Towards a Meta-Modelling Approach to Configurable Middleware

Nelly Bencomo[1], Gordon Blair[1], Geoff Coulson[1], and Thais Batista[2]

[1] Comp. Dept., InfoLab21, Lancaster University, Lancaster, LA1 4WA, UK
{nelly, gordon, geoff}@comp.lancs.ac.uk
[2] Comp. Scs. Dept., UFRN, 59072-970, Natal - RN, Brazil
thais@ufrnet.br

Abstract. In our research we are studying how to combine modelling, meta-modelling, and reflection to systematically generate middleware configurations that can be targeted at different application domains and deployment environments. Despite this generality our approach adopts a uniform set of concepts: components, components frameworks, and reflection. Components and component frameworks provide structure, and reflection provides dynamic (re)configuration and extensibility for run-time evolution and adaptation. In this paper we present meta-models that capture the generality inherent to our approach and form a basis for automatic generation of extensible "middleware families" that can be instantiated differently depending on the application domain, QoS, deployment environment and degree of dynamic reconfigurability required.

1. Introduction

Reflection has now emerged as an important technique in the support of more configurable and re-configurable middleware [1]. A number of experimental reflective middleware platforms have been developed and used in industry. In our research we complement the use of reflection with 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) [9], and *middleware families* (abstract collections of component frameworks that are tailored to specific application domains and deployment environments). In addition, middleware families employ reflection [7] to discover the current structure and behaviour of the family instantiation, and to allow selected changes at run-time for dynamically consistent evolution and adaptation. The end result is a flexible middleware architecture that can be straightforwardly specialised to a wide range of domains including multimedia, embedded systems [2], and mobile computing [5].

Challenging new requirements emerge when working with such architecture. Middleware developers are faced 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. In addition, such *ad hoc* approaches do not offer a formal foundation 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 in systematically generating configurations of middleware families. In our MDSD-based approach, we propose the capture of the fundamental component-based programming concepts in a core set of meta-model elements called a *kernel*. All middleware family members regardless of their domain share this minimum set of concepts. On top of this, we propose a set of extension meta-models which capture the extensibility characteristics of our underlying (concrete) component model and which can be plugged in as appropriate.

In the remainder of this paper we first, in section 2, introduce the main concepts of our concrete component model. Then, in section 3, we discuss the MDSD-based modelling of these concepts and show our current model-based realisation. Then in section 4 we discuss the application of the meta-models and models in generating middleware families. Finally, we present conclusions and discuss future work in section 5.

2. A happy family: Lancaster's reflective middleware

As mentioned, our notion of middleware families is based on three key concepts: *components*, *components frameworks*, and *reflection*. Both the middleware platform and the application are built from interconnected sets of components. The underlying component model is based on OpenCOM[3], 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) [9]. A CF is a set of components that cooperate to address a required functionality or structure (e.g. service discovery and advertising, security 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 policies and constraints.

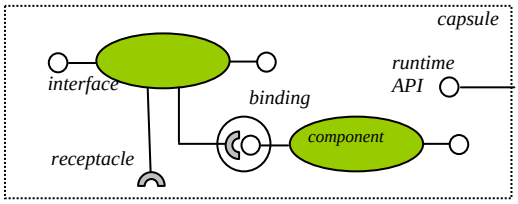

Figure 1. The OpenCOM main concepts

The basic concepts of OpenCOM are depicted in figure 1. Specifically, capsules are containing entities that offer a component run-time (CRT) API for the loading, binding etc. of components. Components are language-independent units of deployment that support interfaces and receptacles (receptacles are "required interfaces" that indicate a unit of service requirement). Bindings are associations between a single interface and a single receptacle. The CRT API is roughly as follows (many details have been omitted for reasons of space). The role of the *notify()* call is discussed below.

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

The architecture into which this fits is shown in figure 2. The layer immediately above the CRT consists of the so-called caplet extensions and a set of reflective extensions. The role of the caplet CF is to provide structured support for extensibility at the deployment environment level in terms of pluggable caplets which are subscopes within a capsule that are used for a variety of purposes including sandboxing and supporting heterogeneous programming languages. The reflective services then provide generic support for target system reconfiguration—i.e. inspecting, adapting and extending the structure and behaviour of systems at runtime (see below). Both the caplet and reflective extensions are independently and optionally deployable (using the CRT), and their precise configuration can be tailored to the needs of the target system and deployment environment.

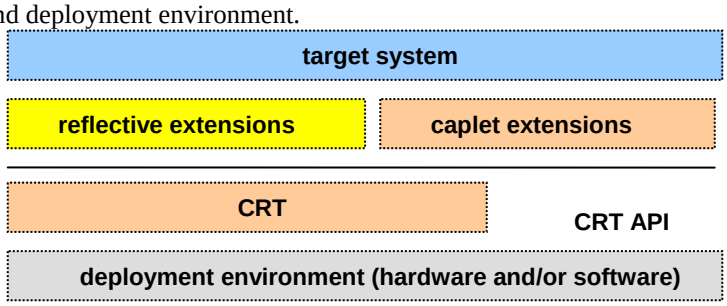

Figure 2.OpenCOM Architecture

The Reflection Services

As mentioned, 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 meta-models manage both evolution and consistency of the base-level system. The motivation of this approach is to provide a separation of concerns at the meta-level and hence reduce complexity. Three reflective meta-models are currently 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 together 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.

Causal connection between the base-level system and the meta-models is achieved via the above-mentioned *notify()* operation from the CRT API. This operation allows meta-models to register a callback that is invoked every time a subsequent call (bind, load, etc) is made on the CRT. The callback invocation contains all the parameter values of the call and so gives the callback holder a complete picture of all activity in the capsule. As an example, the architecture meta-model uses a notify callback to keep itself updated with information associated with the internal topology of the capsule contents. In case a meta-model needs to change the base-level configuration in some way, it simply invokes the respective operation (bind, load etc.) in the API. In this way, the causal-connection relation between the base and the meta level is maintained.

3. Modelling

In this section we present a set of UML meta-models that support the abstract specification of families of middleware. Figure 3 shows the three packages that comprise the OpenCOM metamodel. The *Kernel* package includes the fundamental model elements of OpenCOM: viz. component, capsule, interface, receptacle, binding, composite component and component framework. On top of this, the *Caplet Extensions* package includes the fundamental model elements of the caplet extensions in OpenCOM: viz. caplets, loaders, and binders. This package provides structured support for extensibility at the deployment environment level in terms of pluggable extensions. The *Reflective Extensions* package includes the fundamental model elements of the OpenCOM reflective meta-models (see the three reflective packages in Figure 3).

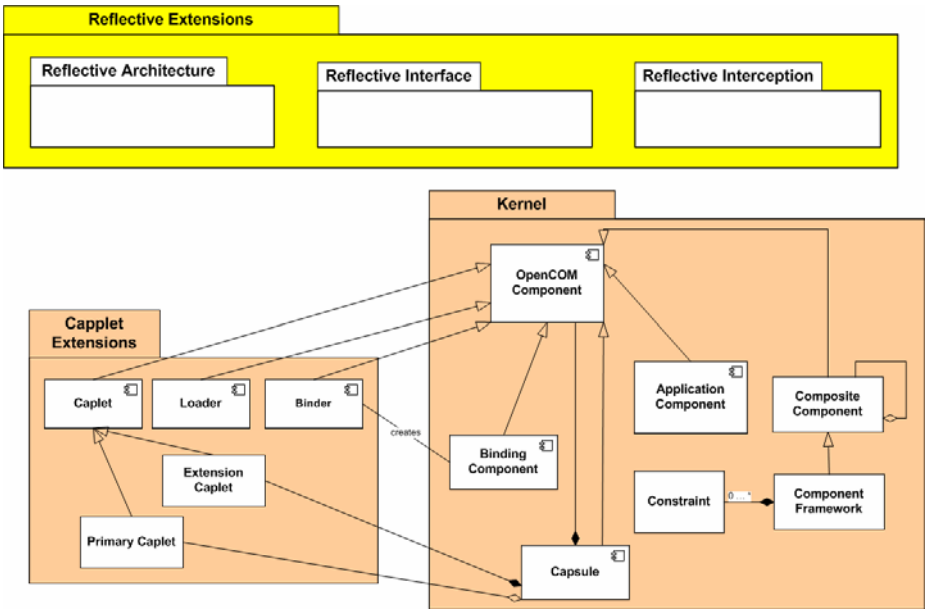

Figure 3. The OpenCOM Metamodel

Crucially, these meta-models are organised in terms of a structure with two orthogonal dimensions. One axis of this is the UML hierarchy (with its layer M0, M1, M2, and M3) and the other is a division into base level and (reflective) meta-level[1]. This is illustrated in Figure 4. All the meta-models (packages) we have described above populate the UML M2 level; however, while the kernel and caplet extensions packages live in the base-level, the reflective extensions live in the meta-level. The intention is that middleware family specifications will populate the UML M1 level and, of course, instantiations of these specifications will populate the UML M0 level. As a consequence, in implementation the meta-objects are optional and can be dynamically loaded/unloaded when required.

Figure 4 also shows an example instantiation of the model in terms of a very simple application example that illustrated the intended use of the model. This is a "calculator" which contains three sub-components; an adder, a multiplier and a calculator. The calculator component offers the services of adding and multiplying based on the services of the adder and multiplier components.

Figure 4 only shows details about the reflective architecture package. Work related to the reflective interception and interface packages have been done but it is not shown in this paper for reasons of space.

[1] Note that there is a potentially-confusing terminological clash here between the UML "meta-level" and "reflective meta-levels". These two concepts are entirely distinct; nevertheless we are forced to employ both of these terms because they are so well established in their respective communities.

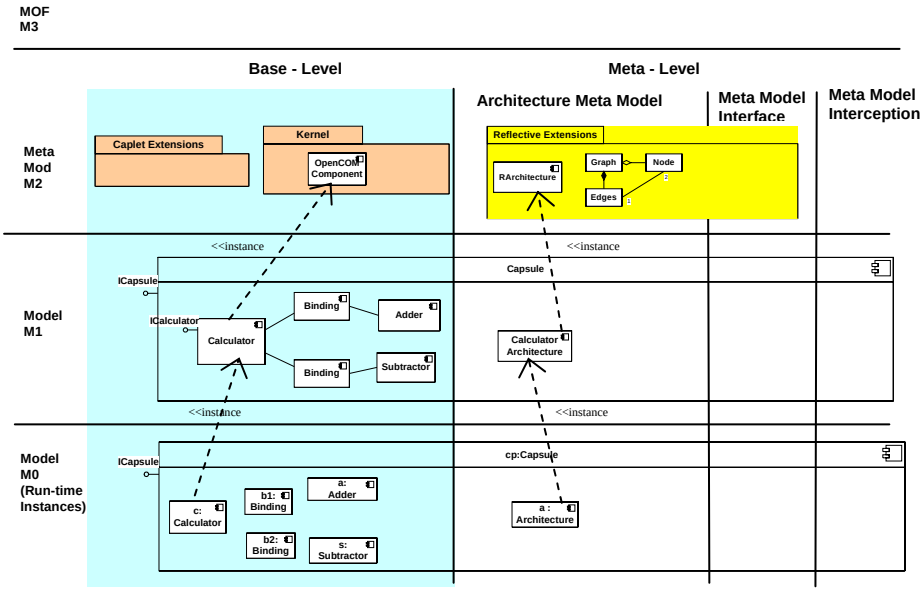

Figure 4. UML Models and Reflective Architecture of a Calculator Configuration

Finally, Figure 5 shows how the causal connection between the base and meta-levels is generically modelled in terms of a UML sequence diagram. This is based on the semantics of the *notify*() operation discussed above. The sequence diagram represents (part of) the dynamic behaviour specification of OpenCOM as opposed to the static structure of the models that was shown in Figure 4. Different models at level M1 will reuse or instantiate this generic causal-connection diagram.

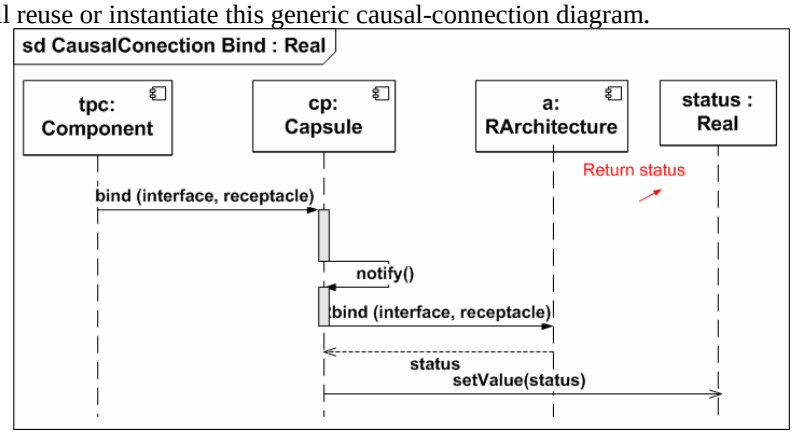

Figure 5. Sequence Diagram for Causal Connection when calling the *bind* operation

4. Discussion

We now turn to the application of the meta-modelling concepts to give support to the specification and efficient generation of middleware families. We apply the approach 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).

In outline, different middleware configurations are generated from models that are written in terms of the above meta-models. The models are sufficiently abstract that a number of different concrete OpenCOM-level configurations of components can be generated from them (i.e. the mapping of UML to Open COM components is not simply 1:1). The concrete configurations that are generated are determined by the following dimensions of variability:

- quality of service (QoS)
- deployment environment
- (re)configurability

The *QoS dimension* allows the abstract-to-concrete mapping to be influenced by consideration 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 Open COM-level configuration, this will indicate the inclusion of the caplet extension. The caplet extension is needed because of its support for sandboxing untrusted components, and for its provision of specialised loaders that are able to load remote objects. It will also indicate the inclusion of a security CF 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. 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 package might be deployed. This would preclude the use, e.g., of the caplet 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 meta-model would be included but not the others. Alternatively, if the application might need components to be added or replaced at runtime, the architecture meta-model would additionally be needed [6].

The above example 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..

5. Conclusions and Future Work

We have developed a set of metamodels that assist in the specification of middleware families and in the generation of specific family members which are determined by *quality of service*, *deployment environment* and *configurability* dimensions of variability. The metamodels capture the main concepts of the design philosophy of our middleware family: components, components frameworks, reflection for dynamic (re)configuration and extensibility. First, a package called *Kernel* containing the meta-model of the fundamental concepts is proposed. The UML specifications of reflective metamodels and caplets as extensions of the kernel are then presented in the packages *Caplet Extensions* and *Reflective Extensions*. As a result, at runtime the components/CFs related to caplets extensions and the meta-objects are optionally dynamically (un)loaded when pluggable extensions and reflective capabilities are required. In the particular case of the modelling of reflection, this paper describes how metamodels and models specify the causal connection between the base and meta-level.

We are now investigating how to generate different middleware configurations while keeping decisions that are generic to a set of configurations at the metamodel level design. More work has to be done to completely identify the variability among the related configurations (members) of middleware families to support an efficient generation of configurations. Another key area of future work will be to maintain the UML models at runtime and to keep this causally connected with the underlying running system in order to support reconfiguration. We also plan to investigate how solutions for the crosscutting problems we described can be found in the area AOSD.

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), (2004), 262-267.
2. Costa, P., Coulson, G., Mascolo, C., Picco, G.P., Zachariadis, S.: The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems, PIMRC05,(2005)
3. 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)
4. 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
5. 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)
6. Grace, P., Coulson, G., Blair, G., Mathy, L., Yeung, W.K., Cai, W, Duce, D., Cooper, C.: GRIDKIT: Pluggable Overlay Networks for Grid Computing, Proc. Distributed Objects and Applications (DOA'04), (2004)
7. Maes, P., "Concepts and Experiments in Computational Reflection", Proc. OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, pp147-155, ACM Press, 1987.
8. 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
9. Szyperski C.: Component Software: Beyond Object-Oriented Programming, Addison-Wesley, (2002)