Experiments with Fractal on Modular Reflection

Jérémy Buisson, Fabien Dagnat

Experiments with Fractal on modular reflection

Jérémy Buisson and Fabien Dagnat
TELECOM Bretagne / Université Européenne de Bretagne
Technopôle Brest-Iroise – CS83818
29238 Brest Cedex 3, France
Email: first.last@telecom-bretagne.eu

Abstract

In most reflective systems, the model of reflection objects often mirrors (a part of) the metamodel of the system. As a result, reflection is commonly tightly bound to the rest of the system. In this paper, we investigate the loosening of that coupling.

With the rise of domain-specific modeling the need for separation of concerns and reuse when designing metamodels become critical. Therefore, we advocate the use of general design patterns abstracting the details of modeling languages when working on crosscuting concerns (such as reflection) of a metamodel. Once the abstract patterns for reflection are built, they are mapped onto concrete modeling languages thanks to model engineering tools. In this paper, we apply this approach to the Fractal component model.

Following this process, reflection mechanisms built at the abstract level are straightforwardly reused and the resulting reflection system gains modularity.

1. Introduction

In model-driven approaches for design and engineering, one of the trends advocates for the design of many domain-specific modeling languages [1], [2]. In comparison to a single general-purpose language, this practice allows one to preserve the ontologies used by domain experts. Parts of the semantics of those ontologies can be hardcoded in the domain-specific tools supporting the engineering process, capitalizing on the experience of domain experts.

One counterpart of this approach is that designing metamodels becomes quite frequent. Similarly to what happens in applications, metamodels exhibit several concerns. Some of them appear to crosscut application domains and must be included in many domain-specific languages. This is for instance the case of hierarchical decomposition, assembling facilities and reflection. Despite new technologies help separate concerns in metamodels [3], [4], [5], those concerns are commonly hardwired within domain-specific metamodels. Worse, they are often tightly entangled.

In this paper, we investigate how to raise modularity of metamodels, considering the example of reflection. To do so, we propose a collection of metamodel patterns for architectural units, composite structures and assemblies. The patterns are used to build reflection at an abstract level. Namely, they abstract a subset of the capabilities of modeling languages. When designing reflection, only the ones abstracting the desired capabilities are integrated in order to target specifically the items that are required to be reflected. Mixed together, the patterns give an indication of the metadata that are required by reflection mechanisms. Futhermore, some of the mechanisms can even be implemented at the abstract level, allowing for reuse across several modeling languages. In this paper, we apply a slightly different process in order to reimplement an abstract reflective system on top of the Fractal [6] component model. This experiment still helps having a better understanding of how the proposed patterns interact when designing modeling languages. It also contributes to make the model of Fractal reflection clearer.

The rest of the paper is organized as follows. Section 2 depicts the metamodel patterns for building the abstract reflective system. Section 3 presents Fractal and a metamodel of that component model. Section 4 details how the above process is applied to rebuild reflection on top of Fractal. Section 5 compares the approach with related works. Section 6 gives some indication of ongoing and future work.

2. Metamodel patterns for an abstract reflective system

The general idea underlying the proposed approach consists in designing the reflective system at an ab-

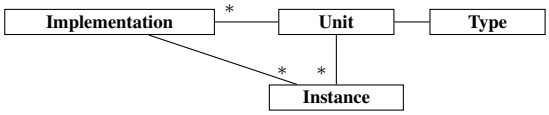

Figure 1. Architectural units and runtime entities.

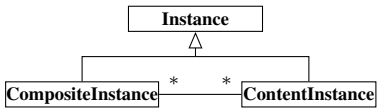

Figure 2. Hierarchical (composite) entities.

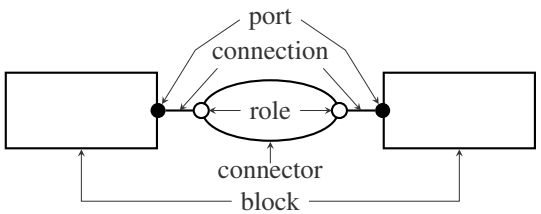

Figure 3. Block and connector units.

stract level, in an independent manner with regard to any concrete modeling language. In so doing, the reflective system can be reused when new domain-specific modeling languages are designed. In addition, details of modeling languages are hidden behind the reflective system when applications are reconfigured. Thus reconfiguration languages and protocols designed for that abstract reflective system are straightforwardly inherited in any concrete modeling language.

To design a reflective system, some knowledge is required about the entities defined by the concrete language. For instance, considering a component modeling language, the design of a reflective system requires to know that components are runtime entities and that bindings between components can be rearranged. Talking about components or whatever else is secondary, as well as the way the modeling language reifies components and bindings. Indeed, in order to identify the operations of the reflective system, it is sufficient to know which runtime entities can be created, destructed and connected together.

Merging the abstract reflective system with the concrete modeling language implies mapping reflective operations onto the concrete concepts. In component modeling languages, components are runtime entities. Therefore, the create and destruct operations of the reflective system would be merged with the concept of component. Obviously, that merge action makes extensive use of the properties of the concepts that have been identified in each concern. In addition, concept reification in the concrete modeling language should not anticipate the design of the reflective system. For instance, reifying attributes, i.e. breaking encapsulation in order to expose some parts of the internal state, is usually specific to reflection. The scope of reification is typically connected to the operations of the reflective system, as applying an operation to a concept often requires that concept to be reified. Again, the choice of reifying concepts results from the merge action of the reflective system with the concrete modeling language.

2.1. Metamodel patterns

In order to apply the above methodology, we first define a collection of metamodels that describe the properties the abstract reflective system focuses on. Similarly to design patterns in object oriented modeling [7], they give some solutions to model some properties that are common to several modeling languages.

In the following patterns, in order to ease the understanding, we adopt a common terminology whatever the modeled property. It is defined when modeling the basic concepts of architectural units and runtime entities in Figure 1. In this pattern, the central concept is the notion of unit, which denotes what models are made of[1]. In object oriented modeling, the concept of unit would alias the concept of class. Each unit has a type and is associated with some implementations. A unit is represented at runtime by instances that model a state associated with an implementation. Notice that, beyond the structural aspect described in Figure 1, associations are constrained. For instance, every unit implementation must conform to the type of the unit.

Figure 2 models hierarchical runtime entities. This is a slight variation of the composite pattern found in [7]. The latter defines a recursive tree structure, while the former only introduces the concept of composite instance, which has some content. Nevertheless, like the other patterns, it should not be thought independently of any context. In particular, when merging with a concrete modeling language, some concrete concept can be said to be both content and composite instance, resulting in recursive decomposition. Another specificity of the pattern is the fact that any content instance can be shared by several composite instances.

The latest pattern concerns the assembling capability of instances. This pattern is based on the distinction between connectors and blocks. This distinction is a standard technique [8], [9] that allows software architects to model explicit communication objects.

1. Several kinds of unit may coexist.

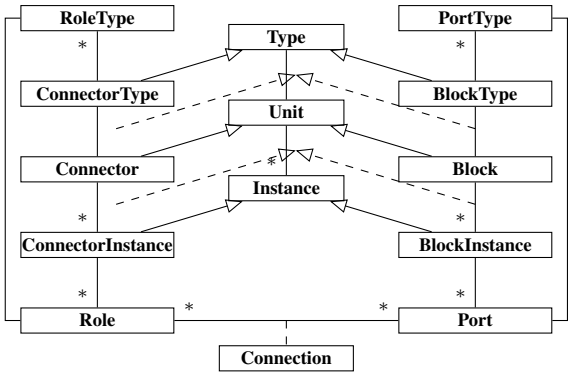

Figure 4. Block/connector design pattern for assembly.

Figure 3 illustrates these concepts using a very simple assembly and defining the associated concepts. The ellipse in the middle models a connector that connects 2 (rectangle) blocks. Connectors reify the semantics of the communications and the connection rules of legal assemblies (e.g. relations over the types associated to the connected ports). Ports and roles are the endpoints through which blocks and connectors interact. Connections, here drawn as lines, denote which port of blocks plays each role of the connector. Generalizing bindings, connectors can be of arbitrary complexity, possibly having more than 2 roles.

Both concepts (connector and block) specialize the notion of unit. Together, they are used to describe software architectures (here called assemblies). As represented in Figure 4, connectors and blocks introduce some specificities with regard to type and instance. At the level of instances, roles (resp. ports) are introduced in order to reify the interaction points of connectors (resp. blocks). They are specified in the types of the connectors (resp. blocks). The association modeling the assembling capability between roles and ports is reified by the concept of connection. This association is constrained by a typing rule.

2.2. Abstract reflective system

Relying on the concepts of the previous patterns, Figure 5 shows the metamodel for the abstract reflective system. This metamodel is structured in two parts. An observability facility is first introduced through observable instances, which expose a collection of attributes. Attributes denote the observable part of the state. Their list and types are specified in the type of the unit. Attributes can also be used to model the exposure of connections and composite content. On top of observability, a reconfigurability facility is added. At the level of the unit, it allows to create and destruct instances. At the level of instances, it allows to change the implementation. At the level of attributes, it allows to modify the state of instances. At the level of the type, it allows to change the collection of attributes, roles and ports. Last, at the level of the system, it allows to load and unload units dynamically.

2.3. Discussion

The patterns are not supposed to be isolated one from the others. They would rather be considered in combination with one another. For instance, combining composite and content instance as a "component" concept assumes a recursive hierarchical component model. Combining connector and composite instance allows one to model some kind of "composite binding" concept. Combining several concepts into concrete entities makes it possible to give a flexible description of the actual properties of concrete languages.

The proposed metamodel for the abstract reflective system assumes several design choices. The most important one is the set of operations and the concepts to which they apply. The metamodel could have been made more complete if for instance observable units and types were added. This would have allowed the introspection of those concepts. Therefore, the given abstract reflective system is only one among the many possible ones. Several reflective systems could even coexist in a single concrete modeling language. That way, each concrete concept is equipped with only its own capabilities, independently of the other concepts. The approach thus provides high flexibility in order to make reflective capabilities fit requirements.

Interestingly, the metamodel of Figure 5 does not explicitly use the metamodels of assembly and hierarchy. Indeed, we have chosen to rely on the general attribute framework in order to expose connections and content. This is a design choice. Connections and content could have been exposed through specific methods as well. The chosen design allows to add seamlessly new observable and reconfigurable properties, such as concurrency and time constraints. Nevertheless, it reports most of the difficulty on the implementation of attributes. Furthermore, interactions between the concerns do not appear explicitly. Making content and connections appear in the type as attribute specifiers may in addition prevent substituability and reuse.

3. The Fractal component model

Fractal [6] is a component model that is intended to structure applications at runtime. Fractal components

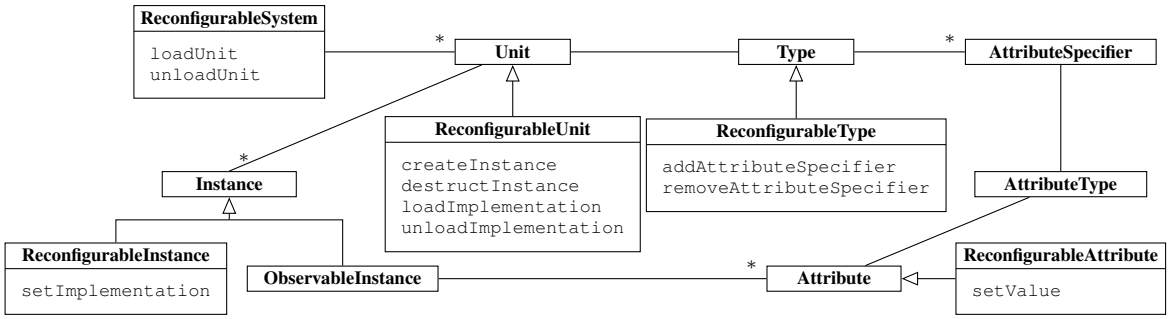

Figure 5. Metamodel for observability and reconfigurability.

are runtime entities that are bound to one another. Each component is made of two parts: a content implements the business logic thanks to explicit dependencies; a controller encapsulates the content and provides technical services for component management.

Fractal is a recursively hierarchical model. Each component can contain an assembly of subcomponents. Futhermore, each component can be contained by several components. Usually, two kinds of components are distinguished in hierarchical architectures: primitive components, the leaves of the hierarchy, do not expose their subcomponents; while composite components expose their subcomponents. It is also commonly assumed that the content of primitive components consists in one behavioral entity of some programming language such as Java or C; and that only composite components can truly contain Fractal components.

In order to interact, components have interfaces. Bindings in Fractal are largely inspired from the client-server and object models. Each binding connect one client interface to one server interface by forwarding method calls. To do so, Fractal defines a typing rule for legal connection of interfaces. Bindings are not reified. In addition, Fractal introduces the concept of composite binding as a collection of mediating components and bindings. As this concept has no concrete existence in Fractal, we do not consider it in this paper. We do not consider either collection interface cardinality, which is mostly a mechanism for spawning similar interfaces on-the-fly. Nevertheless, these features are typical applications of the connector concept of our metamodel pattern for assemblies.

Fractal supports reflection natively. Standard controllers give some means to introspect the architecture at runtime. One can list the interfaces of a component, retrieve the component owning an interface, list the subcomponents of a composite component, list the components that contain a component, get the server interface connected to a client one. Standard controllers also allow to add a component to and remove it from a

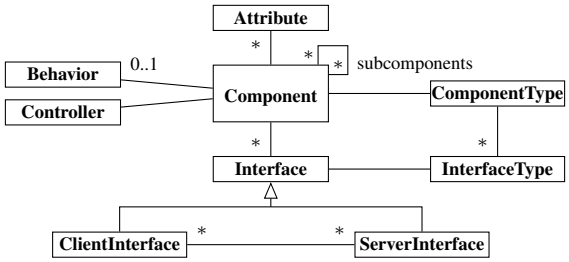

Figure 6. One metamodel of Fractal.

composite component, and to connect and disconnect interfaces. In addition, Fractal includes a framework such that components can reflect some attributes.

Fractal has some other features that are not relevant in this paper, such as the management of components' life cycle. Readers may refer to [6] for a more complete description.

Metamodels of Fractal have been rarely proposed. Figure 6 depicts the one that is used in the rest of the paper. This metamodel simply formalizes the above description of Fractal. In comparison to other proposals, this one focuses much more on the component model. For instance, the one of Tibermacine [10] is much closer to Fractal ADL [11] than to Fractal itself.

4. Mapping patterns onto Fractal

Given the patterns of Section 2 and the metamodel of Fractal, it is possible to implement the described abstract reflective system for the Fractal component model. To do so, the properties modeled in Section 2 have to be mapped onto the concrete concepts of Fractal. The design of the patterns and the one of the Fractal metamodel are somewhat different. For instance, the former one reifies connectors while Fractal does not; the Fractal metamodel hardcodes the recursivity of the hierarchical decomposition while the patterns do not; as Fractal components are run-

time entities (i.e. instances), no Fractal entity is the counterpart of the "unit" concept. These differences do not matter at the design level. Nevertheless, they impact on the implementation of the reflective system. Missing concepts may be built *ex nihilo* as additions to the concrete language, while the other concepts are combined in order to reflect the actual properties of the concrete language. The following gives the results of that process when it is applied to Fractal.

The mapping between Fractal and the patterns is the following one. Regarding the basic metamodel (Figure 1), a Fractal component is an instance; the component's type is a type; and its behavior is an implementation. The "unit" concept is added to aggregate implementations with a type and a collection of belonging instances. Regarding the assembly metamodel (Figure 4), a Fractal component is a block instance; its type is a block type; an interface is a port and an interface type is a port type. As already stated, Fractal has no counterpart for connectors and related concepts. Actually, the component model implicitly defines one single connector for simple client-server connections, whose instances are the associations between client and server interfaces. Instead of adding a generic "connector" concept, an alternative design would consist in simply defining the only specific connector relevant for Fractal, depending on the capabilities we require from the reflective system. Regarding the hierarchy metamodel (Figure 2), a Fractal component is a composite and a content instance. The "implementation" concept is specialized such that it denotes assemblies. Last, regarding the metamodel of the abstract reflective system (Figure 5), a component is an observable and a reconfigurable instance; the component's type is reconfigurable and Fractal attributes are observable and reconfigurable. In addition, the "unit" concept is extended to become a component factory able to create and destroy components.

Once the mapping is completed, the concrete reflective system relies only on the terminology and the concepts defined in Section 2. The concrete language of Fractal is abstracted and enriched when components are manipulated. Consequently, programs using such a reflective system need not know about the Fractal component model.

4.1. Implementation considerations

Mirror-based reflection [12] is a natural strategy for the implementation of a reflective system. The general idea underneath mirrors consists in separating the implementation of the reflective system from the implementation of the (modeling) language. As the above methodology enforces the separation at the level of the design, it is natural to preserve that separation at the level of the implementation. Nevertheless, as noticed in [12], a minimal kernel of reflection must be integrated into the implementation of the concrete modeling language. In our case, that kernel is inherited from the native reflective capabilities of Fractal. Resulting from the above mapping and from the adoption of mirror-based reflection, the implementation of the reflective system looks like the diagram of Figure 7. Each mirror object is simply said to inherit the properties of the generic patterns.

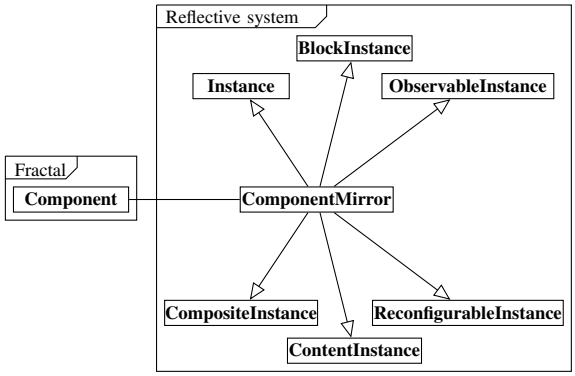

Figure 7. Mirror-based reflection for Fractal.

Using mirror-based reflection rises the need for synchronizing changes in the Fractal components with the changes in their mirrors. That synchronization is implemented by a propagation mechanism in both directions. On one side, mirrors invoke the native reflective operations of Fractal. On the other side, Fractal controllers must be enhanced to notify the mirrors of any change. Further discussion of the causal connection between Fractal components and their mirrors is beyond the scope of this paper.

Fractal and the reflective system are structured differently. Consequently, associations between Fractal runtime entities and mirrors are not necessarily one-to-one associations. One example appears when reifying connector instances which have no Fractal counterpart. The implementation of the reflective system requires the mirrors of connector instances to be associated to all of the connected Fractal components.

In addition, the association between runtime entities and mirrors can be weakened such that one can exist without the other. This would allow to drop mirrors during normal execution, and to connect mirrors only when they are required to perform reflection operations. Mirrors can either persist in a repository, or be dynamically instantiated on demand. In our prototype, the latter facility is implemented thanks to native Frac-

tal introspection facilities, though it could have resulted from interactions with the development environment.

Mirrors are not required to reify the whole application. They would rather focus on the components involved in reflection operations. Therefore, a specific view of the application is provided for the design of reconfigurations. This view focuses on points of interest, and it is presented thanks to the specific language defined by the abstract reflective system.

Some concepts of the abstract reflective system of Section 2 are missing in the Fractal component model. For instance, Fractal does not model connectors. This does not mean that those concepts have to be implemented in order to extend the component model. Indeed, applications are still modeled thanks to regular Fractal; connectors are only used when applying reflection operations. Therefore, it is sufficient to implement a minimal system for a specific Fractal connector, whose instances mirror the Fractal bindings. Operations applied to those instances simply reflect as method calls to the standard Fractal controllers.

4.2. Intrusiveness

Regarding the metamodel, the described approach is not intrusive at all. The two metamodels, Fractal and abstract reflective system, are independent. They define two distinct languages for modeling applications. The first one, Fractal, is used by designers and developers for operational purpose. The second one, the abstract reflective system, is used to design reconfigurations. Neither the component model nor the programming model of Fractal are changed.

At the level of the implementation, Fractal needs to be enhanced even if it natively supports reflection. Those modifications implement an automatic synchronization scheme of the running application with its mirrors in the abstract reflective system. New features such as the concept of connector lie outside the Fractal implementation. If the choice was made to make the synchronization explicit, it would not have been required to update the Fractal implementation. If Fractal were not natively reflective, larger modifications would have been required in order to integrate some minimal support for reflection.

5. Related works

5.1. Reflective systems

Mirror-based reflection [12] is an implementation strategy that consists in separating the reflective system from the implementation of the modeling concepts. Bracha and Ungar advocate for designing the reflective system in tandem with the programming language. The size of the reflective API is then considered as a clue of excessive complexity of the designed language. Even if this usage must be acknowledged, it should not be decisive for the design of reflective systems.

In addition, Bracha and Ungar [12] propose that several reflective systems coexist in order to reflect both the virtual machine language and the high-level programming languages. In this paper, we propose to go one step further as we propose to design a specific language for the reflective system. Application models are then automatically mapped onto that specific language.

Lorenz and Vlissides [13] propose to abstract the interfaces of the reflective system. This idea mainly aims at reducing the coupling between reflective systems and their clients. Our work rather focus on a methodology for the design of reflective systems. Working at the metamodel level, it implicitly retains the idea of defining an abstract interface for the reflective system and enforces the separation with the modeling language.

In Fractal [6], the native reflective system is integrated within component controllers. It provides operations to manipulate at runtime the abstractions introduced by the component model. Despite different implementation strategies, most of the other reflective systems are likewise integrated with their respective programming or modeling language (OpenCOM [14], Java [15]), or designed specifically for a particular language (SOUL [16]). Consequently, techniques such as making components quiescent [17] (or similar properties) can hardly be reused “off the shelf”, even if they implement orthogonal concerns.

5.2. Separation of concerns in metamodels

The methodology presented in this paper strongly relies on separating concerns in model engineering. On top of metamodelization languages and tools such as MOF, Kermeta [2] and ATL [18], some projects have appeared to help the modularization of models and metamodels. Some of them may support the methodology that is depicted in this paper.

Theme/UML [19], [20] and its extension KerTheme [4] transpose the ideas of aspect oriented programming [21] to model engineering. Those projects give a mean to model aspects at the same level as objects and classes. The considered models are made of an executable class diagram associated with some sequence diagrams. Join points and advice blocks target those two diagrams.

Muller [3] defines the concept of parameterized model. Parameters consist in a pattern model, which can be matched against a base model. In order to compose the parameterized model with a base model, parameters are substituted by some elements of the base model that match the pattern. The resulting model is merged into the base model.

In comparison to KerTheme, Muller proposes that effective parameters are given explicitly, while aspects advocate for matching automatically the join points. Muller focuses on structural models, while KerTheme considers behaviors.

Aspect oriented programming and templates have proven useful to express object oriented design patterns. We can therefore expect them to be helpful when designing and assembling the patterns of Section 2.

France *et al.* [5], [22] propose a generic model composition framework. The framework distingues the selection of elements and the merge of matching elements. This model composition operator seems to fit the mapping phase of the abstract reflective system onto the concrete modeling language of Section 4. That mapping is actually a selection of matching elements.

In addition, the methodology underlying the work of France *et al.* [22] is similar to the one we propose in this paper. Indeed, France *et al.* propose that their generic composition framework is merged into a concrete metamodel thanks to some composition strategy. The composition strategy gives the list of composable elements and the merge operation, in a similar way we indicate the list of reflected elements and the reflection operations.

In the work of France *et al.*, the composition framework has a secondary role as soon as it is merged into the composable metamodel. Unlike France *et al.*, we give the abstract reflective system the central place. Specifically, details of the concrete modeling language shall be abstracted when reconfigurations are performed. That way, reconfigurations might be expressed independently of the concrete modeling language.

6. Conclusion and future works

In this paper, we have described an approach that revisits how to build reflective systems. It has been experimented with the Fractal component model.

With regard to traditional methods, this approach abstracts from any concrete modeling or programming language. The proposed approach fits well the context of domain-specific modeling languages, as it discourages the common practice consisting in designing reflective systems specifically to each language. In fact, language implementers still have to design a reflective kernel. But that minimal kernel needs only provide manipulation primitives, while high level operations are outsourced in the external reflective system. In addition, the coupling between reflective system clients and concrete languages is completely removed. Therefore, mechanisms implemented at the abstract level can be reused whatever the concrete language is. Last, the approach gives a mean to describe the reflective capabilities of existing systems.

Even if we focus in this paper on reflective systems, the described approach applies to several other fonctionalities. France *et al.* [22] follow a similar method when they design their generic framework for static composition of model elements. One could also envision to use the same technique for distribution, deployment and temporal constraints. All those functionalities have in common to cut across the modeling languages.

Currently, the proposed approach is manual. Combining metamodel patterns into the abstract reflective system, mapping the latter onto a concrete language, identifying missing concepts: everything is done by hand. The only exception is the abstraction of the concrete language during reconfigurations, which results from the mirror-based implementation strategy. In our ongoing work, we are considering the use of existing model engineering tools in order to automate (or at least verify) the process. We are particularly interested in composition operators and aspect-oriented modeling, which seem to be good candidates to support the separation of the reflection concern.

The same methodology is currently applied to other concrete languages. In order to ensure some kind of completeness of the abstract reflective system, we consider languages outside component and object paradigms such as functionnal or synchronous languages. Furthermore, we are working on systems with no built-in reflective support, where the process presented in this paper becomes more complex as the reflection concern has to be implemented from scratch. We study in addition how to enrich that systems at the abstract level with properties such as temporal constraints. Our long-term goal is to target distributed real-time systems.

Finally, we are beginning work to build a tool to specify reconfiguration of a system based on its model. Switching from a patch approach (where the reconfiguration must be programmed) to an abstract specification is needed in systems with very complex architecture, especially when the runtime architecture does not meet the design time one. In this context, the presented approach seems to be a good base to make this tool generic with respect to the modeling language.

Acknowledgement

This work has been funded by the French ministry of research through the SPaCIFY consortium (ANR 06 TLOG 27).

References

[1] G. Nordstrom, J. Sztipanovits, G. Karsai, and A. Ledeczi, "Metamodeling: rapid design and evolution of domain-specific modeling environments," in *Conference and Workshop on Engineering of Computer-Based Systems*, Nashville, USA, Mar. 1999, pp. 68–74.

[2] P.-A. Muller, F. Fleury, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J.-M. Jézéquel, "On executable meta-languages applied to model transformations," in *Model Transformations In Pratice Workshop*, Montego Bay, Jamaica, Oct. 2005.

[3] A. Muller, "Reusing functional aspects: from composition to parameterization," in *Aspect-Oriented Modeling Workshop*, Lisbon, Portugal, Oct. 2004.

[4] O. Barais, J. Klein, B. Baudry, A. Jackson, and S. Clarke, "Composing multi-view aspect models," in *International Conference on Composition-Based Software Systems*, Madrid, Spain, Feb. 2008, to appear.

[5] R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh, "Providing support for model composition in metamodels," in *International Enterprise Distributed Object Computing Conference*, Annapolis, USA, Oct. 2007, pp. 253–266.

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The Fractal component model and its support in Java," *Software: Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, Sep. 2006, spec. issue on experiences with auto-adaptive and reconfigurable systems.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, ser. Addison-Wesley professional computing series. Addison-Wesley, 1995.

[8] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213–249, Jul. 1997.

[9] S. Matougui and A. Beugnard, "How to implement software connectors? a reusable, abstract and adaptable proposal," in *Distributed Applications and Interoperable Systems*, ser. LNCS, vol. 3543, Athens, Greece, Jun. 2005, pp. 83–94.

[10] C. Tibermacine, "Contractualisation de l'évolution architecturale à base de composants: une approche pour la préservation de la qualité," Ph.D. dissertation, University of South Brittany, Oct. 2006, in French.

[11] M. Leclercq, A. E. Ozcan, V. Quéma, and J.-B. Stefani, "Supporting heterogeneous architecture descriptions in an extensible toolset," in *International Conference on Software Engineering*, Monneapolis, USA, May 2007, pp. 209–219.

[12] G. Bracha and D. Ungar, "Mirrors: design principles for meta-level facilities of object-oriented programming languages," in *Conference on Object Oriented Programming Systems Languages and Applications*, Vancouver, Canada, Oct. 2004, pp. 331–344.

[13] D. Lorenz and J. Vlissides, "Pluggable reflection: decoupling meta-interface and implementation," in *International Conference on Software Engineering*, Portland, USA, May 2003, pp. 3–13.

[14] G. Coulson, G. Blair, M. Clarke, and N. Parlavantzas, "The design of a configurable and reconfigurable middleware platform," *Distributed Computing*, vol. 15, no. 2, pp. 109–126, Apr. 2002.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification, 3^{rd} edition*. Addison-Wesley, 2004.

[16] R. Wuyts, "Declarative reasoning about the structure of object-oriented systems," in *Technology of Object-Oriented Languages*, Santa-Barbara, USA, Aug. 1998, pp. 112–124.

[17] J. Kramer and J. Magee, "The evolving philosophers problem: dynamic change management," *IEEE Transaction on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, Nov. 1990.

[18] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "ATL: a QVT-like transformation language," in *Conference on Object Oriented Programming Systems Languages and Applications*, Portland, Oregon, Oct. 2006, pp. 719–720.

[19] E. Baniassad and S. Clarke, "Theme: An approach for aspect-oriented analysis and design," in *International Conference on Software Engineering*, Edinburgh, UK, May 2004, pp. 158–167.

[20] S. Clarke and R. Walker, *Aspect-Oriented Software Developement*. Addison Wesley, 2005, ch. Generic aspect-oriented design with Theme/UML, pp. 425–258.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *European Conference on Object-Oriented Programming*, ser. LNCS, vol. 1241, Jyväskylä, Finland, Jun. 1997, pp. 220–242.

[22] F. Fleurey, B. Baudry, R. France, and S. Ghosh, "A generic approach for automatic model composition," in *Aspect-Oriented Modeling Workshop*, Nashville, USA, Oct. 2007.