Evolution of an Adaptive Middleware Exploiting Architectural Reflection

Francesca Arcelli and Claudia Raibulet

DISCo – Dipartimento di Informatica Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca, I-20126, Milan, Italy
{arcelli, raibulet}@disco.unimib.it

Abstract: Nowadays information systems are required to adapt themselves dynamically to the ever changing environment and requirements. Architectural reflection represents a principled means to address adaptivity. It also represents an emerging approach to deal with the software evolution issues. In this paper we aim to point out how systems exploiting architectural reflection to achieve adaptivity evolve in an organized, linear manner controlling easier their growth and complexity than systems based on ad hoc solutions. To sustain this affirmation we present the possible evolution improvements we gain through our Adaptive and Reflective Middleware (ARM).

Keywords: Software evolution, architectural reflection, adaptive systems.

1. Introduction

One of the most challenging issues raised by nowadays information systems is to adapt themselves dynamically and automatically in the attempt to accomplish the anytime, anyone, anywhere paradigm in a constantly changing reality. In this context, we describe our approach with the aim to provide support to identify, choose, and exploit the appropriate system's components able to satisfy users' requests according to different levels of quality of services (QoS) in a dynamic mobile-enabled heterogeneous environment.
Our solution for adaptive systems exploits architectural reflection [1, 3], which introduces additional layers playing an intermediary role between the representation and implementation of the system's components/functionalities and applications. These reflective layers enable applications to adapt to the systems' features and, vice-versa, systems to adapt to the applications' requirements.
The usage of reflection at the architectural level has both advantages (i.e., a principled, as opposed to ad-hoc, way to achieve adaptivity [5], an explicit representation of architectural aspects exploited at run-time) and disadvantages (i.e., a significant increase of the number of software components which reduces the overall efficiency, modifications of the reflective components may cause overall damage). In this paper we focus on an additional and implicit advantage, which can be considered a side-effect of applying reflection at the architectural level: the support for software evolution. The benefits of using architectural reflection as a mechanism to achieve software evolution are treated in several scientific works [2, 4, 6, 10, 14].
Our aim is to point out those design aspects of a reflective architecture (or more precisely of the reflective knowledge and its management) which ensure implicitly its proper and consistent evolution. To achieve this goal, we propose our solution for reflective and adaptive middleware (ARM) considering the main laws of software evolution introduced by Lehman [7]: continuous adaptation, increasing complexity, continuing growth, declining quality, organizational stability, and conservation of familiarity.

The rest of the paper is organized as follows. Section 2 provides a brief overview of our software architecture exploiting reflection to achieve adaptivity. Section 3 focuses on the design aspects of our approach that enable and ensure the evolution of ARM considering the Lehman's software evolution laws. Conclusions and further work are dealt within Section 4.

2. ARM: An Adaptive and Reflective Middleware

Our solution for adaptive systems is a reflective middleware composed of two layers. The first layer defines the reflective knowledge. It reifies the system's components in terms of reflective objects (capturing their current state) and their related QoS [12]. In addition, we have introduced the concept of *property*. Properties [13] express the characteristics of the system's components that are not directly measurable at run-time, but which are exploited together with the QoS to achieve adaptivity. Currently, ARM defines three properties: structural, representing the physical structure of an ARM-enabled node, topological, representing the connections between ARM-enabled nodes, and location, representing the physical location of a component.
The second layer introduces the view concept [13] representing an organizational mechanism of the reflective knowledge. Views organize reflective objects based on their QoS, structure, location, and topology. Each view has associated strategies that implement the logic necessary to take decisions. For example, strategies identify the most appropriate system's component to execute a service based on its QoS or on its location. Strategies depend strongly on the application domain, thus they have not been inserted into views, but represented through a separate class and associated to views.

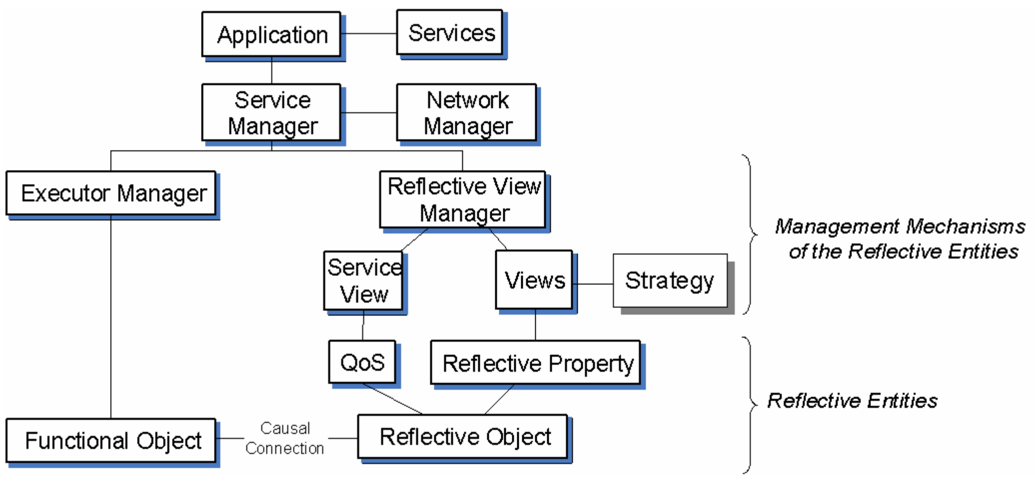

Figure 1. An Adaptive and Reflective Architecture

To explain how our architecture exploits its reflective elements to achieve adaptivity we describe (see Figure 1) how it chooses the most appropriate component to execute a service characterized by specific QoS and properties. Users' requests of services and their QoS/properties arrive to the service manager, which may interrogate both the local ARM node and/or the remote ARM-enabled nodes. Once the service manager identifies the type of request (i.e., service execution, inspection of the available services, etc.), the request arrives to the reflective view manager. Based on the service required and its QoS/properties, the reflective view manager determines how to better organize the reflective knowledge to search for the most appropriate component that provides the service. It creates dynamically views on the reflective knowledge to address efficiently a request. Views identify the most appropriate component to execute a service based on their own semantic. By composing the partial results provided by each view, we obtain the best local component, which is further compared with the remote results provided by the network manager in order to obtain the best overall solution. Eventual modifications performed on the identified reflective entities for the execution of the requested services are propagated to the functional objects through a causal connection [9] mechanism. Finally, the service manager requires the executor manager to execute the service exploiting the components identified by the reflective view manager.

3. Evolution of ARM-enabled Systems Exploiting Architectural Reflection

It is well known that supporting the activities involved in software evolution is a very expensive task. Hence developing software architectures or environments which enable the development of software easier to change, extend and adapt itself to new requirements or contexts is certainly of great relevance for software evolution and maintenance.
Continuous adaptation is one the well known Lehman's laws of software evolution [7] and certainly one of the main aims of our software middleware for adaptive information systems, where adaptation can occur at different levels and at different steps during the development lifecycle. In the following, we focus on the advantages provided by our approach from the software evolution point of view by considering the main Lehman's laws.

Continuous adaptation refers to the ability of a system to address new requirements and changes. In ARM evolution aspects may regard the representation and/or the management of the reflective knowledge. The representation can be easily extended or changed because reflective objects capture only the state of a system component, while QoS and properties are modeled as separate entities. Furthermore, QoS and properties can be modified, added or removed independently because they depend only on the underlying system's components and features.
The management of the reflective knowledge is performed through two mechanisms: views and strategies. Views organize reflective knowledge based on various semantics, each one capturing an independent and orthogonal aspect of the reflective entities. Each view has its own strategies, which implement the policies to choose the most appropriate component to execute a service. Views and strategies can be modified, added or removed independently of other software entities.
In addition, the causal connection between the base and meta levels ensures the consistency between the functional, the reflective and the adaptive part of the architecture.
Separation of concerns [9], the fundamental requirement of reflection, is achieved: reflective knowledge (Reflective Objects) is separated from the base knowledge (Functional Objects). The reflective layers provide only non-functional information about the system being causally connected with the physical layer which provides its functional information. In this way overall change is avoided due to the fact that modifications within the reflective layers cannot modify the functionalities of a system, it can only influence its performances

Increasing complexity as a consequence of the system evolution states that changes in a system lead to the modification of its structure and, implicitly, to an increase of its complexity.
We assert that the complexity of our architecture does not change during its evolution. Its skeleton is composed of five main elements: reflective entities, QoS, properties, views, and strategies. A modification of the reflective and adaptive part of the system regards one or more of these elements. They are modeled independently by separate entities, hence modifications are made separately on each type of element. Their changes cannot increase the overall complexity. In our approach, reflective knowledge is managed through strategies hence modifications at the reflective layers should not cause modifications at the application layer. The addition of further QoS or properties, or views or strategies maintains complexity unchanged. For example, strategies are implemented exploiting the Strategy design pattern [8], hence a new strategy means the addition of a new object. In this case evolution is translated into an increase of the number of objects.

Continuing growth regards the continuously increase of the functionality offered by a system to maintain user satisfaction.
The functionality of the reflective and adaptive part of our architecture is to identify the proper system's components to satisfy service requests. Due to the fact that the reflective knowledge is causally connected to the base entities, when additional services are added to the system they are reified at the meta-level, too. To improve the functionality of the reflective part, various properties and views can be added, for example, a property that specifies the cost of a service or its provider, hence we can have a view on the reflective knowledge based on the newly added property.

The continuing growth does not lead to a *declining quality*, because the reflective layers maintain their primary structure. To maintain or improve the quality of the reflective layers old properties and views can be replaced by new once. New organizations of the reflective objects do not lead to a re-engineering of the reflective layers, the main mechanisms of the representation and management of the reflective knowledge remain unchanged. This ensures implicitly both the *organizational stability* and the *conservation of familiarity* with the reflective layers.

3.1. ARM's Design Issues Improving Evolution

In this section we introduce further aspects which contribute to the evolution and maintenance of ARM.
As previously mentioned, views organize reflective entities based on various semantics according to QoS and properties. From the evolution point of view they provide at least two main advantages: the possibility to extend the number of views on the reflective objects, hence to improve the system's adaptivity, and to represent and exploit information such as location, costs, topology, providers in the adaptivity process together with the QoS of the reflective entities. Note that a reflective entity may be used in various views, but it has only one representation in the system, each view containing a list of references to the entities it manages. In this way consistency among views is implicitly achieved. Based on its semantic, a view associates a score to its reflective entities. The reflective entity with the highest score represents the most appropriate one to provide the service claimed by the current request.
To further improve its evolution and maintenance several design patterns [8] have been applied:

- *chain of responsibility pattern* to implement the service view; it addresses two main problems: the dynamic control of a collection of service views, and the management of complex services (i.e., a *send e-mail service* may be seen as a composition of two elementary services *type* and *send e-mail*);
- *composite pattern* to implement the structure of strategies of the service view; the strategy analyzing a complex service is a composition of strategies related to the elementary sub-services of the complex one; this improves significantly the implementation of strategies by requiring the definition of the elementary strategies, and the definition of the complex once as the composition of the already defined strategies; modifications of elementary strategies are automatically propagated to the complex once;
- *strategy pattern* to implement the policies based on which views assign scores and choose the most appropriate entity for a service request;
- *observer pattern* to implement the causal connection mechanism; this pattern provides an efficient mechanism for the synchronization between objects.

Trying to accomplish a well-defined delimitation of the various aspects of an adaptive and reflective architecture/system, we implicitly achieve maintainability, reusability and integrability.

4. Current and Further Work

In this paper we briefly tried to describe how systems developed through ARM and hence exploiting architectural reflection are easier to maintain. Evolution of these systems, as the capacity to adapt themselves to environment and requirements changes, is largely improved.
Reflection is a key feature for *architecture centered evolution:* all the architectural relevant changes made at the architectural level have to be reflected at the code level, assuring synchronization between the two levels.
Further work will focus on other important aspects such as resource negotiation and allocation as well as mechanisms to choose dynamically strategies based on the application domain. We would like also to explore, as outlined in [10], how reflective modeling of software architectures can support run-time adaptive software evolution.
ARM has been developed by extending the software architecture designed during the MAIS (Multichannel Adaptive Information Systems) Project [11].

References

1. F. Arcelli, C. Raibulet, F. Tisato, M. Adorni, “Architectural Reflection in Adaptive Systems”, Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE’2004), Banff, Alberta, Canada, June, 21st-24th, 2004, pp. 74-79.
2. W. Cazzola, A. Ghoneim, G. Saake, “Software Evolution through Dynamic Adaptation of Its OO Design”, Objects, Agents and Features, LNCS 2975, Springer-Verlag, 2004, pp. 67-80
3. Cazzola, W. Sosio, A. Savigni, A., Tisato, F.: Architectural Reflection. Realising Software Architectures via Reflective Activities. In Proceedings of the 2nd International Workshop on Engineering Distributed Objects, LNCS, Springler Verlag (2000) 102-115
4. J. Dowling, V. Cahill, "Dynamic Software Evolution and the k-Component Model", Proceedings of OOPSLA 2001 Workshop on Software Evolution, 2001
5. Elianssen, F., Andersen, A., Blair, G. S., Costa, F., Coulson, G., Goebel, V., Hansen, O., Kristensen, T., Plagemann, T., Rafaelsen, H. O., Saikoski, K. B., and Weihai Yu. Next Generation Middleware: Requirements, Architecture, and Prototypes. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS’99),* 1999, 60-65.
6. P. Ebraert, T. Tourwe, “A Reflective Approach to Dynamic Software Evolution”, Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution, 2004, pp. 37-44
7. M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, W. M. Turski, “Metrics and Laws of Software Evolution – The Nineties Views”, Proceedings of the 4th International Symposium on Software Metrics, IEEE CS Press, 1997, pp. 20-32
8. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: elements of reusable object-oriented software, Addison Wesley, Reading MA, USA, 1994
9. P. Maes, “Concepts and Experiments in Computational Reflection”, Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA’87), 1987, pp. 147-155.
10. H. Masuhara, A. Yonezawa, "A reflective Approach to support software evolution", Proceedings of International Workshop on the Principles of Software Evolution, 1998, pp.135-139.
11. MAIS Project – www.mais-project.it
12. OMG Adopted Specification. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. ptc/2004-06-01, http://www. omg.org, 2004.
13. C. Raibulet, F. Arcelli, S. Mussino, M. Riva, F. Tisato, L. Ubezio, “Components in an Adaptive and QoS-based Architecture”, Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2006), 2006
14. S. Rank, “Architectural Reflection for Software Evolution”, Proceedings of the 2nd Workshop on Reflection, AOP and Meta-Data for Software Evolution, 2005