Preening: Reflection of Models in the Mirror
a Meta-Modelling Approach to Generate Reflective Middleware Configurations

Nelly Bencomo[1]

Comp. Dept, InfoLab21, Lancaster University, Lancaster, LA1 4WA, UK
nelly@acm.org

Abstract. This paper outlines my PhD research in the area of Model-Driven Software Development applied in the generation of OpenCOM-based Reflective Middleware family configurations. These configurations might address different domains but will follow the main foundation concepts: components, components frameworks and reflection for dynamic (re)configuration. A Kernel Metamodel for the Reflective Middleware family is proposed and the UML specifications of Reflective Metamodels as extension of the Kernel are presented. As a result at runtime the meta-objects are optional dynamically (un)loaded when required. In particular this paper describes how metamodels and models specify the causal connection between the base and meta-level. From the proposed models a range of research opportunities are stated at the end of the paper.

1. Introduction

Metalevel architectures and the concept of reflection are useful for modifying programming systems dynamically in a controlled way [10]. A number of experimental reflective middleware platforms have been developed and used in industry. At Lancaster University, we have investigated the role of reflection [9,11] in supporting both customisation and dynamic re-configuration of middleware platforms. To complement the use of reflection, we also investigate the use of component technologies, and the associated concept of component frameworks [12] in the construction of our open middleware solutions.

Challenging new requirements have emerged when working with dynamically reconfigurable component frameworks with (un)pluggable components. Middleware developers deal with a large number of variability decisions when planning configurations at various stages of the development cycle. These include decisions in design, component development, integration, deployment and even at run-time. These factors make it error-prone to manually guarantee that all these decisions are consistent. Such *ad hoc* approaches do not offer formal foundations for verification that the ultimately configured middleware will offer the required functionality.

[1] PhD work being developed at Lancaster University under the supervision of Prof. Gordon Blair

Our main research topic focuses on whether Model-Driven Software Development (MDSD) techniques can be successfully used to address the challenges described above and what are the implications of its application. MDSD is a new paradigm that encompasses domain analysis, metamodeling and model-driven code generation. We believe that Model Driven techniques provide a key solution when systematically generating configurations of the Middleware families.

2. Background: Reflective Middleware at Lancaster University

The reflective middleware research [14] carried out for more that eight years at Lancaster University is based on three key concepts: *components, components frameworks and reflection*. At Lancaster both the middleware platform and the application are built from interconnected sets of components. The underlying component model is based on OpenCOM [4], a general-purpose and language-independent component-based systems building technology. OpenCOM supports the construction of dynamic systems that may require run-time reconfiguration. It is straightforwardly deployable in a wide range of deployment environments ranging from standard PCs, resource-poor PDAs, embedded systems with no OS support, and high speed network processors. Components are complemented by the coarser-grained notion of *component frameworks* (CFs) [12]. A CF is a set of components that cooperate to address a required functionality or structure (e.g. Discovery and Advertising, Security, Interactions, etc). CFs also accept additional 'plug-in' components that change and extend behaviour. Many interpretations of the CF notion foresee only design-time or build-time plugability. In our interpretation run-time plugability is also included, and CFs actively police attempts to plug in new components according to well-defined, per-CF, policies and constraints that are expressed in a language such as OCL [13].

Reflection is used to support introspection and adaptation of the underlying component/ CF structures [1]. A pillar of our approach to reflection is to provide an extensible suite of orthogonal meta-models each of which is optional and can be dynamically loaded when required, and unloaded when no longer required. The motivation of this approach is to provide a separation of concerns to reduce the complexity of the overall metalevel. Three reflective meta-models are now supported:

The architecture meta-model represents the current topology of a composition of components within a capsule; it is used to inspect (discover), adapt and extend a set of components. For example, we might want to change or insert a compression component to operate efficiently over a wireless link. This meta-model provides access to the implementation of the meta-component that has a component graph where components are nodes and bindings are arcs. Inspection is achieved by traversing the graph, and adaptation/extension is realized by inserting or removing nodes or arcs.

The interface meta-model supports the dynamic discovery of the set of interfaces defined on a component; support is also provided for the dynamic invocation of methods defined on these interfaces [1]. Both capabilities enable the invocation of interfaces whose types were unknown at design time.

The Interception meta-model supports the dynamic interception of incoming method calls on interfaces and also the association of pre- and post-method-call code [1]. The code elements that are interposed are called interceptors. For example, in the above wireless link scenario we might want to use an interceptor to monitor the conditions under which the compressor should be switched.

3. Progress to date

The main topic of this PhD research is the use of Model Driven techniques to systematically generate configurations of the OpenCOM-based Reflective Middleware family. The first phase of this research has the following goals (i) to design a metamodel that captures the implications of using reflective architectures, (ii) to investigate the generation of specific members of the family and the tool support.

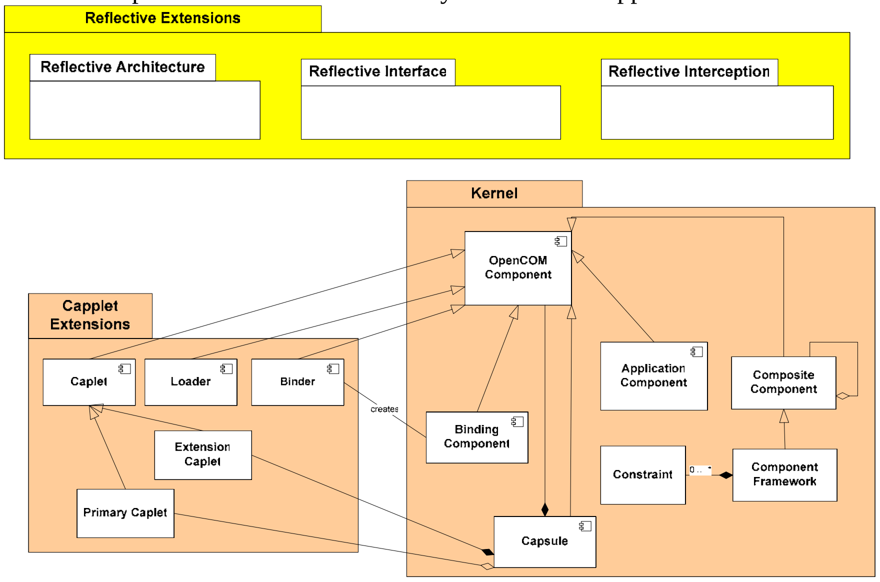

Figure 1. The OpenCOM Metamodel

Figure 1 shows the three packages that comprises the OpenCOM metamodel. The Kernel package concepts are extended by the Reflective and Caplet packages. The metamodel includes the fundamental model elements of OpenCOM such as OpenCOM Component, Capsule, Binding Components, Composite Components, and Component Frameworks from the *Kernel Package*, and Caplets, Loaders, and Binders from the *Caplet Extensions Package*. The reflective metamodels Architecture, Interface and Interception are also represented as part of the *Reflective Extensions Package*. A particular contribution of the proposed models is the specification of how the Reflective Metamodels extend the *Kernel* and Caplet packages. As a consequence in Implementation the meta-objects are optional and can be dynamically loaded/unloaded when required.

Different Middleware configurations might be generated, for example, (i) configurations that address specific domains such as tiny Embedded Systems where resources are constrained and reflection capabilities might be minimised to avoid overhead and (ii) configurations that exploit computational reflection to provide a broad range of dynamic middleware-level communications services like binding mechanisms to cope with the different styles of service interaction paradigms (e.g. Remote Procedure Call, Publish-Subscribe and tuple spaces).

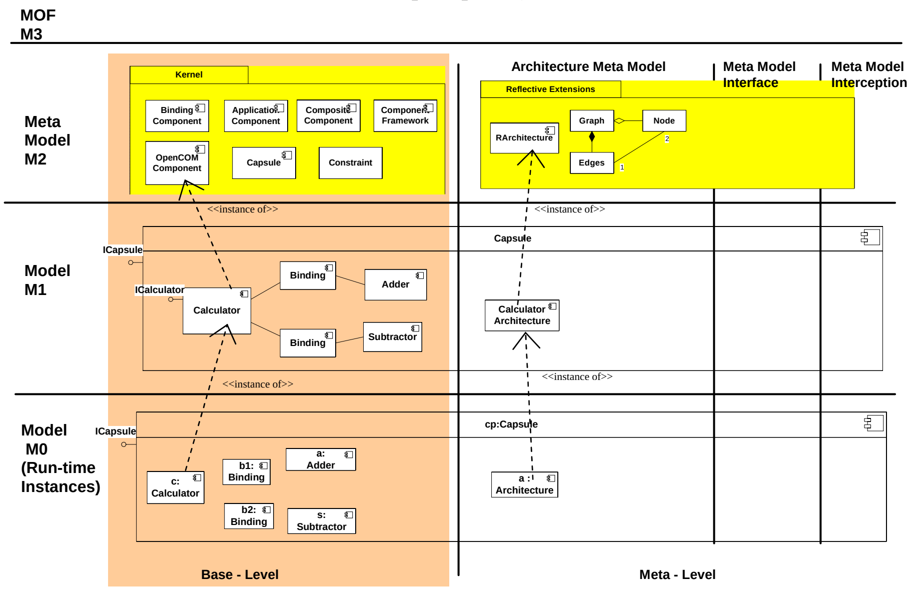

Figure 2. UML Models and Reflective Architecture of a Calculator Configuration

Figure 2 illustrates the configuration for a simple Calculator example: there are three components; an adder, a multiplier and a calculator within a Capsule (container). The calculator component offers the services of adding and multiplying based on the services of the adder and multiplier components. There are two key OpenCOM concepts to explain; Capsules and Bindings. Capsules are containing entities that offer a runtime API. Components can be (un)loaded into a capsule at runtime using the API. Bindings are associations between a single interface and a single receptacle. The horizontal layers correspond to the four-level UML hierarchy and the vertical layers correspond to the base-level and meta-level of the reflective architecture hierarchy. Both hierarchies are orthogonal. Figure 2 shows the OpenCOM metamodel (M2), the model of the application Calculator (M1) and the instances of the application (M0).

Causal Connection

Capsules offer a component runtime-level API [4]. This API mainly offers services to (un)load, instantiate and destroy components and bind interfaces and receptacles. It offers as well the *notifyCallback* that acts as a primitive to give support to reflective metamodels. Relevant operations in the API offered capsules are as follows:

struct load(comp_type name);
status unload(struct t);
comp_inst bind(ipnt_inst interface, ipnt_inst receptacle);
status notifyCallback();

When the callback is registered every subsequent API call (bind, load, etc) is reported to the callback. The callback will keep the architecture meta-model updated with all the information associated with the internal topology of the capsule contents. In the case that the meta-model changes its configuration, it will invoke the respective operation bind, load, unload in the API. This way, the causal-connection relation between the base and the meta level is maintained. This generic behaviour is specified by a Sequence Diagram in the Metamodel. Thus, different models at the M1 layer will reuse or instantiate such diagrams. These sequence diagrams are part of the dynamic behaviour specification of OpenCOM. The model elements shown in Figure 2 are part of the static structure of the models. For reasons of space these sequence diagrams are not shown in this paper.

4. Related Work

The application of MDSD to middleware platforms is attracting a considerable amount of research interest. Significant related work has been done in the context of the CoSMIC Project [18]. CoSMIC had common goals with the work described in this paper. However, the idea of generating middleware family configurations with optional capabilities (extensions) that can be (un)plugged at deployment or run-time (caplets and reflective extensions) is unique to our work. The modelling and generation of reflective capabilities is a crucial part of our research.

5. Conclusions and Future Work

A core contribution of this doctoral research is the design of a set of Metamodels for the specification of our OpenCOM-based Reflective Middleware family. The metamodels have captured the main concepts of the design philosophy of our middleware platform and the relationship among them. The proposed metamodels have revealed a range of concrete approaches and techniques of how to use MDSD to generate appropriate component and CF machinery.

Several model-driven techniques have been proposed. One very well known is OMG Model Driven Architecture (MDA) [16]. 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 [19], (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. We are now investigating how to generate code from OpenCOM-based models to different middleware configurations while keeping decisions that are generic to a set of configurations at the metamodel level design. As part of the first phase of the research we plan to identify the variability among the related configurations of the family to support an efficient generation of configurations. Some partial results are shown in [2,3]. For the second phase of our research we will investigate how to model and generate configurations dealing with cross-cutting concerns characteristic of middleware platforms [2].

Acknowledgments. Special thanks to the Middleware Group at Lancaster University but specifically to my supervisor Gordon Blair, Geoff Coulson, Paul Grace, Thais Batista, and Pete Sawyer. 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.

References

1. Blair, G., Coulson, G., Grace, P.: Research Directions in Reflective Middleware: the Lancaster Experience, Proc. 3rd Workshop on Reflective and Adaptive Middleware (RM2004), Canada, (2004), 262-267.
2. Bencomo N., Blair G.: Raising a Reflective Family, Models and Aspects - Handling Crosscutting Concerns in MDSD, ECOOP, Scotland, 2005
3. Bencomo N., Blair G., Coulson G., Batista T.: Towards a Meta-Modelling Approach to Configurable Middleware, 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, ECOOP, Scotland , 2005
4. Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., Ueyama, J.: A Component Model for Building Systems Software, Proc. IASTED Software Engineering and Applications (SEA'04), USA, (2004)
5. Dunkels A, Grönvall B., Voigt T.: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In Proc. of First IEEE Workshop on Embedded Networked Sensors, USA, (2004)
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. Kleppe A., Warmer J., Bast W.: MDA Explained The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003
9. Maes, P.: Concepts and Experiments in Computational Reflection, Proc. OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, pp147-155, ACM Press, 1987.
10. Okamura H., Ishikawa Y., Tokoro M.: Metalevel Decomposition in AL-1/D, Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software (1993), 110-127
11. Smith B.: Reflection and Semantics in a Procedural Language. PhD thesis, MIT Laboratory of Computer Science, 1982
12. Szyperski C.: Component Software: Beyond Object-Oriented Programming, Addison-Wesley, (2002)
13. Warmer J., Kleppe A: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Publishing Company, 1999
14. Middleware at Lancaster: http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/index.php
15. AspectJ Project: http://eclipse.org/aspectj
16. OMG Model Driven Architecture: http://www.omg.org/mda/
17. The JBoss Project: http://www.jboss.org/index.html
18. Cosmic Project: http://www.dre.vanderbilt.edu/cosmic/
19. Wireless Sensor Networks Devices: http://www.moteiv.com/