Using Model Driven Engineering technologies for building authoring applications

Olivier Beaudoux, Arnaud Blouin, Jean-Marc Jézéquel

Using Model Driven Engineering technologies for building authoring applications

Olivier Beaudoux
ESEO-GRI
Angers, France
olivier.beaudoux@eseo.fr

Arnaud Blouin
INRIA
Rennes, France
arnaud.blouin@inria.fr

Jean-Marc Jézéquel
INRIA/IRISA
Rennes, France
jezequel@irisa.fr

ABSTRACT

Building authoring applications is a tedious and complex task that requires a high programming effort. Document technologies, especially XML based ones, can help in reducing such an effort by providing common bases for manipulating documents. Still, the overall task consists mainly of writing the application's source code. Model Driven Engineering (MDE) focuses on generating the source code from an exhaustive model of the application. In this paper, we illustrate that MDE technologies can be used to automate the development of authoring application components, but fail in generating the code of graphical components. We present our framework, called Malai, that aims to solve this issue.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; I.7.1 [**Document and Text Processing**]: Document and Text Editing

General Terms

Document design

Keywords

MDE, authoring applications, Malai, Malan

1. INTRODUCTION

As pointed out by Quint and Vatton, "*Traditional methods for editing structured documents are not sufficient to address the new requirements. New techniques must be developed or adapted to allow more users to efficiently create advanced XML documents*" [10]. This induces the development of more complex authoring applications that propose rich user interfaces. Such development thus becomes a tedious and complex task that requires more and more programming effort. XML technologies can help in reducing such an effort by providing common bases for managing XML documents. However, these technologies do not avoid the need of writing numerous lines of code. To solve that kind of complex problem, one usually tries to break down a system into as many *models* as needed in order to address all the relevant concerns. Models have been used for a long time as *descriptive* artifacts, which is already extremely useful. Here we want to go beyond that, *i.e.* we want to be able to perform computations on models [9].

In this paper, we propose the use of MDE technologies for generating the code of powerful authoring applications. We analyze well known MDE technologies that come with the Eclipse platform [5] and its complementary Kermeta platform [9], regarding their capability to generate components of authoring applications (section 2). We underline that current MDE technologies offer good generative capabilities except for the important case of graphical components; to solve such an issue, we propose our Malai framework [3], and its MDE integration within Eclipse and Kermeta (section 3).

2. USING CURRENT MDE TECHNOLOGIES

In order to fit the way we teach RelaxNG to ESEO students, we have developed our own RelaxNG authoring application based on Eclipse and Kermeta MDE platforms (see figure 1). The *RelaxNG editor* allows editing grammars in the compact format ①. The editor source code has been generated by EMFtext [6] from two models ②: the RelaxNG abstract syntax specified in file *RelaxNG.ecore*, and the compact format concrete syntax specified in the *RelaxNG.cs* file. Files *constraints.kmt* and *rnc2rng.kmt* complement this generated code by respectively defining the constraints that RelaxNG grammars must respect ③, and the transformation of RelaxNG grammars from its compact format to its XML format ④.

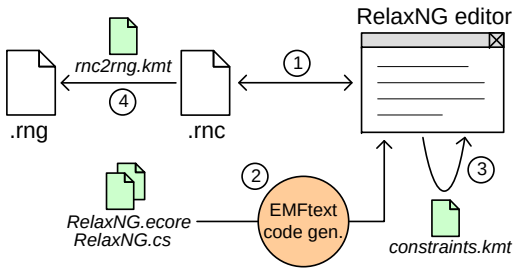

Figure 1: Synopsis of our RelaxNG application

The following sections briefly explain the design of the four MDE files involved in figure 1. Full versions of all these files can be freely downloaded from `http://gri.eseo.fr/software/relaxng`.

2.1 A model of the RelaxNG abstract syntax

The first step of any MDE development consists of specifying the model of the application's domain data. Under Eclipse, such a model is specified by an Ecore diagram that represents a simplified UML class diagram. Figure 2 is the excerpt of file *RelaxNG.ecore* that concerns the hierarchical definition of XML documents.

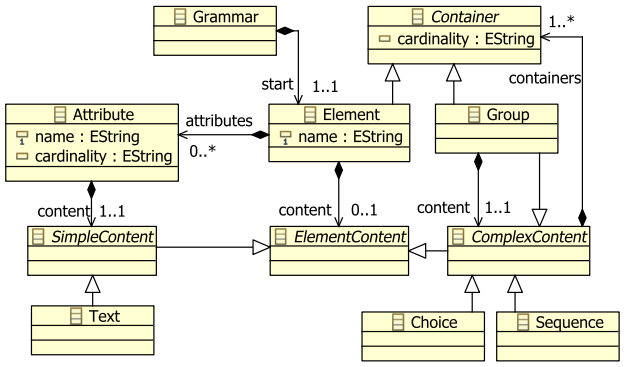

Figure 2: An Ecore model for RelaxNG grammars

The *Grammar* starts with the definition of its root element (relation *start*). An *Element* definition includes a *name*, some *attributes*, and optional *content*; it is considered as a *Container* and thus includes a *cardinality*. An *Attribute* definition includes a *name* and a *content*, and can be optional (attribute *cardinality*). The content of an attribute is always a *SimpleContent*. The *ElementContent* can be either a *SimpleContent*, a *ComplexContent*, or a *Group*; a *ComplexContent* can be a *Choice* or a *Sequence* of *Element*s or *Group*s.

2.2 A model of the RelaxNG concrete syntax

EMFtext allows the generation of powerful text editors [6] from the Ecore model of the abstract syntax (*e.g.* file *RelaxNG.ecore*) and the EMFtext model of the concrete syntax (*e.g.* file *RelaxNG.cs*). The following code gives a small excerpt of the *RelaxNG.cs* file:

```
TOKENS {
   DEFINE ID $('a'..'z'|'A'..'Z'|'_'|'-')+$;
   DEFINE CARD $'?'|'*'|'+'$;
}
RULES {
   Grammar ::= start;
   Element ::=
      "element" name[ID]
      "{" (
         "empty" | ( children ) |
         (( attributes ",")* ( attributes | children ))
      ) "}"
       cardinality [CARD]? ;
}
```

The concrete syntax defines the language tokens and its grammar rules. The rules match the classes defined in the abstract syntax (see figure 2), and specify the textual syntax of attributes and relations of these classes. For example, rule *Grammar* defines that a grammar is directly represented by the *Element* of the *Grammar.start* relation; rule *Element* defines that an element is represented by: keyword *"element"*, its.*name* attribute, and its content surrounded by brackets '{' and '}'.

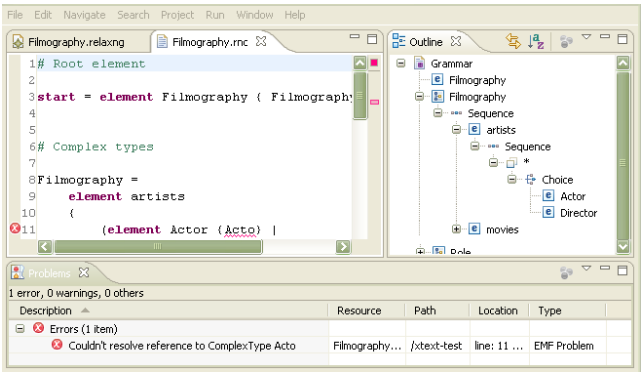

Figure 3: Screenshot of our RelaxNG editor

Figure 3 illustrates the power of the generated editor: the main window allows the editing of RelaxNG grammars with auto-completion features and error detection on the fly; the outline gives a hierarchical view of RelaxNG objects as they are defined within the model.

2.3 Definition of constraints

The model of the abstract syntax must be refined with the definition of constraints. For example, constraint *attributeUnicity* ensures uniqueness of attribute names within their element. OCL is widely used for such a purpose [11]; Kermeta complements Eclipse by allowing the definition of OCL constraints within class invariants. The following code is the excerpt of the *constraints.kmt* file that implements the *attributeUnicity* constraint using the *isUnique* OCL function; it illustrates how aspect programming of Kermeta allows the addition of operational code into classes defined in the Ecore model:

```
aspect class Element {
   inv attributeUnicity is do
      attributes.isUnique{a | a.name}
   end
}
```

2.4 Document transformation

Writing Kermeta transformations is based on adding a transformation operation on each class of the model. Such an addition is based onto the aspect capability of Kermeta: an aspect supplements an existing class with new operations without requiring the modification of the class [9]. The following Kermeta code gives the excerpt of transformation *rnc2rng.kmt* regarding class *Element*:

```
aspect class Element {
   operation toText(): String is do
      result := "<element name='" + name + "'>"
      attributes.each{a | result.append(" " + a.toText())}
      content.each{c | result.append(c.toText())}
```

```
        result .append("</element>\n")
    end
}
```

Operation *toText* returns the string representing the element in the RelaxNG compact format. It consists of tag "<*element name=...*>" built by calling operation *toText* on each element attributes and on the optional element content. This operation is implemented in the same way for each classes of the RelaxNG model.

2.5 Discussion

File *RelaxNG.ecore* features 26 classes and file *RelaxNG.cs* counts 71 lines; the resulting generated code counts around 14.000 lines of code *inside methods*. The RelaxNG authoring application has been initially developed by two students during their final-year project. They have good skills in OOP, but had no experience in using MDE and defining language abstract syntax. Consequently, they often introduced concrete aspects of the language in the model (*e.g.* class *Parenthesis),* or defined concrete aspects of the language without any entry in the model (*e.g.* no definition of attribute *cardinality*). They spent 16 days for building files *RelaxNG.ecore* and *RelaxNG.cs*, and these both files have been entirely rebuilt by the professor in charge of the student project (an expert of the domain) in 4 days. Writing files *constraints.kmt* and *rnc2rnc.kmt* only required 1 day for the professor. Even if this evaluation is superficial, it clearly shows the gain of using MDE technologies.

Graphical components are essential components for authoring environment. However, MDE does not fully help in building such graphical components. For example, the Eclipse project proposes the Graphical Modeling Framework (GMF) for generating the code of diagramming tools [5]. GMF illustrates its ability to generate the code of powerful tools, but the resulting tools remain *stereotyped* and cannot be easily personalized. For example, the Ecore editor of Eclipse has been generated by GMF, but *not* its XML Schema editor, thus showing that GMF is not well suited to the development of a RelaxNG graphical editor. The next section proposes our framework to encompass such a limitation of current MDE technologies.

3. MALAI: A MDE FRAMEWORK FOR GRAPHICAL COMPONENTS

3.1 The conceptual framework

This section explains our conceptual framework, called Malai [3], through a case study: a tree-based editor dedicated to the specification of RelaxNG grammars.

Malai organizes the user interface (UI) as depicted by figure 4. The UI is composed of the three main elements as defined by the DPI model [2]: the domain *Data*, their *Presentations*, and the *Instruments* used to interact with the data through the presentations. For our case study, the domain data is a *RelaxNG Grammar*, the presentation is a *Tree* widget with editing capabilities, and the instruments are: *Creator*s for creating RelaxNG objects (one per concrete class of the RelaxNG model), an *Eraser* for deleting objects, a *Mover* for reparenting objects (*e.g.* moving an attribute definition to another owner element), and a *Pencil* for edition object properties (*e.g.* the name of attributes or elements).

The UI is split into two parts: the *abstract part* that does *not* depend on the platform (*e.g.* a desktop PC or a mobile phone), and the *concrete part* that depends on the platform. Each UI element is modeled by its *static part* that consists of a *class diagram*, optionally supplemented by a *dynamic part* that consists of a *state machine diagram*, a *Malan* mapping, or *Malai* code.

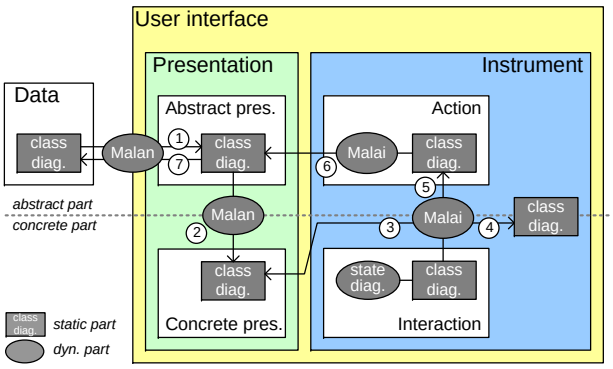

Figure 4: Principle of the Malai framework

The *abstract presentation* defines the presentation data through its class diagram that does *not* include any graphical information; in our case study, it is a tree model, analogous to the Swing's *TreeModel*. The abstract presentation is initially built from the domain data by the Malan mapping ① that maps the data, specified by a class diagram, to the presentation. The *concrete presentation* complements the abstract presentation by defining platform-dependent graphical data through its class diagram; the concrete presentation of our tree is the widget tree itself, such as the Swing's *JTree*. It is initially built by a second Malan mapping ②.

An instrument transforms user *interactions* into *actions* that operate on the abstract presentation, thus linking the abstract part of the UI with its concrete part. An *action* defines its data through a class diagram. For example, instrument *Creator* produces action *Create* that defines the *type* of the object to create, and its *owner* object within the edited grammar; instrument *Mover* produces action *Reparent* that defines the *target* object to move, and its *newOwner*. An interaction is defined by a class diagram that includes the interaction data and the events consumed by the interaction; these data are modified accordingly to the state machine diagram that handles UI events (*e.g.* a "mouse moved" or a "key pressed"). For example, a *KeyTyped* interaction is bound to *KeyPressed* and *KeyReleased* events; a *DragAndDrop* interaction is bound to *MousePressed*, *MouseMove*, *MouseReleased* and *KeyPressed* (for cancellation) events.

Whenever the state of an interaction changes, the instrument performs a *feedback* to the concrete presentation ③, and to the instrument itself ④. For example, when interaction *DragAndDrop* is in the "drag" state, instrument *Mover* highlights the possible "drop" target of the concrete presentation ③, and changes its cursor shape ④. In the same time, the instrument updates the related action ⑤ thus transforming the interaction into a tangible action. For example, instrument *Mover* transforms interaction *DragAndDrop* into action *Reparent* by initially setting its *target* and subsequently modifying its *newOwner*. Both the feedback and the transformation are specified by Malai code. In turn, an

updated action modifies the abstract presentation in a way defined by the Malai code ⑥. For example, action *Reparent* induces a move of the *target* object of the presentation into the *newOwner*. Since Malan mappings establish a durable link between its source and its target, the concrete presentation is subsequently updated when the presentation is updated ②. Moreover, domain data is also updated by the reverse-side ⑦ of Malan mapping ①, thus allowing the data to be shared by multiple presentations.

3.2 MDE integration

Research works have already proposed frameworks dedicated to interactive documents [2, 7, 8]. However, they are not based on an MDE approach and do not focus on generating the code of the final authoring application[1].

Although Malai has been designed in the spirit of MDE, it does not currently provide MDE tools. The Malai framework is currently implemented as a Java GUI toolkit, and has been partially evaluated through the development of three interactive applications [3]. The Malai code is currently defined using a pseudo-language [3], and should evolved into a concrete language. Moreover, Malan defines it own source and target model, rather than using a predefined one such as Ecore [4].

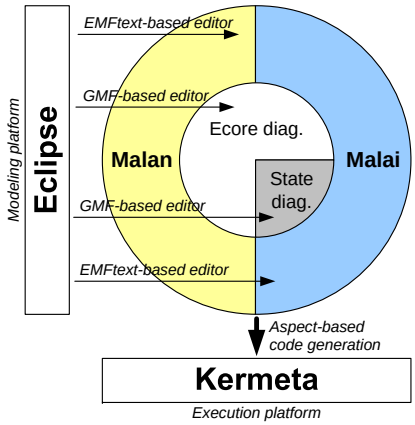

Figure 5: MDE integration of Malan-Malai

We are currently working on integrating Malan and Malai in a complete MDE environment, as described in figure 5. In such a scheme, Malan uses Ecore class diagrams for specifying source and target models, and Malai uses Ecore diagrams and state machine diagrams for specifying interaction models; these models are graphically edited with Eclipse GMF-based editors. Malan and Malai are textual languages respectively dedicated to the definition of mappings and interactions that are textually edited through EMFtext-based editors. These four graphical or textual languages (Ecore, State machine, Malan and Malai) allow the generation of code based on the aspect-programming capabilities of Kermeta; moreover, Kermeta active operations [1] will be used to implement Malan mappings. In such a schema, Eclipse thus plays the role of the modeling platform, while Kermeta plays the role of the execution platform.

[1]Related works on model-based user interface development environments (MB-UIDE) can be found in [3].

4. CONCLUSION AND PERSPECTIVES

In this paper, we propose the use of MDE technologies for generating the code of authoring applications. We demonstrate through a real example that current Eclipse and Kermeta platforms allow the generation of text editor components that can include powerful editing capabilities such as on-the-fly document validation, outline view of the document, text completion, constraint validation and document transformation. However, MDE technologies currently failed in generating the code of graphical components, except for stereotyped and limited ones. We thus propose a MDE integration of our Malai framework [3, 4] to solve such an issue. The integration is based on Eclipse being used as the modeling platform, and on Kermeta as the execution platform.

The next step of our work is to use the resulting MDE Malai platform for generating applications such as a RelaxNG graphical editor. This generative process will allow us to evaluate the performance of Malai in term of expressiveness and development cost, and in terms execution time and usability of the generated authoring applications.

5. REFERENCES

[1] O. Beaudoux, O. Barais, J.-M. Jézéquel, and A. Blouin. Active operations on collections. In *Proc. of MoDELS'10 (in press)*, 2010.

[2] O. Beaudoux and M. Beaudouin-Lafon. OpenDPI: A toolkit for developing document-centered environments. In *Enterprise Information Systems VII*, pages 231–239. Springer, 2006.

[3] A. Blouin and O. Beaudoux. Improving modularity and usability of interactive systems with Malai. In *Proc. of EICS'10*, pages 115–124. ACM, 2010.

[4] A. Blouin, O. Beaudoux, and S. Loiseau. Malan: A mapping language for the data manipulation. In *Proc. of DocEng'08*, pages 66–75. ACM, 2008.

[5] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley, 2009.

[6] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wend. *Model Driven Architecture - Foundations and Applications*, chapter Derivation and Refinement of Textual Syntax for Models, pages 114–129. Springer, 2009.

[7] J. Lumley, R. Gimson, and O. Rees. A framework for structure, layout & function in documents. In *Proc. of DocEng'05*, pages 32–41. ACM, 2005.

[8] T. Malloy. The future of documents. In *Proc. of DocEng'05*, pages 1–1. ACM, 2005.

[9] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proc. of MODELS'05*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

[10] V. Quint and I. Vatton. Techniques for authoring complex XML documents. In *Proc. of DocEng '04*, pages 115–123. ACM, 2004.

[11] J. B. Warmer and A. G. Kleppe. *The object constraint language: getting your models ready for MDA.* Addison-Wesley, 2003.