parameter to monitor. Instead, it is possible to define a setpoint of stability of the system, identifying which events perturb such stability. Another issue is that computing systems nowadays are hardly isolated, and they interact with other systems building a distributed systems with different feedback loops interacting.

Figure 1 depicts the components that form part of a feedback loop, and how they interact. Every component is a concurrent one and has a specific role in the system. From the picture we can see that the subsytem is constantly monitored. This information is given to another component in charge of analysing the monitored information, and deciding a corrective action. This action is performed through another concurrent component known as the actuator. Since every component runs concurrently, the interaction between them can be implemented using message passing, event-based communication, remote method invocation or other mechanisms from concurrent programming. As a rule for using feedback loops in the design of a system, actuators and monitors appear as verbs, while the subsystem and the computing component appear as substantives, as it will be shown in the following section. The reason why it is not like this in Figure 1, is because that is a description of the model, and not the model applied to a system.

3 Feedback Loops for Self-Adaptable Context-Aware Systems

Modelling software systems using feedback loops implies for the developers to identify which kind of information needs to be monitored, dedicating particular agents for this task. Once the monitored information is collected, another component is in charge

of deciding correcting actions, using an actuator agent to apply the corrections to the system.

Consider the case of a computer-assisted system for managing the lights of a so called *intelligent house*. This system consists of a set of lights and sensors that detect the presence of people in the house. The information related to motion detection is passed by the sensor to the light controller, which is a specialised component that decides whether to turn on or off the lights, or simply modify their intensity. Such systems can already be found in the market providing only that specific behaviour. The inner loop of Figure 2 describes this behaviour.

More advanced systems [3, 4] provide pre-configured settings to adapt the behaviour of the systems to different predefined *contexts*. For the automated house, we consider *context* as a situation that is happening in the house, such as watching a film, arriving home, sleeping, etc. Adaptation to the context can be for instance, if nobody is at home, motion detection triggers an alarm. We can model this by enriching the light controller system with a context reasoner. The context reasoner can decide to adapt the behaviour resulting in a different modification of the light intensity depending on the context. To adapt the context, we also make the controlling component able to receive messages from the user. The outer loop is not independent of the inner loop in Figure 2. A message received from a user arriving home adapts the context, and modifies light intensity as actuating action.

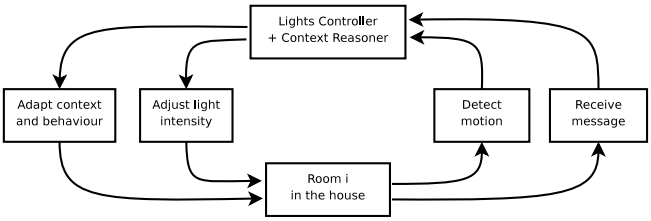

Fig. 2. Feedback loops modelling an automated light system enriched with a context reasoner

Since the use of mobile devices such as phones, PDAs, media players or GPSs are becoming very common, we can expect that users will use her/his mobile device to communicate with the house. We also expect that these devices can adapt their behaviour according to their context. The concept of *context* can be different for mobile devices. The context can represent locality, CPU use, battery load, or a particular situation such as being busy, being in a meeting, etc. Events representing context are constantly monitored giving the information to a *context reasoner*, which decides the behaviour of the device in order to react to other external events or messages. The actuating action can also be to trigger certain events to communicate with other devices. The feedback loop modelling a self-adaptable context aware device is depicted in Figure 3.

These simple loops already provide self-adaptability to the house lights system and to the user's mobile device. The former adapts light's intensity according to the detection of users, and the later adapts its behaviour depending on the context. Consider

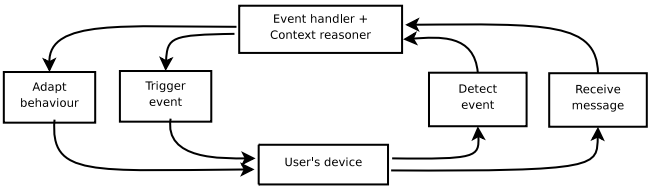

Fig. 3. Feedback loops modelling a self-adaptable and context-aware mobile device.

now both models collaborating as a self-organising system. Here is where the context reasoner of the light controller systems makes more sense, because the intensity of the lights can be adapted accordingly to particular context-dependent scenarios. For instance, you do not want to turn on the lights and wake up the kids when they are in the sleeping context.

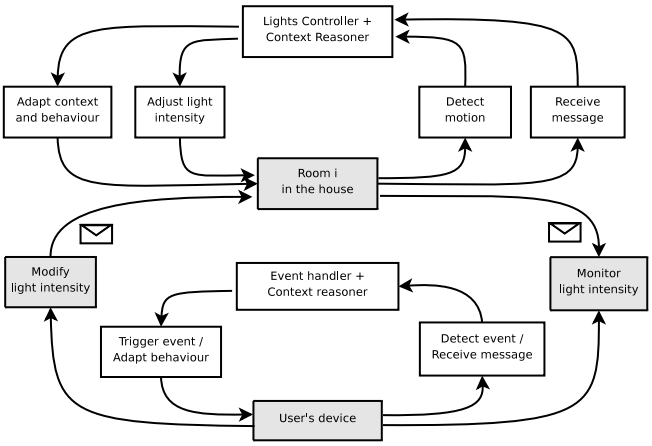

Fig. 4. Communicating two feedback loops.

Figure 4 depicts the interaction between both loops. The two loops of the mobile device have been compressed in one in order to make the global loop more readable. The loop depicts that user's device monitors the intensity of the lights while still handle events and receives messages. As we already mentioned, being in the context of *arriving home* triggers an event to turn on the lights. Since the house lights system is enriched with a context reasoner, some events triggered from user's device may not have always the same result. For instance, turning on the lights when arriving home may not work as expected if kids are in the sleeping context. Like this, two users communicate through the lights systems as stigmergy. We can also observe that sensors and lights serve as stigmergy for the communication of user's device, and the controller of the house, be-

cause both of them monitor the system, and trigger events to modify the intensity of lights.

4 Implementing Feedback Loops

We have started to implement a prototype of the system using Mozart [5], a multi-paradigm programming system implementing the Oz language [6]. We have identified several ways of communicating components of a loop, which can be done using an event-driven approach, or stream communication, which can achieve by pulling or pushing information (lazy or eager execution). To communicate distributed components, message passing seems to be the most appropriated paradigm.

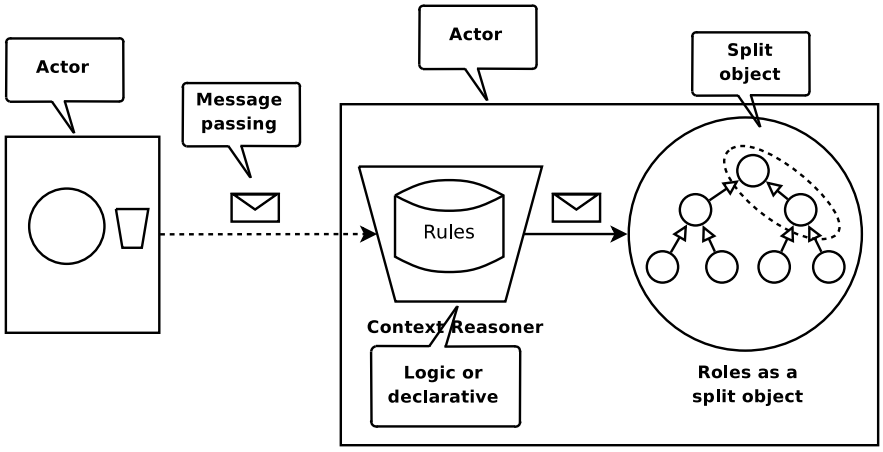

Fig. 5. Context-actor architecture with indications about the different paradigms used for each part of the system.

Figure 5 describes the architecture to implement a *context-actor*, which is a context-aware object, running on its own execution thread as in the actor model. User's devices follow naturally the actor model [7], but inside the actor we can introduce other paradigms as well. For instance, the context reasoner applies a set of rules to the monitor information in order to determine the correspondent rule. This component fits better logic or declarative programming. To implement adaptive behaviour, we have chosen a model representing roles [8], where split objects [9] are used as the general architecture.

Communication between context-actors is implemented using asynchronous message passing, which can also be seen as event-based communication. Since messages are asynchronous, it is easier to handle network failures, because actors do not have to wait for other actors as it would happen using RMI. The actor model also allows

us to easily encapsulate the state avoiding problems of shared-state concurrency which would involved sophisticated locking mechanisms. This scheme makes communication between actors quite independent of the middleware used for distribution support. Mozart is moving towards the Distribution SubSystem (DSS) [10] as its middleware for distribution, and we are also working with AmbientTalk [11] in the specific case study of the intelligent house.

Let us have a look at some parts of the code of the prototype implemented using Mozart. As depicted in Figure 5, the context-actor is one of the main entities. To implement it, we need to run its code in its own execution thread, and we need a way of communicating with it. The following function `NewContextActor` returns a port to communicate with the actor, which is created with three arguments: the attributes of the actor, the roles and the rules. Line 2 shows how a split object is created with the attributes and the roles. The list of rules is used for creating a context reasoner. The variable `Str` is associated with the port that is created and return in line 15. Every message that is sent to the port will appear in the stream `Str`. The thread is launched in line 14, applying the procedure `Loop`, which is defined from line 5 to 12.

`Loop` is a very important procedure. It keeps the current context as one of its arguments, and it reads every message that arrives in the stream via the port. To read the stream, it uses a pattern matching obtaining the message `Msg` and the new stream. In Line 8, it gives the message and the current context to the context reasoner, which returns a new context and the role that will handle in the message inside the split object. In line 9, the split object is called with the message and the correspondent role.

```
 1: fun {NewContextActor Attrs Roles Rules}
 2:    Obj  = {NewSplitObject Attrs Roles}
 3:    CtxR = {NewContextReasoner Rules}
 4:    Str
 5:    proc {Loop Context Stream}
 6:       case Stream
 7:       of Msg|NewStr then Role NewCtx in
 8:          pair(NewCtx Role) = {CtxR Msg Context}
 9:          {Obj Msg Role}
10:          {Loop NewCtx NewStr}
11:       end
12:    end
13: in
14:    thread {Loop init Str} end
15:    {NewPort Str}
16: end
```

The code of `NewContextActor` already shows the integration of some programming paradigms, and shows a possible implementation of feedback loops. With respect to the programming paradigms, as a general view we can say that we are using high-order functional programming. `NewSplitObject` and `NewContextReasoner` are functions returning functions that are invoked in lines 8 and 9. The procedure `Loop` is completely declarative and running in its own thread. The synchronization is done with the pattern matching with the `case` statement. If the variable `Stream` does not contain any new message, it is an unbound variable that does not match any pattern yet.

Once the message arrive, the procedure continues with the loop. It is a fully declarative procedure because there is no explicit state in `Loop`.

As implementation of the feedback loop, the receiving of the message in the stream corresponds to the monitoring action. As soon as a new message arrive, it is given to the context reasoner. The context reasoner can be implemented in whatever paradigm. The only constraint that must respect is that it must return a context and a role to handle the message. The rules will determine the context and the role to be used, adapting the behaviour of the context actor according to the information that has been monitored.

```
 1: fun {NewHouseLights}
 2:    Controller
 3:    Lights = lights(room1:{NewLight off}
                       room2:{NewLight off}
                       room3:{NewLight off})
 4:    Sensors=sensors(room1:{NewSensor init Controller}
                       room2:{NewSensor init Controller}
                       room3:{NewSensor init Controller})

 5:    General = {Object
 6:                meths(switchOn: proc{$ Id}
                                      {Lights.Id on}
                                   end
 7:                      switchOff:proc{$ Id}
                                      {Lights.Id off}
                                   end)}

 8:    Sleep = {Extends General
 9:                meths(switchOn: proc{$ Id}
                                      skip
                                   end)}

10:    Roles = roles(general:General
                     sleep:Sleep)

11:    Rules = use(role:sleep
12:                iff:fun{$ Msg Context}
                          Msg.id == room3 andthen {GetTime} > 10pm
                       end)
13: in
14:    Controller = {NewContextActor attrs(Lights Sensors)
                                      Roles Rules}
15:    Controller
16: end
```

The code of `NewHouseLights` represents the implementation of the part of the system described in section 3. `NewContextActor` already implements the feedback loop. The behaviour is what is given in `NewHouseLights`. Looking at line 14, we can see that the Controller is created a feedback loop using the set of `Lights` and `Sensors` as attributes. Lines 3 and 4 show that all lights start off, and that the sensors know the `Controller` in order to transmit motion detection. Line 10 composes

`General` and `Sleep` as the roles that the split object can have. From line 5 to 7 we can see that `General` is an object having methods `switchOn` and `switchOff`. `Sleep` extends this behaviour by overriding method `switchOn` with a procedure that does nothing, i.e., do not turn the lights on.

To know when to use the `general` or the `sleep` role, line 11 and 12 define the rule that has to be applied by the context reasoner. The rule says to use role `sleep` if and only if we are in `room3` and it is later than 10 o'clock in the evening. The context reasoner will decide for the `general` role otherwise.

5 Conclusions and Future Work

Building context-aware systems is a complex problem that can be divided in different component targeting a particular task. Every task can be solved using the most appropriate programming paradigm, and therefore, we see the advantage of multiparadigm languages. In order to also provide self-adaptability, we have decided to use feedback loops to model the system. Feedback loops allow us not only to modularise the system but also to integrate the different paradigms. The system is able to monitor and adapt itself according to the context in which events are handled.

The implementation of split objects is not straight forward and we realise that a better syntax support is needed. This is part of our future work, where we must continue with the implementation of the prototype. We also identify some limitations in the way of expressing rules, and we are studying alternatives paradigms.

Acknowledgement This work has been partially funded by the European projects EVERGROW and SELFMAN, and by the flemish project of Context-Driven Adaptation of Mobile Services (CoDAMoS).

References

1. Group, I.A.: Ambient intelligence: from vision to reality (2003)
2. Van Roy, P.: Self management and the future of software design. In: Formal Aspects of Component Software (FACS '06). (2006)
3. Dynalite: Dynalite control systems (2003)
4. Crestron: Crestron electronics, inc. (2003)
5. Mozart Community: The Mozart-Oz programming system. http://www.mozart-oz.org (2007)
6. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)
7. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proc. of the 3rd IJCAI, Stanford, MA (1973) 235–245
8. Vallejos, J., Ebraert, P., Desmet, B., Cutsem, T.V., Mostinckx, S., Costanza, P.: The context-dependent role model. In Indulska, J., Raymond, K., eds.: 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS '07). Lecture Notes in Computer Science, Springer-Verlag (2007) 277–299

9. Bardou, D., Dony, C.: Split Objects: a Disciplined Use of Delegation within Objects. In: Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96), San Jose, California, USA (1996) 122–137 Publié en tant que ACM SIGPLAN Notices 31(10).
10. Klintskog, E., El Banna, Z., Brand, P., Haridi, S.: The design and evaluation of a middleware library for distribution of language entities. In Saraswat, V., ed.: ASIAN'03, Springer Verlag (2003)
11. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'hondt, T., De Meuter, W.: Ambient-Oriented Programming in AmbientTalk. (2006)