ABSTRACT
Model-Driven Architecture (MDA) as a model-based approach to software development facilitates the synthesis of application programs from models created using customized, domain-specific model processors. MDA model compilers can be realized by graph rewriting-based model transformation. In Visual Modeling and Transformation System (VMTS), metamodel-based rewriting rules facilitate to assign OCL constraints to model transformation steps. This approach supports validated model transformation. The goal is to validate not only the individual transformation steps, but the whole transformations as well. Unfortunately, the validation introduces a new concern that often crosscuts the functional concern of the transformation steps. To separate these concerns and make them reusable, an aspect-oriented solution is presented for constraint management. This paper introduces a new type of aspect, the constraint aspect to separate the crosscutting constraints from model transformation steps. Moreover, algorithms are given to create and weave constraint aspects to model transformation steps prior to the execution of the transformation. The presented method results in a more efficient and optimized weaving, furthermore, we can require that not only a single transformation step, but a whole transformation validates, preserves or guarantees certain properties about the output of the transformation.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques;

General Terms
Algorithms, Design and Languages.

Keywords
Aspect-Oriented Constraints, Constraint Aspect, Constraint Weaving, Metamodel-Based Model Transformation, OCL.

1. INTRODUCTION
Model-driven development approaches (for example Model-Integrated Computing (MIC) [18] 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.

MIC advocates the use of domain-specific concepts to represent the system design. Domain-specific models are then used to synthesize executable systems, perform analysis or drive simulations. Using domain concepts to represent system design helps increase productivity, makes systems easier to maintain and evolves and shortens the development cycle.

MDA offers a standardized framework to separate the essential, platform-independent information from the platform-dependent constructs and assumptions. A complete MDA application consists of a definitive platform-independent model (PIM), one or more platform-specific models (PSM) including complete implementations, one on each platform that the application developer decides to support. The platform-independent artifacts are mainly UML and other software models containing enough specification to generate the platform-dependent artifacts automatically by model compilers.

MDA as a model-based approach to software development facilitates the synthesis of application programs from models created using customized, domain-specific model processors. Model-Driven Software Development (MDS) employs domain-specific models to represent the software, its environment, and their relationships, and thus, it is well-suited for the rapid design of complex computer-based systems.

Model transformation lies at the heart of the model-driven approaches [13] [19]. With MDS, a modeling environment operates according to a modeling paradigm, which is a set of requirements that define how a system within a domain is modeled. The modeling paradigm is captured in the form of formal modeling language specifications referred to as metamodel. Once a metamodel is created for a particular domain, a modeling environment allows a modeler to create domain-specific models that can be synthesized into various artifacts.

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 [2]. (iii) Analysis and verification; analysis algorithms can be expressed as transformations on the design [3].

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. Contrarily, 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 [10].

For example, from a transformation that transforms class model to relational database management system (RDBMS) model (transformation Class2RDBMS) we would like to require that it 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 OCL constraints assigned to the transformation steps. Unfortunately, often, the same constraint is repetitiously applied in many different places in a transformation, therefore the constraints crosscut the transformation steps and their management becomes hard.

In [8] a solution of the case study Class2RDBMS is provided where a transformation is presented with 11 transformation steps and two constraints are emphasized, from which the first one appears 30 times in 11 transformation steps and the second one 16 times in 6 transformation steps. It is very difficult to manage manually these scattered 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 an mistake during the modification. A few tangling constraints can make the whole transformation scattered. Using aspect-oriented constraints, a method has been given to solve the problem of the crosscutting constraint in model transformations. The main idea is to handle constraints similarly to AOP aspects, to provide the transformation constraints with the properties of the AOP aspects. AO constraints are created separately from transformation steps and, using a weaver method, they are woven back to the transformation steps 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 new type of aspect, the constraint aspect, which, replacing the AO constraint, also solves the problem of the crosscutting constraints. Constraint aspect is not a textual constraint, it has structure. Moreover, it fits the visual environments better, and working with it is more efficient than using AO constraints.

The rest of the paper is organized as follows: Section 2 gives some background and related work information, Section 3 introduces the problem of the crosscutting constraints in model transformation steps. Section 4 discusses the constraint aspect-based aspect-oriented solution along with the necessary constructs and algorithms. Finally, conclusions are given.

2. BACKGROUND AND RELATED WORK
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.

VMTS [11] [20] 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 [16] 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.

Models can be considered special graphs; they simply contain nodes and edges between them. This formal background makes possible to treat models as labeled graphs and to apply graph transformation algorithms to models using graph rewriting. 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.

Previous work [11] has shown that the 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 steps.

The Object Constraint Language (OCL) [15] is a formal language for analysis and design of software systems. It is a subset of the UML standard [16] that allows software developers to write constraints and queries over models. A constraint is a restriction on one or more values of an object-oriented model or system: A precondition to an operation is a restriction that must be true just prior to execution. Similarly, a postcondition to an operation is a restriction that must be true just after its execution.

Aspect-Oriented Software Development (AOSE) [1] provides a technique to address separation of concerns (SoC). The methods of AOSD facilitate the modularization of crosscutting concerns within a system. AOSD attempts to decompose a system further into core application concerns, and a special type of concern called a 'crosscutting concern'. A crosscutting concern cannot easily be modularized, because the concern tends to be scattered across a system. AOSD allows crosscutting concerns to be separated from the core concerns so that they can be reasoned about in isolation and woven back later to form a functioning system.

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.

Constraint-Specification Aspect Weaver (C-SAW) [6] is an aspect-oriented approach to modeling. The C-SAW weaver framework serves as a generalized transformation engine for manipulating models. C-SAW is a plug-in for the Generic Modeling Environment (GME) [12]. The result of model weaving is a new model that contains adaptations that are spread across the model hierarchy. These adaptations can be undone and new concerns can be woven. The weaver can be used to distribute any system property endemic to a specific domain across the hierarchy of a model. A weaver can also be used to instrument structural changes within the model according to the dictates of some specific constraint propagations.

3. CROSSCUTTING CONSTRAINTS
In model transformation, the dominant decomposition is the functional behavior of the transformation steps. 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 specified after the first draft of the transformation, and treated with secondary importance. They crosscut the transformation, and it is almost impossible for the designer to perform the intuitive steps of verifying the transformation.

In many cases not only a transformation step but a whole transformation is required to validate, preserve or guarantee a certain property. To meet this expectation, all the transformation steps have to be taken into consideration. If one defines a constraint for several transformation steps or for a whole transformation, then the same constraint appears numerous times in the transformation and crosscuts it. The modification and deletion of a crosscutting constraint is not consistent, because such an operation must be performed on all of its occurrences. Moreover, it is often difficult to reason about the effects of a complex constraint when it is scattered across the numerous rule nodes in transformation steps.

In the transformation Class2RDBMS the first step (CreateTable) creates a table for each non-abstract class, creates the columns and adds a public key (existence-based identity implementation [4]) based on the class attributes and the name of the class. The remaining part of the transformation (10 transformation steps) modifies the properties of the created tables. We want to require from the whole transformation that it guarantees that the public key, which is generated in the first step, remains unmodified.

```
context Table inv primary_key:
  self.column->exist(c | c.name = tableHelper.table.name and c.type.name = 'int' and c.isPrimaryKey)
```

It is certain that the transformation guarantees this property if the constraint is defined for all transformation step node of type Table. This means that the constraint can appear in each transformation step several times. Therefore the constraint crosscuts the whole transformation [9] [20]. Constraints appearing several times and crosscutting a transformation increase the time of constraint handling and the possibility of making a mistake during the modification. The motivation is to achieve an optimized and consistent constraint modification, deletion and propagation with separating crosscutting constraints, and weaving them back later automatically.

4. SOLUTION WITH ASPECT-ORIENTED METHODS
Aspect-oriented constraints are OCL constrains defined separately from the transformation and transformation steps and woven to the steps using a weaver method (GLOBALCONSTRAINTWEAVER) [8].

VMTS is a visual approach: metamodels, models and transformations are defined by graphical models. But OCL constraints used to specify the transformations steps are textual expressions. Therefore, the concept of constraint aspect has been developed. A constraint aspect expresses the same conditions as an OCL constraint in a visual way.

![Figure 1. (a) Example Constraint Aspect, (b) Transformation Step with Propagate Constraint Aspect.](image)

**A constraint aspect** is a pattern (structure) built from metamodel elements to which OCL constraints are assigned. A constraint aspect contains not only textual conditions described by the OCL constraints but structure, type and multiplicity conditions and weaving constraints as well. The structure, type conditions and weaving constraints are checked at propagation time, while the OCL constraints are validated during the model transformation.

**A propagated constraint aspect** is a constraint aspect linked to the elements of a transformation step. It forms a weaving configuration that contains the constraint aspect with the required conditions and the transformation step.
A constraint aspect is a visual constraint. It fits the visual transformation system better than the OCL constraint. In Fig. 1a, a constraint aspect is depicted: Class, HelperNode, Table, and the associations between them represent the structure of the constraint aspect. Cons_C1 and Cons_T1 are OCL constraints propagated to the pattern of the constraint aspect. Fig. 1b represents a transformation step with a propagated constraint aspect. The dashed line shows the propagation of the constraint aspect to RHS of the transformation step.

4.1 Creating Constraint Aspect from OCL Constraints

Fig. 2 introduces the creation process of the constraint aspects from an OCL constraint along with the normalization of the created constraint aspect. The lines with numbers from 1 to 4 show the steps of the constraint aspect creation: the algorithm identifies the context type (Attribute) and the referred types by the association ends (Class, DataType). Based on the types, it builds the pattern. Finally, the algorithm assigns the OCL constraint to the root rule node of the created constraint aspect. In Fig. 2b the dashed lines denote that the constraint Cons_A1 contains path expressions. The lines 5 and 6 show the constraint normalization (decomposition and relocation). The pseudo code of the algorithm CREATECONTRAINTASPECT is as follows.

```plaintext
CREATECONTRAINTASPECT (Constraint C): ConstraintAspect
1) CA = CREATECA(C.Context)
2) foreach NavigationStep N in C
3) PRN = CREATEPRN(TYPEOF(N.DestNode))
4) LINKPRN(CA, PRN)
5) endforeach
6) PROPAGATECONTRAINT(CA, C)
7) return CA
```

![Diagram](image)

**Figure 3. Creating constraint aspect from OCL constraint and its normalization.**

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 validation method. In general, the larger part of the validation is the first step, because of its computational complexity. Each navigation step in a constraint means several queries on the model database. Therefore, it would be beneficial to reduce the navigation steps contained by the constraints, because the eliminated navigation steps accelerate the first part of the constraint evaluation.

In [7], a method is provided to normalize OCL constraints in transformation steps and constraint aspects. This means eliminating as many navigation steps from the constraints as possible using constraint relocation and constraint decomposition. The method is also applicable for arbitrary UML class diagrams.

The computational complexity of the CREATECONTRAINTASPECT method is \(O(n)\), where \(n\) is the number of the navigation steps contained by the processed constraint.

If the constraint aspect \(CA\) is created from an OCL constraint \(C\) using CREATECONTRAINTASPECT algorithm, and the normalized constraint aspect \(CA'\) is created from the \(CA\) with NORMALIZECONSTRAINT algorithm [7], then \(C, CA\), and \(CA'\) are equivalent in the sense of the contained conditions. Furthermore, after their propagation to the transformations the constrained steps are also equivalent during the execution.

4.2 Constraint Aspect Weaving

The weaving is a static algorithm; it should be accomplished once for a set of constraints and a transformation. Its result, the constrained transformations, can be used an optional number of times.

In the proposed approach, there are two constructs that support the weaving process.

(i) The context information of the aspect-oriented constraints is a type-based pointcut - it selects rule nodes based on their metatype. The weaving process driven by the type-based pointcuts is referred to as type-based weaving.

(ii) A weaving constraint is similar to a property-based pointcut, it is also an OCL constraint that specifies the weaving but it is not propagated or used during the transformation. Weaving constraints facilitate optional conditions during the weaving process. Therefore, it is referred to as constraint-based weaving.

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.

```plaintext
class.name = "NonAbsClass"
```

This means that the constraint assigned with the presented weaving constraint is propagated only to those places, where the name of the Class starts with the term ‘NonAbsClass’. The asterisk (*) is a wild-card that stands for any combination of letters. The constraint aspect queries structures with the appropriate metatype and the weaving constraint selects the appropriate structures from the result, based on their name.

The CONSTRAINTASPECTWEAVER (CAW) algorithm propagates constraint aspects to model transformation steps. The input of the CAW is the transformation step and the constraint aspects, whereas the output is the constrained transformation step (Fig. 3).

The output of the weaving is not stored as a new transformation step. The result is handled as a linking between the constraints and a transformation step. This linking is referred to as weaving configuration.

The CAW algorithm checks each transformation step individually. In each step, it searches for an isomorphic submodel to the structure of \(CA\) (Line 3) and validates the found structures, based on the weaving constraints. For each constraint \((CA C_1...CA C_n)\) contained by \(CA\), the algorithm selects rule nodes from the matches with the metatype that corresponds to the context information of the actual constraint (lines 5-6). CAW uses the ISREFERREDTOWEAVE method to decide if a transformation step requires \(CA\) to be propagated (Line 7). If at least one of the
constraints (CA_{C_1}...CA_{C_s}) contained by CA needs to be propagated, then the whole constraint aspect is linked (lines 8-9).

```
CONSTRANTASPECTWEAVER(ConstraintAspect[] CAs, Transformation T)
1 foreach ConstraintAspect CA in CAs
2    foreach Transformation Step S in T
3        matches = METATYPEBASEDMATCHING(pattern of the CA, S)
4    foreach Constraint C in CA
5        nodesToCheck = GETNODESBYTYPE(ContextType of C, matches)
6    if (ISREQUIREDTOWEAVE(C, nodesToCheck, out nodesToWeave)) then
7        WEAPEXTRAINTASPECTASPECT(, )
8    break
9    end if
10  end foreach
```

Figure 3. The Weaving process with the input and the output artifacts of the Constraint Aspect Weaver algorithm.

4.3 Computational Complexity of CONSTRANTASPECTWEAVER Algorithm

The computational complexity of the CONSTRANTASPECTWEAVER algorithm is at most \( O(\sum_{i=1}^{c_i} \sum_{j=1}^{n_i} (n_j^{k_j} + \sum_{p=1}^{m_{jp}})) \), where \( c_i \) denotes the number of the constraint aspects, \( s \) is the number of the transformation steps, \( c_i \) is the number of the constraints contained by the constraint aspect \( i \), \( n_j^{k_j} \) is the complexity of the metatype-based searching (worst case), \( n_j \) is the number of rule nodes contained by the transformation step. Furthermore, \( j \) and \( k_j \) are the number of rule nodes contained by the constraint aspect \( i \). The \( m_{jp} \) is the number of rule nodes selected by the GETNODESBYTYPE method.

This method facilitates the approach toward managing constraints using AO techniques. Similarly to aspects, the constraint aspects are specified and stored independently of any model transformation step or rule node and are linked to transformation steps by the CAW.

However, it is more efficient to work with constraint aspects than OCL constraints, because during propagation of the constraint aspects we can use the metatype-based searching for pattern matching [11] to reduce the possible places of the constraint assignment. Then we execute the ISREQUIREDTOWEAVE algorithm only on the selected places to decide if it is necessary to assign the given constraint aspect. In case of OCL constraints, the GCW algorithm uses the CHECKSTRUCTURE method to find the structurally appropriate places in the transformation steps, while the complexity of the metatype-based searching is equal to the complexity of finding an isomorphic submodel only in the worst case.

After the weaving process it is more efficient to execute transformations with propagated normalized constraint aspects than OCL constraints. Normalized constraint aspects contain the constraints on their most adequate place, which results that there are as few navigation steps as it is possible. Therefore, evaluating constraints using propagated normalized constraint aspects requires less computational time.

The whole weaving process must be executed once for a transformation. This obviously takes time, but all of they are static algorithms, preceding execution time. Once they are accomplished, their results can be reused for arbitrary number of models.

5. CONCLUSIONS

Model-based software development requires the transformation of the models between various stages. These transformations must be precisely specified, which can be accomplished along with constraints enlisted in transformation steps.

A disadvantage of our earlier metamodel-based model transformation approach can be seen in many tangling constraints throughout of the transformation steps. In metamodel-based model transformation, the two main advantages of the aspect-oriented constraint management are the following. (i) It eliminates the crosscutting constraints from model transformations. (ii) Using AO methods, constraints become aspects. This means that the transformation steps can be executed with or without the propagated constraints as well. Moreover, the optimized transformation steps and constraints can be reused. Hence, the transformation can be executed with different propagated constraint set based on the required conditions. Furthermore, constraints are defined and stored independently from the transformations. Therefore they can be propagated to different transformations, thus, the constraint themselves can also be reused.

Using aspect-oriented constraint management in visual model transformation, it has been observed that the maintainability and understandability of the transformation steps have been increased along with the attached constraints. With the help of this
constraint modification and simple constraint removal has become possible. The same constraint does not appear repetitiously in many different places. Moreover, it is not necessary for the transformation steps or for the designer of the transformation steps to be aware of the constraints. The provided weaver algorithm works on the whole transformation, it handles all transformation steps contained by the transformations. Therefore, the result of the weaving, the constrained transformation as a whole satisfies the conditions required by the constraints [10]. The discussed 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 [9] and .NET Compact Framework mobile platform.

Working with constraints aspects is more efficient than OCL constraints. The normalization has to be accomplished once and not at each constraint validation. During the propagation the structure of the constraint aspects facilitates the more efficient weaving process. Therefore, constraint aspects make the weaving and the transformation processes more efficient.

The introduced approach can be generalized to other transformation languages which facilitate to assign constraints to transformation steps. The presented concepts and algorithms can be reused with minor, approach-related modifications. The main limitation of the aspect-oriented constraint management is that it requires more preprocessing steps than the general approach. Constraints and transformation steps have to be defined separately, and then the propagation of the constraints to the transformation steps must be performed automatically.

6. ACKNOWLEDGMENTS
The fund of “Mobile Innovation Centre” has supported in part, the activities described in this paper.

7. REFERENCES