Genie: a Domain-Specific Modeling Tool for the Generation of Adaptive and Reflective Middleware Families

Nelly Bencomo and Gordon Blair

Lancaster University, Comp. Dep., InfoLab21,
Lancaster, UK, LA1 4WA
[nelly, gordon] @comp.lancs.ac.uk

Abstract. At Lancaster University we are investigating about the two following challenges (i) how to develop new, scalable and adaptable middleware systems offering richer functionality and services, and (ii) how to do it in a more efficient, systematic, and if possible automatic way that guaranties that the ultimately configured middleware will offer the required functionality. This article is centered on how we face the second challenge. We describe Genie, our proposal of how to use Domain Specific Modeling (DSM) to support a development approach during the life cycle (including design, programming, testing, deployment and execution) of reflective middleware families. **Keywords**: Domain-Specific Modeling, Domain-Specific Languages, Model-Driven Engineering, Family Systems, Reflective Middleware.

1. Introduction

Middleware is a term that refers to a set of services that reside between the application and the operating system and its primary goal is to facilitate the development of distributed applications[13]. To pursue this goal many middleware technologies have been developed. All share the purpose of providing abstraction over the complexity and heterogeneity of the underlying distributed environment. With the advance of time other goals have been added, for example; adaptability is emerging as a crucial enabling capability for many applications, particularly those deployed in dynamically changing environments such as environment monitoring and disaster management [10, 19]. One approach to handling this complexity at the architectural level is to augment middleware systems with intrinsic adaptive capabilities [8, 18, 24]. Under these circumstances, the development of middleware systems is not straightforward at all. Application developers have to deal with a large number of complex variability decisions when planning middleware configurations and adaptations at various stages of the development cycle (design, component development, integration, deployment and even at runtime). These include decisions such as what kinds of components are required and how these components must be configured together. Tracing these decisions manually and using ad-hoc ways do not guarantee their validity to achieve the required functionality. Software engineers who work in the area of adaptive middleware development are consequently two-fold challenged in that they should (i) develop new, scalable and adaptable middleware systems offering richer functionality and services, and (ii) the approaches they use should be more efficient and systematic and should guarantee a formal foundation for verification that the ultimately configured middleware will offer the required functionality.

At Lancaster University we are researching how to meet these challenges. We use reflection and system-level component technologies and the associated concept of component frameworks, in the construction of our open, adaptive and re-configurable middleware families to face the first challenge identified above. More information about this facet of our research can be found in [1]. This article focuses on how we face the second challenge. We use DSM to raise the level of abstraction beyond

programming by specifying solutions using domain concepts. We advocate working with DSM to improve the development of middleware families, systematically and in many cases automatically, generating middleware configurations from high level specifications. In this paper we describe the prototype tool Genie, our proposal of how to use DSM to support a development approach during the life cycle (including design, programming, testing, deployment and even execution) of reflective middleware families. The paper is organized as follows. Section 2 introduces the Lancaster's middleware platform and its basic concepts. Section 3 presents Genie; relevant aspects and basic concepts of Genie are discussed. Section 4 discuses aspects related with different levels of abstraction in Genie and future work. Finally section 5 gives some final remarks.

2. Lancaster's Reflective Middleware : Meeting the Family

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 [9], 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* [22]. A component framework is a set of components that cooperate to address a required functionality or structure (e.g. service discovery and advertising, security etc). Component frameworks also accept additional 'plug-in' components that change and extend behaviour. Many interpretations of the component framework notion foresee only design-time or build-time plugability. In our interpretation run-time plugability is also included, and component frameworks actively police attempts to plug in new components according to well-defined policies and constraints. Similar to product family area's approach, we use component frameworks to design the middleware families that can be adapted by reconfiguration. The architecture defined by the component framework basically describes the commonalities and we achieve variability by plugging in different component.

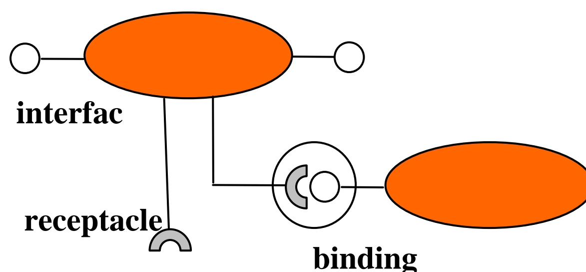

Figure 1: The OpenCOM main concepts

The basic concepts of OpenCOM are depicted in

Figure **1**. 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.

Reflection is used to support introspection and adaptation of the underlying component/component framework structures [7]. 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 reflective services then provide generic support for target system reconfiguration—i.e. inspecting, adapting and extending the structure and behaviour of systems at runtime. 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[1] are currently supported:

- The *architecture reflective meta-model* to inspect (discover), adapt and extend a set of components.
- The *interface reflective meta-model* to support 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.
- The *interception_reflective meta-model* to support the dynamic interception of incoming method calls on interfaces and the association of pre- and post-method-call code.

3. Genie

Genie is a prototype for a development-tool that offers a Domain Specific Language for the specification, validation and generation of artifacts for OpenCOM-based middleware platforms. Genie enables the construction and validation of models that drive the life cycle of the reflective middleware families at Lancaster University; this includes design, programming, testing, deployment, and even execution [4]. From the models specified not only source code can be generated but configuration and deployment files, results associated with model checking and validations, and documentation.

Genie has been developed using MetaEdit+ [20]. MetaEdit+ has proved to be a mature tool that offers a simple and elegant approach to develop DSLs. MetaEdit+ offers symbol and diagram editors that allow users to develop the same graphic concepts experts, designers, and programmers use. The generation of artifacts is done using reports. Reports access models information and transform it into various text-based outputs; in the case of Genie these outputs can be XML configuration files, programming code, or test code. The new version of MetaEdit use protected blocks in the text-based output. It means (i) manual changes to generated files are preserved each time new code is generated and (ii) the programmer who adds handwritten code knows exactly where to add it. This way, unwanted changes in the generated code is avoided. It was one of the drawbacks of our approach that has been fixed. The next sections discuss some relevant aspects of Genie.l

3.1. Modeling Process with Genie

DSM provides a systematic use of Domain Specific Languages (DSLs) to express different facets of information systems. In many cases DSM includes the creation of

[1] Note that there is a potentially-confusing terminological clash here between the "meta-level" and "reflective meta-levels" terms. 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.

domain-specific generators that create code and other artifacts directly from models [16, 17]. Getting the benefits of DSM was limited as it was common to develop the supporting tool besides the DSLs and the generators. Nowadays we have modern metamodel-based DSM tools available which are used by developers to just focus on the development of DSLs and the generators. Using these tools, the process for implementing model-based development generally presents the following four phases [23]:

- Identification of abstractions and concepts and specification of how they work together
- Specification of the language concepts and their rules (metamodel). These rules will guide the modeling process that developers follow.
- Creation of the visual representation of the language (notation); this is done in the case we have a Domain Specific Visual Language.
- Definition of generators. These generators will produce source code, documentation, results related to model validation, etc.

The process in Genie essentially follows these steps (see Figure 2). More details are shown in the next sections.

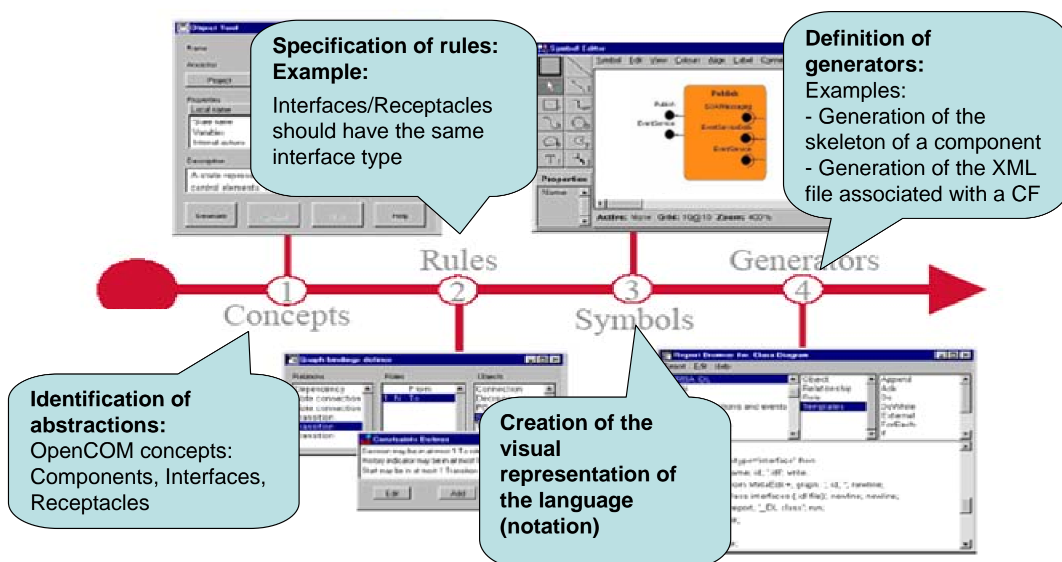

Figure 2: Steps for implementing a Domain Specific Modeling Language (DSML): Case study Lancaster Middleware Platform

3.2. Genie: basic Concepts

As in other program family techniques, our approach uses component frameworks to manage and accomplish variability and development of systems that can be adapted by re-configuration. A component framework enforces architectural principles (constraints) on the components it supports; this is especially important in reflective architectures that dynamically change. Reconfiguring a running system using our approach implies the insertion, deletion and modification of the structural elements represented in the component frameworks: components, interfaces, receptacles, binding components and constraints. Models associated with component frameworks are used

to represent the possible variants (configurations) of the different families. Models can be effective and valuable in this sense as they can be verified and validated a priori (before the execution of the reconfiguration).

Existing models of OpenCOM-based middleware families use a wide variety of notations that depend on the domain that is being modeled. However, the basic concepts of any OpenCOM-based model use the basic notions that OpenCOM dictates (i.e. components, interfaces and component frameworks). Genie offers a common modeling notation for all the models called the OpenCOM DSL. The specification of how these concepts work together is described in the graphs associated with the components and component frameworks. An example of a model associated to a component framework is shown in

Figure **3**. The component framework specified is the Publisher [15]. In the figure we can see that components offer and require interfaces and interfaces can be bound together to connect components. Component frameworks can export interfaces from internal components. In the same way, component frameworks can require interfaces to satisfy the requirements of some of their internal components.

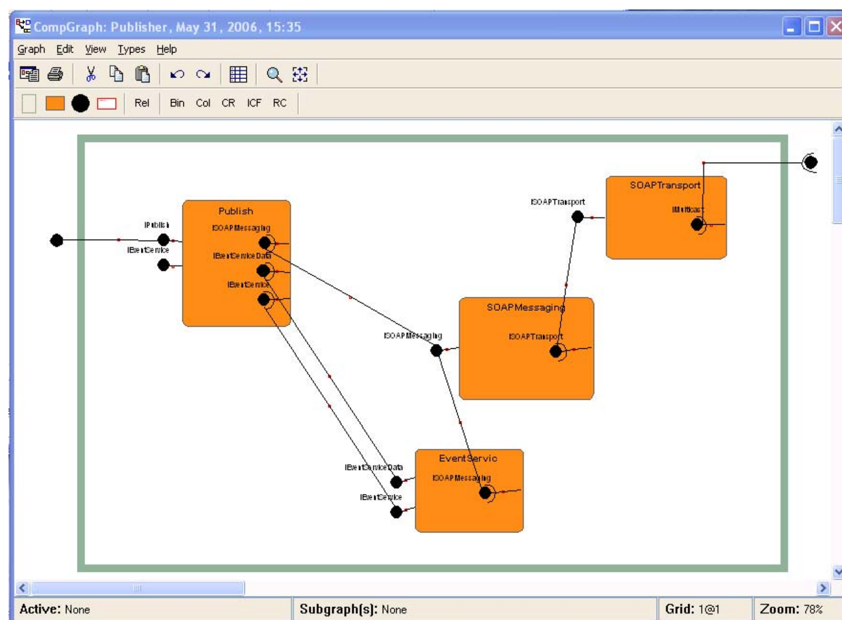

Figure 3: A Component Framework (Publisher) modeled in Genie

Many artifacts can be generated from component-framework models. Some examples of artifacts that can be generated from these models are:

- the XML files associated to policies that rule the configuration and reconfigurations of the component frameworks.
- test code that use hardcode connections of the components in the component framework. These test code is executed as isolated experiments before performing the tests that use reflective capabilities and do not use hardcode connections.
- reports of validations and checkings; for example, a report can show notifications of interface mismatches meaning that interfaces of different types are mistakenly connected. More details and examples are in Section 3.3.
- documentation

Figure 4 shows other examples and details of models, relations between models, and generation of different artifacts. Arrow (a) shows how from the graph of a component framework a component can be chosen to get more details. From the model (graph) associated to a component more details associated with required and offered interfaces, author, version, etc can be found . If the user wants to explore the interfaces associated with a component; she could open a window with the data associated with the interfaces (signature, parameters, etc.). In the same way, the user could open a window with the data associated with the author/responsible of the component. Arrow (b) shows how from the graph of a component, the skeleton code of the component can be generated and/or accessed. Finally, arrow (c) shows a policy (XML file) associated with the configuration of components shown in the graph of the component framework. These policies are stored in a Knowledge Repository that will be accessed by the middleware configurators at run-time. The configurators will read the policies to perform the re-configurations connecting and disconnecting components to perform adaptations [5].

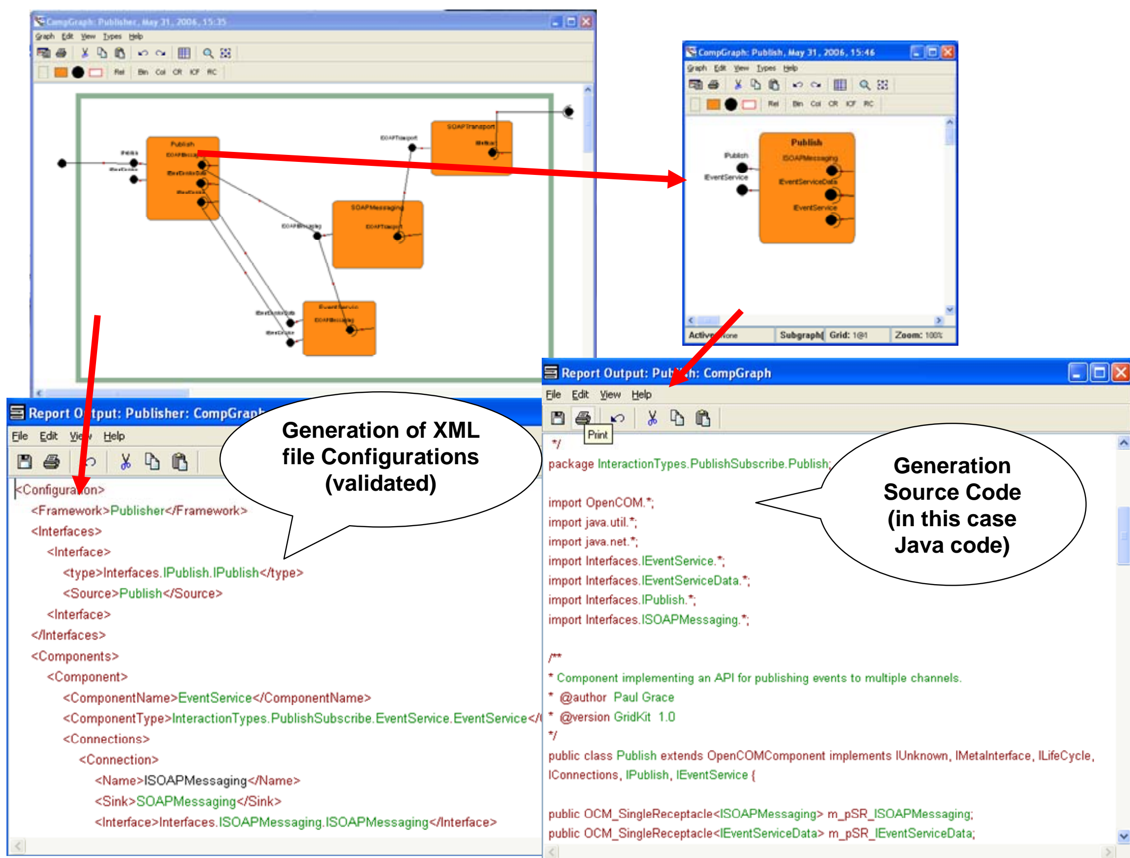

Figure 4: Generation of different artifacts

3.3. Validation of Models

In MetaEdit+, the validation of models can be performed while the modeler is editing a model or once the edition has been completed. The second option is faster and is the option we prefer to use. Any generation of artifacts (source code, XML file, etc.) does

require validation and checking. To understand the important role of validation in Genie let us focus on the case of component frameworks.

As noted above, a component framework imposes constraints on the components it supports. Consequently the basic checking is related to these architectural constraints. When designing the validations of the component frameworks we exploit known variabilities in architectural structures so that common checking infrastructure can be built once and then used by any user of Genie in the corresponding component framework. Not only does this approach decrease the cost of models validation, but it makes it easier the technology since the modeler needs just to be concern about the domain-specific aspects of the problem; in this case the behavior of components and specific domain-related constrains (architectural styles and new constraints).

An example of basic validation is the verification that all the connections between required interfaces and offered interfaces conform to the same type (therefore the configurator does not need to check these conditions at run-time). Examples of more specific validations are related to the specific constraints enforced by the component frameworks: a specific component may appear only once at the most, a connection between two components must exist, etc. These validations should be written for all the component-framework models.

4. Different levels of abstraction in Genie

OpenCOM DSL models in Genie are defined essentially in terms of configurations of OpenCOM components and individual components. These concepts are not about code but about much higher-level abstractions as shown in the previous sections. Genie offers the OpenCOM-based DSL but also allows the specification of models using UML [12]. Every OpenCOM component is specified using a UML class that inherits from the superclass called OpenCOM Component [3, 6]. In Figure **5**, arrow (a) shows how a component is inspected and shows a partial view of the corresponding UML specification. The component Subscriber is specified by the class *Subscriber* that inherits from the superclass *OpenCOMComponent*. Arrow (b) shows how, from the graph of a component, the skeleton code of the component is generated and/or accessed. Genie will traverse the UML models related to the component to generate this source code. The code generated in the example is Java code. More detail can be found in [6]

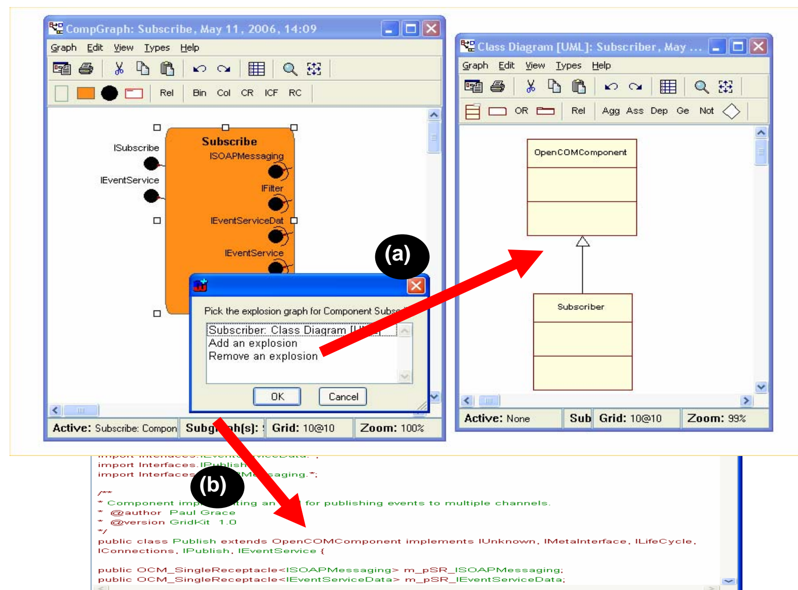

Figure 5: OpenCOM DSL and UML

Figure 6 shows the different levels of modeling corresponding to different levels of abstraction. At the bottom level we have the models corresponding to the *underlying code framework*. This code framework offers modules (i.e. components) that will be used from the DSL environment at higher levels. At a high level of abstraction, models defined using the DSLs, are used to generate the code that relies on the code framework. Higher levels are at a more coarse level of granularity and it is here that we deal with concepts that are closer to the problem domain. Lower-level modeling entities are about source code and implementation details. In general, programmers will work at the lower level (programming level) or generating the underlying framework code. This fits well with the vision MetaCase has for domain-specific modeling where applications are built on top of a software platform and possibly a code generation framework [2].

Future Work

It is on this specific aspect of Genie that we would like to focus our future work. We aim to introduce higher levels of abstraction in Genie to focus on different domains like grid computing [14, 21] (using more specific notions like overlay network frameworks, or resource management framework) and service discovery protocols. For example, we envisage having pre-designed and specialized components frameworks with characteristics and constraints focused on specific requirements of a class of applications (family). We are already working on the specification of models for families of service discovery protocols with a common architecture [11]. This way, we can minimize resource usage through not just component code re-use, but architecture too.

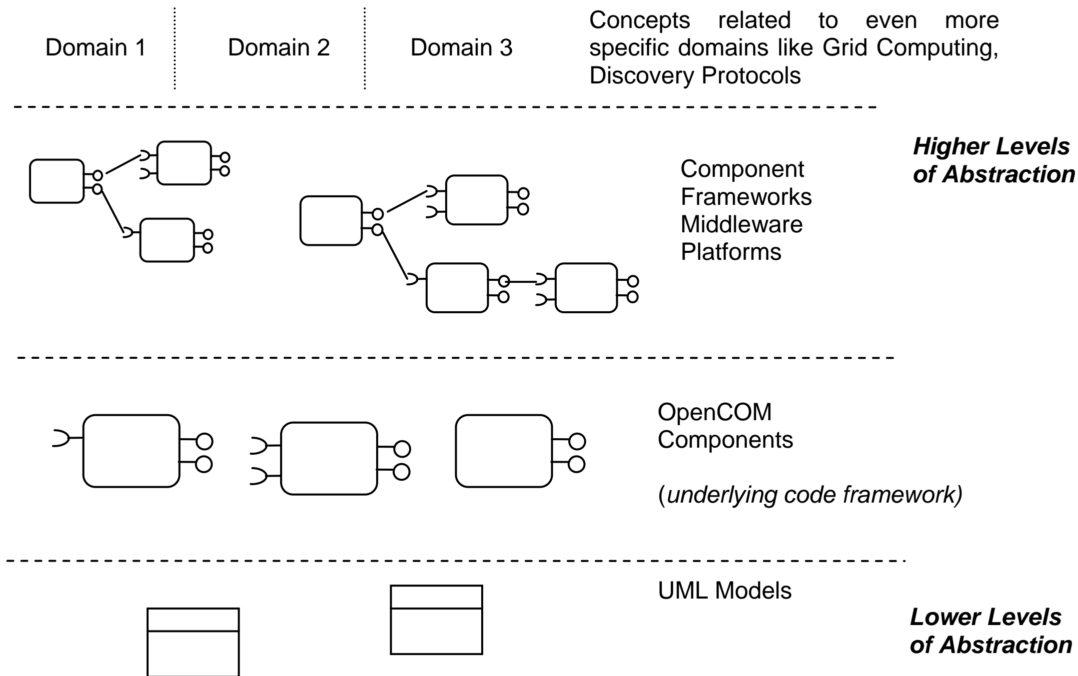

Figure 6: Level of modeling corresponding to different levels of abstraction

5. Final Remarks

Reflective and adaptive middleware platforms require the creation of multiple dynamically deployed variations of reconfigurable systems. A systematic approach is needed to model, generate and finally validate these variations. Genie represents the way in which we have met this challenge. Genie is a DSM environment prototype to support the development during the life cycle of reflective middleware families. The environment simplifies the development of middleware families offering a platform that guides the development process. Genie is proven to generate the policies for configuration of our Gridkit middleware platform [5].

In this paper, we have described the OpenCOM DSL offered by Genie, a domain specific language for the specification, validation and generation of artifacts for OpenCOM-based middleware platforms. Among the benefits of Genie are reusability of code and knowledge. Genie promotes valid code and artifacts offering a less error-prone approach.

Genie has been developed using MetaEdit+. DSM-based metamodeling tools like MetaEdit+ make it easier to construct DSL-based environments to automate software development. However, while DSL approaches raise the levels of abstraction and allow the development of systems considerably faster than UML-based approaches, UML has the advantage of visualizing code using the well understood UML models. We advocate combining both approaches [6]. DSLs and UML can give benefits by providing an intermediate representation that is validated and translated into well understood UML-based models. Following this philosophy, Genie offers tool support for different levels of abstraction using common semantics. It offers supports from the source code level up to domain-specific and higher levels, and consequently for different users. Our future work focuses on adding support for higher levels of abstractions including more specific domains and adaptability requirements [21]. We

think this offers the additional advantage of better communication between participants in development projects and therefore generating potential for more successful projects.

Acknowledgments

Grateful acknowledgment is made to MetaCase for permission to use their tool MetaEdit+.

References

1. Next Generation Middleware @ Lancaster University. http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/index.php.
2. Ambler, S.W. Unified or Domain-Specific Modeling Languages? *Sofware Developmet's Agile Modeling Newsletter* 2006.
3. Bencomo, N., Blair, G., Coulson, G. and Batista, T. Towards a MetaModelling Approach to Configurable Middleware *2nd ECOOP'2005 Workshop on Reflection, AOP and MetaData for Software Evolution RAM-SE* Glasgow, Scotland, 2005.
4. Bencomo, N., Blair, G. and France, R. Models@runt.time. Workshop in conjunction with MoDELS / UML 2006, 2006.
5. Bencomo, N., Grace, P. and Blair, G. Models, Runtime Reflective Mechanisms and Family-based Systems to support Adaptation submitted to Workshop on MOdel Driven Development for Middleware (MODDM), 2006.
6. Bencomo, N., Sawyer, P. and Blair, G. Viva Pluralism!: on using Domain-Specific Languages and UML *Submitted to Multi-Paradigm Modeling: Concepts and Tools (MPM'06)*, Genova, 2006.
7. Blair, G., Coulson, G. and Grace, P., Research Directions in Reflective Middleware: the Lancaster Experience. in *3rd Workshop on Reflective and Adaptive Middleware*, (2004), 262-267.
8. Blair, G., Coulson, G., Robin, P. and Papathomas, M., "An Architecture for Next Generation Middleware. in *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98),* , (The Lake District, UK, 1998), 91-206.
9. Blair, G., Coulson, G., Ueyama, J., Lee, K. and Joolia, A., OpenCOM v2: A Component Model for Building Systems Software. in *IASTED Software Engineering and Applications*, (USA, 2004).
10. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K. and Gjorven, E. Using Architecture Models for Runtime Adaptability. *Software IEEE*, *23* (2). 62-70.
11. Flores, C., Blair, G. and Grace, P., Service Discovery in Highly Heterogeneous Environments. in *4th Minema Workshop*, (Lisbon, Portugal, 2006).
12. Fowler, M. and Scott, K. *UML Distilled*, 1999.
13. Geoff, C. "What is Reflective Middleware?" *IEEE Distributed Systems Online*.
14. Grace, P., Coulson, G., Blair, G., Mathy, L., Duce, D., Cooper, C., Yeung, W.K. and Cai, W., GRIDKIT: Pluggable Overlay Networks for Grid Computing. in *Symposium on Distributed Objects and Applications (DOA)*, (Cyprus, 2004).
15. Grace, P., Coulson, G., Blair, G. and Porter, B., Deep Middleware for the Divergent Grid. in *IFIP/ACM/USENIX Middleware*, (Grenoble, France, 2005).
16. Greenfield, J., Short, K., Cook, S. and Kent, S. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools* Wiley, 2004.
17. Kelly, S. and Tolvanen, J.-P., Kelly, S., Tolvanen, J-P, "Visual domain-specific modelling: Benefits and experiences of using metaCASE tools", . in *International workshop on Model Engineering in ECOOP 2000*, (France, 2000).
18. Kon, F., Costa, F., Blair, G. and Campbell, R. The case for reflective middleware. *Communications of the ACM*, *45* (6). 33-38.
19. McKinley, P.K., Sadjadi, S.M., Kasten, E.P. and Cheng, B.H.C. Composing Adaptive Software. *IEEE Computer*, *37* (7). 56-64.
20. MetaCase. Domain-Specific Modeling with MetaEdit+ (http://www.metacase.com/).
21. Sawyer, P., Bencomo, N., Grace, P. and Blair, G., Ubiquitous Computing: Adaptability Requirements Supported by Middleware Platforms. in *Workshop on Software Engineering Challenges for Ubiquitous Computing*, (Lancaster, UK, 2006).
22. Szyperski, C. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2002.

23. Tolvanen, J.-P. Domain-Specific Modeling: How to Start Defining Your Own Language, DevX.com, 2006.
24. Wang, N., Schmidt, D.C., Parameswaran, K. and Kircher, M. Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications. *EEE Distributed Systems Online special issue on ReflectiveMiddleware*.