Aspect-Oriented Techniques in Metamodel-Based Model Transformation

László Lengyel, Tihamér Levendovszky, Hassan Charaf

Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary
lengyel@aut.bme.hu, tihamer@aut.bme.hu, hassan@aut.bme.hu

Abstract: Aspect-Oriented Software Development (AOSD) is an emerging area with the goal of promoting advanced separation of concerns throughout the software development lifecycle. AOSD started on programming language level, but it must be applicable on a higher abstraction level as well. This paper provides an overview of aspect-oriented software development, discusses the key AOSD concepts, and presents the aspect-oriented programming and aspect-oriented modeling. Furthermore, introduces an aspect-oriented constraint management approach applied in a metamodel-based model transformation system.

Keywords: Aspect-Oriented Software Development, Aspect-Oriented Modeling, Aspect-Oriented Constraint Management, Metamodel-Based Model Transformation.

1 Introduction

Aspect-oriented (AO) techniques are popular today for addressing crosscutting concerns in software development. Aspect-oriented software development (AOSD) methods enable the modularization of crosscutting concerns within software systems. AOSD techniques and tools, applied at all stages of the software lifecycle, are changing the way software is developed in various application domains, ranging from enterprise to embedded systems.

A growing area of research in the field of software development is concentrated on bringing aspect-oriented techniques into the scope of analysis and design [1] [2] [3]. The motivation of these efforts is the systematic identification, modularization, representation, and composition of crosscutting concerns such as security, mobility, distribution, and resource management. Its most important potential benefits include improved ability to reason about the problem domain and the corresponding solution; reduction in software model and application code size, development costs and maintenance time; improved code reuse; architectural and design level reuse as well as better modeling methods across the lifecycle.

The remainder of this paper is organized as follows: Section 2 gives an overview of the aspect-oriented software development, illustrates the principles of the AOSD, aspect-oriented programming (AOP) and aspect-oriented modeling (AOM). Section 3 introduces an aspect-oriented constraint management (AOCM) approach applied in a metamodel-based model transformation system, and finally concluding remarks are presented.

2 Aspect-Oriented Software Development Overview

Aspect-Oriented Software Development (AOSD) [1] [4] is a technology that extends the separation of concerns (SoC) in software development. The methods of AOSD facilitate the modularization of crosscutting concerns within a system. Aspects may appear in any stage of the software development lifecycle (e.g. requirements, specification, design, and implementation). Crosscutting concerns can range from high-level notions of security to low-level notions like caching and from functional requirements such as business rules to non-functional requirements like transactions. Researchers in AOSD are driven by the fundamental goal of better separation of concerns (SoC).

AOSD is an emerging paradigm that provides explicit abstractions for concerns that tend to crosscut over multiple system components and result in tangling in individual components. By representing crosscutting concerns *or aspects* as *first-class abstractions*, and by providing new composition techniques for combining *aspects* and components, the modularity of the system can be improved leading to a reduced complexity of the system and easier maintainability.

The aim of this section is to provide a conceptual discussion on the problems that are tackled by AOSD. The discussed concepts and problems appear to be general for the complete software development life cycle. We will discuss the following important issues: the concerns and their separation, the problem of the crosscutting and tangling concerns, the aspect-oriented decomposition, and aspect weaving. Furthermore, aspect-oriented programming and aspect-oriented modeling is presented in more detail.

2.1 General AOSD Concepts

Separation of concerns. To understand the ideas in AOSD, we have first to look at the *separation of concerns principle*, which can be actually considered one of the key principles in software engineering. This principle states that a given problem involves different kinds of concerns, which 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

separation of concerns principle is a ubiquitous software engineering principle, which can be applied in various ways.

Concerns. Despite a common agreement on the necessity of the application of the separation of concerns principle, there is not a well-established understanding of the notion of concern. For example, in object-oriented methods the separated concerns are modeled as objects and classes, which are generally derived from the entities in the requirement specification and use cases. In structural methods, concerns are represented as procedures. In aspect-oriented programming, the term concern is extended with the so-called crosscutting properties such as synchronization, memory management and persistency. In a sense one can consider this as a generalization of the notion of concern in the context of programming languages. Although we consider this as a natural development, it increases the necessity of renewed understanding of what concerns are because concerns are not anymore restricted to objects and functions. Moreover, the task of separating the right concerns is complicated because one has now to deal with larger set and variety of concerns.

Any engineering process has many things about which it cares [1]. These range from high-level requirements (*"The system shall be stable"*) to low-level implementation issues (*"Remote values shall be cached"*). Some concerns are localized to a particular place in the emerging system (*"When the Ctrl+A hot key is pressed, a defined window shall pop up"*), some involve systematic behavior (*"All exception handling shall be traced"*). Generically, we call all these concerns, though AOSD technology is particularly directed at the last, systematic class.

2.1.1 Problem Statement

Modular Decomposition. The separation of concerns principles states actually 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. This decomposition process is illustrated in Fig. 1a. The design problem is decomposed into concerns (C_1, C_2…C_n) and each of these concerns is mapped to a separate module (M_1, M_2…M_n). A module is an abstraction of a modular unit in a given design language (e.g. class or function) [5].

Crosscutting Concerns. Many concerns can indeed be mapped to single modules. Some concerns, however, cannot be easily separated, and given the design language we are forced to map such concerns over many modules. This is called *crosscutting***.** In Fig. 1b, for example, concern C_2 is mapped to the modules M_1, M_2 and M_{n-1}. We say that C_2 is a crosscutting concern or an aspect. Examples of aspects are e.g. tracing, synchronization, and load balancing. 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 are dealing with these crosscutting concerns this is in principle not possible, 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 the concern because it is spread over many places. Finding the places where the crosscutting occurs is the first problem, adapting the concern appropriately is another problem.

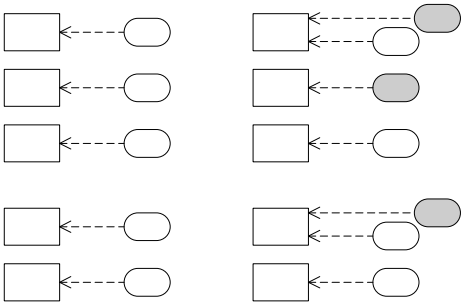


Figure 1
(a) Mapping concerns C_1, $C_2 \ldots C_n$ to modules M_1, $M_2 \ldots M_n$, (b) Concern C_2 crosscuts modules M_1, M_2 and M_{n-1}

Things may even worsen if we have to deal with multiple crosscutting concerns. For example in Fig. 2a we have to deal with 2 crosscutting concerns C_2 and C_3.

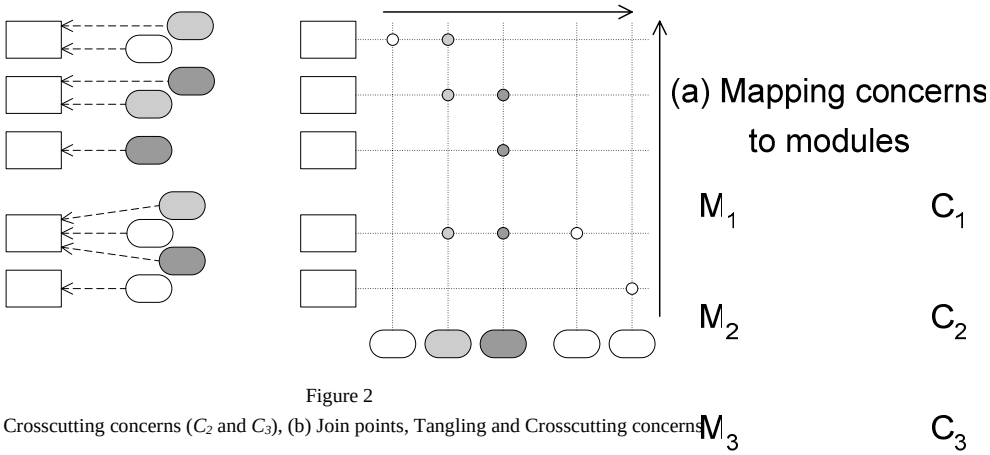


Figure 2
(a) Crosscutting concerns (C_2 and C_3), (b) Join points, Tangling and Crosscutting concerns

Tangled Concerns. Since we cannot easily localize and separate crosscutting concerns several modules will include more than one concern. We say that the

concerns are tangled in the corresponding module. For example in Fig. 2a, the concerns C_2, C_3 and C_{n-1} are tangled in the module M_{n-1}. Note that concern C_{n-1} is not crosscutting.

Join points. In Fig. 2b the same information is depicted as in Fig. 2a. The modules are aligned vertically and the concerns horizontally. The circles represent the places where the concerns crosscut a module. These are called *join points*. Join points can be at the level of a module (class) or be more refined and deal with sub-parts of the module (e.g. attribute or operation). Crosscutting can be easily identified if we follow a concern in a vertical direction (multiple join points). Tangling can be detected if we follow each module in the horizontal direction.

Formatted: Bullets and Numbering

2.1.2 Aspect Decomposition and Weaving

Aspect. Obviously, a given design problem can have crosscutting concerns and conventional abstraction mechanisms fail to cope appropriately with these concerns. AOSD provides explicit abstractions for representing crosscutting concerns, referred to as *aspects*. As such, a given design problem is decomposed into concerns that can be localized into separate modules and concerns that tend to crosscut over a set of modules.

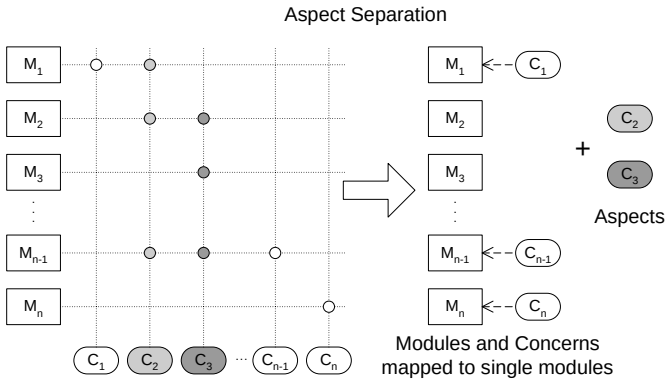


Figure 3
Aspect separation

Pointcut specification. To specify the points that the aspect crosscuts a *pointcut specification* is used. A pointcut specification is, essentially, a predicate over the complete set of join points that the aspect can crosscut. A pointcut specification can enumerate the join points or provide a more abstract specification. In an abstract sense, aspects can thus be specified as follows.

Aspect name

Pointcut specification

Advice. The crosscutting is actually localized in the pointcut specification. The pointcut specification indicates which points the aspect crosscuts but it does not specify what kind of behavior is needed. For this the concept of *advice* has been introduced. An advice is a behavior that can be attached before, after, instead of or around a join point in the pointcut specification.

Aspect name

Pointcut specification

Advice

Weaving. Having separated the aspects, their management (e.g. maintenance or reuse) become easier and consistent. In order to obtain a complete system from the separated artifacts, AO provides the weaving mechanism. *Weaving* is the process of composing core functionality modules with aspects, thereby yielding a working system. The various AO approaches have defined several different mechanisms for weaving.

There are several aspect-oriented approaches and languages. Although they differ in the way of specifying aspects, pointcuts, advices, and weaving basically adopt the concepts presented above.

In summary, AOSD emphasizes the separation of concerns and is designed to handle complex structures. Both AOP and AOM are part of the AOSD paradigm.

2.2 Aspect-Oriented Programming

The history of programming has been a slow and steady climb from the depths of direct manipulation of the underlying machines to linguistic structures for expressing higher-level abstractions. The progress in programming languages and design methods has always been driven by the invention of structures that provide additional modularity. Subroutines assembled the behavior of unstructured machine instructions, structured programming argued for semantic meaning for these subroutines, abstract data types recognized the unity of data and behavior, and object-orientation (OO) generalized this to multiplicity of related data and behaviors.

The current state-of-the-art paradigm in programming is the object-oriented (OO) technology. With objects, the programmer is supposed to think of the universe as a set of instances of particular classes that provide methods, expressed as imperative programs, to describe the behavior of all the objects of a class.

Object-orientation has many advantages, particularly in comparison to its predecessors. Objects facilitate modularization. The notion of sending messages to objects helps concentrate the programmer's thinking and aids understanding code.

Inheritance mechanisms in object systems provide a way both to assign the related behaviors to multiple classes and to make exceptions to that regulation.

Objects are not the last improvement in programming paradigms. AOSD techniques are the next step in this progression. Aspects introduce new linguistic mechanisms to modularize the implementation of concerns. Each of the earlier steps (with the minor exception of multiple inheritance in OO systems) focused on centralizing on a primary concern. AO, like its predecessors, is about recognizing that software systems are built with respect to many concerns and those programming languages, environments, and methodologies must support modularization mechanisms that honor these concurrent concerns. AO is technology for extending the kinds of concerns that can be separately and efficiently modularized [1].

AOP is a technology for separation of crosscutting concerns on programming language level into single units referred to as aspects. An aspect is a modular unit of crosscutting implementation. It encapsulates behaviors that affect multiple classes into reusable modules. Aspectual requirements are concerns that introduce crosscutting in the implementation. Typical aspects are synchronization, error handling or logging. With AOP, each aspect can be expressed in a separate and natural form, and can be automatically combined together into a final executable form by an aspect weaver. As a result, a single aspect can contribute to the implementation of a number of procedures, modules or objects, increasing reusability of the codes. The differences between AOP and traditional programming are shown in Fig. 4. Compared to traditional approaches AOP allows separation of crosscutting concerns at source code level. The aspect code and other part of the program can be woven together by an aspect weaver before the program is compiled into an executable program.

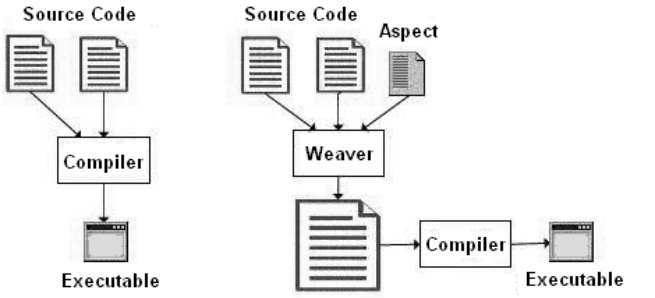


Figure 4
Traditional and AOP approaches

2.3 Aspect-Oriented Modeling

Modeling is a key tool in software engineering, allowing the software production process to be represented at a variety of stages and levels of detail. Aspect-oriented modeling techniques allow system developers to address crosscutting and quality objectives, such as security separately from core functional requirements during system design [12]. An aspect is a pattern of structure and behavior such that it is a crosscutting realization of common structural and behavior characteristics [13].

An aspect-oriented design model consists of a set of aspects and primary models. An aspect model describes how a single objective is addressed in the design, while the primary model addresses the core functionality of the system as given by the functional requirements.

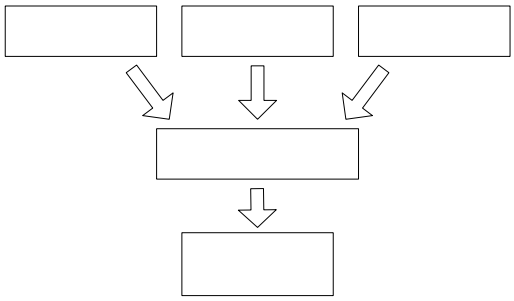


Figure 5
An overview of the AOM approach

In AOM, weaving rules are used to weave aspect models with the primary model. These rules are stored separately from the aspect and the primary model, which makes both the aspect models and the rules reusable. The aspects and the primary model are composed before implementation or code generation. The composed model supports design analysis. Composition is most often performed manually, but there are tools that automate part of the composition. Fig. 5 gives an overview of AOM.

An aspect-oriented approach is introduced in [14] 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.

Aspect Models Weaving Rul

Weaving

3 Aspect-Oriented Methods in VMTS

Visual Modeling and Transformation System (VMTS) [15] [16] is an n-layer metamodeling environment which supports editing models according to their metamodels, and allows specifying OCL constraints. Models are formalized as directed, labeled graphs. VMTS uses a simplified class diagram for its root metamodel ("visual vocabulary").

Also, VMTS is an UML-based [13] model transformation system, which transforms models using graph rewriting techniques. Moreover, the tool facilitates the verification of the constraints specified in the transformation step during the model transformation process.

The modularization of crosscutting concerns is also useful in model transformation. Model transformation means converting an input model available at the beginning of the transformation process to an output model or to source code. Models can be considered special graphs; simply contain nodes and edges between them. This formal background facilitates to treat models as labeled graphs and to apply graph transformation algorithms to models using graph rewriting. Therefore a widely used approach to model transformation applies graph rewriting [17] as the underlying transformation technique, which 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 the rule being applied to (host graph), and replacing this subgraph with RHS.

In the VMTS approach, LHS and RHS of the transformation steps are built from metamodel elements. This means that an instantiation of LHS must be found in the host graph instead of the isomorphic subgraph of LHS. These metamodel-based rewriting rules extended with control structure are called Visual Model Processors (VMP) in VMTS. Previous work [15] has shown that the rules can be made more relevant to software engineering models if the metamodel-based specification of the transformations allows assigning Object Constraint Language (OCL) [18] constraints to the individual transformation steps.

The Object Constraint Language (OCL) is a formal language for analysis and design of software systems. It is a subset of the UML standard [13] that allows software developers to write constraints and queries over object models.

The increasing demand for visual languages (VL) in software engineering (e.g. Unified Modeling Language - UML; Domain-Specific Languages - DSLs) requires more sophisticated transformation mechanisms for diagrammatic languages. Although these VLs can often be modeled with labeled, directed graphs, the complex attribute dependencies peculiar to the individual software

engineering models cannot be treated with this general model. Consequently, often it is not enough to transform graphs based on the structural information only, we want to restrict the desired match by other properties, e.g. we want to match a node with a special integer type property whose value is between 4 and 12.

To define the transformation steps precisely beyond the structure of the visual models, additional constraints must be specified which ensures the correctness of the attributes, or other properties can be enforced. Using OCL constraints in modeling is a suitable choice. It provides a solution for the unsolved issues. In our experience, constraints are proven to be useful in model transformations as well.

Often, the same constraint is repetitiously applied in many different places in a transformation and crosscuts it. It would be beneficial to describe a common constraint in a modular manner and designate the places where it will be applied. Therefore, the motivation of our aspect-oriented technique-based research is to eliminate the crosscutting constraints in visual model transformation steps.

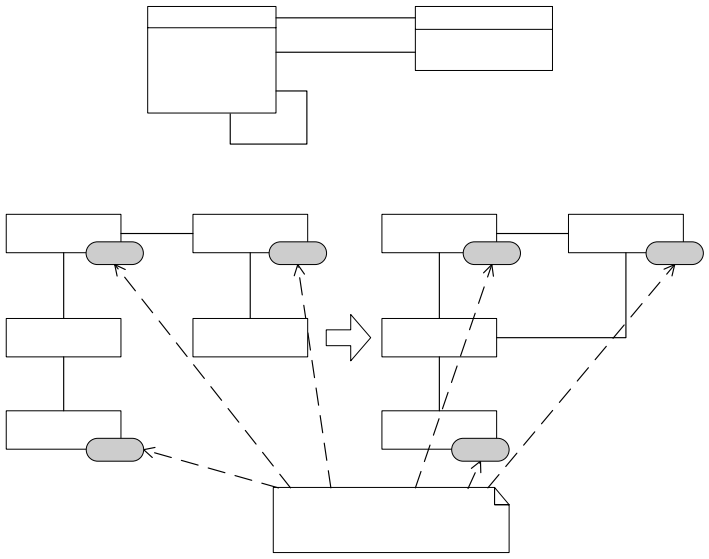
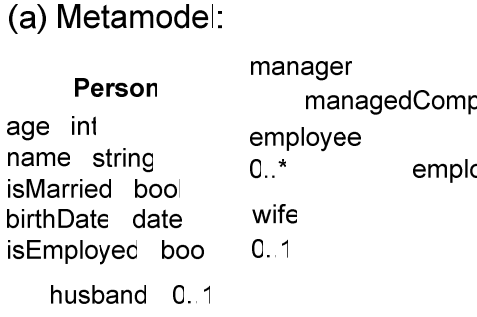


Figure 6
A sample metamodel and a transformation step with a crosscutting constraint

In Fig. 6 an example is depicted for crosscutting constraints. There is a transformation which modifies the properties of *Person* type objects and we would like the transformation to validate that the *age* of a *Person* is always under 200 (*Person.age < 200*). It is certain that the transformation preserves this property if the constraint is defined for all rewriting rule element whose type is *Person* (Fig. 6b). This means that the constraint can appear several times, and therefore the



constraint crosscuts the whole transformation, its management is not consistent. E.g. constraint maintenance must be performed on its all occurrence. Furthermore, often it is difficult to reason about the effects of a complex constraint, when it is spread out among the numerous nodes in a transformation step.

Therefore, we developed a mechanism to separate this concern. Having separated the constraints from the pattern rule nodes (nodes of the transformation steps), we also need a weaver method which facilitates the propagation (linking) of constraints to transformation step elements.

This means that using separation and a weaver method, we manage constraints using AO techniques: Constraints are specified and stored independently of any model transformation step or pattern rule node and linked to the rule nodes by the weaver. Therefore our constraints are similar to the aspects in AOP.

Conclusions

In this paper, an overview is given about the aspect-oriented software development, aspect-oriented programming and aspect-oriented modeling. Furthermore, our aspect-oriented constraint management-based model transformation approach is presented.

We have found that the source of our transformation problems is often related to the lack of support for modularizing crosscutting concerns. As we have extended our metamodel-based visual transformation language with an aspect-oriented constraint management approach, it was observed that the maintainability and understandability of our transformation steps have been increased along with the attached constraints. Considering the correctness of the transformation, the understandability of the constraints is crucial, since the verification is still left to the intuition and the experience of the designer.

Using AO constraints in metamodel-based model transformations, we achieved several benefits. Consistent constraint modification and simple constraint removal became possible. The same constraint does not appear repetitiously in many different places. Moreover, it is not necessary for the transformation steps to be aware of the constraints, or for the modeler who creates the transformation steps. Transformation steps are applicable with or without constraints. We can assign constraints to the transformation as a whole, and not only to the individual transformation steps.

These methods have successfully been applied in industrial projects, like generating user interface from resource model and user interface handler code from statechart model for Symbian [19] and .NET CF mobile platform [20].

The number of publications related to aspect-oriented techniques is quite large. As a starting point we recommend the special issue of CACM devoted to the topic [6].

References

[1] Robert E. Filman, Tzilla Elrad, Siobhan Clarke, Mehmet Aksit, Aspect-Oriented Software Development, Addison-Wesley, 2004

[2] Sixth International Workshop on Aspect-Oriented Modeling, http://dawis.informatik.uni-essen.de/events/AOM AOSD2005, Chicago, IL, March 2005

[3] Seventh International Workshop on Aspect-Oriented Modeling, http://dawis.informatik.uni-essen.de/events/AOM_MODELS2005/ Montego Bay, Jamaica, October 2, 2005

[4] AOSD Homepage, http://www.aosd.net/

[5] The Aspect-Oriented Software Architecture Design Portal, http://trese.cs.utwente.nl/taosad/aosd.htm

[6] Communications of the ACM Volume 44, Issue 10, October 2001

[7] The AspectJ Programming Guide, http://www.aspectj.org

[8] AspectC++, http://www.aspectc.org/

[9] HyperJ, http://www.research.ibm.com/hyperspace/HyperJ/ HyperJ.htm

[10] ComposeJ, http://trese.cs.utwente.nl/prototypes/composeJ/index.htm

[11] DemeterJ, http://www.ccs.neu.edu/research/demeter/DemeterJava/

[12] G. Georg, R. France and I. Ray, An Aspect-Based Approach to Modeling Security Concerns. In Proceedings of the Workshop on Critical Systems Development with UML, Dresden, Germany, 2002, pp. 107-120

[13] OMG UML 2.0 Specifications, http://www.omg.org/uml/

[14] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck, Handling Crosscutting Constraints in Domain-Specific Modeling, Communications of the ACM, October 2001, pp. 87-93

[15] T. Levendovszky, L. Lengyel, G. Mezei, H. Charaf, A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, Electronic Notes in Theoretical Computer Science, International Workshop on Graph-Based Tools (GraBaTs) Rome, 2004

[16] The VMTS Homepage. http://avalon.aut.bme.hu/~tihamer/research/vmts

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

[18] OMG Object Constraint Language Spec. (OCL), http://www.omg.org

[19] L. Lengyel, T. Levendovszky, G. Mezei, B. Forstner, H. Charaf, Metamodel-Based Model Transformation with Aspect-Oriented Constraints, Accepted to International Workshop on Graph and Model Transformation, GraMoT, Tallinn, Estonia, September 28, 2005

[20] L. Lengyel, T. Levendovszky, H. Charaf, Eliminating Crosscutting Constraints from Visual Model Transformation Rules, Accepted to ACM/IEEE 7th International Workshop on Aspect-Oriented Modeling, Montego Bay, Jamaica, October 2, 2005