The Case for Meta-modeling Frameworks Specialisation

S. Temate¹, L. Broto¹, and D. Hagimont¹
¹IRIT/ENSEEIHT, 2 rue Charles Camichel - BP 7122
31071 Toulouse cedex 7 (France)

Abstract—Domain Specific Modeling Languages (DSMLs) are increasingly used today as they allow users to express strategies without being programming experts. This is particularly true for graphical DSMLs, inspired from UML. Implementing a DSML means providing specific editors and an execution machine.

Many experiments showed that implementing specific graphical editors is much manpower consuming. Our analysis is that most framework for building such editors (e.g. EMF/GMF) are generic, i.e. aim at fulfilling the requirements of any field, which leads to increased complexity. If domain specialization was successful for languages, why don’t we apply it to frameworks?

In this paper, we argue for Domain Specific Modeling Frameworks (DSMFs) for building DSML editors in specific application fields. We describe our experiment in implementing such a DSMF devoted to component-based DSML. We then generalize this proposal: we believe that DSMFs could be metamodeled in the same way as DSMLs are metamodeled.

Keywords: Meta-modeling, DSML, autonomic management, components

1. Introduction

In the context of the TUNe project [1], [2], we are investigating the design and implementation of an autonomic administration system, relying on a component-based middleware. In order to help the definition of administration policies, we explored the design of graphical Domain Specific Modeling Languages (DSMLs) for describing software to deploy, configure, monitor and reconfigure. From our experiments, we noticed that there is a need to design different DSMLs according to the considered administration facets and application domains. To implement the editors associated with these DSMLs, we initially relied on well known frameworks like EMF/GEF/GMF, but we found that they are overly complex. Therefore, we decided to implement our own framework. This framework called yTUNe allows designing a DSML in terms of its metamodel and provides a fully automatized generation of the associated editor. This is possible because we are focussed on component-based DSMLs (in the context of TUNe), thus allowing to wire in yTUNe how the concepts from the DSML metamodel are represented in the editor.

From a wider point of view, we consider that:

• yTUNe is a Domain Specific Modeling Framework (DSMF) devoted to the design of component-based DSMLs (and generation of their editors)
• it could be possible to implement different DSMFs for different domains
• DSMFs could be metamodeled in the same way as DSMLs are metamodeled
• the DSMF implementation could rely on a common graphical framework.

The rest of the paper is structured as follows. Section 2 presents the motivations which led us to the design of a DSMF for building component-based DSML editors. Section 3 describes the design and implementation of this framework. Section 4 explains how we plan to generalize this approach. We conclude in Section 5.

2. Motivations

In this section, we present the motivations which led us to the design of a DSMF for building DSML editors.

2.1 Application needs

The research described in this paper takes place in the context of a project which aims at designing and implementing an autonomic administration middleware called TUNe [1]. An autonomic administration system allows the deployment, supervision and reconfiguration of software in a distributed environment. TUNe is a successor of the Jade [3] system which is an autonomic management system based on the Fractal component model [4]. Each software managed by Jade is encapsulated in a Fractal component which provides a generic interface allowing its management (essentially starting, stopping and configuring it). However, with Jade we observed that Fractal interfaces were too low level and difficult to use. The management behavior had to be implemented in Java and had to invoke Fractal’s API. The main motivation in TUNe was to introduce higher abstraction formalisms (DSML) to hide the details of Fractal. To enable such an autonomic administration, we had to introduce several DSMLs:

• for node diagrams. We defined a DSML to describe hardware environments in which applications are deployed and administrated. This DSML allows a graphical description of the topology of the environment as a set of interconnected clusters (groups of machines sharing the same characteristics).
for deployment diagram. We defined a DSML to describe the initial deployment of applications. This DSML allows a graphical description of a set of interconnected software elements, which forms a software architecture.

- for reconfiguration diagram. We defined a DSML to describe reconfigurations which are triggered by monitoring probes. Such a reconfiguration is defined as a state diagram where states are annotated with administrative actions.

Moreover in the TUNe project, we experienced that it could be very convenient to specialize the above DSMLs for particular application contexts:

- intensional diagrams. Grid applications generally require hundreds of servers and it is not practical to describe each of them in extension. Therefore we designed a language which allows describing a software architecture in an implicit way, we called it *description in intension*. The principle is to describe the application deployment pattern by using cardinalities as in a class diagram. Therefore, a described component or link can lead to the deployment of many (component or link) instances.

- application specific deployment diagrams. It is often interesting to issue a DSML specific to a given application as it captures constraints from this application. For instance, we designed such a DSML for clustered JEE applications [5] composed of web (e.g. Apache), application (e.g. tomcat or Jboss) and database (e.g. MySQL) tiers. This DSML only allows defining coherent JEE architectures (e.g. it is not possible to interconnect an Apache server with a MySQL server).

In summary, our experience in the TUNe project revealed the need for administration languages which are specific to the considered administration facets and specific to the application domains. If not all, many of these DSML have a graphical syntax and the implementation of editors associated with these DSML is the main issue addressed in this article. An important common denominator of the above DSML is that they all address the description of architectures composed of components, connectors, bindings and attributes.

Notice that in other previous research projects related to the definition of component models (such as Fractal [4] or SCA [6]), we also had to implement such graphical editors for defining component architectures, which emphasizes the need for tools that help implementing such editors.

### 2.2 Existing tools

In the MDE context, the *abstract* syntax of a DSML is defined by its *meta-model* and an instance expressed with this DSML is called a *model*. Each model must conform to the corresponding meta-model. A *concrete* syntax associated with a DSML provides users with a textual or graphical formalism to manipulate the DSML’s concepts. The abstract syntax (meta-model) and the concrete syntax are used to generate an editor for editing models which conform to the language definition.

The Eclipse Modeling Project (EMP) stands in this context. It is an open source project which is composed of several sub-projects centered on different MDE’s aspects. Some of these sub-projects such as EMF [7] (Eclipse Modeling Framework), GEF [8] (Graphical Editing Framework) and GMF [9] (Graphical Modeling Framework) address DSML design and implementation.

**EMF** is a modeling framework for specifying meta-models and managing models. From the specification of the meta-model (.ecore), a set of Java classes are produced, where each element described in the meta-model is represented as a Java class. These Java classes can later be used as the foundation for developing tools. One interesting aspect of EMF is the possibility to generate an editor from the meta-model in order to manage instances (models). But this editor uses a Tree-viewer to display these instances, and typically doesn’t provide an architectural view, thus lacking expressiveness.

**GEF** provides support for building graphical editors on top of the EMF framework. It allows the association of a graphical view with each element of the meta-model. This graphical view is implemented in the lower-level Draw2D framework which is a standard 2D drawing framework based on SWT from Eclipse. Designing a graphical view using GEF consist in writing Java code which can be a very painstaking work and this is why GMF came into existence.

**GMF** provides the ability to generate graphical representations of DSMLs, on top of EMF and based on the infrastructure provided by GEF. To implement a graphical editor for a DSML with GMF, developers need to construct a number of intermediate models which are used to generate the Eclipse plug-in which implements the editor. The *domain* model defines the abstract syntax of the language (.ecore meta-model). The *graph* model specifies the shapes that will be used in the editor. The *tooling* model specifies which tools will be available in the editor palette (e.g. a button to create an element in the language). Finally, the *mapping* model specifies how the shapes of the graph model are mapped with the concepts of the meta-model. GMF also provides a wizard which can derive first rough versions of the intermediate models (graphical, tooling and mapping) from the domain model. Then these generated intermediate models can be refined.

GMF is presented as the most complete framework for generating model’s graphical editors. However, similarly to experiments reported in [10], our experiments revealed that GMF is quite complex, and not as user friendly as we had like it to be. Evaluations reported in [11] show that only 12% of users find GMF easy to use. The wizard which is responsible for generating the initial versions of
the intermediate models works well only for very simple DSMLs, and when the user diverts from the wizard default solution, he has to hand-code what he really wants (with GEF).

Our analysis is that most framework for building such editors (e.g. EMF/GMF) are generic, i.e. aim at fulfilling the requirements of any field, which leads to increased complexity. Specializing such a framework according to the constraints of a domain (in our case component-based DSML) would allow keeping the definition of a specific editor simple, while fulfilling the requirements of the considered domain. Therefore, in the next section, we describe the design of a Domain Specific Modeling Framework which allows generation of DSML editors in the area of component-based languages.

3. A Domain Specific Modeling Framework

3.1 Design principles

We designed and implemented a modeling framework called yTUNe. This framework provides a meta-modeling environment for editing meta-models. From the edited meta-models, users can generate graphical editors for models edition. The yTUNe meta-modeling language basically uses MOF concepts and is dedicated to the design of component-based languages. The aim of yTUNe is then to provide a user-friendly support allowing fully automatized generation of graphical editors, for modeling component-based architectures.

Regarding fully automatized generation of model editors, our idea consists in designing the yTUNe meta-model (which allows defining component-based DSML meta-models) so that a common concrete syntax can be applied to the targeted languages. It means that the yTUNe meta-model defines all the component-based concepts which shall be used in a DSML meta-model, and the common concrete syntax defines the types of graphical representations these concepts should have. The definition of the yTUNe meta-model is based on our knowledge of the type of concepts which are usually used in our component-based domain, and we also know the usually used graphical representations. However, the graphical representations can be configured/adapted when designing a particular DSML with yTUNe. Then the editor generation is achieved by simply interpreting the graphical elements associated with the concepts included in the DSML meta-model.

3.2 The yTUNe Modeling Framework

3.2.1 Meta-modeling environment

The meta-modeling environment provides specific component-based constructions which are:

- **Component**: It is a cohesive unit of attributes and properties, which represents the structuring unit that we call component in our domain. The graphical representation of a component is a shape which can be specialized in the yTUNe meta-modeling editor.
- **Connector**: This concept corresponds to a connection point (also called interface in the component domain). The graphical representation of a connector is a shape which can be specialized in yTUNe.
- **Binding**: In the component domain, components’ connectors can be linked together to constitute a component architecture. The Binding concept represents these links. The graphical representation of a binding is a link (which can also be graphically specialized).
- **Aggregation**: an aggregation is a relationship which allows combining a component to a connector. Graphically it is characterized by the shape of the connector stuck with the shape of the component. This relationship always has a 1..n cardinality where only the n is settable. This cardinality is a constraint which enforces that in the generated editor a component can be stuck to n connectors but a connector can be stuck to only one component.
- **Association**: an association is a relationship with two ends, where one end is on a connector and the other end on a binding. A binding can only be associated with two connectors. If the user wants (in a meta-model) to associate more than two connectors with a binding, e.g. he wants in a model to allow a binding of type B which links connectors of type C1, C2 or C3, he can define in the metamodel an abstract connector C from which C1, C2 and C3 inherit and associate B with C.

The graphical representations described above are statically defined in yTUNe, but they can be specialized. There are two Components properties which are concrete syntax elements:

- **Schematic property**: It is one of the graphical elements added in the meta-model definition. This property allows to assign a graphical representations to a component thanks to a drawing editor provided in yTUNe. The user can draw its desired representation or import an existing image.
- **Abstract property**: This graphical property allows to specify whether a component can be instantiated or not. According to its value, a creation tool will be available in the intended editor palette or not.

Besides these specific concepts, the meta-modeling environment also offers some elementary constructions provided by most of the existent meta-modeling environments: attributes, composition and inheritance relations.

We use cardinalities to restrain our languages, but to allow complete definition of DSL, we are working on integrating an OCL like constraint definition.
3.2.2 Editor generation

Once the DSL (meta-model) is specified and saved, generation of the models editor is achieved in only one click. In fact at this step the meta-model includes all the necessary information for generating the editor on the fly. The meta-model is interpreted and a model editor is generated according to the semantic assign to each concept defined in the meta-model (semantic described above) and their graphical representation.

The generated editors are enough user-friendly, and the edited models are conform to their corresponding meta-model definition. Next we illustrate the yTUNe editor generation process using two running examples.

3.3 illustrative examples

The two running examples we present here are representative enough of our specific requirements (components, binding, connectors, attributes . . . ).

a) J2EE: The first example is a configuration language for a J2EE platform. The J2EE platform defines a model for developing web applications in a multi-tiered architecture. Such applications are typically composed of a web server an application server and a database server. This use case consist in designing a configuration language for a J2EE platform, and then generating the corresponding J2EE graphical model editor. This configuration language designed with yTUNe is depicted in Figure 1. The web server is the apache component with its mod_jk connector, the application server is the tomcat component with its ajp13 and datasource connectors and the Database server is the mysql server with its WAClients connector.

![Fig. 1: J2EE Configuration language modeled with yTUNe](image)

b) Fractal: The second example is a description language of the Fractal component model. Components have functional (client and server) and non-functional (control) interfaces, they are linked together thanks to Links to constitute a software architecture. The Fractal component model enables composite components which are components that contains other components. A basic meta-model of Fractal’s ADL (Architecture Description Language) designed with yTUNe is depicted in Figure 2 and shows all the basic concepts of Fractal, which are: components, functional interfaces (Client, Server) and non-functional interfaces (Control).

Figure 3 shows the two editors generated from the above meta-modeled DSML.

![Fig. 2: Fractal ADL modeled with yTUNe](image)

![Fig. 3: (a)J2EE editor. (b) Fractal editor.](image)

Notice that in the meta-models described in Figures 1, all the components which describe servers has been assigned the corresponding server’s icon as graphical representation. Then models edited in the J2EE generated editor (Figure 3(a)) are very expressive. Also the defined J2EE configuration language enforces that only a consistent j2EE architecture can be defined (e.g. it is not possible to interconnect an Apache component with a MySQL component).

Regarding the Fractal meta-model, Figure 2 shows that the Interface connector is aggregated with the Component component. In the generated Fractal model editor (Figure 3(b)) it is represented with Interface instances stuck on Component
Fully automatized generation of such editors has been made possible because we have specialized the meta-model of our modeling framework (for the area of component-based languages) and associated a graphical concrete syntax with it (even though this syntax can be specialized for a given language). We believe that such an approach can be generalized in order to allow generation of different types of yTUNe frameworks according to different domain requirements. This vision is explained in the next section.

4. Generalization

4.1 Vision

We believe that this approach (implementing a DSMF for component-based DSMLs) can be generalized to issue different DSMFs associated with different fields. For instance, we are targeting a DSMF which could be devoted to the design of DSMLs for describing timing diagrams such as Gantt charts. Therefore, in the same way as DSMLs are defined in terms of a DSML metamodel (from which an editor is generated thanks to a DSMF), a DSMF can be defined in terms of a DSMF metamodel (from which an DSML metamodel editor is generated). Figure 4 illustrates this generalized approach. The generation of a

Fig. 4: Our vision

DSML metamodel editor from a DSMF metamodel cannot be fully automatized. At this step, the designer of a DSMF must associate with the DSMF metamodel a specification of the edition behaviour (a concrete syntax). For instance, the component-based DSMF metamodel defines Component, Connectors, Binding, Association and Aggregation concepts. The edition behaviour specification must define how these concepts are managed graphically. We are currently refactoring the yTUNe prototype to support both a component-based DSMF and a timing-based DSMF, while factorizing libraries and interfaces as much as possible. This should led us to the design of a framework for implementing DSMFs, which would allow combining a DSMF metamodel and reusable graphical libraries.

4.2 Positioning

This vision can be positioned compared to the Eclipse Modeling Project as follows.

From the point of view of the generation of the DSMF, we believe that this generation cannot be fully automatized, but that a implementation framework can help developing DSMF and promote software component reuse (libraries of graphical behaviors). This approach can be compared to GEF as the level of abstraction is quite similar. However, this comparison only holds from a narrow part of our proposal, as GEF does not target DSMF but DSML.

From the point of view of a particular DSMF which allows fully automatized generation of DSMLs' editors, this approach can be compared to GMF which aims at generating graphical editors with minimal development. However, since GMF is generic and does not target specific domains, it fails addressing the needs of these specific domains.

Concerning the related work, we did not found any similar proposal in the literature. The closest work to our is that of Nordstrom et al. [12] who propose to describe DSL meta-models with UML and OCL, and to associate presentation specification with such a meta-model to obtain a modeling environment (editor). However, this association of the graphical syntax with the concrete syntax is not detailed and will probably require development. This proposal is similar to our regarding the generation of DSMFs (which are modeling environments for DSMLs). We believe that this DSMF generation will require much expertise, which should be only required to generate DSMF, DSMFs being dedicated to a specific domain and much simpler to use for generating a DSML editor.

5. Conclusions

We are conducting research on a key issue in the MDE community which is the development of editors associated with DSMLs. Many experiments showed that the standard tools in the domain (i.e. EMF/GEF/GMF) are overly complex and difficult to use. Our believe is that these frameworks are too complex because they aim at being generic. Instead, we claim that we should apply the principles of DSML (Domain Specialization) to these frameworks. By doing so, DSML designers in a given field will be able to easily design DSMLs and generate the associated editors.
Regarding this orientation, the goal of this paper was two-folds:

- to describe our experience in designing and implementing one domain specific modeling framework (DSMF) devoted to (i) the design of component-based DSMLs and (ii) the generation of dedicated editors.
- to present a vision of the ideal framework which would allow generation of DSMF.

References


