Raising a Reflective Family

Nelly Bencomo, Gordon Blair
Computing Department, InfoLab21, Lancaster University, Lancaster, LA1 4WA, UK
nelly@acm.org, gordon@comp.lancs.ac.uk

Abstract

This paper gives an outline of our research on how to combine modelling and meta-modelling to methodically generate reflective middleware family configurations. From our partial results we have identified some research challenges and questions that bring together the areas Model-Driven Software Development, Reflection, and Aspect Oriented Software Development.

1. Introduction

Adaptability is one of the main goals of middleware platforms. In order for middleware platforms to be adaptive, their services and properties need to support a wide variety of deployment environments, application domains, and QoS properties. At Lancaster University, our middleware research group has been investigating how to combine the notions of components (language-independent units of dynamic deployment), component frameworks (collections of components that address a specific area of concern and accept additional plug-in components) [10], and middleware families (abstract collections of component frameworks that are tailored to specific application domains and deployment environments). We apply our research approach [1] in terms of both configuration (i.e. establishing an initial set of components in a target deployment environment, and reconfiguration (i.e. making changes to the initial set of components at runtime). Reflection is applied to discover the current structure and behaviour of the component configurations, and to allow selected changes at run-time for dynamic adaptation [7]. The end result is a flexible middleware platform that can be straightforwardly specialized to a wide range of domains including multimedia, embedded systems [2, 4], and mobile computing [5].

New challenges have emerged from working with this platform. Middleware developers deal with a large number of variability decisions when planning configurations. These include decisions in design, component development, integration, deployment and even at run-time. This large number makes it error-prone to manually guarantee that all the decisions are consistent. Such *ad hoc* approaches do not offer formal foundations for verification that the ultimately configured middleware will offer the required functionality.

To address these issues, we are currently investigating the use of Model-Driven Software Development (MDSD) techniques. MDSD is a new paradigm that encompasses domain analysis, meta-modelling and model-driven code generation. We believe that MDSD has great potential for systematically generating configurations of middleware families. In this paper we summarize our approach and partial results and identified some research challenges and questions.

2. 2. Our approach: a summary

We have specified a kernel UML meta-model that embraces the set of fundamental concepts of our approach [1]. All middleware family members regardless of their domain share this minimum set of concepts. On top of this, we propose a set of extensions (the *caplet extensions* and the *reflective extensions*) which capture the extensibility characteristics of our underlying component model and which can be (un)plugged at deployment or run-time as appropriate. The role of the caplet extension is to provide structured support for extensibility at the deployment environment level in terms of pluggable extensions that are used for a variety of purposes including supporting heterogeneous programming languages and sandboxing. Three reflective extensions are supported *architecture*, *interface*, and *interception*. The *architecture extension* provides a metalevel representation of the architecture in terms of components, connections, and architectural styles rules and enables the discovery and adaptation of the architecture of component frameworks. The *interface extension* supports the dynamic discovery of component interfaces and the dynamic invocation of methods in those interfaces. Finally, the *interception extension* supports the dynamic interception of incoming method calls on interfaces and also the association of pre and post-method-call behaviour.

In short, the extensions provide (i) structured support for extensibility at the deployment

environment level that is used for a variety of purposes supporting heterogeneous programming languages and hardware platforms, and (ii) generic support for target system reconfiguration through the use of architecture, interception, and interface reflective extensions.

Different middleware configurations are generated from the models and meta-models specified. The particular domain and required functionality will determine the primary configuration. The ultimate concrete configurations are determined by three dimensions of variability we have identified: (i) QoS, (ii) deployment environment, (iii) degree of (re)configurability.

The QoS dimension allows the abstract-to-concrete mapping to be influenced by considerations such as mobility (e.g. whether the components should be able to migrate), dependability (e.g. whether certain components should be replicated), or security (e.g. whether certain components are allowed to dynamically load other components). For example, consider an application with a QoS requirement for mobile code. In the generated configuration of components, this will indicate the inclusion of some component framework extensions. These extensions are needed to support sandboxing of untrusted components, and for provision of specialised loaders that are able to load remote objects. It will also indicate the inclusion of a security component framework to validate the remote components. All of this machinery will be transparently instantiated without having to be explicitly present in the UML model.

The deployment environment dimension refers to the resource capabilities of the hardware/software environment in which the system will be deployed. Consider, for example, a distributed application that is deployed in a heterogeneous environment consisting of PCs, PDAs and resource-poor sensor motes [11]. While it would be unproblematic to deploy the whole of the reflective extensions package on the PCs and maybe the PDAs, this may not be possible on the sensor motes where perhaps only components related to the kernel should be deployed. This would preclude the use of more elaborated and complex extensions on the motes and thus restrict the functionality available in that environment. We are currently working on the design of specific middleware configurations addressing embedded systems domains where extremely resource-constrained environments are found [2].

Finally, the configurability dimension refers to the degree of reflective support that will be required at runtime. This essentially determines which of the reflective extensions will be instantiated. For example, if performance monitoring for QoS purposes is required, the interception reflective extension would be included but not the others. Alternatively, if the application might need components to be added or replaced at runtime, the architecture reflective extension would be needed. Another example might be the need for high configurability in a poorly resourced deployment environment; this might indicate the selection of a 'lightweight' architecture reflective extension.

The above examples raised the possibility of multiple dimensions potentially cross-cutting each other (i.e. QoS and configurability). Such cross-cutting is expected to be a common occurrence. Aspect Oriented Software Development (AOSD) offers techniques that may help us address this problem.

3. 3. Research Challenges and Questions

As stated above, we believe that MDSD has great potential in systematically generating configurations of middleware families. However, achieving this systematic support requires that we address a number of research challenges and questions:

- **Crosscutting of the variability dimensions**: we plan to investigate how solutions for the crosscutting problems we described above can be found in the area of AOSD. In this sense, AOSD techniques should also be applicable and defined at the design level using UML as computational reflection has been applied and defined in our approach [1]. How do we represent the crosscutting concerns identified above (i.e. aspects) in the proposed metamodels?
- **AOP and reflection as complementary techniques**: AOP and reflection can be seen as complementary techniques. One possible approach for implementing AOP mechanisms is reflection [3, 9]. In our specific case we see that composition filters might be used as an approach for implementing the causal connection between the meta-level and base-level related to the interception reflective extension. AOP offers a useful abstraction principle to structure the meta-level for complex systems such as middleware [5]. To what extent do AOP and reflection overlap? How can we model this synergy between AOP and reflection?
- **Model-driven techniques**: several model-driven techniques have been proposed. One very well known is OMG Model Driven Architecture (MDA). Currently the main focus of MDA is on the design of distributed applications and how to map them to specific middleware technologies. A Platform Independent Model (PIM) is said to be portable to a number of Platform Specific Platforms (PSMs) or

target middleware platforms. Our focus is different. What we address is the generation of the middleware platform itself. MDA-style transformations might be applied to the middleware. In this sense, we see promise in the use of MDA concepts in the generation of middleware families. Let's study two working examples: (i) the case were reflective middleware configurations are deployed in a heterogeneous environment consisting of PCs, PDAs, and resource-poor sensor motes [11], (ii) the situation were several configurations provide a broad range of dynamic middleware-level communication services like binding mechanisms to cope with the different interaction paradigms as found in mobile environments [7]. In example (i), we might think of a generic model of the middleware as the PIM and the different generated middleware configurations for the different environments as the PSMs. In the case of the example (ii), the generic model of the binding mechanism corresponds to the PIM and the configurations that match the broad range of interaction paradigms in mobile environments correspond to the PSMs. In this sense, what are the implications of applying MDA concepts to the middleware itself?

- **UML models at runtime**: a key area of future work is to investigate how to maintain the UML models at runtime and to keep them causally connected with the underlying running system in order to support reconfiguration. How should a runtime UML model look like? How do we maintain the causal connection?

Answers and solutions to these questions and problems would lead to mechanisms that add new capabilities to our middleware platform in a cost-effective manner.

Acknowledgement: This research is part-financed by the RUNES project. RUNES is supported by research funding from European Commission's 6th framework Programme under contract number IST-004536.

4. References

1. Bencomo, N., Blair, G.S., Coulson, G., Batista, T., "Towards a Meta-Modelling Approach to Configurable Middleware", Proc. 2nd ECOOP'2005 Workshop on Reflection, AOP and Meta-Data for Software Evolution, Glasgow, Scotland, July 2005
2. Blair, G., Coulson, G., Grace, P.: Research Directions in Reflective Middleware: the Lancaster Experience, Proc. 3rd Workshop on Reflective and Adaptive Middleware (RM2004), (2004), 262-267
3. Blair G., Blair. L., Rashid A. Moreira A., Araújo J., Chitchyan R.: Aspect-Oriented Software Development, chapter 17 - Engineering Aspect-Oriented Systems, pages 379-406. Addison-Wesley, 2005
4. Costa, P., Coulson, G., Mascolo, C., Picco, G.P., Zachariadis, S.: The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems, Proc. of PIMRC05,(2005)
5. Costa, F.: Combining meta-information management and refection in an architecture for configurable and reconfigurable middleware. Ph.D. Dissertation, University of Lancaster, (2001)
6. Gabriel R., Bobroe D., White J., CLOS in Context – The Shape of the Design Space, in Object-Oriented Programming – the CLOS perspective, Chapter 2, MIT Press, 1993, 29-61
7. Grace P., Blair G. Samuel S.: "ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability". Proc of International Symposium on Distributed Objects and Applications (DOA), (2003)
8. Maes, P., "Concepts and Experiments in Computational Reflection", Proc. OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, pp147-155, ACM Press, 1987
9. Bouraqadi N., Ledoux T.: Aspect-Oriented Software Development, chapter 12 - Supporting AOP using Reflection, pages 261-282. Addison-Wesley, 2005
10. Szyperski C.: Component Software: Beyond Object-Oriented Programming, Addison-Wesley, (2002)
11. http://www.moteiv.com/