Towards a More Effective Coupling of Reflection and Runtime Metamodels for Middleware *

Fábio M. Costa, Lucas L. Provensi, Frederico F. Vaz

[1]Institute of Computing – Federal University of Goiás (UFG)
74690-815 – Goiânia – GO – Brazil

fmc|lucas|frederico@inf.ufg.br

Abstract. *Reflection plays an important role in the flexibilisation of middleware platforms. Through dynamic inspection, middleware interfaces can be discovered and invoked at runtime, and through adaptation the structure and behaviour of the platform can be modified on-the-fly to meet new user or environment demands. Metamodeling, on the other hand, has shown its value for the static configuration of middleware (and other types of system in general). Both techniques have in common the pervasive use of meta-information as the means to provide the self-representation of the system. However similar they are, these two techniques usually fall on different sides of a gap, namely development time and runtime, with little interplay between them. In this position paper, we review an approach for the combination of reflection and metamodeling, as well as the main applications we envisage in the context of middleware platforms. The main goal is to highlight the importance of such combination, contributing to a wider discussion of the topic during the workshop.*

1. Introduction

Meta-information is at the core of both reflection [Maes 1987] and metamodeling [Odell 1995] techniques. It is the means through which the reified features of a base-level system (such as a middleware platform) are represented in reflective architectures. It is also the reason for metamodeling techniques to exist, that is, to represent meta-information in a consistent way. Although metamodeling usually deals with meta-information in a well-structured way, reflection typically handles it in an ad hoc fashion. On the other hand, while metamodeling is traditionally limited to the static (*cf.* design time) representation of meta-information, reflection enables its dynamic use and evolution. It thus seems natural to combine the two techniques, enabling the dynamic use of well-structured meta-information.

*This work was funded by CNPq-Brazil (the Brazilian Government's agency for the promotion of scientific and technological development), grants number 478620/2004-7 and 506689/2004-2.

In the approach advocated in this paper, the structures of meta-information represented in a metamodel are kept accessible at runtime. As such, they can be used both at static configuration time, before the system is put to run, and at runtime, as the basis for the reflective meta-objects to construct the representation of the reified base-level system. While other approaches can be found in the literature for the combination of metamodeling and reflection, such as in [Yoder and Razavi 2000] and [Bencomo et al. 2005], we believe that the approach discussed here is a more natural fit, besides its potential to be applied in conjunction with those other approaches.

This position paper discusses the basic ideas of our approach and presents some of its concrete applications in the context of middleware platforms. It also discusses current work aiming to refine the metamodeling technique in use, as well as some areas of application. A brief comparison with other approaches from the literature is also presented, aiming at future collaborations with those research efforts in order to further advance the state of the art in the field.

2. Basic Approach

2.1. Reflection

The adopted approach to reflective middleware is based on the Lancaster Open ORB project [Blair et al. 2001]. The architecture is clearly divided into a *base-level*, which contains the usual middleware functionality (such as remote binding, remote method execution, and object references), and a *meta-level*, which provides the reification of the base-level features. Both the base- and meta-level are defined in terms of a uniform component model, which facilitates the identification, at runtime, of the several functional elements of the platform. The meta-level is accessed through meta-interfaces that expose a well-defined meta-object protocol for both inspection and modification (a.k.a. adaptation) of the base-level. Furthermore, in order to cope with complexity, the meta-level is further divided into four meta-space models: *interface* (for the discovery of an interface's methods and attributes, as well as for dynamic invocation); *interception* (for interposing extra behaviour in the pre/post-processing of interactions); *architecture* (for the manipulation of the component configuration of the platform); and *resources* (which allow the underlying resources, such as memory and processor, to be inspected and reconfigured). Critically, all these meta-space models are concerned with

particular kinds of meta-information about the features of the base-level system, although not necessarily in a structured or unified way.

2.2. Metamodeling

Metamodeling is achieved through the four-level approach of the Meta-Object Facility (MOF) [OMG 2000]. The first level (level 0) represents the actual system entities, while, level 1 contains the model from which those entities were instantiated. Level 2 then consists of the meta-model, which defines a language for describing models, while Level 3 is the core language for describing meta-models. In the context of metamodeling for middleware, we are particularly interested in levels 2 and 1, which represent, respectively, the platform's type system and the actual definitions of the (types and templates of the) entities that comprise particular middleware configurations. Using a metamodeling tool such as the MOF or, more recently, the Eclipse Modeling Framework (EMF) [Budinsky et al. 2004], a middleware metamodel can be defined and customised, allowing the automatic generation of tools (such as a repository) that facilitate the definition and storage of types and templates. Finally, from the type and template meta-information items that are kept in the repository, generic factories can be used to instantiate concrete platform configurations (level 0).

The implementation of this meta-information management tool in EMF is currently under work. When using EMF, it is worth noting that this technology is mainly meant for model-driven development, where a model of a system is defined and tools are used to automatically generate code for the system. That is, metamodeling is not a main intent of EMF. However, the distinction between a model and a metamodel is only a matter of reference point, that is, a model can be taken as the model of a another model, instead of directly modeling a level 0 system. In our case, thus, the metamodel of a middleware platform is defined as a usual core model in EMF. Then, the code that is generated from this model (extended with a few customisations) is actually the implementation of a meta-information repository to store definitions of custom middleware components. This repository contains facilities for creating, accessing, deleting and evolving (see below) such meta-information. The interpretation and use of this meta-information is outside the scope of EMF and is performed, as described above, by external component factories, which are provided in the form of middleware core services. In addition, although EMF is mainly meant to be used within Eclipse, with the exception of the UI editor the generated implementation can be used by standalone applications. Indeed, the generated EMF.edit plug-in can be customised with extensions and used as a true repository of model-related meta-information, which can be accessed by standalone Java applications (such as by the factories mentioned above) through a programmatic interface.

3. Combining Reflection and Metamodeling

The reflection and meta-information approaches described above have been combined in a reflective middleware architecture called Meta-ORB [Costa 2001]. This architecture was first prototyped in Python [Costa 2002] as a reference implementation. It was later reimplemented in Java, using both J2SE and J2ME, aiming at its deployment in portable devices. This latter implementation is called MetaORB4Java [Costa and Santos 2004]. Although a complete description of this platform is outside the scope of this position paper, in the following we describe its main features in what concerns the combination of reflection and metamodeling.

3.1. Building the platform meta-model

The Meta-ORB architecture is actually a meta-architecture for middleware. Its core consists of a type system defining a language of constructs that can be used to define special-purpose entities used to build custom middleware configurations. The main constructs are components (which encapsulate functionality) and binding objects (which encapsulate interaction behaviour). A particular middleware configuration can be built in terms of a composition of components interconnected by binding objects (which are, themselves, defined in terms of component and binding compositions).

This type system is represented as a metamodel in UML and the EMF Eclipse plug-in was used to generate a basic repository implementation. This basic repository consists of the implementation of model elements, with the standard EMF accessor methods, plus editting functions (both programmatic and UI-based). This repository is currently being extended with more elaborate accessor methods, in a way that resembles the CORBA Interface Repository [OMG 2003]. For instance, the *lookup_name* and *lookup_id* methods were introduced which enable the search for type/template definitions based on their short names or fully qualified names, respectively.

3.2. Using the model to instantiate platform configurations

As a representative example of the use of the repository, component and binding definitions are expressed as instances of their corresponding metamodel elements and stored in the repository. This can be easily done using the generated UI-based editor plug-in. Two generic factories were implemented, respectively for the creation of components and binding objects. These factories obtain the necessary meta-information elements from the repository (using the *lookup*-like methods described above) and use them as the blueprint to create concrete, customised, middleware configurations composed of components interconnected by

binding objects. Note that a single type definition can be used to define the whole platform configuration. This can be the definition of a distributed binding object composed of internal components and other, lower-level, bindings. Although this recursive structure is mirrored in the repository (in terms of separate type definitions that contain or reference other type definitions), the factories (the binding factory in this case) only needs to be given the name (or *id*) of the outermost type definition; the type definitions for the internal components and bindings are implicitly obtained by the factories without the intervention of the middleware developer.

3.3. Using the model to instantiate reflective meta-objects

From the moment particular middleware configurations have been instantiated and put to run, they can also be subject to the reflection mechanisms of the platform (see Section 2.1). This means that the components and binding objects that make up a configuration can be inspected and adapted at run time. In order to enable this, the meta-objects that perform the reflection mechanisms need precise meta-information about such components and bindings in order to reify them. The meta-objects obtain this meta-information from the respective component and binding definitions stored in the repository. For instance, in order to reify the internal configuration of a binding object, the architecture meta-object needs to obtain the part of that binding's definition that describes the component graph (where the nodes are component types and the edges are either local or distributed bindings) used to instantiate the binding.

The above subsections illustrate one side of the combination of reflection and metamodeling proposed in this work. The other side is related to the use of reflection to build new component and binding types that are persisted in the repository, and is described next.

3.4. Creating new model elements using reflection

This side of the combined approach refers to the evolution of a model's elements as a result of reflection. More specifically, architectural reflection can be used to dynamically change a component or binding object so that it becomes more suitable to varying operating conditions or user requirements. Often, this process leads to new component and binding definitions that might be useful in other contexts. Therefore, there is a case for making such evolved definitions persistent so they can be reused later. In our approach, these evolved definitions take the form of *versions* of the original component or binding definitions (so as to avoid potential conflicts with other existing definitions) and are stored in the repository alike. Later, they can be retrieved by factories in order to generate new instances of components and bindings that incorporate, from the begining, the adaptations that were made through reflection.

4. Other Applications of the Approach

The integrated management of middleware configuration and dynamic adaptation is a direct benefit of the uniform treatment given to meta-information. More precisely, the same constructs used to statically configure a middleware platform also constitute the primitives through which dynamic adaptation is achieved. Resuts in this respect have already been published elsewhere.

More recently, we are investigating the use of this technique to model other aspects of middleware platforms, in particular: resource management in grid computing middleware, and context-awareness in middleware for mobile computing environments. In the former, an extension of the metamodel is being defined to model the several kinds of resources available in a grid, which will be used as a flexible way to (re-)configure the allocation of tasks to processors. In the latter, a metamodel is being defined to allow the representation of context-related meta-information, which enables context-aware architectural adaptation of the middleware platform (in a similar way as proposed in [Capra et al. 2001]). In both cases, the advantage of using a runtime explicit metamodel is the intrinsic extensibility, which allows new kinds of resource or context meta-information to be seamlessly integrated into the system in a dynamic way. In general, we believe that any kind of meta-information present in a system (such as middleware) can be leveraged by the ability to represent it as a runtime metamodel integrated with reflective capabilities.

5. Concluding Remarks

This position paper has reviewed the main ideas and applications of our approach for the combination of runtime metamodels with reflection in the context of distributed systems middleware. The approach focuses on the integration of the design-time and runtime use of an explicit middleware metamodel and related meta-information with reflective introspection and adaptation. The main benefit of this is the uniform treatment given to static configuration and dynamic reconfiguration of the platform, which are based on the common constructs and abstractions defined in the metamodel.

Other approaches have been proposed in the literature to integrate reflection and metamodeling, such as [Bencomo et al. 2005], which use metamodels to configure the reflective capabilities of a middleware platform. The approach discussed here, however, is different in the sense that the modeled meta-information provides the actual self-representation of the reified base-level entities, instead of being a model of the meta-level. Indeed, we believe that

both approaches can be effectively used in a complementary way.

Although the core ideas of our approach have been around for a while [Costa 2001, Costa and Blair 2000], its potential contribution is still underexploited. We believe that the exploitation of the proposed ideas in mainstream middleware platforms, in conjunction with more ellaborate techniques for architectural reflection and separation of concerns, can be of great benefit for the advancement of flexible and adaptive middleware technologies. It is our goal to contribute to raise this discussion in both the middleware and the modeling communities, aiming to identify new research opportunities in the area of reflective middleware and runtime metamodels.

References

Bencomo, N., Blair, G. S., Coulson, G., and Batista, T. (2005). Towards a meta-modelling approach to configurable middleware. In *2nd ECOOP2005 Workshop on Reflection, AOP and Meta-Data for Software Evolution*, Glasgow, Scotland.

Blair, G. S., Costa, F. M., Saikoski, K., and Clarke, N. P. H. D. M. (2001). The design and implementation of Open ORB version 2. *IEEE Distributed Systems Online Journal*, 2(6).

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., and Grose, T. J. (2004). *Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley.

Capra, L., Emmerich, W., and Mascolo, C. (2001). Exploiting reflection and metadata to build mobile computing middleware. In *Middleware 2001 Workshop on Mobile Computing Middleware*, Dresden, Germany.

Costa, F. M. (2001). *Combining Meta-Information Management and Reflection in an Architecture for Configurable and Reconfigurable Middleware*. Ph.D. thesis, University of Lancaster, Lancaster, UK. http://www.comp.lancs.ac.uk/computing/users/fmc/pubs/thesis.pdf.

Costa, F. M. (2002). Meta-orb: A highly configurable and adaptable reflective middleware platform. In *Proceedings of the 20th Brazilian Symposium on Computer Networks*, pages 735–750, Buxzios-RJ-Brazil. Brazilian Computer Society.

Costa, F. M. and Blair, G. S. (2000). Integrating reflection and meta-information management in middleware. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'00)*, Antwerp, Belgium. IEEE, IEEE.

Costa, F. M. and Santos, B. S. (2004). Structuring reflective middleware using meta-information management: The meta-orb approach and prototypes. *Journal of the Brazilian Computer Society*, 10(1):43–58.

Maes, P. (1987). Concepts and experiments in computational reflection. In *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*, Orlando, FL USA. American Computer Machinery, ACM Press.

Odell, J. (1995). Meta-modeling. In *Proceedings of OOPSLA'95 Workshop on Metamodeling in Object-Orientation*. ACM.

OMG (2000). *Meta Object Facility (MOF)*. Object Management Group, Needham, MA. OMG Document formal/2000-04-03.

OMG (2003). *The Common Object Request Broker: Architecture and Specification*. Object Management Group, Needham, MA USA, rev. 3.0 edition.

Yoder, J. W. and Razavi, R. (2000). Metadata and adaptive object-models. In *ECOOP2000 Workshop Reader*, volume 1964 of *Lecture Notes in Computer Science*. Springer-Verlag.