Orchestrating Context-Aware Systems

A Design Perspective

Alfredo Cádiz, Sebastián González, Kim Mens
Département d'Ingénierie informatique, Université catholique de Louvain
Louvain-la-neuve, Belgium
alfredo.cadiz | s.gonzalez | kim.mens@uclouvain.be

ABSTRACT

The notion of context is becoming increasingly important for the development of applications that can adapt dynamically to their changing environment of use. The demand for dynamic behaviour variability and behaviour interoperation affects the whole engineering process of such applications, and it is yet unclear how different existing solutions fit in the process and what unsolved questions remain. In this paper we present our view on the design of context-aware applications, identifying some of the main aspects that need to be addressed from a software design perspective, and how existing approaches fit this global picture.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; C.5.3 [**Computer System Implementation**]: Microcomputers—*Personal computers, portable devices*

General Terms

Design, Languages

Keywords

Context-aware systems, system architecture

1. INTRODUCTION

The steady convergence towards systems that are aware and reactive to their execution environment brings new functional and technical challenges that were a non-issue upon the time traditional desktop and server systems dominated the computing platform spectrum. Thanks to the real-time availability of information coming from their physical and logical environment, context-aware systems have the potential to adapt swiftly to changing running conditions and match user needs and expectations more tightly than traditional systems.

At the software level, the platform shift from systems that are oblivious of their environment to interconnected systems equipped with physical sensors must be accompanied by a shift in software architectures, moving from relatively fixed, isolated structures to more interoperable and adaptable services. New software architectures should avoid anticipating and articulating behaviour, favouring instead a causal connection between the application's behaviour and its changing execution environment, so that graceful adaptation to changing contexts can take place.

In this paper we discuss main challenges and considerations that need to be addressed when building systems that can adapt to their execution context. We take a design perspective, having therefore as main concern the architecture of context-aware systems: how they should be structured to enable seamless adaptability. We start by describing a motivating example of a system which enhances the user's experience by adapting itself to context. Then we present the general software architecture we envision and continue in subsequent sections by explaining in more detail the different architectural components we introduce.

2. A CONTEXT-AWARE APPLICATION

To illustrate the concepts we present in this paper, we describe the case of a software application for a context-aware smartphone. A smartphone is usually equipped with localisation, battery and movement sensors. It also provides connectivity through WiFi, EDGE, or 3G, allowing, for instance, access to E-mail and VoIP.

In order to optimise the smartphone's battery life, the software installed on the device can adjust its behaviour according to the current context perceived by the sensors. We focus on the case when the device in running out of battery: The base behaviour, when context information does not present useful resources for this functionality, is just warning the user about the issue. The system notifies the user showing a message on the screen and it delegates to him the task of finding an electricity source for recharging the batteries. In case of having data access and localisation information, we can enhance the previous message by providing a map showing the electrical plugs or charging stations close to the user.

Another scenario raises when the smartphone, in addition to low battery levels, detects the user in constant movement in unknown locations. With this supplementary information the device can infer the user is probably unable to plug the device to the electrical network. In this case the device can automatically switch off the 3G connectivity and use, the more battery-conservative, EDGE technology for all forthcoming communications. A possible conflict can arise when

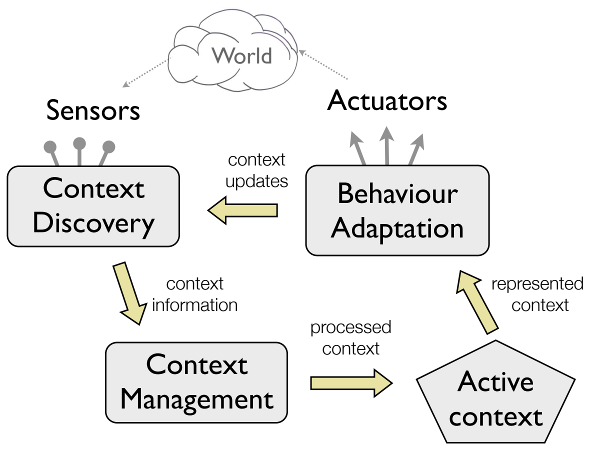

Figure 1: Context-aware system architecture

the user is already on VoIP call. In order to avoid an abrupt end of the communication, when the battery reaches a low level of charge, in addition to a warning, the current 3G mode should be maintained until the call is over.

In order to design such a context-aware application, we need to consider a number of additional concerns introduced by the use of context information: how to discover context information, how to properly manage the available information, how to correctly represent the current context and how to provide a set of specialised behaviours for different situations.

In the next sections we present an architecture for context-aware systems which isolates and describes the concerns listed above.

3. GENERAL SYSTEM ARCHITECTURE

Figure 1 shows the general architecture we propose for context-aware systems. This architecture encodes the overall scheme of things for which we are engineering new context-aware system programming technology. So far, we have been active in the development of programming language abstractions [3, 4]. The architecture helps us positioning past research, and in highlighting our related research that should be looked at to future research.

Figure 1 serves as a kind of graphical index for the remainder of the paper. First a *context discovery* subsystem acquires and translates raw context information into a more useful format for the application. Then a *context management* subsystem resolves possible conflicts between contradictory sources of context information. Once the context is consistent it can be represented as *active context* for allowing the *behaviour adaptation* mechanism adapt the software according to context. The forthcoming sections explain more in detail each subsystem.

4. CONTEXT DISCOVERY

Context awareness necessarily begins by acquiring context information. Any piece of data that is computationally accessible can become part of the context [6]. In our example, we can identify four sources of context information: localisation, battery level, movement and phone state. The last source comes from the application itself.

Since devices can incorporate many different sensors (physical and logic), we need to provide homogeneous access to all possible context information, allowing the application to abstract itself from the sensing details. This means we need to translate the raw sensed data into a more useful format. For instance, when measuring movement, we obtain values regarding the device's acceleration, but the application might just care about *still*, *walking speed* or *vehicle speed*.

This leads naturally to the introduction of the first main component in our proposed architecture, the *context discovery* subsystem, responsible for the extraction, aggregation and deduction of contextual information [7, 9]. Such contextual information should be an accurate depiction of the surrounding environment at any moment in time. Besides the surroundings, contextual information also encompasses the internal state of the device. Applications can adapt their behaviour according to this joint computational snapshot of the external world and the device's state.

The context discovery subsystem helps applications cope with the heterogeneity of context sources. Each kind of context information is sensed in a different way: a battery sensor provides the remaining battery charge, a location sensor provides the geographical coordinates, and a movement sensor shows values on the three spatial axes.

Although having such a context discovery component seems reasonable, even though modern Software Development Kits (SDKs) such as the iPhone's free the application from handling low-level communications with sensors, they do not provide predefined abstractions and frameworks for modularising and customising context sensing, leaving this task to the developer. Approaches such as the Context Toolkit [9] and WildCat [2] do allow applications to abstract away from discovering and presenting context. These approaches provide two modes for providing context information to the base application: pull mode and push mode. The first allows the application to ask for the information when needed, and the latter allows the application to subscribe to updates on any source of information. All the context sensing logic is centralised in an independent module, allowing the application to query for data using an homogeneous protocol.

However, gathered information can sometimes be contradictory or irrelevant. The system should be able to filter information and resolve conflicts to select the most appropriate adaptation. This is the role of the next subsystem in the architecture.

5. CONTEXT MANAGEMENT

Even if we can sense and translate context information in a useful format for applications, problems arise when the information becomes contradictory. As applications become more aware of their surroundings, and information comes from many different sources, two or more different pieces of context information can promote different and, in some cases, inconsistent adaptations to the application.

In our example in Section 2 we defined different adaptations according to possible situations regarding a low battery scenario. Additionally, we also considered an exceptional case when the smartphone is in use and its behaviour cannot be adapted until the call is over. In this case we should hold the adaptation until the phone is idle again and avoid affecting the user's current call.

General-purpose languages provide little or no support for resolving possible conflicts caused by contradictory context information. In addition, depending on the level of conflicts in a particular system, resolving them can become an extremely complex task, requiring more elaborated solutions.

The conflict resolution concerns should all be handled by a *context management* component which is in charge of processing the context information gathered by the context discovery component. We can find many different approaches for such a subsystem: SOCAM [5] in addition to its context discovery capabilities, allows the definition of first-order logic rules for combining context and triggering related actions. CRIME [8] introduces a federated fact space for allowing the construction and maintenance of a knowledge base of context information. This, combined with a logic coordination language allows the definition of rules for processing the knowledge and reaching the most appropriated action for each state of the knowledge base.

The resulting set of processed context information must be consistent enough for allowing the application to adapt itself according to the current world state. Then, for allowing dynamic adaptation to context we need to create a link between the application logic and the active context at run-time. This will be discussed in the next Section.

6. ACTIVE CONTEXT

Once context information is captured as raw data; translated into an appropriate format by the context discovery subsystem, and the context management subsystem has filtered and solved conflicts in incoming context information, we obtain the *active context*, which is consistent and can be made visible, at run-time, to the application.

Context information should be meaningful enough for letting the application know the relationships between different pieces of context information. By relating different pieces of context we allow the creation of families of contexts sharing similar characteristics and the application can exploit these relationships for reusing adaptation logic. These relationships are provided by the developer when implementing the application's context-aware logic.

In Figure 2 we can see part of the context information related to the example presented in Section 2. In this case, the `home` and `office` contexts are related to the `known-location` context and this last context will promote adaptations which are valid for all the known places, for instance, redirecting the call to the closest fixed line. This is possible because when either `home` or `office` becomes active, also `known-location` will be active as well.

This kind of active context representation can be found in Ambience [3, 4], a context-aware prototype-based language with multiple dispatch. Ambience features first-class contexts and supports representation of context information as object graphs. Contexts can be related using delegation links and thus families of contexts can be created. This allows defining specialised behaviour for a context and include all the subcontexts delegating to it.

The *active context* does not imply a specialised subsystem, but needs to be considered as the glue between the context information and the application's logic. This allows the latter select the most appropriate adaptation at run-time by checking the current active context, which represents the current state of the surrounding world. In the next section we discuss how languages can adapt their behaviour according to their active context.

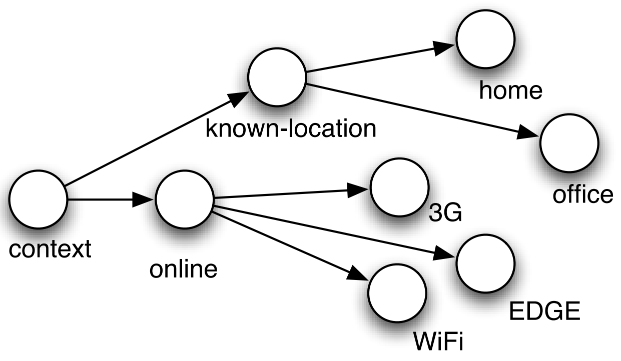

Figure 2: Partial Context organisation for example of Section 2.

7. BEHAVIOUR ADAPTATION

So far we have presented abstractions for sensing, reasoning and representing the active context information. Once the application can see a consistent representation of the context information it can enhance its functionality for selecting the most appropriate behaviour according to the current context. The effective use of that information is still a challenging problem for application programmers.

General purpose languages such as C++, Java or Objective-C can provide dynamic adaptation to context, with some considerations. We need to foresee the application's variation points at design time, since we need to implement and integrate the mechanisms for selecting from different behaviours in the application's code. To select the most appropriate context we can implement a design pattern, such as the strategy pattern, for encapsulating different possible behaviours.

The schema mentioned above can be acceptable when the application is limited to a few fixed and previously defined variation points. Problems arise when we need to introduce new variation points into the application. We would need to rewire our defined machinery for the new variation points. Also we experiment an explosion of classes related to the design pattern implementing our dynamic adaptations and each encapsulated behaviour.

Considering more powerful and sensor-enriched devices, we cannot assume few variation points. Every functionality can be modified by context at run-time and new adaptations can be needed as the environment evolves. As mentioned above, using traditional design patterns we experiment a huge overhead related to all the machinery needed for providing dynamic adaptation to context.

Context-oriented programming languages (COP) have been proposed as an alternative and more elegant solution for dealing with the increasing need to adapt software to context at run-time. ContextL [1] is a CLOS extension which adds additional behaviour to classes in the form of layers. When a layer is activated the class will present the new behaviour defined in the layer. Ambience [3, 4] proposes a prototype with multiple dispatch object model. Ambience provides first-class contexts and they can be associated with methods. When a context is active, the specialized version of the method is invoked. ContextL and Ambience allow the separation of base behaviour from context-dependent adaptations.

In our smartphone example from Section 2 we have described behaviour which is affected by context information. Using a COP language such as Ambience it is possible to explicitly separate each piece of behaviour corresponding to a different context:

```
(defmethod call ((from @phone) (to @phone))
  ... Make a call ...)

(with-context (@low-batt)
  (defmethod call ((from @phone) (to @phone))
   ... Show warning message
   (resend)))

(with-context (@low-batt @online)
  (defmethod call ((from @phone) (to @phone))
    ... Search for electricity and show map
    (resend)))

(with-context (@low-batt @unknown-location @3G)
  (defmethod call ((from @phone) (to @phone))
   ... Switch to EDGE...
   (resend)))
```

The code above shows how the four possible situations can be implemented. All the methods share the same signature. The first one represents the base case, while the other three contain the specialised behaviour according to the possible contexts. The active context information is explicitly associated to the methods defined in the scope of the `with-context` construct and they do not override the methods defined in another set of context. The `resend` keyword calls the next most specialised implementation of the method. This is how code can be reused. In the snippet above, the code implementing a phone call is implemented at the first method only.

When an application executes its adapted behaviour, it can either explicitly or implicitly change the state of the context information. The first can happen when the application updates its internal state, such as being idle or performing a call. The latter happens when adapted behaviour stimulates its surrounding world in such a way that sensed context information changes thanks to the new behaviour. In our example from Section 2 switching to EDGE changes the context information about the device's connectivity. This reaction on the context information closes the cycle presented in Figure 1.

8. CONCLUSIONS

Context-aware systems raise questions as to how such applications should be architected. Potentially large amounts of information need to be assimilated by the system to build an accurate representation of the current context. Besides discovering the context and managing its computational representation, we need adequate tools to build applications that can adapt to such context.

In this paper we have presented our vision on the global architecture of the kind of context-aware system we aim at obtaining. The different architectural components follow from the main basic requirements that need to be addressed to enable dynamic adaptation to context. Even though this architecture is fairly general, it already provides a first definition of the issues that should be taken into account when building context-aware systems.

9. ACKNOWLEDGEMENTS

This work has been supported by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels and by the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy.

10. REFERENCES

[1] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In *Proceedings of the Dynamic Languages Symposium*, pages 1–10. ACM Press, Oct. 2005. Co-located with OOPSLA'05.

[2] P.-C. David and T. Ledoux. Wildcat: a generic framework for context-aware applications. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, New York, NY, USA, 2005. ACM.

[3] S. González, K. Mens, and A. Cádiz. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008.

[4] S. González, K. Mens, and P. Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *Proceedings of the Dynamic Languages Symposium*, pages 77–88. ACM Press, Oct. 2007. Co-located with OOPSLA'07.

[5] T. Gu, H. K. Pung, and D. Q. Zhang. A service-oriented middleware for building context-aware services. *J. Netw. Comput. Appl.*, 28(1):1–18, 2005.

[6] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March–April 2008.

[7] C.-H. Hou, H.-C. Hsiao, C.-T. King, and C.-N. Lu. Context discovery in sensor networks. In *International Conference on Information Technology: Research and Education (ITRE)*, pages 2–6. IEEE Computer Society Press, 2005.

[8] S. Mostinckx, C. Scholliers, E. Philips, C. Herzeel, and W. De Meuter. Fact spaces: Coordination in the face of disconnection. In *9th Int. Conf. on Coordination Models and Languages*, volume 4467, pages 268–285, June 2007.

[9] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, New York, NY, USA, 1999. ACM.