Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

Proceedings of the Third Workshop on

Models and Aspects –

Handling Crosscutting Concerns in MDSD

at the 21st European Conference on

Object-Oriented Programming (ECOOP)

Berlin, Germany

Organizing Committee
Christa Schwanninger, Siemens AG, Munich, Germany
Markus Voelter, Independent Consultant, Heidenheim, Germany
Iris Groher, Siemens AG, Munich, Germany
Andrew Jackson, Trinity College Dublin, Ireland

Michael Cebulla (Ed.)

Bericht-Nr. 2007 – 6

ISSN 1436-9915

Workshop Papers

A. Beugnard and C. Kaboré
Interests and drawbacks of AOSD compared to MDE

Z. Altahat, T. Elrad, and D. Vojtisek
Using Aspect Oriented Modeling to localize implementation of executable models

A. Rummler, B. Grammel, and C. Pohl
Improving Traceability through AOSD

A. van den Berg, T. Cottenier, and T. Elrad
Reducing Aspect-Base Coupling Through Model Refinement

L. Lengyel, T. Levendovszky, and H. Charaf
Identification of Crosscutting Concerns in Constraint-Driven Validated Model Transformations

T. Reiter, M. Wimmer, and H. Kargl
Towards a runtime model based on colored Petri-nets for the execution of model transformations

B. Morin, O. Barais, J.-M. Jézéquel, and R. Ramos
Towards a Generic Aspect-Oriented Modeling Framework

Interests and drawbacks of AOSD compared to MDE

A position paper

A. Beugnard
ENST Bretagne
CS 83818
F-29238 Brest cedex 3
antoine.beugnard@enst-bretagne.fr

C. Kaboré
ENST Bretagne
CS 83818
F-29238 Brest cedex 3
eveline.kabore@enst-bretagne.fr

Categories and Subject Descriptors
D.2.10 [**Software Engineering**]: Design; D.1.5 [**Programming techniques**]: Object-Oriented Programming

General Terms
Aspect programming, Model engineering

ABSTRACT
This position paper is the result of experiences we made using model driven engineering to define an automatized design process that injects different concerns during the development of components. We compare our approach with classical aspect oriented programming techniques. We argue that the MDE approach is more flexible.

1. INTRODUCTION
The quest for modularity is a long term research activity. Historically, functions were the first natural modules. Good practices were promulgated to search for low coupling and high cohesion. But functions are often interleaved and module boundaries are difficult to identify. Software architectures, architecture styles are topological abstractions that help reasoning on modules, called components, and their interactions, called connectors. But, considering functional properties is not enough; a lot of non-functional properties are worth trying to be isolated, placed in modules.

A few years ago, aspect oriented programming [4] was introduced to separate concerns. Concerns, also called aspects, are described independently of the functional part of the system, and are *weaved* with it during the development process. Concerns (or aspects) are a way to describe another dimension of modularity. But, concerns are, as functions, highly interleaved [2]. To our knowledge, no standard classification is agreed today.

More recently, the object management group (OMG) has defined the model driven architecture (MDA^{TM}) [1] that proposes an other class of concerns: the platform. The platform describes the target environment with its own features. The principle of the MDA is to *merge* a platform independent model (PIM) with a platform description model (PDM) in order to obtain a platform specific model (PSM). Beyond this simple principle, the model driven engineering approach generalizes and suggests to define more models and more models merging operations (called transformations).

We experienced the MDE approach to automatize the development of a special kind of components called communication components. These components have functional specification and we identified at least 4 concerns: data type implementation, data distribution, data replication, data representation. The full process is described in [3]. This experimentation leads us to a comparison of the AOSD and MDE approaches.

2. ASPECT ORIENTED PROCESS
Aspect oriented approaches rely on a description language and a weaving mechanism. The language allows to specify the different concerns. The mechanism offers operations that are used during the weaving to merge the concern with the program. The weaving mechanism defines the aspect technology approach; which operators are available, which pointcuts can be used, etc. This defines the join point model. Knowing this model, designers have to specify their concerns using the aspect language.

When many concerns are defined one of the not yet solved problem is to select the order of their weaving. This question point out that weaving aspects (or concerns) is included in a process.

The AOP process can be summarized as follows: first, select a programming language; second, define join points and a way to identify them (the join point model), and eventually defines concerns. For instance: using Java, and AspectJ, the designer defines a logging aspect and weaves it to the program.

3. MODEL DRIVEN PROCESS
The way model driven approach is used is less formalized than AOP, and still in construction. We used it with a special interpretation; instead of having only a PIM and a PDM model to merge into a PSM, we keep the PIM (considered as an abstract specification) and define many models, one for each the concerns we were interested in. Each concern gives

raise to a meta-model (a specific grammar) that was used to define variants of the same concern. And for each meta-model we defined a tailored transformation that injects this concern into the trunk model (program).

Having many concerns we had to choose their order of application. This is a design choice, that leads to adapted model transformations. Transformations are developed *knowing* the result of the previous concerns merging.

The MDE approach we used can be summarized as follows: first select a modelling language, second define a meta-model that can be used to specify the weaving transformations.

4. DISCUSSION

Separation of concern is an essential design process. Two challenges are how to describe a concern and how apply it?

The aspect approach makes the choice to offer an universal, generic, mechanism of weaving and requires that the concern designer adopt it and expresses concerns knowing this universal mechanism. All the flexibility is in the concern description.

On the contrary, the model driven approach offers more flexibility. In fact, the concern designers decides first the way he describes the concern, selecting a concern meta-model, and after, elaborates a transformation that injects concerns into the base model. No universal merging (weaving) transformation is required. Every transformation is tailored.

We argue the MDE approach can be used to separate concerns in a more flexible way that the usual AOP does. Transformations implement automatized steps of the design process. Parts of this process are related to the woven concern and, hence, can be implemented thanks to model transformations. Our approach is pragmatic and consists in selecting an order of application of the chosen concerns. Other approaches may attempt to define transformations and (meta) models that are independent of the order of application. At that time, our approach seems more tractable since more constrained; building commutative and associative transformations is difficult. The resulting transformations of our approach depend on the development process.

We also argue that concerns must be selected, analyzed, specified, modeled prior to their weaving process. The concern model influence the weaving transformation, but the implementability of the transformation may also influence the concern model. This is why the flexibility offered by MDE is so important.

5. REFERENCES

[1] A. Board. Model driven architecture (mda). Technical report, Object Management Group, July 2001. `http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01`.

[2] L. Chung, B. A. Nixon, and E. Yu. *Non-Functional Requirements in Software Engineering*. Springer, January 1999. ISBN-10: 0792386663, ISBN-13: 978-0792386667.

[3] E. Kabore and A. Beugnard. Conception de composants répartis par transformations de modèle (, antoine beugnard). In *Journées de l'Ingénierie Dirigée par les Modèles*, pages 117–131, Toulouse, France, 29–30 mars 2007.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, volume 1241, pages 220 – 242, 1997.

Using Aspect Oriented Modeling to localize implementation of executable models

Zaid Altahat
GE Healthcare
Illinois Institute of Technology
Zaid.Altahat@ge.com

Tzilla Elrad
Illinois Institute of Technology
Elrad@iit.edu

Didier Vojtisek
INRIA
Didier.Vojtisek@inria.fr

ABSTRACT
Executable models are essential to define the behavior of models, such as constraints put on model elements. However their implementation crosscut multiple model elements. Model semantics will facilitate Model Driven Development, without it, Design and Implementation won't necessarily represent different abstractions of the same system. This paper introduces a mechanism to query executable models and weave constraints in order to localize their implementation, which improves code redundancy and modularity.

Keywords
Executable Models, Aspect-Oriented Modeling (AOM), Model Driven Development (MDD), Model Driven Architecture (MDA), Aspect-Oriented Software Development (AOSD).

1. INTRODUCTION
Models have been limited in use to design and documentation. Designs are lost by interpretation when moved from system architects (Design) to software engineers (Implementation). The design doesn't dictate the models semantics. Semantics are "the underlying meaning of exchanged models, that is, the *constraints* that models place on the runtime behavior of the specified system." [7]. Design By Contract (DBC) is an example of *constraints* on the model behavior; however their implementation is not localized and crosscut [3] multiple model elements.

A programming language consists of syntax and semantics. Syntax is the language constructs, such as UML class diagrams; while, semantics give the syntactic constructs their meaning. Leaving out the semantics of models created a gap that lead to a wide range of interpretations of the same model. The gap also created a chain of tools that can only exchange the syntax of models. Executable models came to fill in this gap.

Executable models *constrain* how models *behave* at run time. Code generated from models should have a *unique* execution behavior. Unique in the sense that if different codes, programming languages such as Java or C++, to be generated, all should have the same execution behavior. UML is in the process of fully defining Executable UML [7]. KerMeta [5] on the other hand has already defined full behavioral language to specify semantics of models. Section 3 briefly presents KerMeta.

One way to constrain the behavior of a model element, a Class for example is to define *Invariant* condition on the class, and *pre* and *post* conditions on its operations. These three are what is referred to as DBC, which KerMeta already provides capabilities of; however it achieves that *individually* for each class and operation. To manually define constraints for each class and operation may lead to code redundancy and reduction in modularity. The added constraints *crosscut* [3] multiple classes and operations. Aspect-Oriented Modeling (AOM) can help in obviating this problem.

AOM provides separation of crosscutting concerns at the models level. Most popular among these models are behavioral models, which are used in software development, not just for design and documentation but for code generation as well. To set foundations for the code generation and model transformation, new standards are being defined as part of Model Driven Architecture (MDA) group. MDA standards are being set in parallel to AOM. MDA *transforms* a model from high abstract level to platform specific level then to code. AOM also help in keeping crosscutting models separate, as well as transforming Platform Independent Models (PIM) to Platform Specific Models (PSM) by weaving in platform dependent model implementation.

Using AOM approach we will demonstrate how to *localize* the implementation of a crosscutting behavior that intersect multiple classes and/or operations. In AOM a pointcut model, and an *advice* model are defined. Both models and the original models are fed into a weaver. The weaver adds the advice, added behavior, to the join points matched by the pointcut in the model. This paper introduces a novel approach for the modularization and weaving of executable models.

The main contribution of this paper is to provide a model driven approach to query and weave executable model elements into models. The problem this approach tackles is to localize the DBC constraints for executable models; moreover, localize the implementation of operations. Which reduces code redundancy and increases modularity. The project was done in KerMeta for both querying and weaving executable model elements, it is a pure model driven approach that operates on executable models.

Paper is organized as follows: Section 2 presents related work and section 3 briefly describes KerMeta. Section 4 is the core of this paper; it presents the details of the metamodels used, as well as the weaving process. Besides, Section 5 demonstrates the querying and weaving process on an example model. Original model and modified model are presented in Appendices A and B, respectively.

2. Related Work

There are other attempts to localize DBC constraints. However they were designed with a specific programming language in mind. A C++ approach [10] presented a mechanism to localize DBC implementation using Constraint-Specification Aspect Weaver (C-SAW) [9]. ECL [4] was used to locate operations, in addition to weave *assertions* at the beginning and end of an operation to represent *pre* and *post* conditions, respectively. No support for *Invariant* condition. Another approach, Contract4J [2], uses AspectJ to support DBC in Java. It uses Java 5 annotations to mark elements to be amended and define the *pre*, *post*, and *Invariant* conditions. It uses AspectJ behind the scenes to weave in the added code. In contrast with my approach, both of these approaches are geared more towards a specific programming language and are not based on executable models.

3. KerMeta

Meta-languages such as MOF1.4 [5] , MOF2.0 [6] , and Ecore[1] are used to specify the structural and syntax parts of a model but not its behavior. For example EMOF specifies an operations signature and stops there, without defining its behavior. A mix of pseudo code and natural language is used to define its behavior. KetMeta on the other hand uses an operational semantic to specify the precise behavior of models. The example [11] presented in Listings 1 and 2 shows how the definitions of the same method in both MOF and KerMeta.

> *Operation* ***isInstance(element : Element) : Boolean***
>
> "*Returns true if the element is an instance of this type or a subclass of this type. Returns false if the element is null*".

Listing 1. MOF definition of method isInstance()

KerMeta proposes a rich model oriented environment for metamodeling. It provides support for many use cases, including:

- Implementation of operations directly in metamodels.
- Execution of simulation of metamodel behavior.
- Transformation and weaving of models.
- Verification and validation of models against metamodels (as given by a set of static and dynamic constraints).
- Building new Domain Specific Languages under the shape of metamodels, Building any model-driven tools, including tools that generate tools (generative programming).

```
operation isInstance(element : Element) : Boolean is do
        // false if the element is null
        if element == void then result := false
        else
                // true if the element is an instance of this
                type
                // or a subclass of this type
                result := element.getMetaClass == self or
                element.getMetaClass.allSuperClasses.cont
                ains(self)
        end
end
```

Listing 2. KerMeta definition of method isInstance()

This work is a demonstration of several of them. The most important is that it reflectively applies MDA to itself[12]. In this paper, KerMeta is applied at two levels. First, the language is used to define the transformation that constitutes the metamodel weaver. Second, the weaving is applied to KerMeta itself by introducing the advices in models written in KerMeta.

4. Metamodel Weaver

The weaver consists of several metamodels, a pointcut metamodel, an advice (added behavior) metamodel, a link metamodel, and the weaver itself, presented in Figure 1. The following sections present each of these metamodels in details.

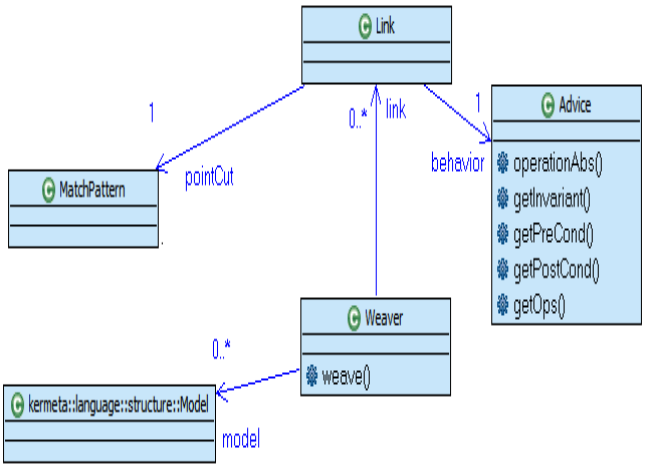

Figure 1 Weaver Metamodel

Weaver::weave, shown in Figure 1, is the starting operation, it is passed a collection of *Link* and a collection of *Model*. *Link* defines a relation between a pointcut *MatchPattern* and an *Advice*. For ach join point matched by a *pointCut behavior* is added.

Advice has multiple operations *getInvariant, getPreCond,* and *getPostCond* to retrieve *Invariant*, *pre* and *post* condition, respectively. An instance advice inherits from *Advice* and overwrites *operationAbs* in order to define *pre* and *post* conditions. Instance advice can also hold other operations definitions that needs be added to the model. They are retrieved using the operation *getOps*. This is to provide an operation implementation into a class. Section 5 presents an example with added behavior.

Figure 2 shows the pointcut metamodel, *MatchPattern*. All elements, except *MatchPattern*, inherit from *MatchPattern* and with it they inherit the string *namePattern* to define its name signature. The matching signature consists of a *ClassPattern* class that has a collection of *AttributePattern* and a collection of *OperationPattern*, which in turns has a collection of *ParamPattern*. All elements inside *ClassPattern* are optional, that's a pointcut in its simplest format is a class name pattern, for example **Account* that will match all classes that end with *Account*. Match patterns used here are similar to AspectJ name matching. Section 5 presents an example with pointcut.

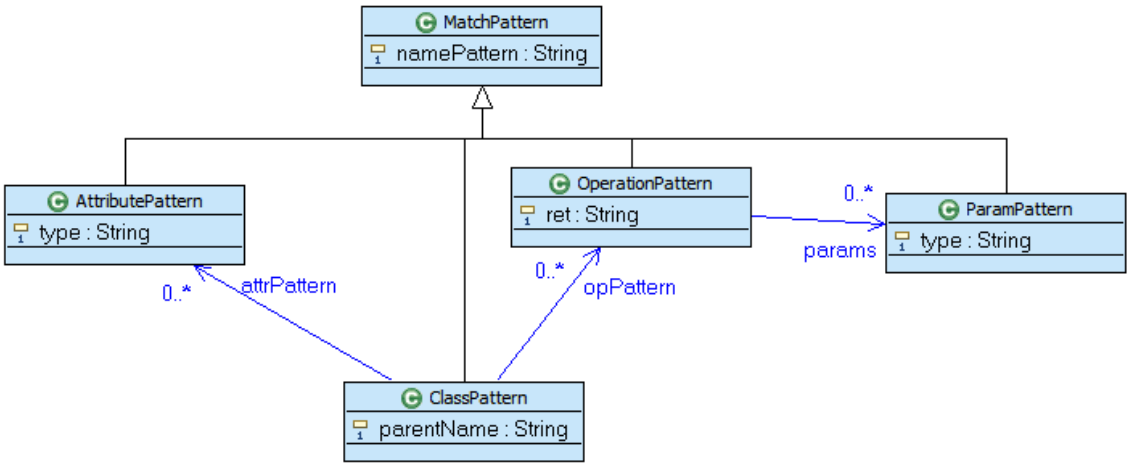

Figure 2 Pointcut metamodel

5. Example

Next we'll introduce an example where blocks of executable models were weaved into model elements, classes and operations. Figure 3 introduces a basic Bank system with different type of accounts. One thing to note about the class *Account* is that the class itself doesn't have *Invariant* condition and none of its operation has a *pre* or a *post* condition, which will be added using the weaver. Also the operation *applyInterest* is abstract where its implementation will be weaved in for two of *Account* subclasses only. KerMeta representation of the model is presented in Appendix A.

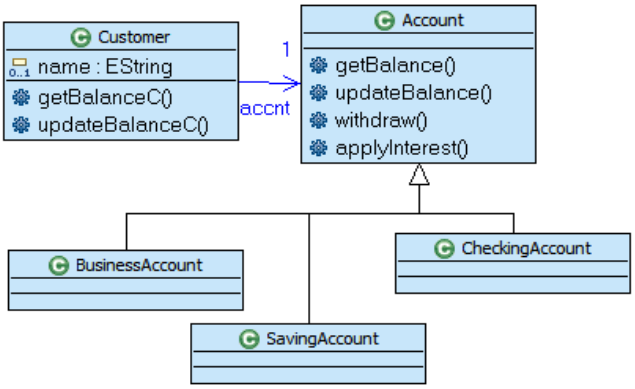

Figure 3 Bank metamodel

* Class Account doesn't have *Invariant*. None of the operations has *pre* or *post* conditions, and *applyInterest* is abstract.

The model in Figure 3 represents the element *model* in Figure 1. The *Weaver* needs *link* element(s) to define what behavior to add for a matched pointcut. Figure 4 introduces two of these *Link* elements. In Figure 4-a a *Link* is created with a *pointCut* that matches the operations *updateBalance* and *withdraw*. *Advice1* defines the behavior to be added, it introduces the *Invariant* condition to the matched class, and the *post* condition to the matched operations.

Figure 4-b defines another Link with a *pointCut* that matches classes *CheckingAccount* and *BusinesAccount* that inherit from the class *Account*. The behavior to be added is an implementation for the operation *applyInterest*. More elements could be used to define more model queries, like number and type of parameters to an operation and its return type. More involved queries were run on larger models, but for sake of simplicity I introduced these queries on the Bank system.

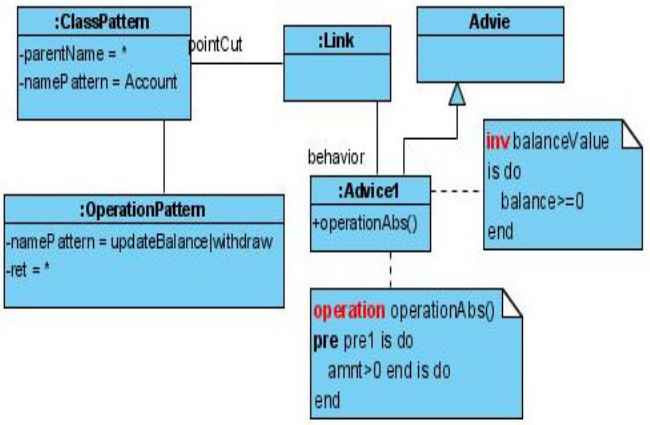

Figure 4-a An instance of Link with a pointcut that matches operations *updateBalance* and *withdraw* in class *Account*, and *pre* and *post* conditions.

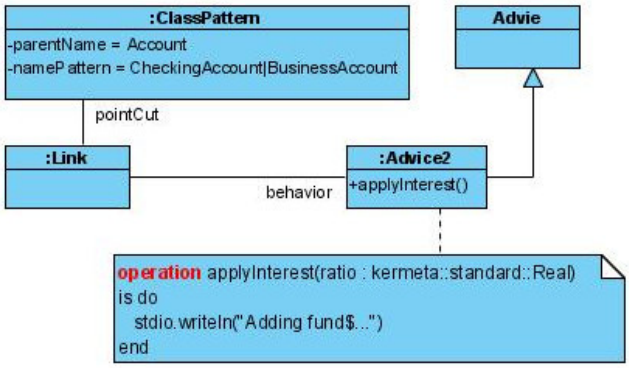

Figure 4-b An instance of Link with a pointcut that matches classes *CheckingAccount* and *BusinessAccount* whose parent are *Account*, and implementation for operation *applyInterest*.

The weaver iterates on each element in the Bank model and applies each link on it. It iterates on the elements twice, once for each link. In the first pass it adds the *Invariant* condition to the class *Account* and the *post* condition to the operations *updateBalance* and *withdraw*. In the second pass it adds the operation *applyInterest* to the classes *CheckingAccount* and *BusinessAccount*. Appendix B presents the generated modified model, and Appendix A present the original model before any modifications.

6. Conclusion and future work

Executable models are getting high attention in order to add semantics to PIM models. In this paper we presented a novel way to query and weave executable models. We chose to localize the implementation of DBC constructs and shared operations implementations in order to improve code redundancy and modularity.

In pointcut metamodel we used strings to define many of the match pattern elements, as shown in Figure 2. In the future we'd like to change the parameter type and operation return type to *kermeta::language::structure::Type*. This will enable us to check for types and super-types, such as *Integer* and *Collection,* without having to use strings. It will also enable us to check for validity of operation arguments. However, using the element *Type* will complicate writing queries and the actual querying process.

7. REFERENCES

[1] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R. and Grose, T. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[2] Dean Wampler, "AOP@Work: Component design with Contract4J". http://www28.ibm.com/developerworks/java/library/j-aopwork17.html

[3] Diotalevi, F., "Contract Enforcement with AOP," *IBM DeveloperWorks*, July 2004, http://www-106.ibm.com/developerworks/library/j-ceaop/

[4] Gray, J., Sztipanovits, J., Schmidt, D., Bapty, T., Neema, S., and Gokhale, A., "Two-level Aspect Weaving to Support Evolution of Model-Driven Synthesis," in *Aspect-Oriented Software Development*, (Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhán Clarke, eds.), Addison-Wesley, 2004.

[5] OMG. Meta Object Facility (MOF) Specification 1.4, Object Management Group,http://www.omg.org/cgi-bin/doc?formal/2002-04-03, 2002.

[6] OMG. MOF 2.0 Core Final Adopted Specification, Object Management Group,http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 2004.

[7] OMG, "Semantics of a Foundational Subset for Executable UML Models" 2006.

[8] http://www.kermeta.org/

[9] http://www.gray-area.org/Research/C-SAW/

[10] Jing Zhang, Jeff Gray and Yuehua Lin. "A Model-Driven Approach to Enforce Crosscutting Assertion Checking".

[11] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer and Jean-Marc Jézéquel. "On Executable Meta-Languages applied to Model Transformations"

[12] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. -- Reflective model driven engineering. -- In G. Booch P. Stevens, J. Whittle, editor, Proceedings of UML 2003, volume 2863 of LNCS, pages 175--189, San Francisco, October 2003. Springer.

8. Appendix A

/* Class Customer was not modified, it was left out from this appendix.*/

```
class Account
{
        attribute balance : kermeta::standard::Integer

        operation updateBalance(amnt : kermeta::standard::Integer)
        is do
                balance := amnt
        end

        operation getBalance() : kermeta::standard::Integer
        is do
                result := balance
        end

        operation withdraw(amnt : kermeta::standard::Integer)
        is do
                balance := balance - amnt
        end

        operation applyInterest(ratio : kermeta::standard::Real) is
        abstract
}

class CheckingAccount inherits Account
{ }

class SavingAccount inherits Account
{ }

class BusinessAccount inherits Account
{ }
```

9. Appendix B

This Appendix shows only modified classes and operations. Unmodified classes were left out and are identical to the originals presented in Appendix A. Weaved code is in the red box.

woven models.

```
class Account
{
        inv balanceValue is
                do
                        balance.isGreaterOrEqual(0)
                end
        attribute balance : kermeta::standard::Integer
        operation applyInterest(ratio : kermeta::standard::Real):
                kermeta::standard::~Void is abstract
        operation
        updateBalance(amnt:kermeta::standard::Integer) :
                kermeta::standard::~Void
                post post1 is do
                result.isNotSameAs(void).~and(balance.isGre
                        aterOrEqual(0))
                end
        is do balance := amnt end
        operation getBalance() : kermeta::standard::Integer is
        do result := balance end
        operation withdraw(amnt : kermeta::standard::Integer) :
                kermeta::standard::~Void
                post post1 is do
                        result.isNotSameAs(void).~and(bal
                          ance.isGreaterOrEqual(0))
                end
        is do balance := balance.minus(amnt) end
}
class CheckingAccount inherits Account
{
        operation applyInterest(ratio : kermeta::standard::Real) :
                kermeta::standard::~Void is do
                        stdio.writeln("Adding fund$...")
                end
}
class BusinessAccount inherits Account
{
        operation applyInterest(ratio : kermeta::standard::Real) :
                kermeta::standard::~Void is do
                        stdio.writeln("Adding fund$...")
                end
}
```

Improving Traceability through AOSD

Andreas Rummler
SAP Research CEC Dresden
Chemnitzer Str. 48
01187 Dresden, Germany
andreas.rummler@sap.com

Birgit Grammel
SAP Research CEC Dresden
Chemnitzer Str. 48
01187 Dresden, Germany
birgit.grammel@sap.com

Christoph Pohl
SAP Research CEC Karlsruhe
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe, Germany
christoph.pohl@sap.com

1. INTRODUCTION

In real world business applications traditional software product line engineering and model-driven software development (MDSD) [6] often cannot properly reflect the decomposition of system features. For instance, Governance, Risk and Compliance (GRC) checks or late introduction of security properties often crosscut the architectural design of a system. To overcome these issues Aspect-Oriented Software Development (AOSD) [1] modularizes such crosscutting concerns in independent aspects. Although aspects can already be captured at requirements stage [2, 7], there is no clear mapping to later development stages. MDSD can address this by model transformation. However, AOSD still increases the complexity of traceability (and hence, maintainability) because it adds yet another dimension of variability [3]. This issue is one of the most important arguments against applying AOSD techniques in an industrial context. Future research has to take care of this issue in order to lay the basis for industry acceptance of AOSD. According to an internal audit of customers of SAP, missing traceability during the whole development cycle is the top-rated weakness. In addition missing traceability information was explicitly mentioned as weakness in 2005 in an external ISO certification audit.

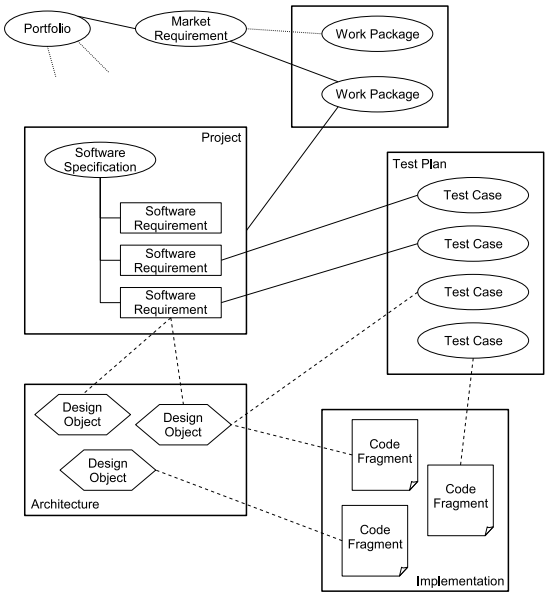

Figure 1: Linking of Artifacts in SAPs Development Process

2. TRACEABILITY SUPPORT INSIDE SAP

The current state of trace support implemented in SAPs internal development process is outlined in figure 1. Market requirements are linked to software specifications comprising software requirements that can be linked to test cases. Further linkage of requirements to design and development artefacts, or linkage of those artefacts to test cases is possible in general, but not sufficiently supported by tools (indicated by dashed lines). Currently, several interesting questions typically arising during the development process cannot be answered automatically: Have all high-level market requirements really been designed and implemented properly? How can this implementation be tested against these requirements? Have artefacts been developed whose behaviour is not covered by any requirements or which is even unwanted? Why has one artefact been chosen over another, alternative one? These question can be answered on a fine-grained level by experts involved in the development process, but not by people dealing with a coarse-grained view. Therefore tools are needed that are able to give reasonable answers to these questions or at least help people in finding those answers. MDSD has its merits for transforming artefacts into each other based on certain models, but crosscutting functionality as outlined above tangles not only code but also such models.

3. IMPROVING TRACEABILITY THROUGH AOSD

There are approaches to Aspect-Oriented Requirements Engineering (AORE) [2, 7] that can help address some of these questions. For instance relationships, dependencies and interactions among existing requirements can be identified at early stages of the development lifecycle. However, AORE approaches do not explicitly define mechanisms for mapping information gathered at the requirements level to later development phases. There is a need for defining mapping guidelines, rules, and heuristics for mapping of entities and trace information across the entire development lifecycle. Assets repositories are also required that may collect and maintain product line assets and the mapping rules, guidelines, and heuristics. In addition, there is a need for a traceability meta-model that defines which trace information: assets, concerns, relationships, dependencies, behaviours, compositions, mappings, needs to be captured and managed.

SAP Research is currently involved in an European funded project called AMPLE, which stands for Aspect-Oriented, Model-Driven Product Line Engineering.[1] The focus is on providing a holistic treatment of variability by addressing each stage in the software life cycle. Another point of interest in AMPLE is providing effective forward and backward traceability of variations and their impact.

The approach followed in AMPLE relies on the modularization of cross cutting concerns at model level. Starting already at the stage of requirements engineering will foster traceability. To be able to track dependencies between AO and non-AO artefacts along the development cycle, explicit aspect interfaces need to be defined. To rely on earlier work already made available in [4] and [5] seems to be promising. The intrusive nature of AO techniques is reduced in its intensity by defining aspect interfaces that form some kind of contract between the to-be-extended system and the extending aspects. Other ongoing work concerns a metamodel for variability including the support of AO concepts and appropriate tracing information. Based on this metamodel a tool chain is designed that supports the definition of SPL, product generation and full support for tracing relationships and dependencies among automatically generated or manually created artefacts. A comprehensive case study is currently being implemented, consisting of a complete example from SAPs core application business.

4. CONCLUSIONS

To summarise this position paper, tracing artefacts throughout the whole development process is a key issue in industry, driven by internal *and* external forces. Handling variability and documenting decisions on variations is the core issue of traceability. AOSD approaches introduce interesting concepts to modularise cross-cutting concerns at various development stages but it also complicates traceability. Explicit aspect interfaces are one requirement for easier tracking of dependencies between AO and non-AO artefacts. At the workshop, we would like to share our industry perspective on how AOSD and MDSD could further fertilise each other for improving traceability issues, among other challenges.

5. REFERENCES

[1] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[2] J. Grundy. Aspect-oriented requirements engineering for component-based software systems. In *Proceedings of the 4th IEEE Symposium on Requirements Engineering*, 1999.

[3] S. Katz and A. Rashid. From aspectual requirements to proof obligations for aspect-oriented systems. In *Proceedings of the 12th IEEE International Conference on Requirements Engineering*, 2004.

[4] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the ACM International Conference on Software Engineering*, 2005.

[5] U. Kulesza, V. Alves, A. Garcia, C. Lucena, and P. Borba. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In *Proceedings of the International Conference on Software Reuse ICSR 06*, 2006.

[6] T. Stahl and M. Völter. *Model-driven Software Development*. John Wiley, 2006.

[7] B. Tekinerdogan, A. Moreira, J. Araujo, and P. Clements. Early aspects: Aspect-oriented requirements engineering and architecture design. In *Workshop Proceedings at AOSD Conference*, 2005.

[1]The website for the project can be found at `http://www.ample-project.net`

Reducing Aspect-Base Coupling Through Model Refinement

Aswin van den Berg
Motorola Software Group, Motorola
1303 E. Algonquin Rd,
Schaumburg, IL 60196, USA
+1 (847) 538-2597
Aswin.vandenberg@motorola.com

Thomas Cottenier
Motorola Software Group, Motorola
1303 E. Algonquin Rd,
Schaumburg, IL 60196, USA
+1 (847) 538-2597
Thomas.cottenier@motorola.com

Tzilla Elrad
Illinois Institute of Technology
3100 S. Federal Street,
Chicago, IL 60696, USA
+1 (312) 567-5142
Elrad@iit.edu

Illinois Institute of Technology
3100 S. Federal Street,
Chicago, IL 60696, USA
Cottho@iit.edu

ABSTRACT

Aspect-Oriented Programming languages allow pointcut descriptors to quantify over the implementation points of a system. Such pointcuts are problematic with respect to independent development because they introduce strong mutual coupling between base modules and aspects. This position paper addresses the aspect-base coupling problem by defining pointcut descriptors in terms of abstract views of the base module. These abstract views should be towards the architectural viewpoints of the system under development.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Design Tools and Techniques – *modules and interfaces.* D.2.1 [**Software Engineering**]: Design Tools and Techniques – *Object-oriented design methods.* D.3.3 [**Programming Languages**]: Language Constructs and Features – *classes and objects, modules, inheritance.*

General Terms

Languages, Theory, Algorithms.

Keywords

Aspect weaving, modeling, refinement, aspect-oriented modeling. model-driven software engineering, aspect-oriented programming.

1. INTRODUCTION

Since the inception of Aspect-Oriented Software Development (AOSD) in 1997, it has been known that Aspect-Oriented Programming (AOP) languages introduce strong coupling between base modules and aspects. AOP languages allow pointcut descriptors to refer directly to the implementations of modules to capture joinpoints, points where aspects inject behavior through advices. This practice is problematic with respect to modularity and independent development. Aspects need fine-grained control over the modules they advice and, vice versa, the advised modules need to be aware of those aspects. Therefore, both aspect and base module become hard to evolve independently.

There are three main research directions in addressing this aspect-base coupling problem. The first direction of research advocates restricting the expressiveness of aspects by forfeiting the obliviousness of modules [1][2][3]. A second approach favors investigating alternative ways to modular reasoning in the presence of aspects. In [4], the authors argue that a global analysis of the system configuration is required before the interfaces of the system modules can be determined. A third direction of research focuses on methods that allow pointcut descriptors to be defined at a higher level of abstraction, in terms of the program semantics [5]. Our work with Motorola ***WEAVR*** in [6] introduces pointcut descriptors that can infer implementation joinpoints from higher level descriptions. This paper proposes an approach to AO modeling that is integrated with a model refinement approach with the purpose to reduce the aspect-base coupling.

2. REQUIREMENTS FOR ASPECT-BASE DECOUPLING

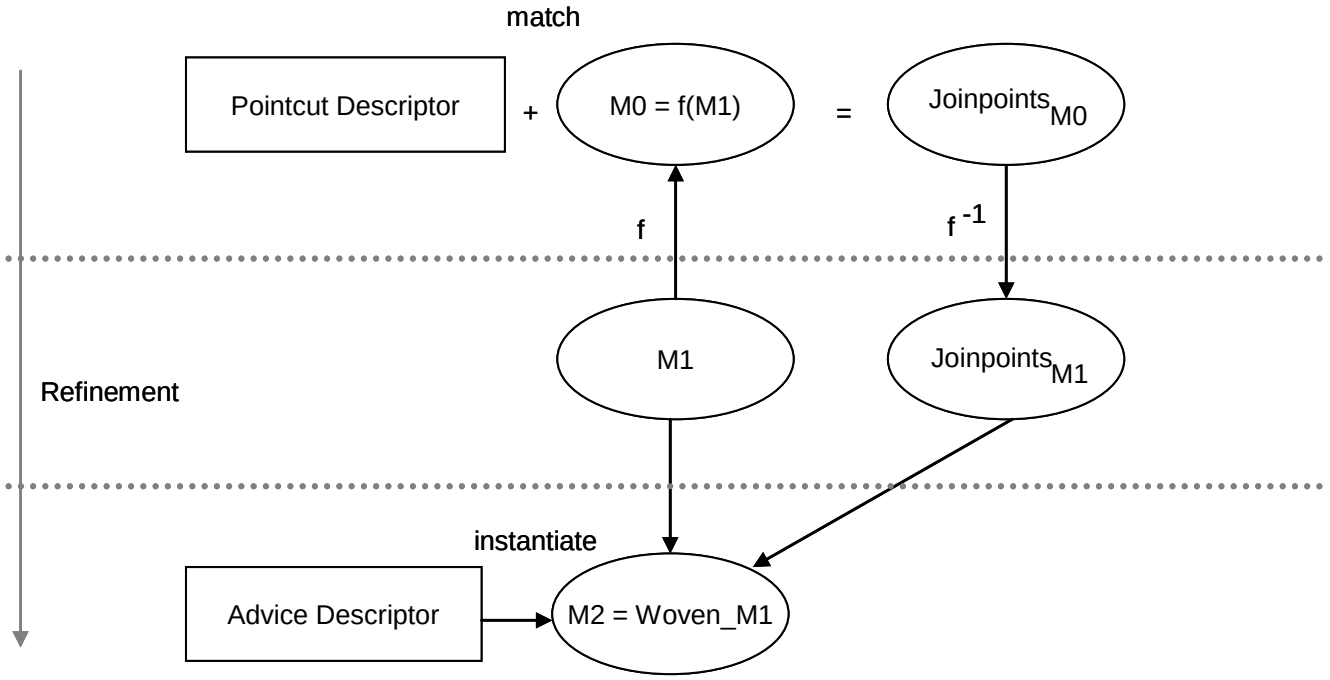

Figure 1: Aspect weaving seen as a model refinement expressed in terms of a realization mapping (f)

Let M1 be the current refinement of a software system. We show five requirements for moving towards our goal (see Figure 1):

1. There needs to be an abstract view M0 of the refinement M1 of the system under development that is sufficiently describing the behavior of its specification towards a particular architectural viewpoint.
2. There needs to be a precise definition of what it means that a refinement is realizing an architectural view. This realization can be described by a well-defined mapping f from the refinement M1 to the view M0.
3. The development process/tool needs to enforce that the refinement of the view is actually realizing the view. That is, the process/tool needs to enforce the realization invariant M0 = f(M1).
4. Define pointcut descriptors in terms of the view M0. The matching produces a set of joinpoints in M0 (denoted by Joinpoints_{M0})
5. Translate these joinpoints in terms of the refinement M1 and instantiate the advice at corresponding points in M1. The resulting woven model M2 is more refined than M1 because it has a new concern incorporated in it.

Since the pointcut descriptor is written in terms of M0 it is completely independent from the refinements that are introduced in M1. Since M0 is an abstract view towards an architectural viewpoint it is not a view that is dependent on the pointcut descriptor. And because M0 is not dependent on the pointcut descriptor it follows that also M1 is not dependent on it. Therefore there is no aspect-base coupling between the aspect and the refinements introduced from M0 to M1.

3. REFERENCES

[1] Aldrich, J. Open Modules: *Modular Reasoning about Advice*. In Proceedings of the 19th European Conference on Object-Oriented Programming, Glasgow, Scotland, LNCS 3586, pp. 144-168, Springer, 2005

[2] Griswold, W.G., Shonle, M., Sullivan, K., Song, Tewari, N., Cai, Y., Rajan, H.: *Modular Software Design with Crosscutting Interfaces.* IEEE Software, 23:1, pp. 51–60, IEEE Computer Society, 2006

[3] Gybels, K., Brichau, J.: *Arranging Language Features for More Robust Pattern-Based Crosscuts.* In proceedings of the International Conference on Aspect-Oriented Software Development, , Boston, USA, pp 60–69, ACM Press, 2003.

[4] Kiczales, G., Mezini, M.: *Aspect-Oriented Programming and Modular Reasoning.* In proceedings of the International Conference on Software Engineering, St. Louis, USA, pp 49–58, ACM Press, 2005

[5] Ostermann, K., Mezini, M., Bockisch, C.: *Expressive Pointcuts for Increased Modularity.* In Proceedings of the 19th European Conference on Object-Oriented Programming, Glasgow, Scotland, LNCS 3586, pp. 214-240, Springer, 2005

[6] Cottenier, T., van den Berg, A., Elrad, T., *Joinpoint Inference from Behavioral Specification to Implementation*, In Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP), Berlin, Germany, 2007

Identification of Crosscutting Concerns in Constraint-Driven Validated Model Transformations

László Lengyel
Budapest University of Technology and Economics
1111 Budapest, Goldmann György tér 3., Hungary
lengyel@aut.bme.hu

Tihamér Levendovszky
Budapest University of Technology and Economics
1111 Budapest, Goldmann György tér 3., Hungary
tihamer@aut.bme.hu

Hassan Charaf
Budapest University of Technology and Economics
1111 Budapest, Goldmann György tér 3., Hungary
hassan@aut.bme.hu

ABSTRACT

Domain-specific model processors facilitate the efficient synthesis of application programs from software models. Often, model compilers are realized by graph rewriting-based model transformation. In Visual Modeling and Transformation System (VMTS), metamodel-based rewriting rules facilitate to assign Object Constraint Language (OCL) constraints to model transformation rules. This approach supports validated model transformation. Unfortunately, the validation introduces a new concern that often crosscuts the functional concern of the transformation rules. To separate these concerns, an aspect-oriented solution is applied for constraint management. This paper introduces the identification method of the crosscutting constraints in metamodel-based model transformation rules. The presented algorithms make both the constraints and the rewriting rules reusable, furthermore, supports the better understanding of model transformations.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Algorithms, Design and Languages

Keywords

Aspect-Oriented Constraints, Constraint Weaving, Identifying Crosscutting Constraints, Model Transformation

1. INTRODUCTION

Model-driven development approaches (for example Model-Integrated Computing (MIC) [19] and OMG's Model-Driven Architecture (MDA) [14]) emphasize the use of models at all stages of system development. They have placed model-based approaches to software development into focus.

Model transformation lies at the heart of the model-driven approaches [11] [20]. Transformations appear in many, different situations in a model-based development process. A few representative examples are as follows. (i) Refining the design to implementation; this is a basic case of PIM/PSM mapping. (ii) Aspect weaving; the integration of aspect models/code into functional artifacts is a transformation on the design [1]. (iii) Analysis and verification; analysis algorithms can be expressed as transformations on the design [2].

One can conclude that transformations in general play an essential role in model-based development, thus, there is a need for highly reusable model transformation tools that support validated model transformation.

At the implementation level, system validation can be achieved by testing. Various tools and methodologies have been developed to assist in testing the implementation of a system (for example, unit testing, mutation testing, and white/black box testing). However, in the case of model transformation environments, it is not enough to validate that the transformation engine itself works as it is expected. The transformation specification should also be validated. There are only few and not complete facilities provided for testing offline transformation specifications in an executable style. However, online validated model transformation can guarantee that if the transformation finishes successfully, the generated artifact is valid, and it is in accordance with the required output [8] [9].

For example, require a transformation that transforms class model to relational database management system (RDBMS) model (transformation *Class2RDBMS*) to guarantee the followings: a class that is marked as non-abstract in the source model is transformed into a single table of the same name in the target model, each table has primary key, each class attribute is part of a table, each many-to-many association has a distinct table, and so on.

These types of requirements can be specified by Object Constraint Language (OCL) [14] constraints assigned to the transformation rules. Unfortunately, often, the same constraint is repetitiously applied in many different places in a transformation, therefore the constraints crosscut the transformation rules and their management becomes hard.

In [7], a solution of the case study *Class2RDBMS* is provided where a transformation is presented with 9 transformation rules and two constraints are emphasized, from which the first one appears 30 times in 9 transformation rules and the second one 16 times in 6 transformation rules. This is very difficult to manually manage crosscutting and scattering constraints, because all of the modifications have to be done on all occurrences of the constraints. Constraints appearing several times in a transformation increase the time of constraint handling and the possibility of making a mistake during the modification. Using aspect-oriented constraints, a method has been given to solve the problem of the crosscutting constraint in model transformations [7] [9]. The main idea is to handle constraints similarly to Aspect-Oriented Programming (AOP) aspects, to provide the transformation constraints with the properties of the AOP aspects. AO constraints are created separately from transformation rules and, using a weaver method, they are woven back to the transformation rules before the execution of the transformation. The result of this method is a consistent constraint management (modification, deletion, and propagation) with crosscutting constraint separation and weaving.

The current work proposes a method to identify the crosscutting constraints in model transformations. Our motivation is driven by the fact that transformation designers prefer defining transformation rules directly with constraints. Therefore, a solution should be provided to extract constraints from existing transformation rules. The proposed method makes both the constraints and the rewriting rules reusable, furthermore, facilitates the better understanding and maintainability of the metamodel-based model transformations.

2. BACKGROUNDS

This section as a background information introduces the Visual Modeling and Transformation System (VMTS), the problem of the crosscutting constraints in metamodel-based model transformation rules, and the methods provided by VMTS to define and apply transformation constraints as aspects.

Graph rewriting [17] is a powerful technique for graph transformation with a strong mathematical background. The atoms of graph transformations are rewriting rules, each rule consists of a left-hand side graph (LHS) and right-hand side graph (RHS). Applying a graph rewriting rule means finding an isomorphic occurrence (match) of LHS in the graph to which the rule is applied (host graph), and replacing this subgraph with RHS.

2.1 Visual Modeling and Transformation System

Visual Modeling and Transformation System (VMTS) [21] supports editing models according to their metamodels, and allows specifying constraints written in Object Constraint Language (OCL) [14]. Models are formalized as directed, labeled graphs. VMTS uses a simplified class diagram for its root metamodel ("visual vocabulary"). Also, VMTS is a model transformation system, which transforms models using graph rewriting techniques. Moreover, the tool facilitates the validation of the constraints specified in the transformation rule during the model transformation process.

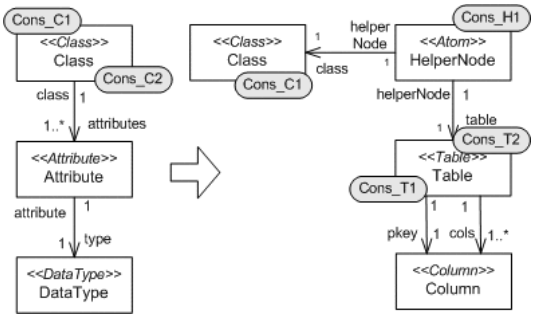

Figure 1: Example transformation rule: *ClassToTable*

In VMTS, LHS and RHS of the transformation rules are built from metamodel elements. This means that an instantiation of LHS must be found in the input graph instead of the isomorphic subgraph of LHS.

Rewriting rules can be made more relevant to software engineering models if the metamodel-based specification of the transformations allows assigning OCL constraints to the individual transformation rules. This technique facilitates a natural representation for multiplicities, multi-objects and assignments of OCL constraints to the rules with a syntax close to the UML notation. An example metamodel-based transformation rule that generates database tables from UML classes is depicted in Fig. 1.

The constraints assigned to the transformation rule guarantee our requirements. After a successful rule execution, the conditions hold and the output is valid, this cannot be achieved without constraints.

2.2 Crosscutting Constraints

In model transformation, the dominant decomposition is the functional behavior of the transformation rules. The constraints ensure the correctness of the transformation only if they are well-defined by the designer. Although they are responsible for the correctness, the constraints are usually treated with secondary importance. They crosscut the transformation, and it is almost impossible for the designer to perform the intuitive activity of verifying the transformation.

Our motivating example (transformation *ClassToRDBMS*) is presented in the previous section. Constraints such as *NonAbstract*, *PrimaryKey*, and *ClassAttrsAndTableColls* (Fig. 1) cannot be encapsulated in any of the rules or rule nodes. However, they express the same constraint concerns on the rules: therefore they should be defined separately from the transformation rules and woven automatically to the appropriate rule nodes later.

Since there is not enough space to present all transformation rules and all occurrences of the constraints appearing in rule *CreateTable*, we provide statistical data only, and the details can be found in [9] and [21]. The transforma-

tion *Class2RDBMS* contains nine transformation rules. In these rules the constraint *NonAbstract* appears 30 times, constraint *Abstract*, which requires the presence an abstract class, appears 16 times. Furthermore, the constraints *PrimaryKey* and *PrimaryAndForeignKey* are linked 6 times, and constraints *OneToOneOrOneToMany* and *ManyToMany*, which are related to processing the associations between classes, appear 4 times [7]. This means that one of the open issues with respect to the transformation is that the same constraints appear several times.

2.3 Aspect-Oriented Constraint Management in VMTS

As it was presented in the previous section, in model transformation, some constraints assigned to transformation rules represent the crosscutting concerns.

In VMTS, aspect-oriented constraints are OCL constrains defined separately from the transformations and transformation rules, and are woven to the rules later using a weaver method. Recall that in VMTS, transformation rules are built from metamodel elements, where a metamodel element can appear arbitrary times in a transformation rule. A rule is not an instance of the metamodel, both of them are on the same meta level. The input and output models are the instances of the metamodels. Each transformation rule node and edge has a metatype that corresponds to a metamodel type. The context information of the aspect-oriented constraints can be used as a type-based pointcut that selects rule nodes based on their metatype. The weaving process driven by the type-based pointcuts is referred to as *type-based weaving* [10]. In Fig. 1, *Class* is the context of the constraint *NonAbstract* and the target rule nodes (rule node *Class* in LHS and rule node *Class* in RHS) of the propagation are selected based on this metatype.

To refine the weaving procedure, *weaving constraints* are applied. A weaving constraint is similar to a property-based pointcut, it is also an OCL constraint, which specifies the weaving, but it is not woven to transformation rules, and thus, it is not used during the transformation process. Weaving constraints facilitate optional conditions during the weaving process. Therefore, it is referred to as *constraint-based weaving* in VMTS [10]. A weaving constraint can be used to represent one or many characters as a means of specifying more than one attribute during a search procedure. This enables to select multiple rule nodes with a single specification. For example, the propagation of constraint *PrimaryKey* (Fig. 1) can be refined with weaving constraints. The constraint *PrimaryKey* can be separated from the transformation rules, and using the weaving constraint *table.name = 'Table*'* it can be woven to rule nodes, whose names start with the *'Table'* character sequence.

Having separated the constraints from rule nodes, we also need a weaver which facilitates the propagation (linking) of the constraints to the rule nodes. Our approach solves the aspect-oriented constraint propagation with the Global Constraint Weaver (GCW) algorithm [9] (Fig. 2).

This mechanism facilitates our approach towards managing constraints using aspect-oriented techniques. Similarly to aspects, the constraints are specified and stored independently of any model transformation rule or rule node and are linked to rule nodes by the GCW.

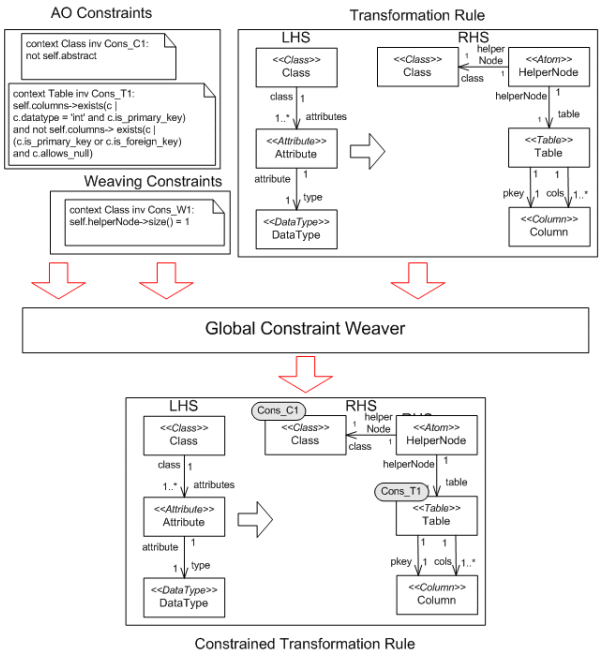

Figure 2: The weaving process and the input and output of the GCW

The output of the weaver is not stored as a new transformation rule. The result is handled as a link between the constraints and a transformation rule. This link is referred to as *weaving configuration* [10]. A weaving configuration can be executed similarly to a transformation. The difference is that it contains the links between the transformation rule nodes and the constraints, therefore, during the execution the transformation engine applies the constraints woven to the transformation rules. Weaving configurations are created once, they are stored in the database, but they require significantly less space than transformation rules, because weaving constraints represent only the rule-constraint relations. Furthermore, this representation makes the transformation rule management transparent: it is not required to modify the rules in each weaving configuration, but only on their original place.

2.4 Constraint Normalization in VMTS

OCL constraints often contain complex expressions with several navigation steps. The constraint evaluation consists of two parts. (i) Selecting the object and its properties that the constraint needs to be checked on, and (ii) executing the checking. In general, the larger part of the evaluation is the first step, because of its computational complexity. Each navigation step in a constraint means several queries on the model database. Therefore the original motivation of the normalization method was to reduce the navigation steps contained by the constraints, because the eliminated navigation steps accelerate the first part of the constraint evaluation. In [9] a method is provided with algorithms to normalize OCL constraints in metamodel-based transforma-

tion rules. This normalization method with the results of the constraint relocation and decomposition algorithms support the aspectification of the crosscutting constraints in model transformations.

3. IDENTIFICATION OF ASPECT-ORIENTED CONSTRAINTS

This section provides a method with algorithms to support the detection of the crosscutting constraints in metamodel-based model transformations. The input of the method is a transformation (transformation rules and a control flow model), and the expected output is the crosscutting constraints separated as aspects. A simple but promising idea is the following:

1. Collect the constraints appearing in the transformation.
2. Identify the repetitive constraints.
3. In fact, not all of the repetitive constraints are crosscutting constraints. Therefore, for each repetitively appearing constraints decide if the actual constraint is crosscutting for the transformation or not.
4. Extract crosscutting constraints as aspects.

The separation of concerns principle states that a given problem involves different kinds of concerns that should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability. The principle declares that each concern of a given software design problem should be mapped to one module in the system. Otherwise, the problem should be decomposed into modules such that each module has one concern. The advantage of this is that concerns are localized and as such can be easier understood, extended, reused, and adapted.

Many concerns can indeed be mapped to single modules. Some concerns, however, cannot be localized and separated easily, and given the design language we are forced to map such concerns over many modules. This is called *crosscutting concern* or *aspect*. Aspects are not the result of a bad design but have more inherent reasons. A bad design including mixed concerns over the modules could be refactored to a neat design in which each module only addresses a single concern. However, if we deal with these crosscutting concerns, this is not possible in principle that is, each refactoring attempt will fail and the crosscutting will remain. A crosscutting concern is a serious problem, since it is harder to understand, reuse, extend, adapt and maintain a concern because it is spread over many rule nodes. Finding the places where the crosscutting occurs is the first problem, adapting the concern appropriately is another problem.

Our crosscutting constraint identification method can be divided into two main parts: coloring (Section 3.1) and extracting (Section 3.2) constraints. Based on the user defined or automatically identified concerns the coloring algorithm assigns colors to the constraints of the processed model transformation. Each color represents a concern that should be modularized. Ideally each constraint will have exactly one assigned color, which means there is no crosscutting. After the coloring, when we realized that there are constraints with more than one color, the extracting is applied to realize crosscutting constraints as aspects (aspect-oriented constraints). Extracting is supported by constraint decomposition [9].

3.1 Coloring Algorithm

The input of the coloring algorithm is the model transformation with the propagated constraints. The output is the coloring, where each of the colors represent a concern. Of course the relevant concerns are also specified by the transformation designer.

A concern, which represents a color can be related to an optional property that is expressed by a constraint: e.g. an attribute value or the existence of specific type adjacent nodes. For example:

```
context Class inv NonAbstract:
not self.abstract
```

The constraint *NonAbstract* represents the concern, which states that the processed class should be non-abstract.

```
context Table inv SourceClass:
self.helperNode.class->exists(c |
(c.name = self.name))
```

The constraint *SourceClass* represents the concern, which predicates that a generated table has a source class with the same name.

If we have only these two constraints, then the coloring is simple: the algorithm assigns to each of the constraint concerns a different color. But if a constraint comprises more concerns, then the coloring will be compound:

```
context Class inv NonAbstractAndProcessed:
not self.abstract and not self.isProcessed
```

The constraint *NonAbstractAndProcessed* incorporates two concerns: (i) the matched class should be non-abstract, and (ii) the matched class should be non-processed. In this case two colors are assigned by the coloring algorithm to the constraint.

The concerns that should be taken into account by the coloring algorithm are defined by meta OCL constraints. These meta OCL constraints form an OCL Set. Each element of this Set represents a color (concern ID). These constraints are evaluated for the model transformation constraints. The result is the coloring, which is refactored by the extracting algorithm (Section 3.2).

Meta OCL constraints are defined as {*context*, *constraint expression*, *color*} triplets. Example simple meta OCL constraints for boolean type attributes: {*Class*, *abstract*, *Color*

1} {*Class*, *isProcessed*, *Color 2*}. Example meta OCL constraint for existing adjacent nodes {*Class*, *self.helperNode.table->size() > 0*, *Color N*}.

An important requirement that the whole method should provide is that each concept can have only one color, e.g. it is not allowed that *abstract* has two or more colors. This means that the elements of the meta OCL Set should be unique.

Algorithm 1 presents the pseudo code of the COLORING algorithm. The model transformation T and the meta OCL Set $metaOCLs$ are passed to the algorithm, which iterates on the constraints propagated to the rule nodes of the transformation T (line 3). In an embedded loop, for each constraint the algorithm iterates on the meta OCL constraints (line 4), and evaluates the relevance of the actual meta constraint ($metaConstraint$) for the actual constraint (C) (line 5). If the constraint C contains the concern represented by the actual meta constraint $metaConstraint$ then the color of the meta constraint $metaConstraint$ is assigned to the constraint C (line 6). Finally the coloring is returned by the algorithm.

Algorithm 1 Pseudo code of the COLORING algorithm

```
1: COLORING (Transformation T, OCLSet metaOCLs): ColoringTable
2: ColoringTable coloringTable = new ColoringTable();
3: for all Constraint C in T do
4:   for all Constraint metaConstraint in metaOCLs do
5:     if CHECKCONSTRAINTRELEVANCE(C, metaConstraint) then
6:       UPDATECOLORINGTABLE(coloringTable, C, metaConstraint.Color)
7:     end if
8:   end for
9: end for
10: return coloringTable
```

This is obvious that meta OCL constraints contain the understanding of the concepts, and this is provided by the developer. Theoretically, this method can provide a 100% solution for our problem. At this point the main question is the quality of the meta OCL constraints, because meta OCL constraints should cover all concerns and should take into account everything from the point of transformations view. To obtain a useful result we should ensure the completeness of the defined meta constraints. (Currently this is the developers responsibility.) Otherwise, the result is relevant only for the covered part of the concerns.

3.2 Extracting Algorithm

The inputs of the constraint extracting algorithm are the model transformation and the result of the coloring algorithm. The outputs are the constraints extracted into aspects.

If the coloring is unambiguous, each constraint has maximum one color, then aspects can be created based on the colors. Constraints without color are not extracted as aspects. But complex constraints may have several colors at the same time. On the level of the source code, crosscutting resulted by the bad design can be solved with refactoring. In the domain of the metamodel-based model transformation, we can apply constraint relocation and decomposition in order to eliminate the annoying consequences of the bad design (crosscutting constraints).

Algorithm 2 presents the pseudo code of the EXTRACTING algorithm, which uses the constraint relocation and constraint decomposition provided by our constraint normalization method [9].

Algorithm 2 Pseudo code of the EXTRACTING algorithm

```
1: EXTRACTING (Transformation T, ColoringTable coloringTable): AspectList
2: AspectList aspectList = new AspectList(coloringTable.Colors.Size);
3: Transformation decomposed = DECOMPOSECONSTRAINT(T)
4: for all ColoringItem coloringItem in coloringTable do
5:   for all Color color in coloringItem do
6:     UPDATEASPECTLIST(aspectList, color, decomposed.GETDECOMPOSEDCONSTRAINT(coloringItem.GETCONSTRAINTBYCOLOR(color)))
7:   end for
8: end for
9: return aspectList
```

The transformation T and the coloring $coloringTable$ is passed to the EXTRACTING algorithm. The algorithm decomposes the constraints of the transformation (line 3) [9]. The algorithm iterates on the coloring items provided by the coloring (line 4), and for each item iterates on the colors assigned to the actual coloring concern (line 5). Based on the actual color the algorithm retrieves the relevant constraint from the actual coloring item. Using this constraint the correspondent constraint is queried from the decomposed version of the transformation. Based on the decomposed constraint the algorithm updates the already prepared aspect list (line 6). Finally the aspect list is returned.

4. RELATED WORK

An aspect-oriented approach is introduced in [5] for software models containing constraints, where the dominant decomposition is based upon the functional hierarchy of a physical system. This approach provides a separate module for specifying constraints and their propagation. A new type of aspect is used to provide the weaver with the necessary information to perform the propagation: the strategy aspect. A strategy aspect provides a hook that the weaver may call in order to process the node-specific constraint propagations.

At the time of the writing we have no knowledge about that any other approach supports aspect-oriented constraint management in model transformation rules, therefore, there is no other method for identifying crosscutting constraints in model transformations. But there are other software development fields, where identifying crosscutting concerns is also crucial.

In [4] an evaluation of clone detection techniques for identifying crosscutting concerns is presented. [6] introduces a tool that finds clones and displays them to the programmer. The

approach is based on program dependence graphs (PDGs) and program slicing. In [18] a method is provided for identifying crosscutting concerns in requirements specifications. [3] provides support for developers to identify aspects early in the software lifecycle. A method is presented for aspect identification and analysis in requirements documentation. The Prism project [22] develops tools and techniques for discovering non-localized units of modularity in large software systems. [12] proposes a model to identify and specify quality attributes that crosscut requirements including their systematic integration into the functional description at an early stage of the software development process.

5. CONCLUSIONS

This paper has introduced an aspect-oriented solution for the problem of crosscutting constraints in metamodel-based model transformations. We have presented the aspect-oriented constraint management of Visual Modeling and Transformation System. So far VMTS is the only environment that provides aspect-oriented methods for constraint management. The main contribution of the paper is the identification of crosscutting constraints in model transformations. We have presented an approach with algorithms that semi-automatically identifies the crosscutting constraints and separates them into aspects.

Of course we would like to automate the largest possible part of the meta constraint definition. Therefore, the next question is: which concerns can be identified automatically, and, of course, in which way. The method can be supported by providing concern suggestions for the developer. This method will not provide a 100% solution, but the suggested concerns can be refined by the developer. Our first suggestion (*Suggestion 1*) is to identify the constraints of the transformation: simple constraints and the subterms of complex constraints based on the boolean separators (*and*, *or*, *xor*). The suggested constraints will be the ones, which are propagated at least twice to any of the rule nodes of the transformation.

A heuristic-based solution could improve the flexibility and usability of the current coloring method: suggestions, mentioned above, should be defined on a higher level. A language should be provided that facilitates to define what should be checked, and the source code that performs the checking is generated automatically based on it. For example: *Suggestion 1* is defined as a heuristic on higher abstraction level, and the checker that performs the control is automatically generated. The research related to the heuristic-based solution is the subject of our future work.

6. ACKNOWLEDGMENTS

The fund of "Mobile Innovation Centre" has supported in part, the activities described in this paper.

7. REFERENCES

[1] U. Assmann and A. Ludwig, Aspect Weaving by Graph Rewriting, Generative Component-based Software Engineering, Springer, 2000.

[2] U. Assmann, How to Uniformly specify Program Analysis and Transformation, Int. Conference on Compiler Construction (CC) 96, LNCS 1060, Springer, 1996.

[3] E. Baniassad, S. Clarke, Finding Aspects in Requirements with Theme/Doc, Early Aspects 2004.

[4] M. Bruntink, A. van Deursen, R. van Engelen, T. Tourw, On the Use of Clone Detection for Identifying Crosscutting Concern Code, IEEE Trans. on Software Engineering, 2005, Vol. 31, No. 10, pp. 804-818.

[5] J. Gray, T. Bapty, S. Neema and J. Tuck, Handling Crosscutting Constraints in Domain-Specific Modeling, Communications of the ACM, October 2001, pp. 87-93.

[6] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, 8th Int. Symposium on Static Analysis, pp. 40-56, 2001.

[7] L. Lengyel, T. Levendovszky, H. Charaf, Eliminating Crosscutting Constraints from Visual Model Transformation Rules, ACM/IEEE 7th Int. Workshop on AOM, Montego Bay, Jamaica, October 2, 2005.

[8] L. Lengyel, T. Levendovszky, H. Charaf, Constraint Validation Support in Visual Model Transformation Systems, Acta Cybernetica, ISSN 0324-721X, Vol. 17(2), pp. 339-357, 2005.

[9] L. Lengyel, Online Validation of Visual Model Transformations, PhD thesis, Budapest University of Technology and Economics, Department of Automation and Applied Informatics, 2006.

[10] L. Lengyel, T. Levendovszky, H. Charaf, Optimizing Constraint Weaving in Model Transformation with Structural Constraint Specification, 8th Int. Workshop on AOM, March 21, 2006, Bonn, Germany.

[11] A. Metzger, A systematic look at model transformations, In Model-driven Software Development, Vol. II of Research and Practice in Software Engineering. Springer, 2005.

[12] A. Moreira, J. Arat'ujo, I. Brito, Crosscutting Quality Attributes for Requirements Engineering, 14th Int. Conf. on Software Engineering and Knowledge Engineering, pp. 167-174. ACM Press, 2002.

[13] OMG MDA Guide Version 1.0.1, 2003. Document number: omg/2003-06-01, `www.omg.org/docs/omg/03-06-01.pdf`

[14] OMG OCL Spec., Version 2.0, 2006. `http://www.omg.org/`

[15] OMG QVT, MOF 2.0 Query/Views/Transformation Specification, `http://www.omg.org/cgi-bin/apps/doc?ad/05-03-02.pdf`

[16] OMG UML Spec., Version 2.1.1, 2007. `http://www.uml.org/`

[17] G. Rozenberg (ed.), Handbook on Graph Grammars and Computing by Graph Transformation: Foundations, Vol.1 World Scientific, Singapore, 1997.

[18] L. Rosenhainer, Identifying Crosscutting Concerns in Requirements Specifications, 2004.

[19] J. Sztipanovits and G. Karsai, Model-Integrated Computing, IEEE Computer, Apr. 1997, pp. 110-112.

[20] J. Sztipanovits, GPCE, vol. 2487 of LNCS, pp. 32-49, Pittsburgh, October 2002.

[21] VMTS Website, `http://www.vmts.aut.bme.hu`

[22] C. Zhang, H.-A. Jacobsen, A Prism for Research in Software Modularization through Aspect Mining, Technical report, Middleware Systems Research Group, University of Toronto, 2003.

Towards a runtime model based on colored Petri-nets for the execution of model transformations

Thomas Reiter
Information Systems Group
Johannes Kepler University
Altenbergerstr. 69
4040 Linz, Austria
+43-732-2468-9236
reiter@ifs.uni-linz.ac.at

Manuel Wimmer
Business Informatics Group
Vienna University of Technology
Favoritenstr. 9-11
1040 Vienna, Austria
+43-1-58801-18829
wimmer@big.tuwien.ac.at

Horst Kargl
Business Informatics Group
Vienna University of Technology
Favoritenstr. 9-11
1040 Vienna, Austria
+43-1-58801-18837
kargl@big.tuwien.ac.at

ABSTRACT

Existing model transformation languages, which range from purely imperative to fully declarative approaches, have the advantage of either explicitly providing statefulness and the ability to define control flow, or offering a raised level of abstraction through automatic rule ordering and application. Existing approaches trying to combine the strengths of both paradigms do so on the language level, only, without considering the benefits of integrating imperative and declarative paradigms in the underlying execution model. Hence, this paper proposes a transformation execution model based on colored Petri-nets, which allows to combine the statefulness of imperative approaches as well the raised level of abstraction from declarative approaches. Furthermore, we show how a Petri-net based execution model lends itself naturally to the integration of an aspect-oriented style of transformation definition, as transformation rules can be triggered not only upon the input model, but on the state of the transformation execution itself.

1. INTRODUCTION

As model transformations play a key role in model driven development, several dedicated languages have emerged that allow to define and execute transformations between source and target metamodels. Compared to transformations implemented in a general purpose programming language or XSL transformations which operate on a models serialization, model transformation languages provide a layer of abstraction by allowing to manipulate models in terms of their abstract syntax given by its metamodel. Apart from this basic commonality, different kinds of model transformation languages exist. These approaches range from purely imperative styles allowing to define *how* an transformation is carried out, to fully declarative transformation definition styles focusing on *what* a transformation's output should be like, according to a certain input.

Declarative approaches (i.e. graph transformations) are typically based on defining rules that are later on interpreted by an execution engine to produce the desired result. Hence, the actual transformation execution as well as the order of rule application generally need not be handled by the user, although approaches based on graph transformations like AGG, or VMTS [7] allow to specify precedence of certain rules. Declarative rules typically consist of a semantically corresponding source and target patterns, whereby for each match of the source pattern in the input model, a target pattern is instantiated in the output model. Additionally, Triple Graph Grammars (TGG) [5] maintain the state of a transformation by traces that link matched source and instantiated target model elements.

Imperative approaches are similar in usage to traditional programming languages and allow the developer to explicitly manipulate transformation execution state and control flow. Although approaches such as the EOL [4], MTL or Kermeta [6] offer great flexibility and ease of use, the programming model does not support the intuitive alignment of concepts that is prevalent in metamodel or schema integration tasks, and one often needs to implement manually what a more succinct declarative description would achieve. However, what a declarative approach gains in abstraction, it loses in flexibility. Naturally, declarative specifications are convenient language constructs for recurring transformation tasks, but for "tricky" problems, a rule-based paradigm can become unwieldy.

To alleviate these limitations, hybrid approaches like ATL [3] or Xtend [8][9] combine imperative and declarative styles of transformation definition. (We regard Xtend as hybrid due to its functional style and rule-like "create" extensions.) Thus, the imperative part of a hybrid language is available to accomplish tasks that cannot be adequately solved declaratively. However, allowing to intermix imperative and declarative statements requires a developer to be aware of how exactly the engine orchestrates transformation execution. For instance, when writing imperative program parts in ATL, one has to be aware that their execution is subject to the engine's scheduling, and one may not assume that certain declarative rules have yet been dealt with, or that a certain internal state is reached. Hence, the imperative part is often necessary simply to work around the confines of the engine's execution procedure, as opposed to enable algorithmic computations. As an example, a common work-around is to explicitly maintain and observe custom state information in global variables, for instance to be able to manually trigger rules at certain points during a transformation's execution, in case the state information (i.e. trace between source and target model) that is automatically maintained by the execution engine does not suffice.

In general, existing declarative and hybrid approaches, are governed by an underlying execution procedure implemented in

This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-810806, and the Austrian Federal Ministry for Education, Science, and Culture, and the European Social Fund (ESF) under grant 31.963/46-VII/9/2002

the respective transformation engine. In our opinion, this rigidness is the main cause for trouble when attempting to solve tricky problems with declarative approaches, or when integrating them with imperative styles. As the actual transformation definitions can be seen as merely parameterizing an intrinsically rigid, pre-defined procedure, we view declarative approaches as *data-oriented*, in the sense that they specify how input data is mapped onto output data. This is reflected in the rationale, that models are seen as graphs, and therefore graph transformations are used to describe and implement model transformations.

As opposed to declarative approaches, imperative approaches express transformations on a very fine-grained level, which is flexible but incurs explicit handling of control flow without support for the alignment of concepts as it is prevalent in schema integration tasks, for instance. Instead of specifying what input data is mapped onto what output data, imperative approaches follow a *procedure-oriented* paradigm and allow to algorithmically define a function that computes the output model from the input model.

We propose to rethink the notion of models as input and output data which is subject to a transformation that is seen either as an explicit or implicit procedure, but understand a transformation as a *process*. In a process-oriented view, a transformation execution is carried out by interacting entities that control streams of information from source to target models. The flowing information stems from the models themselves, and the actual transformation logic is made up by the behavior of individual entities and their interaction which each other.

Consequently, we propose the *transformation net* formalism, which is based on conditional, colored Petri-nets, to represent transformation processes. Such an execution model provides the explicit statefulness of imperative approaches through markings contained in the net's places. The abstraction of control flow from declarative approaches is achieved as transitions can fire autonomously depending on their environment. To describe specific firing rules for transitions, we resort to pre/post rules known from graph transformations.

The following section gives an overview of the transformation net formalism and describes how models and metamodels can be mapped onto transformation nets. The example in section three will describe how higher-level languages can be built on-top of transformation nets and how a process-oriented view favors the incorporation of aspect-oriented rules. Section four concludes with an outlook on future work.

2. TRANSFORMATION NETS

What sets transformation nets apart form existing approaches is their ability of making the transformation process explicit, as opposed to assuming a certain predefined execution rigor. Of course, the Petri-net based formalism needs an execution engine, too. But the Petri-net execution engine is generic and not tailored to a specific task unlike declarative model transformation engines. This makes the transformation net formalism a flexible execution environment to be targeted by generators of higher-level transformation languages, such that specific transformation and integration operators can be defined using the semantics offered by transformation nets.

As symbolically displayed in Figure 1, the “compilation” step produces a transformation net in its initial state (i.e. ready for execution) that uniformly represents models, metamodels and transformation specifications. The static parts of a transformation net that correspond to the transformation process’ inputs and outputs, are generated from models and metamodels, whereas the part that corresponds to the process’ execution logic is created from the integration specification by a custom generator for a certain higher-level language.

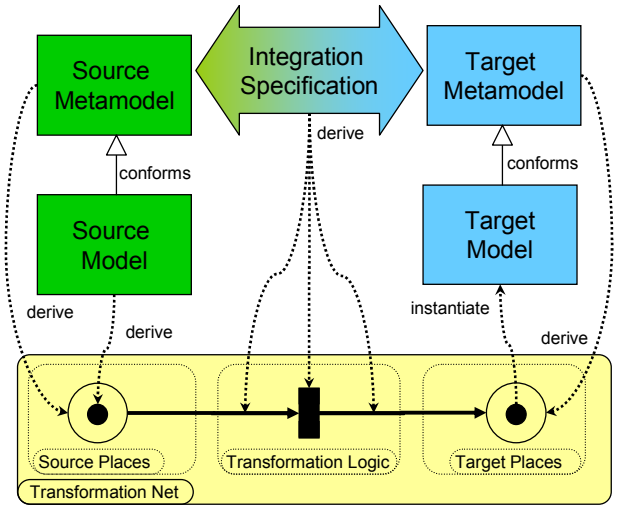

Figure 1. Overall transformation procedure.

The gap between the modeling and the transformation net technical space is bridged by the mapping described in the following. For reasons of brevity, we give a mapping only for the three main elements of metamodels, that are classes, references and attributes, and leave other constructs (e.g.: enumerations) aside.

Classes, references and attributes of metamodels are mapped to *places* of a transformation net.

Objects, as instances of classes are mapped to *one-colored* tokens within a place that corresponds to the object’s class. The token’s color represents an object’s unique ID.

Links between objects conforming to a certain reference are mapped onto *two-colored tokens* within a place that corresponds to the link’s reference. The two colors represent a link’s source (ring color) and target (center color) and stand for the ID of the linked objects.

Values of attributes are mapped onto *two-colored* tokens within a place that corresponds to the values’ attribute. The two colors represent an object’s unique ID and the denoted value.

To complete the transformation net and to provide the actual process logic, a system of transitions and places has to be established that is capable of streaming tokens from the places corresponding to the input metamodels to places corresponding to the output metamodel. Thereby, the transitions represent interacting entities that control the token streams by firing and removing tokens from their input places and adding tokens to their output places accordingly. During execution, state information is explicitly provided by the markings of places, which makes it possible, to trigger transitions according to a certain runtime state, as opposed to only act upon data comprising

the input model. The notion of triggering transitions according to runtime events or states is similar to the notion of point-cuts determining the execution of advice in aspect-oriented programming. Hence, transformation nets naturally cater for the use of aspect-oriented techniques on the runtime level. How to incorporate a weaving mechanism on the language level will be discussed as part of next section's example which introduces a high-level integration language and demonstrates transformation net generation and execution.

3. EXAMPLE

The example in this chapter deals with the specification of a transformation between two metamodels, which is compiled into a net that finally executes the transformation process. Figure 2 shows the source and target metamodels, as well as the input model and the desired output model. As shown, a transformation between these two metamodels has to transform *array* input models into *linked-list* output models.

The transformation specification in-between the metamodels is given in an example language, which comprises several operators whose exact transformation net semantics will be given in the following section when describing the runtime level. On the language level, every operator stands for a certain processing entity, which has inputs and outputs by which individual operators can be assembled in a component-based way. For instance, the C2C (Class2Class) component takes objects from the "Element" class as input, and outputs them into the "Node" class. Analogously the R2R (Reference2Reference) component streams links from "contains" to "head". The C2C component offers another output port "history", of which all this components yet handled tokens can be accessed. The 2-Buf component connected to C2C's "history" sequentially fills an internal buffer of size two, which is again provided as output port. A Linker component takes the two objects in the buffer, and produces a link between them which is streamed into the "next" place. A back-link is produced by the Inverter component that produces back-links from the "next" place and streams them into the "prev" place.

Additionally to these "manually" assembled components, certain operators can cross-cut a transformation specification: Because the target metamodel classes do not have ID attributes, these should be stored within an annotation for eventual round-tripping. This can be accomplished by the Att2Annot component, which henceforth crosscuts the transformation of every object and is therefore woven with every C2C component. The transformation specification is itself a model, and due to the component-like assembly, existing model weavers can be used to merge the aspect operator into the base transformation specification. The top of Figure 2 shows the aspect's definition in a notation inspired from XWeave [2]. The query in the aspect selects all C2Cs, with three additional sub-queries "in.id", "history" and "out", relative to the current C2C operator. The results of "in.id" and "history" are bound to the "values" and "objects" ports of the Att2Annot operator, which for every transformed object instantiates a new Annotation object ("class" port) which is linked up ("ref" port) with the according Node object and sets its text attribute ("att" port) to the value of the source objects "id" attribute. Additionally, the "Annotation" class is woven into the target metamodel, as indicated through the dotted lines in Figure 2. Thereby, the result of the "out" query determines the classes to which an "annot" reference will be added.

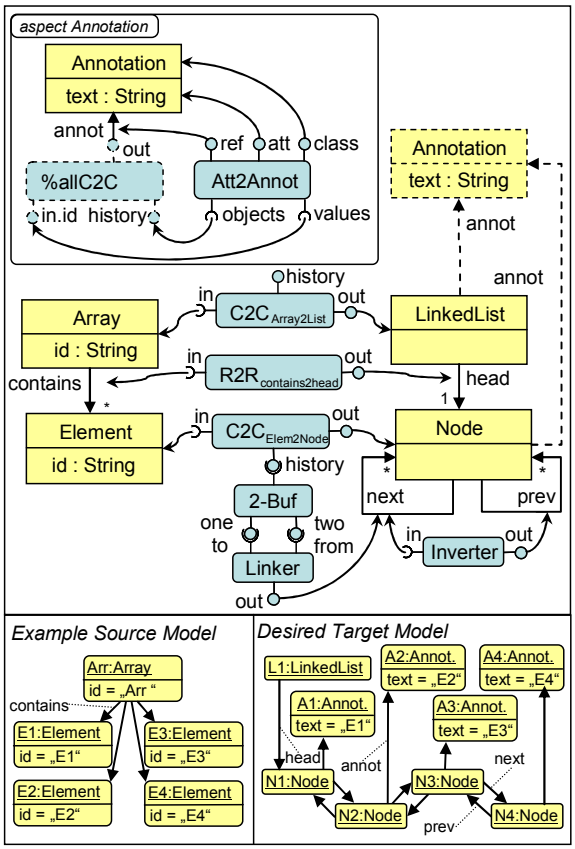

Figure 2. Integration specification between metamodels with example models.

After the weaving process is carried out on the language level, generation takes place to produce a transformation net out of an integration specification. Thereby Petri-net patterns are instantiated according to the transformation net semantics of the operators and assembled according to the overall integration specification. Every such pattern declares input and output arcs which represent the component ports of the respective language operators. The top of Figure 3 shows a transformation net resulting from the above integration specification. The transitions' firing rules are defined with a visual notation that uses pattern-filled tokens that can match for certain input tokens and produce output tokens whose color is either different, the same, or a combination (two-colored tokens) of the matched input colors. Places marked as "ordered" index contained tokens and provide them in a sorted fashion. For instance, the R2R component's transition matches "ArrE1" – the "first" input token. Furthermore, according to the multiplicity of a reference, a place (e.g. "head") can have a capacity, which constrains the amount of tokens a place can hold. Places holding two-colored tokens (references and attributes) have a double-lined border for easier differentiation. For simplicity reasons, the example assumes only a single array object, and since there is only a single ordered reference, the Element place is compiled into an ordered place as well, as not to unnecessarily complicate the example.

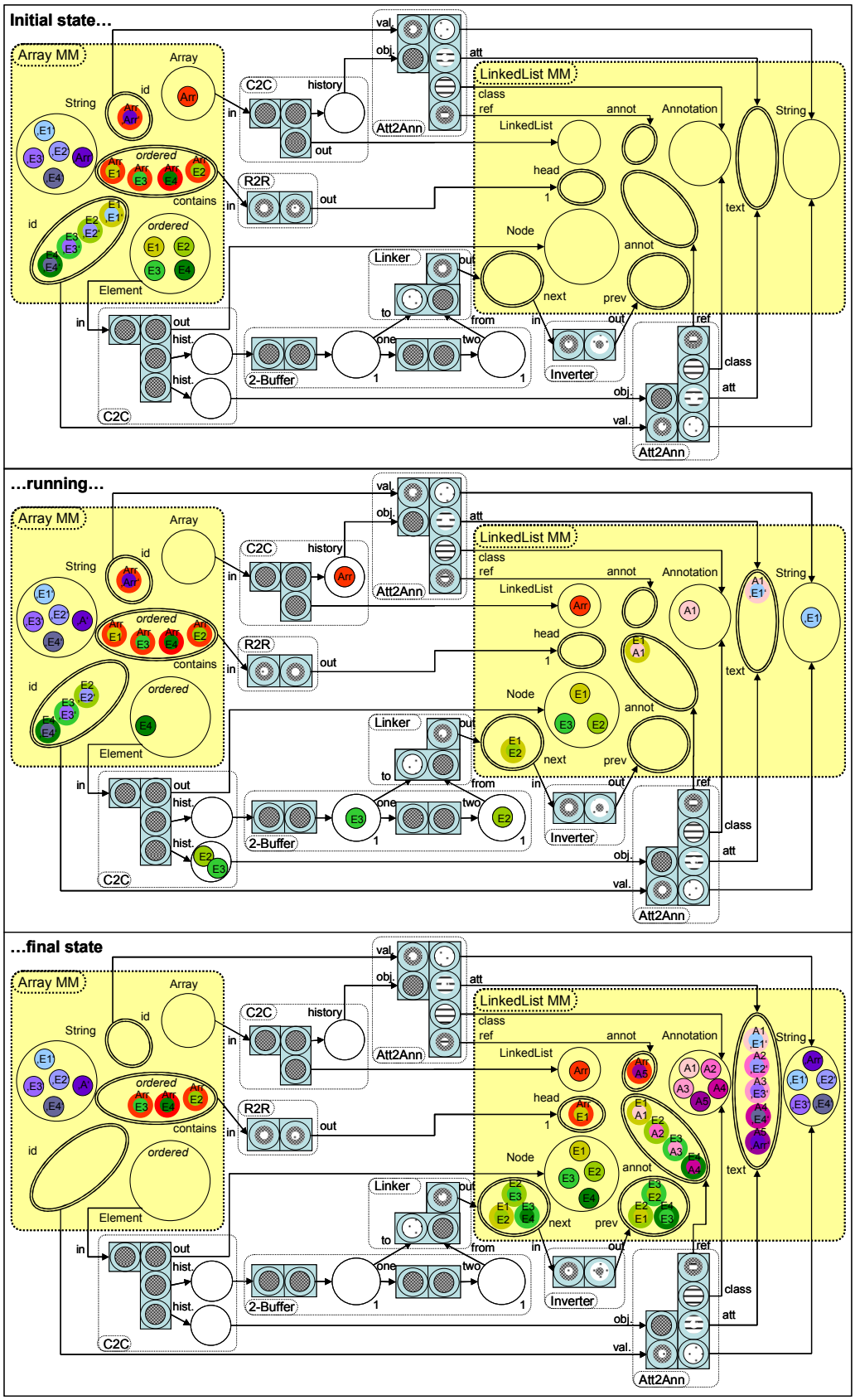

Figure 3. Transformation net execution.

The middle and the bottom of Figure 3 show the transformation net during execution and in its finished configuration. For instance, one can see how the tokens streamed through the C2C component are stored in its "history" place. (The history place is duplicated in the lower C2C, as both the Att2Annot and the 2-Buf components are bound to it.) The 2-Buf component takes in these tokens and fills its two-place buffer. Once the buffer is full (both places have a capacity of just one token), the Linker component's transition can fire and empty the buffer, producing a two-colored token which is streamed into the "next" place. Thereby it is to note, that the creation of two-colored tokens for the "next" link is based on a certain state of the execution, rather than on the input model alone.

Furthermore, one can see how the previously weaved operators form Petri-net patterns that become active after an Array or Element token was streamed. As an example, in the "running" net, the lower Att2Annot pattern has already created an annotation with the according value for the "E1" object, and is currently enabled to do the same for "E2" and "E3", as both have already been handled by a C2C component. Analogously, the rest of the patterns stream tokens from source to target places, possibly depending on other patterns in turn. The actual firing order, however, is handled by the underlying Petri-net engine. Once the transformation process has finished, the final net configuration is used to instantiate a model that conforms to the target metamodel, as shown in the bottom-right corner of Figure 2.

4. CONCLUSION AND FUTURE WORK

In this paper we have presented a new execution model for model transformations based on colored Petri-nets. Such a process-oriented execution model embodies the strengths of imperative and declarative paradigms and is able to explicitly represent a transformation's execution state, which furthermore allows for the natural integration of aspect-oriented transformation rules. Furthermore, although transformation nets are intended as a low-level execution model, transformation tasks like establishing the correct links in the above linked-list example can be expressed elegantly and encapsulated in reusable components.

Currently we have developed the TROPIC prototype (TRansformations on Petri-nets In Color) which can transform integration specifications established with the CARMEN mapping framework [10] into colored Petri-nets that can be executed using the ExSpecT [1] tool. After execution, the resulting Petri-net is transformed into the actual target model. The CARMEN framework builds upon an integration language that provides operators for bridging schematic heterogeneities between metamodels and ontologies. Future work will deal with extending the existing set of integration operators and generators. Due to the fact, that the transformation net approach is very generic, we will furthermore investigate in how well the approach is applicable to other model management tasks, such as model merging or incremental transformations.

Another advantage of a process-oriented view is that a transformation net represents a single artifact which embodies metamodels, models and execution logic altogether. Therefore, we deem a Petri-net based execution model beneficial for debugging purposes and visualization of a transformation's state. Consequently, besides developing generators for further integration languages (e.g.: model merging) or existing model transformation languages, our next steps will focus on developing dedicated tool support in the form of editors and debuggers for the transformation net formalism.

5. REFERENCES

[1] ExSpecT – Executable Specification Tool. http://www.exspect.com

[2] I. Groher and M. Völter. XWeave: models and aspects in concert. *Proceedings of the 10th international workshop on Aspect-oriented modeling, (AOSD 2007)*,Canada, Vancouver: 35-40.

[3] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica. 2005.

[4] D. S. Kolovos, R. F. Paige, and F. A.C. Polack. The Epsilon Object Language (EOL). In *Proc. of European Conference in Model Driven Architecture (EC-MDA)* Bilbao, Spain:128-142, 2006.

[5] A. Königs. Model Transformation with Triple Graph Grammars. *Model Transformations in Practice*, Satellite Workshop of MODELS 2005, Montego Bay, Jamaica, 2005.

[6] P. –A. Muller, F. Fleurey, 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 Practice Workshop*, Montego Bay, Jamaica, October 2005.

[7] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovsky, U. Prange, D. Varro, and S. Varro-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *Proc. Workshop Model Transformation in Practice*, Montego Bay, Jamaica, October 2005.

[8] M. Völter, B. Kolb, . Efftinge and A. Haase. Introduction to openArchitectureare 4.1.x. *MDD Tool Implementers Forum*, TOOLS Europe, 2007.

[9] M. Völter, B. Kolb, . Efftinge and A. Haase. From Front End To Code - MDSD in Practice. *Eclipse Corner Article,* June 2006. http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html

[10] M. Wimmer, H. Kargl, M. Seidl, M. Strommer and T. Reiter. Integrating Ontologies with CAR-Mappings. *First International Workshop on Semantic Technology Adoption in Business (STAB'07)*, Vienna, Austria, May 2007.

Towards a Generic Aspect-Oriented Modeling Framework

Brice Morin
Olivier Barais
Jean-Marc Jézéquel
IRISA Rennes Projet Triskell
Campus de Beaulieu
F-35 042 Rennes Cedex
{bmorin|barais|jezequel}@irisa.fr

Rodrigo Ramos
Centre of Informatics
Federal University of
Pernambuco
P.O. Box 7851, CEP
50732970, Recife, Brazil
rtr@cin.ufpe.br

ABSTRACT

Aspect-Oriented Modeling approaches propose to model reusable aspects, or cross-cutting concerns, that can be later on composed into various base systems. These approaches are often limited to a particular domain: UML class diagrams, UML sequence diagrams, ... and therefore they cannot easily be adapted to other domains. In this paper, we propose to extend the notion of aspect to emcompass an open ended number of domains. We present our Generic Aspect-Oriented Modeling Framework and show how it can easily be specialized for any specific domain.

1. INTRODUCTION

Aspect-Oriented Modeling (AOM) approaches need to deal with two main activities: identifying in a base model points of interest or *join points*, where to compose aspects, and composing the aspects into the base model. Identifying *join points* in a base model require a mechanism to specify the match points: the *pointcut*. In most of the AOM approaches, the *pointcut* is a template model whose elements can match *join points*. Then, the aspect can be woven into the base model thanks to a composition protocol, or weaving directives. Both the *pointcut* expression and the composition protocol are domain-dependent.

In most of the AOM approaches, these two activities are often limited to a particular domain: UML Class Diagrams or Sequence Diagrams [2], State Machines [4], High-level Message State Charts [6]. We argue that the *pointcut* expression and the composition protocol could be generic. Identifying *join points* in a class diagram or in a Finite State Machine (FSM) relies on the same principle: matching model elements. Specifying a composition protocol for class diagram or a FSM needs domain-specific weaving directives, but the underlying process is similar in both cases and boils down to compose model elements.

In this paper we propose a generic aspect-oriented modeling framework that can easily be adapted to different domains. We define the *pointcut* as a model snippet and the composition protocol as a set of adaptations or weaving directives. The remainder of this paper is organized as follows. The need for a generic aspect-oriented modeling framework is motivated in Section 2. Section 3 presents the generic template mechanism we use to point out *join points* while Section 4 presents our generic adaptation metamodel. Section 5 presents two applications of the framework and Section 6 concludes.

2. A MOTIVATING EXAMPLE

This section illustrates the need for a generic aspect-oriented framework, adaptable to almost every domain. In this example, we will compose two "comparable" aspects from two different domains: Class Diagram and Finite State Machine (FSM). Figure 1 illustrates a simple class diagram and a FSM.

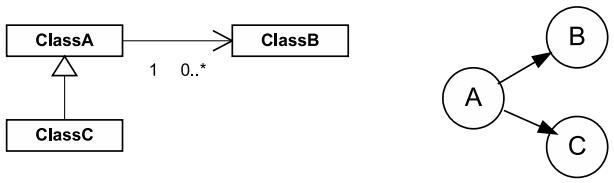

Figure 1: Two models from two different domains

In the class diagram, the aspect consists of making all the classes inheriting from a unique root class "RootClass". In other words, all the classes with no super class will inherit from "RootClass". In the FSM, the aspect consists in introducing a new state "Final"that can be reached from all the other states. In other words, all the states with no outgoing transition will be linked to "Final". Figure 2 shows the *join points* where the aspect can be composed, in each model.

The weaving of each aspect works as follows. In the class diagram, "ClassA" and "ClassB" are inheriting from "RootClass" while in the FSM model, "Final" can be reached from "B" and "C". Figure 3 illustrates the result of this composition.

We can see that these two aspects have strong similarities: in both cases a new element is introduced and then it is linked to existing model elements. We argue that an aspect-

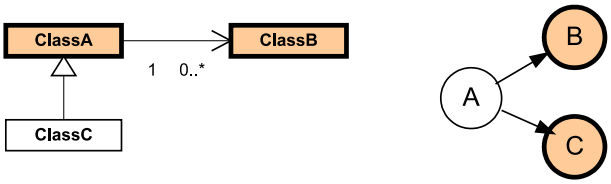

Figure 2: Join Points selection in both models

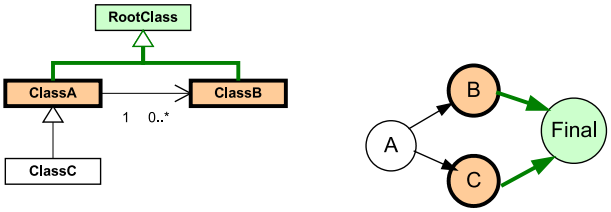

Figure 3: Composition of the aspect in both models

oriented approach should be able to deal with different domains. Our generic aspect-oriented modeling framework is presented in the next two sections.

3. TEMPLATE MECHANISM

As suggested in the introduction, in most of the AOM approaches, the *pointcut* is a template model whose elements can match *join points*. This template model might be expressed as *model snippets*, such as the UML templates [3] in Figure 4. Having patterns expressed in this way, it is possible to allow a user to draw patterns using editors that he is used to, when drawing the models that he intents to match. This section presents the concept of model snippet, how we can build a template mechanism for any domain and how we perform the pattern matching [12].

3.1 Model snippet

Each model-snippet defines a set of information existing in the model that we wish to match. For example, in Figure 4, a class named $Trace$ is declared with two methods ($traceEntry$ and $traceExit$). Whenever, a class contains the same name and methods, it matches with $Trace$. Obviously, the matched class might have also more information, such as methods, attributes or associations.

The snippet in Figure 4 was constructed for UML models [3]. Many other domain specific languages could take advantage of a similar approach. In brief, we can define *model snippets* as:

> A set of objects S is a ***model snippet*** of a metamodel MM iff:
>
> - every object in S is an instance of a metaclass defined in MM;
> - there exists a set M where M is a Valid Model w.r.t. MM and S is subset or equal to M.
>
> where, we assume that a model and a metamodel are respectively sets of EMOF objects and classes.

Every valid model is also a snippet (w.r.t its own metamodel) and so is every model that can be obtained by removing

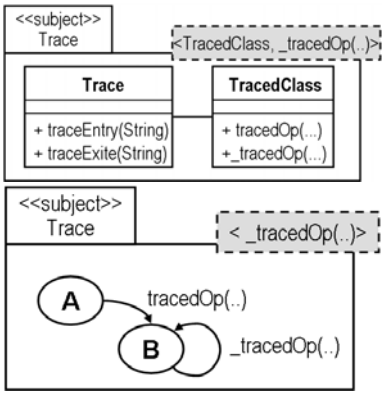

Figure 4: UML Templates of a Class and State Diagram

objects from that model. But not every model that may be obtained by adding objects to such a model.

With this in mind, we show how we can express model-snippets in any domain. We present in Sect. 3.2 a pattern-framework with the minimal elements that form a pattern. In Sect. 3.3, we show how to create *model snippets* and to customise this framework according to a target metamodel.

3.2 Pattern-framework metamodel

Taking a closer look at the model-snippet in Figure 4, we can see the snippet as an instance of the UML metamodel. All elements in the snippets are instances of a UML classifier, and furthermore inherit from a superclass $NamedElement$. For instance, $Trace$ is an instance of a UML $Class$ identified with a feature $name$ equal to $'Trace'$, and the method $traceEntry$ is an instance of $Operator$ with $name$ equal to $'traceEntry'$. The same happens in the model that we want to match. An important finding from this observation is that a snippet specifies a subset of instances, and of associations among them, in the model that we want to match. This set of instances is the information that we use to match models. However, a pattern seems to be a little more complex than just a set of instances of a metamodel.

It is clear from Figure 4 that a pattern is mainly formed by a *snippet* part (in each package of the figure) and a sequence of free-variables over this *snippet* part (in rectangle on the top-right side of each package). The purpose of variables is to define the selection criteria for a particular model element. A variant can also be conceptually seen as a placeholder for any element in the *intended model* that is matched to it. In most cases, variables represent elements that play a significant role in the pattern, and that we have a special interest in matching with. Contrary to variables, non-variables must be directly associated to a unique element in the *intended model*, containing all features which identify the element.

Nearly all metamodels define a special feature that uniquely identifies each element of their models. As we wish to be able to match variables with more than one element in the model, we do not take into account this identifier during pattern-matching of variables. For instance, $TracedClass$ is

a variable in Figure 4. So it matches to any class, with any name, that has the same methods than $TracedClass$ and an association to a class named $'Trace'$. $Trace$ is a non-variable, and, furthermore, we take into account its *name* during pattern-matching. As in most of the cases, UML uses a feature *name* as an identifier. However, the feature can change in other metamodels.

The more information is expressed in the structural part of the pattern, the more precise is the pattern-matching. However, an excessive and detailed *snippet* might also uncover all positive matches. For this reason, as any model, a pattern can have additional constraints, which help to better describe the pattern and to relate variables with other elements in the model.

Note that constraints between variables and non-variables in a pattern should still be valid after they have been matched with elements in the *intended model*. From this standpoint, constraints might also help to describe false positives in a pattern, improving the accuracy of the pattern-matching.

Based on the concepts presented above, we propose a generic metamodel for patterns illustrated in Figure 5. In this metamodel, $Pattern$ represents the whole pattern, and $PatternStructure$ represents its structure. The structural part contains a $PModel$ with a set of instances of classes (*elements*) from a given metamodel, related to the domain metamodel that describes models in which we look for matches (*intended model*). $PatternStructure$ has also a set of $Role$, which express pattern variables in the structural part. Additionally, the pattern can also have some constraints, or *invariants*, that, here, might be expressed in OCL or Kermeta [10].

Figure 5 presents what we call a *Pattern Framework*. $PModel$ is the point (*hot spots*) where the framework can be adapted or specialized by the developer. The specialisation of our framework to the metamodel that describes the *intended models* is described in the next section.

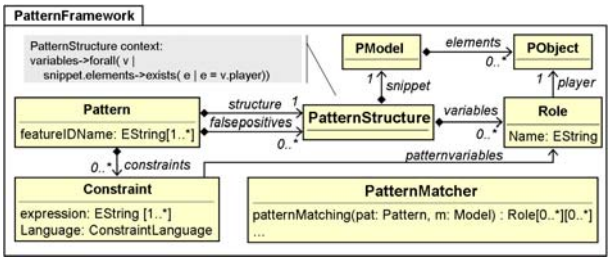

Figure 5: metamodel of the pattern framework

3.3 Constructing model-snippets

Most of the time, the metamodel (MM) of the *intended model* is too restrictive to represent patterns. The reason for that is very simple; patterns have to be expressed in a higher level of abstraction, such as *model-snippets* (see definition in the beginning of Sect. 3.1). For example in the state machine metamodel, it is totally understandable that someone does not want to provide the mandatory *event* of a $Transition$, or even want to instantiate a $Vertex$, which is defined as abstract, in order to match over instances of $State$ or $PseudoState$. We want a snippet to rely as much as possible on the same concepts of MM. In order to do that, we construct on demand a more flexible metamodel (MM') that allows us to represent abstract patterns with all concepts of the metamodel of the *intended model*. The flexible metamodel MM' is equals to MM, except that:

- No invariant or pre-condition is defined in MM';
- All features of all classes in MM' are optional;
- MM' has no abstract element.

Then, we can notice that all concepts in MM are also represented in MM'. MM' describe a wider range of models, including all models described by MM (see Figure 6). This is obtained by removing all the restrictions that exist in MM: invariants, mandatory features, nonexistence of instances of certain class. To allow a feature to be optional, we just set its lower bound as zero. All these restrictions can be expressed as invariants over a group of classes S, and any group of classes with a weaker invariant could be taken as a generalisation of S [14].

Note that any model that conforms to MM also conforms to MM', and, furthermore, any model-snippet that conforms to MM also conforms to MM'. This is an important result, it shows that we can still use existing graphical editors to draw pattern snippets and use them for pattern-matching. It also means that any metamodel that generalizes MM can be used to specify more abstract patterns.

For example, we can generate a new and flexible metamodel (MM') from a state machine metamodel (MM). MM' might describe, for instance, a $Region$ with zero $InitialStates$ or a $State$ with no $Activity$. It also describes $Vertex$ as a concrete class, allowing instances of it.

Finally we need to merge our general framework for patterns (Figure 5) with this flexible metamodel (MM'), that generalizes the intended domain (MM). This composition is called a weaving because it integrates $PObject$ from the $PatternFramework$, as a superclass of all the meta-classes in MM'. This transformation can be compared to an interface introduction in AspectJ [1] that adds a new superclass to a type. Our weaving process is equivalent to this mechanism. The implicit pointcut used in our weaving applies the introduction of the $PObject$ superclass into all the metaclasses with no superclass. The whole process to derive MM' from MM and to permit the specification of valid pattern $snippet(PAT_{snippet})$ is presented in Figure 7.

For example, we can generate a new metamodel that weaves our *pattern framework* and the state machine metamodel. It includes classes from both framework and metamodel, and

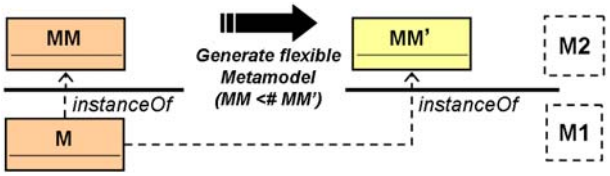

Figure 6: Process for deriving a metamodel for pattern *snippets*.

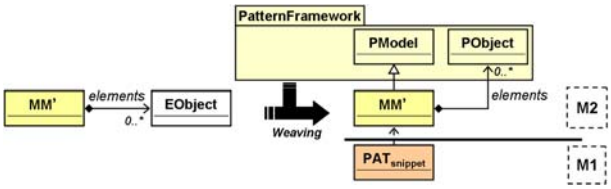

Figure 7: Process for customising the pattern framework.

additionally hook all classes from the latter with *PObject*. Then, all classes that do not have a super classes in MM' inherit from *PObject* after the weaving process. As a result, we obtain a metamodel that can be used to express model-snippets and also can be taken as an input of the pattern-matching mechanism.

3.4 Template Matching

The two previous subsections present how to build a domain-specific pattern matching framework, for any domain. However, an efficient implementation of the pattern matching might not be so easy to construct. This is an extensive topic of research, which has produced several existing languages and APIs with embedded pattern-matching mechanisms [13, 15]. For that reason, we have decided to rely on these existing tools as much as possible in order to integrate our ideas and to contribute with existing tools in this research topic.

Our implementation relies on the Kermeta language [10], an executable and object-oriented DSL (Domain Specific Language) for metamodel engineering. Kermeta is built as a conservative extension of EMOF, giving special attention to the specification of abstract syntax, static semantic (OCL) and operational semantics as well as connexion to the concrete syntax [11]. Consequently, an EMF model is seen as a Kermeta model without operational semantics. Through our implementation, we contribute with pattern-matching mechanisms to the metamodel engineering environment available with Kermeta, which includes model transformations, aspect weaving and loading of EMF models.

For our purpose, we have implemented a pattern-matching front-end in Kermeta. This front-end behaves as an abstract interface between our framework for pattern-matching and existing engines with embedded pattern-matching mechanisms. In order to delegate computation to these engines, we require the implementation of a specialised back-end for each engine.

As a proof of concept, we have constructed a back-end that uses a Prolog engine to perform pattern-matching. Using this approach, facts are derived from the base model in which we want to match a pattern, and are inserted in a knowledge base of the engine. Then, queries are generated from the pattern and are submited to the knowledge base. Finally, subsets of the facts that matches with the pattern (or bindings) are computed.

This generic pattern matching framework is integrated in our generic AOM framework: we use patterns as pointcuts and the bindings resulting from the pattern matching are the join points.

4. ADAPTATION METAMODEL

In this section, we first present our generic adaptation metamodel which is inspired from the SmartAdapters [8] approach for the composition of Java programs. Then, we explain how to specialize this framework for a specific domain metamodel.

4.1 Generic Adaptation Metamodel

The root element of the adaptation metamodel is the *Adapter*. An *Adapter* is composed of an *Aspect* and *Adaptations*. An *Aspect* is composed of a *PatternModel* (*template*) that is used to match base model elements, and a *PModel* (*structure*) that represents the aspect structure. *Adaptations* refer to *PObjects* (*parameters*) from the template or the aspect structure, and describe the composotion protocol of the aspect. The generic adaptation metamodel is illustrated in Figure 8.

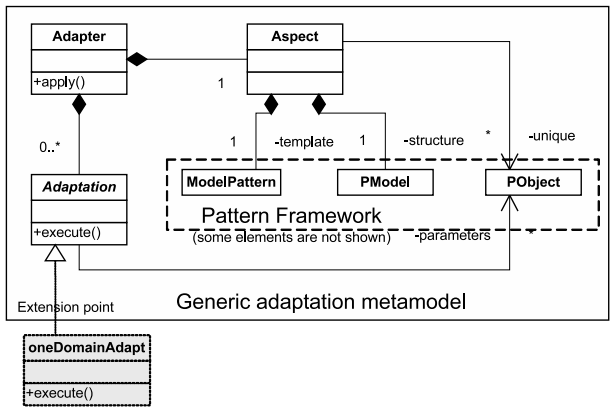

Figure 8: Adaptation Metamodel

The weaving process is quite simple: First, all the elements of the aspect structure are cloned. Then, the pattern matching is executed on a base model using the aspect *template*. For each computed binding, all the *adaptations* of the adapter are executed. When an adaptation introduces an element of the aspect structure, it actually introduces a clone of this element. Finally, a new clone is generated for each non-*unique* aspect structure element and the same process is executed for the next binding.

4.2 Specializing the Adaptation Metamodel

The *Adaptation* meta-class is abstract and therefore cannot be instanciated. This meta-class is an extension point of our framework. In order to specialize the framework for a specific domain, users can define in Kermeta [10] domain-specific adaptations that extends *Adaptation*. These specific adaptations must implement the *execute* method to specify how *parameters* are composed. For example, users can define an adaptation *IntroduceTransition* that inherits from *Adaptation*, and specifies which operations are needed to introduce a *Transition* between a source *State* and a target *State*.

Another complementary solution to specialize the framework for a specific domain is to automatically generate some adaptations e.g. creation or removal of model elements. We use the pattern matching framework to match metamodel snippets (M2 level), for a given meta-metamodel (M3 level : EMOF, ECore). Every pattern is associated with a template adaptation that should be concretized into some

domain-specific adaptations written in Kermeta, according to the computed bindings.

Fisrt, we need to generate an unconstrained ECore metamodel (or EMOF), as we explained in Section 3. Then we design some patterns relevant for code generation and write template adaptations. For example, in most of the domains, it would be very helpful to generate an adaptation that adds a new content to a container. We need a very simple pattern (Figure 9) composed of a class *Container* that is composed of classes *Containment* via a composition relationship *Composed*.

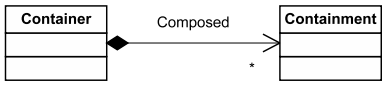

Figure 9: Container pattern

This pattern is associated with a template adaptation (Figure 10).

```
class add[Content]Into[Container]Via[Composed] inherits Adaptation
{
    operation execute() : Void raises TypingException is do
        var container:[Container]
        var content:[Content]

        //Parameter typing
        container?=adapter.getRealObject(parameter.elementAt(0))
        content?=adapter.getRealObject(parameter.elementAt(1))

        if(container!=void and content!=void) then
            container.[Composed].add(content)
        else //at least one parameter is not well typed
            raise TypingException.new
        end
    end
}
```

Figure 10: Template adaptation

We can search for this pattern in every metamodel, for example the FSM metamodel. There are two possible bindings because there are two containment relationship in this metamodel: a *FSM* contains states (*State*) and transitions (*Transition*). Then, one adaptation is generated for each binding: template parameters are substituted with corresponding elements matched in the FSM metamodel. Figure 11 shows the generated adaptation that adds a new state into a FSM. Generation of concrete adaptation is comparable to frame specification [9].

5. APPLICATION

When the framework is specialized, the user can design an adapter, specifying the template, the structure of the aspect, and several adaptations refering to the aspect template or structure model element, in order to specify how the aspect would be composed into the base model. Elements of the template are substituted with corresponding elements of the base model whereas elements of the structure are substituted with cloned elements.

We will describe the aspect presented in Section 2, for the FSM domain. The aspect is illustrated in Figure 12 and aims at introducing a final state for all the states without outgoing transitions. Then, the pattern matching is executed on the base model illustrated on the left of Figure 13.

```
class addStateIntoFSMViaStates inherits Adaptation
{
    operation execute() : Void raises TypingException is do
        var container:FSM
        var content:State

        //Parameter typing
        container?=adapter.getRealObject(parameter.elementAt(0))
        content?=adapter.getRealObject(parameter.elementAt(1))

        if(container!=void and content!=void) then
            container.States.add(content)
        else //at least one parameter is not well typed
            raise TypingException.new
        end
    end
}
```

Figure 11: Generated concrete adaptation

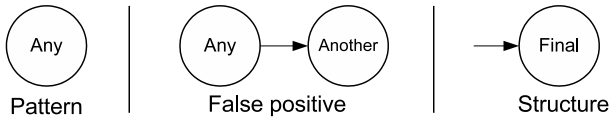

Figure 12: Designing the aspect for FSM

The composition protocol is very simple: an adaptation introduces the state "Final" into the base FSM, an another adaptation introduces the transition between the state playing the role "Any" and the state "Final". The state "Final" is *unique*, so it is introduced only once, while the transition is not *unique*, so a new clone is introduced for every binding. This protocol is applied for all the bindings as shown in Figure 13. Note that if "Final" were not *unique*, there would be two states "Final" after composition: one for "B" and one for "C".

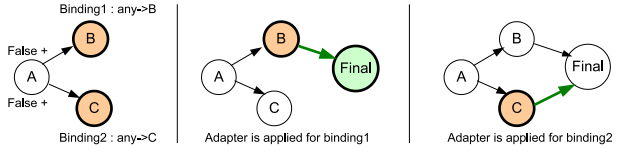

Figure 13: Weaving the aspect into the base model

Now, we can specialize the framework for class models and apply the aspect presented in Section 2. We define the aspect in Figure 14 that is syntactically very close to the previous one in the context of FSMs. This aspect aims at introducing a root class in a target class model. Then, we execute the pattern matching and compose the aspect for each binding, as shown in Figure 15

The composition protocol is similar to the previous one in the context of FSMs: an adaptation introduces the class "RootClass" in the base class model, and a second adaptation makes the class bound to "Any" inherit from "RootClass". The same process is applied for all the bindings, as illustrated in Figure 15.

Both applications are syntactically very close, but with a different semantic corresponding to their respective domain. Our framework can easily be specialized for different domains and can realize both applications.

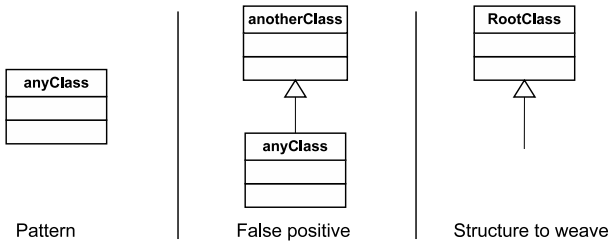

Figure 14: Designing the aspect for Class Model

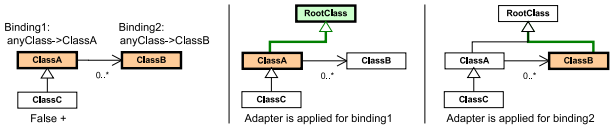

Figure 15: Weaving the aspect into the base model

6. CONCLUSION

In this paper we have presented our generic aspect-oriented modeling framework based on the Kernel Meta-Modeling language Kermeta[1] [10]. This framework is inspired by the SmartAdapters approach [8], but the template mechanism has been revised. Furthermore, the SmartAdapters approach is domain-specific: it first focuses on Java program composition [8] and work is in progress in the domain of EMF models.

In future work, we will improve the automatic generation of domain-specific adaptations, so that the user has even less work to do. Currently we are working on the introduction of variability mechanisms in our approach to make our framework more flexible [7]. For example, it will possible to compose an aspect in different ways, declaring some parts of the protocol as alternatives with several possible variants of composition. We will also leverage the notion of model typing [14]. Currently, when the framework is customized for a given domain metamodel, aspects can only be composed into base models comforming to this metamodel. We want to generalize our approach, making it possible to apply an aspect on every model comforming to a subtype of the metamodel. Finally we will also study how to propose a complete aspect-oriented desgin process such as [5] based on our approach. In particular, we will have to focus on traceability: for example, we can assume that the class diagram and the FSM presented in this paper correspond to two different views of the same system, for two different phases of the lifecycle. We need to be able to trace the aspect from one phase to the next/previous.

7. REFERENCES

[1] AspectJ Team. The AspectJ programming guide. www.eclipse.org/aspectj/doc/released/progguide, 2002-2003.

[2] E. Baniassad and S. Clarke. Theme: An Approach for Aspect-Oriented Analysis and Design. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Siobhán Clarke and Robert J. Walker. Composition patterns: an approach to designing reusable aspects. In *23rd International Conference on Software Engineering, ICSE'01*, pages 5–14, Washington, DC, USA, 2001. IEEE Computer Society.

[4] T. Cottenier, A. van den Berg, and T. Elrad. The Motorola WEAVR: Model Weaving in a Large Industrial Context. *AOSD'06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development, Industry Track*, 2006.

[5] A. Jackson and S. Clarke. Towards a Generic Aspect Oriented Design Process. *7th International Workshop on Aspect-Oriented Modeling,(AOM 2005) Models*, 2005.

[6] J. Klein, L. Hélouët, and J.M. Jézéquel. Semantic-based weaving of scenarios. *AOSD'06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 27–38, 2006.

[7] Ph. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J.M. Jézéquel. Introducing variability into aspect-oriented modeling approaches. In *MoDELS '07: Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, Tennessee, September 30 - October 5, 2007.*

[8] Ph. Lahire and L. Quintian. New Perspective To Improve Reusability in Object-Oriented Languages. *Journal Of Object Technology (JOT)*, 5(1):117–138, 2006.

[9] Neil Loughran and Awais Rashid. Framed aspects: Supporting variability and configurability for aop. In *ICSR*, volume 3107 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 2004.

[10] P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *Proceedings of MODELS/UML'2005*, volume LNCS 3713, Springer-Verlag, October 2005.

[11] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-driven analysis and synthesis of concrete syntax. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 98–110. Springer, 2006.

[12] R. Ramos, O. Barais, and J.M. Jézéquel. Matching model-snippets. In *MoDELS '07: Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, Tennessee, September 30 - October 5, 2007.*

[13] J. Sprinkle, A. Agrawal, T. Levendovszky, F. Shi, and G. Karsai. Domain model translation using graph transformations. In *International Conference Engineering of Computer-Based Systems*, pages 159–168, 2003.

[14] J. Steel and J.M. Jézéquel. On Model Typing. *Software and System Modeling: SoSyM*, 2007.

[15] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *2nd Int. Workshop on Applications of Graph Transformation*, volume 3062 of *LNCS*, pages 446–453. Springer-Verlag, 2004.

[1] see www.kermeta.org