Management of Runtime Models and Meta-Models in the Meta-ORB Reflective Middleware Architecture [⋆]

Lucas Luiz Provensi, Fábio Moreira Costa, and Vagner Sacramento

Institute of Computing, Federal University of Goiás
Campus Samambaia, UFG, 74690-815, Goiânia-GO, Brazil
{lucas,fmc,vagner}@inf.ufg.br
http://www.inf.ufg.br

Abstract. In the Meta-ORB reflective middleware architecture, runtime models provide the necessary meta-information to instantiate specialized platform configurations and to construct the reflective self-representation of base-level systems. Other kinds of useful meta-information may also be provided by the system's runtime model, such as policies that allow the middleware to adapt itself automatically. Evolving the middleware meta-model and extending its infrastructure to handle new kinds of model-based constructs may be considerably complex and would require re-implementation of several parts of the middleware. In this paper we present an approach for the management of runtime models and meta-models, aiming to simplify the evolution of the middleware so that it can support new kinds of constructs defined in its meta-model.

Key words: runtime models, meta-modeling, reflective middleware

1 Introduction

Distributed computing systems, especially those used for real-time applications, often suffer from dynamic changes in their execution environment and requirements, affecting, e.g., available bandwidth and network loss rates. Reflective middleware has been proposed as a way to handle such changes without disrupting the system or its applications. Such platforms are capable of inspecting and adapting their internal structure and behavior at runtime, providing an independent software layer which is ideal to implement the self-management and self-adaptation capabilities needed to handle dynamic requirements [1].

In this paper we review the Meta-ORB reflective middleware platform [2] and discuss how its evolution can be facilitated by the management of models and meta-models at runtime. Autonomic self-adaptation is proposed as a way to enable transparent middleware evolution, a feature that was not fully supported

[⋆] This work was partly funded by FAPEG, The Research Support Foundation of the State of Goiás, Brazil (Call 02/2007; Process number: 200810267000147), and by the HP Technology for Teaching Higher Education Grant.

in previous versions of the platform. This feature was introduced in the current version of the platform, called MetaORB.NET [3]. It enables new functionality and constructs to be dynamically added to the platform, as well as existing ones to be replaced or removed. The approach is based on the reflective manipulation of model and meta-model elements.

The paper is structured as follows. Section 2 discusses the Meta-ORB meta-model and the use of run-time models to instantiate specialized middleware configurations, as well as their use to build the reflective self-representation of base-level entities. Section 3 discusses middleware evolution in terms of its three-level architecture: meta-model, model and system entities. Section 4 discusses related work and Section 5 presents concluding remarks and future work.

2 Meta-ORB Model and Meta-model

The Meta-ORB architecture combines meta-information management techniques, which enable the definition of specialized platform configurations, with computational reflection, which enables dynamic adaptation. Meta-information, in the scope of our work, describes the structure and semantics of entities in a computational system. This description is used for static configuration of the middleware (by instantiating its components at load time) and for its dynamic adaptation (via runtime component-based reconfiguration).

In the Meta-ORB architecture, meta-information is specified in a model, according to an explicit meta-model. This explicit meta-model represents the platform's type system and is maintained in a repository that can be used for the definition, storage and retrieval of models that represent specialized configurations of the middleware and its applications. Once the definition of an entity (a component and its interfaces, for instance) is obtained from the type repository, it may be used to build a runtime model of the entity, allowing its dynamic instantiation by specialized factories and, if necessary, the construction of its reflective self-representation, used for dynamic introspection and reconfiguration.

The main constructs defined in the meta-model are components, that encapsulate functionality and can interact (and be composed) through well defined interfaces, and binding objects, that encapsulate interaction behavior, and can be defined in terms of internal components and other binding objects (a complete description of all meta-model constructs can be found in [2]). Figure 1 shows the creation of a component, where its definition is obtained from the type repository and parsed into a model that is used by the component factory as a blueprint to instantiate the component and its interfaces. The same process is performed by binding factories to instantiate binding objects.

The same meta-information contained in component and binding definitions can also be used to build the self-representation maintained by meta-objects that reify components and bindings. Figure 2 shows the construction of the self-representation of a binding object. First, the meta-object obtains the definition of the binding from the type repository. This definition contains meta-information that describes the internal configuration of the binding in terms of internal com-

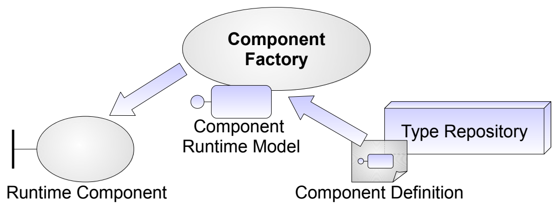

Fig. 1. Creating a component using a runtime model as the blueprint.

ponents and nested bindings. The meta-information is then combined with information maintained by the middleware at runtime, such as the location of each of the endpoints that participate in the binding. The result is compiled into a graph maintained by the meta-object, which contains information about the internal configuration of the binding in each of its endpoints. Once the self-representation is built, meta-objects can be used, via their meta-interfaces, to inspect and adapt the corresponding entities of the base-level system.

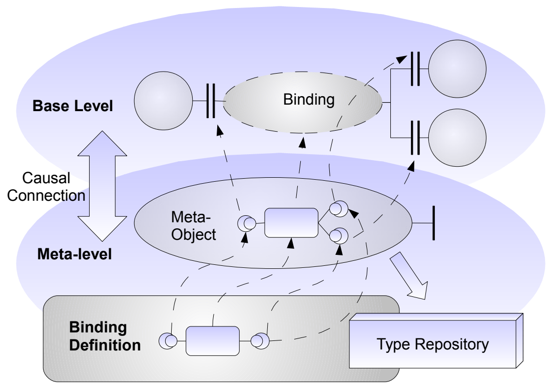

Fig. 2. Building the self-representation for a binding object.

The meta-information used by factories and meta-objects basically refers to the structure of the base-level entities: required and provided interfaces of components and the internal configuration of components and bindings. However, the platform's meta-model allows the addition of other kinds of meta-information to the definition of entities in order to describe their runtime behavior. One example is the definition of QoS constraints, which can be added as annotations to the definition of interfaces. Such annotations may indicate, for instance, the maximum acceptable delay for a media flow passing through the interface. This information in turn can be used in the negotiation process that will determine the type of binding that will be used to connect two or more interfaces.

More recently, the concept of adaptation policy was introduced in the meta-model [3]. As well as QoS constraints, policies describe desirable behavior for the entities they refer to. Policies provide the middleware with meta-information to determine, at runtime, when and how the platform should adapt itself in response to changes in the execution environment or requirements. The introduction of this and other behavioral concepts results in changes to the middleware that affect meta-model, model and base-level entities likewise. However, as such changes are not related to structure, they do not alter the main constructs of the meta-model (components, interfaces and bindings). The management of middleware evolution thus becomes easier, making it possible to apply meta-model changes without disrupting execution, as will be discussed in the next section.

3 Middleware Evolution

Adding new behavioral features, such as self-adaptation, requires several changes to the middleware, comprising three of the four levels described by the Meta-Object Facility (MOF) [4]. First, the middleware meta-model (Level 2) should be extended with constructs to describe the new features. Then, these new features should be incorporated into new and existing middleware models (Level 1). Finally, runtime entities (Level 0) may need to be modified (or created) to interpret the new features that are now part of the runtime model.

As an example, Figure 3 illustrates the changes made to Meta-ORB in order to add self-adaptation support. Firstly, package *policies* was added to the meta-model, with constructs for the definition of adaptation policies and their associations with other meta-model elements. Then, existing models were modified according to the new meta-model, e.g., each binding definition can now be associated with adaptation policies. Finally, the binding factory (which is at level 0) was modified to interpret binding definitions associated with policies, and a new component, the *Adaptation Manager*, was created to manage the application of the policies. Each of these steps will be discussed in more detail next.

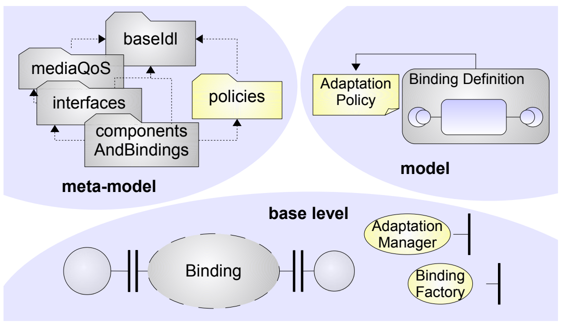

Fig. 3. Middleware extensions to support adaptation policies.

3.1 Meta-model Extension

Extending the meta-model (level 2) is probably the less painful work for the designer of new features, although such extensions usually affect other levels (levels 1 and 0), where the treatment is more complex. The meta-model itself is described using a meta-meta-model (level 3), allowing the meta-modeling of new concepts in accordance with a common language used to describe meta-models (known as the MOF model). Thus, the designer can use a modeling tool and a well-known language to make the necessary changes to the meta-model, making this task as easy as modeling any level 1 system.

In its current implementation [5], the meta-model of Meta-ORB was defined using EMF (Eclipse Modeling Framework) [6]. EMF is a plug-in for the Eclipse platform that facilitates the construction of tools and applications based on structured data models that are defined using a subset of the MOF model. The framework allows the manipulation of the middleware meta-model using graphical plug-ins for Eclipse or other compatible modeling tools. It also allows the generation of the core implementation of the type repository. This core implementation consists of Java classes that represent each of the meta-model constructs.

3.2 Model Transformation

Changes to the meta-model should be reflected in the middleware model. This means that the type repository must be (partly) re-generated to recognize the new features, enabling the definition, modification and access to meta-information elements that represent them.

With this extended repository, it is possible to define new types either programmatically or through the generated Eclipse graphical plug-in. Types created in the repository can be serialized into XMI, a programming language-independent format [7], and stored in local files. When necessary, the middleware model can be reloaded from these files into memory.

To retrieve the types, remote instances of the platform can access the type repository using a Web service, as shown in Figure 4. The service takes a type, which is a runtime instance of an EMF *EObject* (1), serializes it into XMI (2) and returns it to the remote platform (3). The remote platform is free to take the type and convert it into local objects using the appropriate programming language constructs in order to build a runtime model that provides useful meta-information to the middleware (4). For instance, factories can use the model to instantiate entities at runtime (5).

Changes to the meta-model do not affect the way the repository is accessed (via the Web service). However, with the new implementation of the repository (generated after the meta-model extension), model elements defined using the old meta-model (which were serialized and stored in local files) may no longer be reloadable due to incompatibilities between the meta-models. To avoid redefining the middleware model completely, the designer can edit the model manually (using the serialized XMI files) or use some model transformation technique [8]

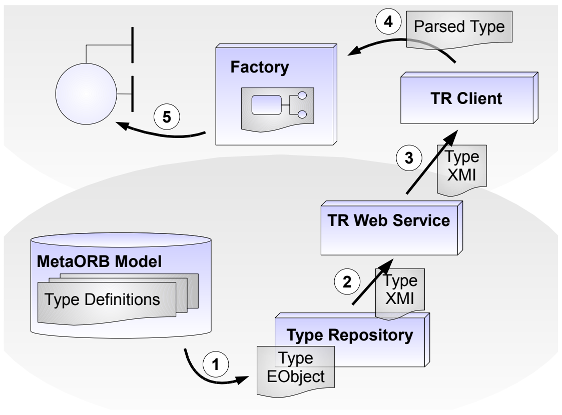

Fig. 4. Type Repository Web Service.

(and a tool compatible with EMF models) to adjust the model to the new meta-model. Adjusting the model manually can be a daunting task, depending on the extent of the changes made to the meta-model. It is generally less difficult to define rules to transform the model described in accordance with the previous meta-model into a model that conforms to the new meta-model.

3.3 Updating Runtime Entities

As seen in Section 2, the middleware makes extensive use of meta-information contained in the type repository. Special entities of the middleware core, such as component and binding factories, are capable of interpreting this meta-information at runtime. Thus, changes to the meta-model that affect existing types, as well as the introduction of new types, require modification of these entities or the creation of new ones to handle the new types at runtime.

The creation of new core middleware entities follows the normal procedure defined by the Meta-ORB platform, which consists in defining the type of the entity in the repository and implementing its primitive code in the host programming language (in the case of primitive components and bindings). The new entity can then be dynamically instantiated using an appropriate factory. An entity whose type has changed, however, may not be immediately replaced by a new entity, as the system may be running and the entity may still be in use and referenced by other entities of the middleware.

To this end, the repository uses type versioning: a change in a type implies the creation of a new version of this type. Operation *UpdateToVersion()* was thus added to the specialized factories of the platform, allowing the dynamic update of entities that have new type versions. This operation gets the new version of the type from the repository and uses the reflective framework of the platform to dynamically adapt the entity in accordance with its new version. In the case of primitive components or bindings, which cannot be adapted through structural reflection, the factory re-instantiates its internal implementation preserving its existing interfaces and adding new ones if necessary.

4 Related Work

In Meta-ORB, meta-modeling and computational reflection are key aspects, which means that middleware evolution is subject to the evolution of its meta-model. At this point, our work relates to other works focused on meta-model evolution, such as [9], which presents an approach for automatic and gradual evolution of generic meta-models. Our approach, however, is more focused on the evolution of a meta-model to provide new sources of meta-information for the middleware. It does not address aspects such as the characterization of the relationships between meta-models and the automatic co-adaptation of models.

Our work is also related to other middleware approaches that use runtime models as a source of meta-information, such as [10] and [11]. Both use architectural runtime models to automatically drive middleware adaptation. However, these works do not define an explicit meta-model as in Meta-ORB. We believe that an explicit meta-model is important for middleware evolution and, with the help of meta-modeling tools such as EMF, it is possible to modify this meta-model as easily as modifying any (level 1) model. In the Meta-ORB platform, the meta-model is also used to generate the type repository implementation, which is an important tool for the management of meta-information.

5 Concluding Remarks and Future Work

In this paper, we presented our approach for managing models and meta-models in the Meta-ORB reflective middleware platform. We also discussed the way it can facilitate middleware evolution. The paper shows how changes made to the meta-model can be reflected in the middleware model and consequently affect the entities that form a (base-level) runtime middleware configuration.

One of the main limitations of this work refers to the possibility of modifying only meta-types that do not structurally affect the middleware programming model. This means that it is not possible to redefine the meta-model concepts of component, interface and binding, as this would imply a complete re-implementation of the middleware core based on these modified concepts. Thus, only changes aimed at providing new types of behavioral meta-information are considered, not affecting the structural part of its programming model. The ability to evolve the middleware programming model still needs further study.

Another important aspect that was not addressed in this work is the creation of an integrated toolkit for defining and manipulating models and meta-models, which can even include a model transformation tool. There are works, such as [12], which propose frameworks for in-place model transformation based on EMF and can be employed for this purpose. In this way, the middleware designer would be able to use the same tool to modify the meta-model, create transformation rules and automatically adapt existing models.

As seen in Section 4, several proposals suggest the use of runtime models to provide information that is important for the autonomy of middleware platforms. The models can hold useful information about the middleware architecture, as well as information about how it must operate. In this paper, we argue that maintaining an explicit and unified meta-model, combined with an infrastructure to support the modeling and meta-modeling of middleware configurations, is an important improvement. This offers a structured and more natural way to evolve the concepts employed in the runtime models used by the middleware.

Finally, as reported in [3], we have successfully used this approach to add autonomic adaptation capabilities to improve the support for a class of multimedia applications in Meta-ORB. We are currently investigating the use of meta-model extension to enhance Meta-ORB's support for other classes of distributed applications, especially as part of a larger grid and cloud computing infrastructure.

References

1. Blair, G.S., Coulson, G., Blair, L., Duran-Limon, H., Grace, P., Moreira, R., Parlavantzas, N.: Reflection, Self-awareness and Self-healing in Open ORB. In: WOSS '02: Proceedings of the first workshop on Self-healing systems, New York, NY, USA, ACM (2002) 9–14
2. Costa, F.M.: Combining Meta-Information Management and Reflection in an Architecture for Configurable and Reconfigurable Middleware. PhD thesis, University of Lancaster (2001)
3. Provensi, L.L., Costa, F.M., Sacramento, V.: Self-adaptive Middleware for Digital Ink Based Applications. In: ARM '08: Proceedings of the 7th workshop on Reflective and adaptive middleware, New York, NY, USA, ACM (2008) 29–34
4. Object Management Group: Meta Object Facility (MOF) Core Specification. "`http://www.omg.org/spec/MOF/2.0/`", accessed in September, 2009 (2006)
5. Costa, F., Provensi, L., Vaz, F.: Using Runtime Models to Unify and Structure the Handling of Meta-information in Reflective Middleware. LECTURE NOTES IN COMPUTER SCIENCE **4364** (2007) 232
6. The Eclipse Foundation: Eclipse Modeling Framework Project (EMF). "`http://www.eclipse.org/modeling/emf/`", accessed in September, 2009 (2009)
7. Object Management Group: XML Metadata Interchange (XMI). "`http://www.omg.org/spec/XMI/2.1.1/`", accessed in September, 2009 (2007)
8. Bézivin, J.: From object composition to model transformation with the MDA. In: Proceedings of TOOLS'USA. (2001) 350–354
9. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. Lecture Notes in Computer Science **4609** (2007) 600
10. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-adaptation with Reusable Infrastructure. IEEE Computer **37**(10) (Oct. 2004) 46–54
11. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E., ICT, S., Trondheim, N.: Using architecture models for runtime adaptability. IEEE software **23**(2) (2006) 62–70
12. Biermann, E., Ehrig, K., Kohler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical definition of in-place transformations in the eclipse modeling framework. Lecture Notes in Computer Science **4199** (2006) 425