A Framework for Requirements Engineering for Context-Aware Services

Anthony Finkelstein Andrea Savigni

Department of Computer Science
University College London
Gower Street
London WC1E 6BT
United Kingdom
E-mail: {A.Finkelstein|A.Savigni}@cs.ucl.ac.uk

Abstract

Context-aware services, especially when made available to mobile devices, constitute an interesting but very challenging domain. It poses fundamental problems for both requirements engineering, software architecture, and their relationship. We propose a novel, reflection-based framework for requirements engineering for this class of applications. The framework addresses the key difficulties in this field, such as changing context and changing requirements. We report preliminary work on this framework and suggest future directions.

1. The Rationale

The purpose of this section is to highlight the key problems associated with requirements engineering in the area of context-aware services. In order to properly classify concepts, we will adopt Michael Jackson's terminology, as introduced in [10] and briefly reviewed in Sect. 2.1. Particularly the critical distinction he maintains between the "world" and the "machine". That terminology will be used throughout the paper.

In this paper, by "context-awareness" we mean the ability of a particular service to adapt itself to a changing context. One classical example is mobile commerce (m-commerce) applications, which should run equally well on full-fledged Web browsers running on desktop computers, on graphic Personal Digital Assistants (PDAs), on Wireless Application Protocol (WAP)-enabled mobile phones, and possibly even on low-end mobile phones, maybe using Short Message System (SMS).

Requirements engineering in the area of context-aware services, especially when these are targeted towards mobile devices, poses new and very challenging problems, that can be summarised as *changing context* and *changing requirements*.

A changing context means essentially that one cannot, while analysing requirements, rely on reassuring assumptions about the world. A changing world complicates the machine by orders of magnitude. In the case of context-aware mobile services, changing context may entail:

- changing location. This means not only that the absolute location of a device can change, but also that the relative locations of two devices must be taken into consideration;
- changing bandwidth for networked devices, most often in unpredictable ways;
- changing display characteristics e.g., graphics PDAs, text-only mobile phones, colour vs. monochrome displays, etc.;
- changing usage paradigm. For example, from a user perspective having a full-screen, button-centred PDA is very different from using a scroll-centred mobile phone;
- target platforms unknown in advance. Note that this problem is *not* implied by any of the preceding points. Platforms may be unknown in advance, and the service should anyway be able to dynamically adapt itself to this aspect of the new context. This means of course performing a very hard abstraction job in order to express the common set of characteristics in a general, uniform way.

This very volatile context of course influences requirements. A key distinction, adapted from Axel van Lamsweerde's work (see Sect. 2.2), is made here between *goals* and *requirements*. We define a goal as a fixed objective of the service, whereas a requirement, in our view, is a more volatile concept that can be influenced by the context. For example, in a m-commerce service, a goal can be “maximise usability of the system”, which is a very abstract objective that the system should tend to [6]. By contrast, a requirement can be: “the display must show both the current state of the shopping basket and a set of available options”. This requirement of course makes sense only if the display is large enough.

One more, fundamental issue related to such services is that they usually belong to the “new economy”. This means in general that these systems have an extremely short time-to-market, which in turn means that traditional, heavyweight methodologies – such as the Rational Unified Process (RUP) [5] – are not applicable.

For all these reasons, we argue that a new approach is needed to tackle this kind of services. Such an approach is the subject of this paper, and will be described as follows. Section 2 will provide the reader with some background information. Section 3 outlines the reflective approach that will be used throughout the work. Section 4 explains the framework itself, while Sect. 5 sets out some of the key challenges it poses. Finally, Sect. 6 sketches some possible ways to move towards an implementation of the framework.

2. Background

The goal of this section is to give a very brief overview of the two main influences behind this paper, namely Michael Jacksons’s “world and machine” work [10], and Axel van Lamsweerde’s “Kaos” [6].

2.1. The World and the Machine

[10] represents a cornerstone in understanding the relationships between a software artifact and the surrounding world. Jackson identifies four facets of relationships between the world and the machine:

- the modelling facet, in which the machine simulates the world;
- the interface facet, where the world touches the machine physically;
- the engineering facet, where the machine controls the world;
- the problem facet, where the shape of the world and of the problem influences the shape of the machine and of the solution.

The discussion of the engineering facet turned out to be particularly useful to us, and particularly the distinction between requirements, specifications, and programs. Requirements are concerned solely with the world, programs are concerned solely with the machine, specifications are the bridge between the two. Section 4 will use these concepts in working out the boundaries between world and machine within our framework.

2.2. Goal-oriented Requirements Engineering

The seminal works by Yue [17] and van Lamsweerde [6] opened a new direction in requirements engineering: the *goal-oriented* approach. The key achievement of this new approach is that it makes explicit the *why* of requirements. Quoting van Lamsweerde, “[before goal-oriented requirements engineering] the requirements on data and operations were just there; one could not capture *why* they were there and whether they were sufficient” [16].

van Lamsweerde’s goal-oriented requirements engineering approach provides for three levels of modelling:

- the meta level, that refers to *domain-independent abstractions*. This model contains concepts such as `goal`, `requirement`, `object`, `entity`, and so on;
- the base level, containing domain-dependent concepts, such as `service`, `telephone`, `bandwidth`, etc. The structure of the meta-level model constitutes a meta-level guide on how to conduct a requirements engineering activity. For example, since `goal` and `constraint` are linked by a `operationalisation` link, every concept in the base level that is an instance of a meta-level concept `constraint` must be linked to an instance of a meta-level `goal` by an instance of a meta-level `operationalisation` link;
- the instance level, containing specific instances of the domain-level concepts.

3. The Reflective Approach

“Computational reflection is the activity performed by a computational system when doing computation about its own computation” [11]. A reflective system maintains, *at run-time*, data structures that materialise some aspects of the system itself.

The problem of allowing a program to reason upon, and possibly change, itself is not new, and has been studied extensively especially in the programming languages community. For example, languages such as LISP and Prolog allow programs to be manipulated as data. More recently, so-called “open languages” (such as OpenC++ [4] or OpenJava [14]) allow programmers to influence the translation process, thus actually providing for the definition of new languages.

For our purposes, reflection means that an explicit, run-time representation of system behaviour is maintained, which *reifies* the actual system behaviour in the sense that changes in the latter are materialised in the meta-level description. Similarly, changes in the meta-level description *reflect* back into the underlying system's behaviour. This "closed loop" approach is called *causal connection*. A reflective system is structured into a (potentially unbound) number of logical levels: the *reflective tower* [13]. In practice, there are seldom more than two of them.

Reflective systems are based on two concepts: consistency between internal and external representations of the system, and separation between meta computation and computation. The consistency is guaranteed by causal connection: computations performed in the base level are reified by the meta level, whereas changes in the meta level reflect back into the base-level. The separation between meta computation (i.e., computation whose domain [11] is the base-level) and computation (whose domain is the world) is essential in order to achieve *transparency*: new functionality can be added to an existing system in a transparent way i.e., without the existing system noticing. This is especially true of functionality implementing non-functional requirements, such as fault-tolerance and security.

Why do we regard a reflective approach as such a fundamental issue? First of all, let us make one point clear: reflection, at least in our view, is a *mechanism*, not a goal. More precisely, it is a mechanism for manipulating meta data in a clean and consistent way. Now, reflection is key in this field because manipulating meta data is essential in this context of highly-dynamic services, as these must be able to dynamically adapt themselves to changing context and changing requirements.

4. The Framework

Figure 1 shows the key concepts of the proposed framework.

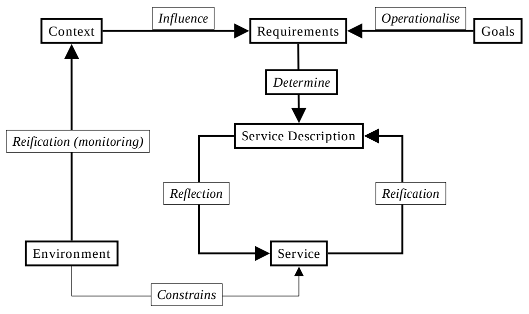

Figure 1. The overall framework.

The rest of this section is devoted to a detailed explanation of the framework constituents. This explanation will follow a precise path that moves from the outside inward i.e., from the outer world towards the boundaries with the machine, and finally inside the machine itself. Therefore, we will start from what is available in the world: goals and environment. We will operationalise goals into requirements, and represent environment information into a context; all of this still belongs in the world. Later we will move from requirements and context towards a service description, which is the bridge between the world and the machine (what Michael Jackson calls "specification"). Eventually we move inside the machine with the notion of a service. Note that throughout the paper we will stick to the notion of the machine as pure software; in other words, we will consider devices (PDAs, mobile phones, etc.) as part of the world.

4.1. Goal

A goal is an objective the system should achieve through cooperation of agents in the software-to-be and in the environment [6]. In our view goals are *immutable* i.e., they do not change with the changing context. They represent the ultimate objective the service is meant to achieve. Changing the goals would mean changing the service itself. Along the lines of [6], a goal is not immediately achievable through actions performed by one or more agents; in other words, a goal is a somewhat abstract and long-term objective.

4.2. Environment

By "environment" we mean whatever in the world provides a surrounding in which the machine is supposed to operate. Taking the environment into account is crucial because it strongly influences the behaviour of the machine. Recall the example of the m-commerce service. In this case the environment comprises such things as bandwidth, location (absolute and relative), service availability, characteristics of the device, and many more issues.

An alternative definition of environment might be: "whatever over which we have no control". If the bandwidth is low, the connection is erratic, the PDA's display is small, the person carrying the mobile phone is driving on a mountain road with many tunnels, this is something that cannot be solved by software. The job of a software engineer can be summarised as a struggle *towards* the goal *despite* the environment; all we can do with the environment is know it and describe it in the best possible way, but we cannot change it.

4.3. Context

Context is defined as the reification of the environment. Note that in this case there is no reflection whatsoever (i.e., no downwards arrow) because, as explained in the previous section, the environment is not modifiable. A context thus provides a manageable, easily manipulatable description of the environment. Most important, such description is continuously, dynamically updated to take into account the fact that the environment also continuously changes.

4.4. Requirement

A requirement represents one of the possible ways of achieving a goal. A requirement operationalises a goal, in that it represents a more concrete, short-term objective that is directly achievable through actions performed by one or more agents. One key assumption that we make is that *requirements can change during system execution*, which differentiates them from goals. In fact, due to a changing environment, the context may change in such a way that the operationalisation of the goals is no longer valid. This calls for monitoring of the context with respect to the goals: changes in the context may yield the necessity for changes in the requirements.

In very informal terms, one may say that requirements are a trade-off between the noble goals and the actual reality. For example, the goal of an m-commerce service might be to provide for a highly interactive user experience. Given this goal, if the context is favorable (e.g., high bandwidth, large colour display, Java Virtual Machine implementation available on the PDA) a requirement might be "use a colorful Java applet to represent the state of the shopping basket", whereas if the connection is slow or there is no JVM available, the requirement may be mitigated into "use a 16-colour animated gif".

4.5. Service Description

A service description is the meta-level representation of the actual, real-world service. As such, it is obviously influenced by the requirements, hence the `Determine` box in Fig. 1. A service description might seem redundant, as one may think of going directly from requirements to service. Why is an intermediate component needed? The answer lies in the reflective approach and in the need for continuously monitoring the service. In fact, the service can be influenced by the environment, and can therefore change in unpredictable ways. These changes can lead to inconsistencies between the service and the requirements. This calls for monitoring of the former with respect to the latter. A service description is a meta-level description of a service. If a suitable formalism is devised for this description, the latter can easily be compared against the requirements in order to establish whether a runtime violation [8] has occurred.

Now, suppose such a violation is detected. We argue that the "reflective way" is a clean and consistent manner of performing run-time changes to the underlying level (which is, at last, the actual system as perceived by the user). This approach consists in manipulating the service description in order to reconcile the service with the requirements. The causal connection, in particular the downwards link (reflection) provides for the consistency between the service description and the service itself. Architectural reflective techniques can be employed to that aim [2, 3, 15].

Since the service description describes the behaviour of the service, it can be regarded as a system specification in the sense used in [10]. Thus, it serves as the bridge between the world and the machine.

4.6. Service

Finally, the service is the heart of the machine. It provides the actual behaviour as perceived by the user. It is worth pointing out that, even though it is only this service that actually interacts with the user, it is the last link in the chain described above; in other words, the actual value delivered to the user is not the service alone, but also the whole hidden reflective infrastructure.

It is also worth pointing out that, apart from goals that are specified off-line and never changed (recall, changing goals means changing what the service provides, and this means at the very least pulling the service down), all the remaining items appearing in Fig. 1 have a run-time image, as emphasised in Fig. 2, where the run-time components are greyed. Finally, Fig. 3 emphasises (in grey) the meta-level

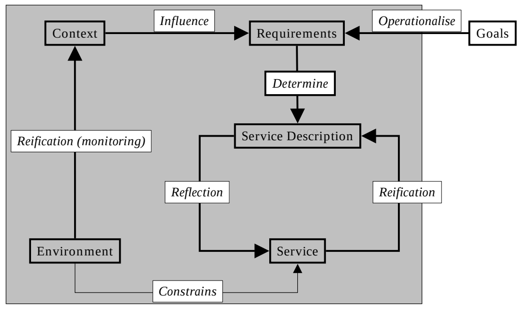

Figure 2. The run-time components.

components i.e., all those components that, even having a run-time image, are not directly visible to the end user.

5. The Challenges

The problems examined in the previous section represent a formidable challenge for any software engineer. More precisely, the following points must be addressed.

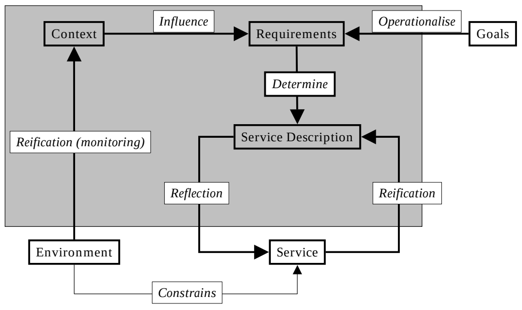

Figure 3. The meta-level components.

5.1. Representing context information at run-time

One of the key issues in these systems is that context is continuously changing. Therefore, requirements, in the first place, and system behaviour must adapt themselves to the changing context. In order for this to be feasible, the context (and its changes) must be represented at run-time. This representation must take place in a way that is both readily understandable by humans and easily manipulatable by machines.

5.2. Bringing requirements information to run-time

In order to be able to perform run-time service description monitoring against requirements, these must be readily accessible at run-time (see Sect. 4.5) [9].

In addition, as explained in Sect. 4.4, requirements typically change over time, so this representation must not simply be a read-only view, but must rather be an evolvable one.

5.3. Bringing architecture information to run-time

This is admittedly one of the most controversial points. It is widely accepted in the software engineering community that a suitable software architecture design phase should always precede the actual implementation. However, in most cases all information about system architecture is lost in the running system [15]. In other words, a running system *implements* a specification; however, this specification is scattered throughout the code, and no explicit representation of it exists at run-time.

6. Implementation Issues

6.1. Describing the Meta Levels

One key question to be answered is: "How to describe the meta levels in an easy and powerful way?" One particularly promising way is the use of XML for such description. The main reasons behind such a choice are sketched in the sequel:

- XML is a world standard. A description implies a formalism, so why not choose a standard one?
- no need to build custom parsers. A number of products implementing the standard DOM and SAX APIs are widely available, often at no cost;
- a number of standards, APIs, and products are available to easily and efficiently manipulate XML files, first of all XSLT;
- a lot of work has been (and is being) done at UCL in this field; in particular, the work on consistency checking of distributed documents (that yielded `xlinkit` [12]) could prove a very useful starting point in determining whether the runtime system behaviour is still aligned with the requirements.

6.2. Where Does All This Belong?

An interesting question to ask is: Where does all the framework belong? Or, in other words, should every single service take care of this on its own? Can all, or at least some, of the framework be collected in a separate product which can be implemented once and for all and customised at will? If so, which parts are strictly service-dependent and which can be made common?

We do not yet have a definitive answer to these questions. However, our current thought is that it should be possible to provide a service-independent set of mechanisms for representing context in a significant class of context-aware services. The mechanisms by which such a context is populated in any particular case is clearly a matter for the device vendor.

On the service description side, the situation is more complex, and service description schemes drawn from existing middleware frameworks [1, 7] may be the right direction.

Acknowledgments

This work was partially funded by UWA (Ubiquitous Web Applications), a EU-funded, Fifth Framework Programme project that the authors are carrying on in cooperation with a number of academic and industrial partners from six European countries. The work described in this paper is intended as an initial contribution to the project.

We are also grateful to Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo for the fruitful discussions that influenced some of the views in this paper.

References

[1] L. Capra, W. Emmerich, and C. Mascolo. Reflective Middleware Solutions for Context-Aware Applications. Submitted for publication.

[2] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering and 6th Reengineering Forum*, Florence, Italy, March 8-11 1998.

[3] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level. In *Proceedings of Automated Software Engineering – ASE'99 14th IEEE International Conference*, pages 263–266, Cocoa Beach, Florida, USA, Oct 12-15 1999.

[4] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA95*, pages 285–299, October 1995.

[5] R. S. Corporation. The Rational Unified Process. `http://www.rational.com/products/rup/`.

[6] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed Requirements Acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[7] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, April 2000.

[8] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling System Requirements and Runtime Behavior. In *Proceedings of IWSSD'98 - 9th International Workshop on Software Specification and Design*, Isobe, Japan, April 1998. IEEE Computer Society Press.

[9] S. Fickas and M. S. Feather. Requirements Monitoring in Dynamic Environments. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, pages 140–147. IEEE Computer Society Press, 1995.

[10] M. Jackson. The World and the Machine. In *Proceedings of the 17th International Conference on Software Engineering*, pages 283 – 292, Seattle, Washington, USA, April 24 – 28 1995.

[11] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA87, Sigplan Notices*. ACM, October 1987.

[12] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. `xlinkit`: a Consistency Management and Smart Link Generation Service. Technical Report RN/00/66, University College London – Department of Computer Science, December 2000. Submitted for publication.

[13] B. C. Smith. Reflection and Semantics in Lisp. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, USA, January 1984.

[14] M. Tatsubori and S. Chiba. Programming Support of Design Patterns with Compile-time Reflection. In *OOPSLA98 Workshop on Reflective Programming in C++ and Java*, pages 56–60, Vancouver, Canada, 1998.

[15] F. Tisato, A. Savigni, W. Cazzola, and A. Sosio. Architectural Reflection. Realising Software Architectures via Reflective Activities. In *Proceedings of the 2nd Engineering Distributed Objects Workshop (EDO 2000)*, Davis, California, USA, November 2–3 2000. To appear.

[16] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In *Proceedings of ICSE'2000 - 22nd International Conference on Software Engineering*, Limerick, 2000. ACM Press. Invited Paper.

[17] K. Yue. What Does It Mean to Say that a Specification is Complete? In *Proceedings of IWSSD-4 – the Fourth International Workshop on Software Specification and Design*, Monterey, CA, USA, 1987.