An Early Aspect for Model-Driven Transformers Engineering

Valdemar Vicente Graciano Neto
Instituto de Informática
Universidade Federal de Goiás
P.O. Box 131
Goiânia-GO, Brazil
valdemarneto@inf.ufg.br

Juliano Lopes de Oliveira
Instituto de Informática
Universidade Federal de Goiás
P.O. Box 131
Goiânia-GO, Brazil
juliano@inf.ufg.br

ABSTRACT

This paper presents preliminary results of application of aspects to address one of the Model-Driven Development (MDD) approach key issues, namely the *pluggability principle*. According to this principle, a MDD based software solution should provide transformation definition rules that can be plugged into a model-driven transformer to provide automatic mapping between models. We identified the invocation of these transformation rules as a new early aspect for model-driven transformers engineering. This paper presents the main ideas behind this new early aspect, discussing its impact on model-driven transformers engineering.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Model-Driven—*model transformation rules, transformers modularization*

General Terms

Model-Driven Development, Aspects

Keywords

Aspects, Model-Driven Development, transformation rules modularization

1. INTRODUCTION

Aspect-Oriented Software Development (AOSD) aims at providing improved modularization and composition techniques to handle crosscutting concerns. Crosscutting concerns are encapsulated in separate modules, known as aspects, and composition, or weaving, mechanisms are later used to weave them back to the base modules in compilation, loading or execution time [16, 3].

Aspects are concerns that crosscut an artifact's dominant decomposition or base modules derived from the dominant separation-of-concerns. Early Aspects (EA) are aspects identified since the earliest phases of software development process [3]. Separation of crosscutting properties in early stages of the development and identification of their mapping and influence on later development stages makes it possible to identify conflicts and establish possible trade-offs early in the development cycle and promotes traceability of broadly scoped requirements and constraints throughout system development, maintenance and evolution [14].

Techniques to identify EA have been proposed [1, 6], but it is still an open issue for AOSD [12]. To deal with this challenge, we investigated EA on the model-driven transformers engineering specific application domain. For this domain, this paper proposes a new early aspect: the transformation rules set and its invocation.

This paper claims that these transformation rules are a crosscutting concern in model-driven transformers. The paper also argues that encapsulating these transformation rules in one aspect addresses the pluggability principle, one of the main issues in model transformers engineering.

The remainder of the paper is organized as follows: Section 2 discusses the model-driven transformers engineering; Section 3 analyses the transformation rules aspect, and the impact of this aspect on the model transformers engineering; Section 4 concludes the paper and discusses future works.

2. MODEL-DRIVEN TRANSFORMERS

According to Model-Driven Development (MDD), software should be modeled at a high-level of abstraction and *transformers* - software components specialized in converting source models into target models - should be provided to automatically map input models into source code for some specific platform or technology. This mapping is tipically based on a set of transformation rules (TR) [17].

In spite of its benefits, there is still a large gap between MDD specifications and their software implementations. In fact, MDD depends on transformers, but it does not suggest how to construct transformers and that is where considering TR as an early aspect may contribute.

The TR pluggability principle requires the transformer's software architecture to be continuously modified to deal with new transformation definitions. More specifically the pluggability principle demands TR to be connected to a transformer without impacting the transformer's software architecture while at the same time providing transparency to the software users.

Thus, the transformers architecture must be designed considering the pluggability principle. Treating TR as an aspect from the earliest phases of the transformers engineering improves the capability of their architecture to support changes in the TR set.

There are some works in literature dealing with transformers architecture issues. AndroMDA is an open source MDD framework developed for multiple target platforms based on the concept of *cartridge*. The cartridges consist of one or more files combined in a Java archive (jar), a cartridge description XML file and one or more code templates [19]. A cartridge provides a metamodel, a set of TR, and code templates to define how UML models are transformed into specific platform code. It defines transformations for web platforms, such as Spring, JSF, Hibernate, Struts, and others [15].

However, it is necessary to have deep knowledge of the AndroMDA architecture to specialize a cartridge. In our proposal, the user only needs to be aware of (1) the object that encapsulates the source model; and (2) how to construct TR. Our "cartridge" is a skeleton. Thus, just a few completions are necessary and there is no need to cope with technological issues (such as AspectJ).

The openArchitectureWare (oAW) is a tool under the Eclipse Modeling Project (EMP) [13]. Models are imported in XMI format, and there are files and Java classes that perform transformations. A cartridge is expressed as an ".oaw" file. When changes are required, it is necessary to extend a default transformer (a Java class) to add modifications in order to cope with the new cartridge. Our approach intends to create TR as an aspect that will be connected to the transformer in weaving time, which prevents direct code manipulation.

3. ASPECTS IN MODEL TRANSFORMERS ENGINEERING

Many tools and frameworks has been created to support transformations between models. For instance, the Software Engineering Research Group of the Federal University of Goiás, in Brazil, has developed a model-based domain-specific software development framework to address Enterprise Information Systems (EIS) software synthesis based on high-level conceptual models which specify data, business rules, and user interface features [2, 4].

Figure 1 shows the main components of the framework's architecture. There are five main tiers: user interface, interface-application, application, business and persistence. Each tier addresses one concern of the EIS engineering and has its own TR set, generating corresponding components of the EIS software. Inversion of Control is an important element of this architecture since the generated EIS software execution is controlled by the framework run-time engine.

Complex EIS software for the agriculture domain was successfully generated using that framework. However, when new IS requirements demanded evolutions, the high coupling of its architecture was revealed. The framework architecture did not address modularization in TR, causing changes to the framework TR to be difficult and error-prone. The framework code which dealt with model transformations was repetitive, spread, replicated and highly coupled throughout its architecture. Figure 1 illustrates such scenario.

Since each TR set is totally dependent of the framework's metamodels, the transformations are quite repetitive. Each

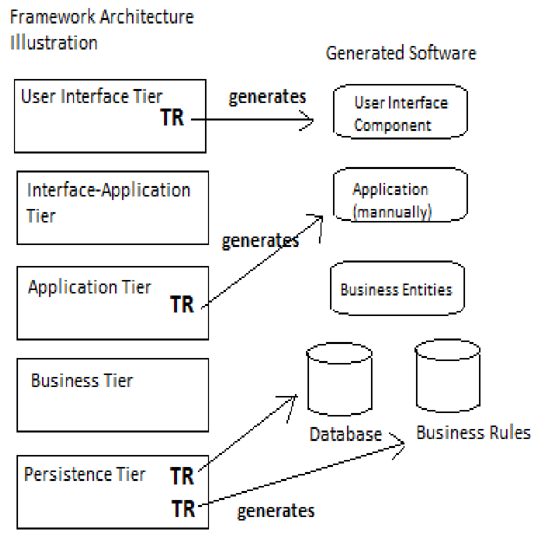

Figure 1: Framework architecture representation and TR as an aspect.

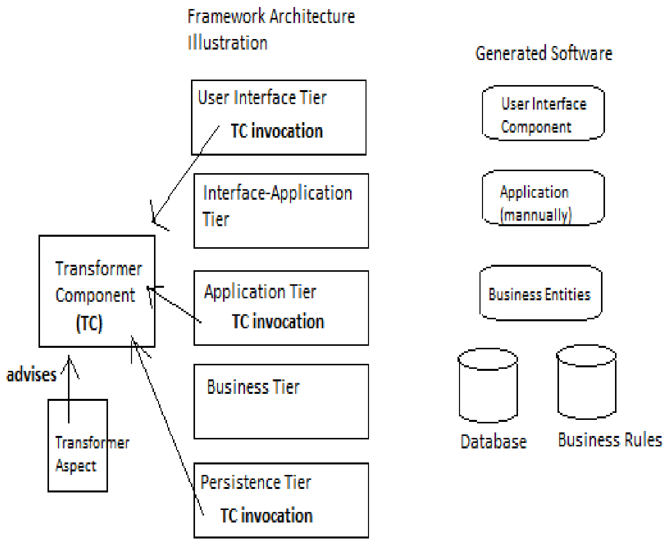

Figure 2: Framework architecture representation after refactoring.

source metamodel element is transformed in one corresponding element in a target code (for user interface, or business rule control, for example). The set of transformations for each part of the generated code leads to IF-ELSE chains (relative to each source metamodel element) been repeated along the framework's architecture, changing only the condition within the IF statement. Thus, TR are crosscutting concerns in this architecture due to metamodels dependencies.

A software refactoring of the framework architecture was initiated, as it is shown in Figure 2. The driving forces for this refactoring were: 1) to apply Aspect-Oriented Programming, benefiting from aspects' capabilities to modularize crosscutting concerns and code composing [10]; and 2) to encapsulate TR in aspects since they are variability points in model transformers components architecture [18, 5] and crosscuting concerns of the presented framework architecture.

Figure 3 presents the conceptual model that represents the proposed scenario. The tool responsible for transforming models is a **Transformer**, which loads and checks the consistency of a **SourceModel**. A SourceModel can be loaded from a file and there is a software component responsible for handling the received format, that is, receiving the model, constructing an equivalent Java object and verifying its consistency with the respective metamodel.

An aspect called **TransformationAspect** advises the transformers' code. After model loading and checking is completed, the **TransformationDefinition** is applied to convert a SourceModel into a **TargetModel**.

A TransformationDefinition is a set of **TransformationRules**. TransformationAspect works as a cartridge, analogous to the AndroMDA cartridge (including the similar content: source metamodel, source model and transformation definition) [19].

Every SourceModel conforms to a **Source Metamodel** and every TargetModel conforms to a **Target Metamodel**. A **Mapping** between elements of both metamodels configures a **TransformationRule**.

Listings 1 and 2 show, respectively, simplified samples of a transformer and a transformation aspect code. A transformer contains a SourceModel attribute, its access and modifier methods, and some other methods responsible for performing Source Model load, Source Model checking against its metamodel, and a main method to simulate the execution of a transformation.

In our approach, TR and target models are modularized into an aspect. It is possible to change the TR according to the platform because aspects technology enables us to access the transformers context and data transferred in method calls (parameters originally passed to the method) [7].

In this case, a Source Model is passed to the *checkInputModel* method and, after it finishes its execution, the transformation definition code that was woven into the transformers code converts the source model into a target model according. After the target model is produced, another software component is responsible for exporting it in the desired format.

Thus, TR become totally pluggable, since a change to a transformation definition consists of an aspect substitution to advice the same transformer, but with a new transformation code into the *transform* advice body. This approach results in modularization and pluggability, separating target technology concerns in distinct aspect bodies.

Listing 1: Transformer's code

```
package mddtransformer;
public class Transformer {
    private SourceModel source;
    private SourceMetamodel sourceMeta;

    public Transformer(){
        source = new SourceModel();
        sourceMeta = new SourceMetamodel();
    }


    private SourceModel getSource() { return ↩
        this.source; }
    private void inputSourceModel() { System.out↩
        .println("Source model was loaded"); }

    /** Check the model against the respective ↩
        model. */
    private void checkInputModel(SourceModel s) ↩
        {
        /** Code that perform checks.*/
    }

    public static void main(String[] args){
        Transformer t = new Transformer();
        t.inputSourceModel();
        t.checkInputModel(t.getSource());
    }
}
```

Listing 2: Transformation Aspect code

```
package mddtransformer;
public aspect TransformationAspect{
    pointcut transform(SourceModel s): execution↩
        (* *.*.checkInputModel(SourceModel)) &&↩
         args(s);

    after(SourceModel s): transform(s){
        TargetModel target;
        TargetMetamodel targetMeta;
        TransformationDefinition t;

        //It uses source model to convert it in ↩
            a target model using transformation ↩
            definition and
        //target metamodel.

        //TRANSFORMATIONS GOES HERE

        //After transform, it deliver the target↩
             model
        this.outputTargetModel();
    }

    public void outputTargetModel(){
        //deliveries the target model
        System.out.println("Target model ↩
            produced");
    }
}
```

TR are specified as a Java IF-ELSE chain (referenced as *direct model manipulation* [11]), the default solution to transform models. Nevertheless, we are investigating the use of emerging formats to specify TR, such as MOFScript

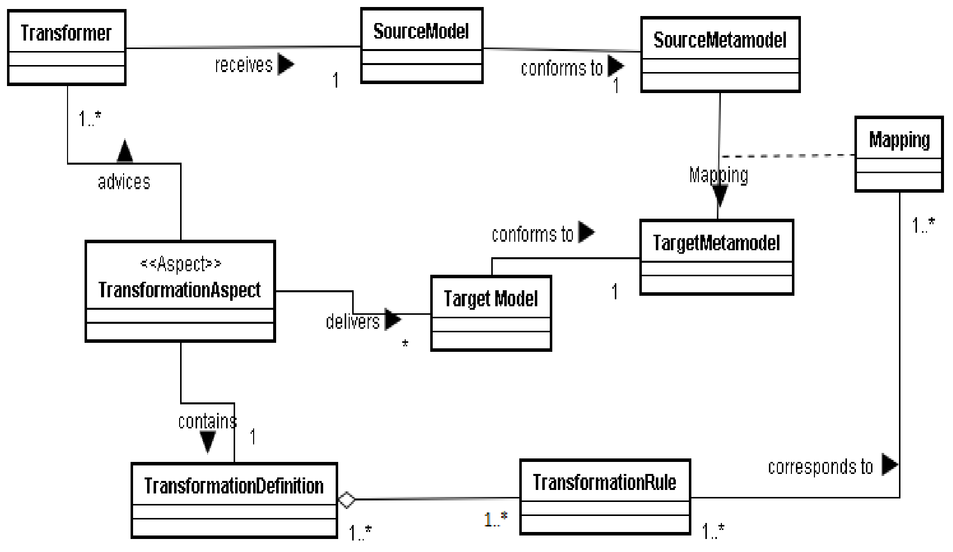

Figure 3: MDD and Aspect components after framework refactoring

[1], Velocity [2], ATL [3], QVT [4], and XSLT [5].

All the TR are encapsulated into one aspect, what reduces scattering, tangled and repetitive code and increases evolvability and maintainability of the framework. This enabled the pluggability required by MDD.

Thus, the experience acummulated with this refactoring indicates TR as a recommended early aspect for the model-driven transformers engineering and the transformers call as crosscutting concerns in frameworks that use MDD.

4. CONCLUSIONS

This paper showed how the composition capabilities of AOP can be used to weave transformation definitions in a model-driven transformers' architecture. This is a contribution to early aspects research since transformation definitions can be considered as aspects since the earliest phases of a model-driven transformers engineering.

Our initial results confirm that AOSD is a suitable paradigm to address modularization and separation of concerns problems in MDD. These results enforce the tendency that the scientific community has reported of migrating aspects from solutions to non-functional requirements to non-trivial requirements and variability points modularization [9, 8].

This proposal benefits the software industry and their investments since business changes, which reflect in market pressures and technologies evolutions, could be well managed with aspects, changing the target technology in an easy and maintainable way.

Finally, the main contribution of this paper is to reveal a new concern in MDD that can be treated as an aspect since the earliest phases of a model-driven transformers engineering.

[1]http://www.eclipse.org/gmt/mofscript/
[2]http://velocity.apache.org/
[3]http://www.eclipse.org/atl/
[4]http://www.omg.org/spec/QVT/1.0/
[5]http://www.w3.org/TR/xslt

Additionally, this paper presents a discussion about model-driven application frameworks. Since there is no a reference architecture for such kind of software, treating transformer invocation as a crosscutting concern and candidate aspect is conceivable.

5. ACKNOWLEDGMENTS

The first author would like to thanks CAPES (Brazilian Coordination for Improvement of Higher Education Personnel), for the finantial support. The authors would also like to thank the Informatics Institute which provided the infrastructure and resources that fostered this research work.

6. REFERENCES

[1] V. Abdelzad and F. S. Aliee. A Method Based on Petri Nets for Identification of Aspects. In *Proc. of Workshop on Early Aspects in AOSD 2010*, 2010.

[2] A. C. Almeida, G. Boff, and J. L. Oliveira. A Framework for Modeling, Building and Maintaining Enterprise Information Systems Software. In *Proceedings of XXIII Brazilian Symposium on Software Engineering*, pages 115–125, 2009. Fortaleza, Brazil.

[3] E. Baniassad, P. C. Clements, J. Araujo, A. Moreira, A. Rashid, and B. Tekinerdogan. Discovering early aspects. *IEEE Softw.*, 23:61–70, January 2006.

[4] G. Boff and J. L. de Oliveira. Modeling, implementation and management of business rules in information systems. *INFOCOMP Journal of Computer Science*, pages 17 – 28, 2010. ISSN: 1807-4545.

[5] R. Bonifácio and P. Borba. Modeling scenario variability as crosscutting mechanisms. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 125–136, New York, NY, USA, 2009. ACM.

[6] C. Duan and J. Cleland-Huang. A Clustering Technique for Early Detection of Dominant and

Recessive Cross-Cutting Concerns. In *Proceedings of the Early Aspects at ICSE: Workshops in Aspect-Oriented Requirements Engineering and Architecture Design*, EARLYASPECTS '07, pages 1–, Washington, DC, USA, 2007. IEEE Computer Society.

[7] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[8] I. Groher and M. Voelter. Aspect-Oriented Model-Driven Software Product Line Engineering. In S. Katz, H. Ossher, R. France, and J.-M. Jézéquel, editors, *Transactions on Aspect-Oriented Software Development VI*, volume 5560 of *Lecture Notes in Computer Science*, pages 111–152. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-03764-1_4.

[9] A. Kellens, K. D. Schutter, T. D'Hondt, V. Jonckers, and H. Doggen. Experiences in modularizing business rules into aspects. In *Proceedings of 24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*, pages 448–451, 2008.

[10] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[11] K. Ma and B. Yang. A Hybrid Model Transformation Approach Based on J2EE Platform. *Education Technology and Computer Science, International Workshop on*, 3:161–164, 2010.

[12] E. A. Nasser and H. S. Hamza. Towards a Domain-Oriented Approach for Identifying Aspects in Software Requirements. In *Proceedings of Workshop on Early Aspects at AOSD 2010*, 2010.

[13] openArchitectureWare.org. openarchitectureware.org - official openarchitectureware homepage. Available at: <http://www.openarchitectureware.org/>. Last Access: 25th Oct 2010, 2009.

[14] A. Rashid, P. Sawyer, A. M. D. Moreira, and J. a. Araújo. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, RE '02, pages 199–202, Washington, DC, USA, 2002. IEEE Computer Society.

[15] B. L. Romano, G. Braga e Silva, A. Marques da Cunha, and W. I. Mour ao. Applying MDA development approach to a hydrological project. In *ITNG '10: Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations*, pages 1127–1132, Washington, DC, USA, 2010. IEEE Computer Society.

[16] P. Sánchez, A. Moreira, L. Fuentes, J. a. Araújo, and J. Magno. Model-driven development for early aspects. *Inf. Softw. Technol.*, 52:249–273, March 2010.

[17] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20:19–25, 2003.

[18] M. Völter and I. Groher. Handling variability in model transformations and generators. In *Proc. of Workshop on Domain Specific Modelling*, 2007.

[19] P. Wittman. MDA using AndroMDA. In: <http://www.wittmannclan.de/ptr/cs/mda$_a$*ndromda.pdf* > , 2010.