Solving Constraints in Model Transformations

Andreas Petter[1], Alexander Behring[1], and Max Mühlhäuser[1]

Technische Universität Darmstadt, Department of Computer Science,
Telecooperation, Hochschulstr. 10, D-64289 Darmstadt, Germany,
{a_petter,behring,max}@tk.informatik.tu-darmstadt.de

Abstract. Constraint programming holds many promises for model driven software development (MDSD). Up to now, constraints have only started to appear in MDSD modeling languages, but have not been properly reflected in model transformation. This paper introduces constraint programming in model transformation, shows how constraint programming integrates with QVT Relations - as a pathway to wide spread use of our approach - and describes the corresponding model transformation engine. In particular, the paper will illustrate the use of constraint programming for the specification of attribute values in target models, and provide a qualitative evaluation of the benefit drawn from constraints integrated with QVT Relations.

Keywords: model driven engineering, constraint solving, constraint programming, model transformation

1 Introduction

Declarative model transformation languages have gained a lot of attention in the model transformation community, especially by discussing and proposing transformation languages like QVT [1]. They are deemed to have several advantages [2]. Some declarative approaches follow the relational approach, which could be called "logic programming for model transformations". These languages relate model elements from the source model to model elements of the target model.

Some models benefit from the use of constraints to overcome possible underspecification. Therefore, constraints have seen widespread usage in practical modelling [3, 4]. Furthermore, constraints have shown their declarative nature for programming tasks. Freuder even states that constraint programming would yet be the closest approach to finding the holy grail of programming [5].

Models with constraints may be transformed using standard model transformation languages. But, as transformation languages are not able to cope with constraints to specify aspects of target models, transforming these models will not take into account the constraints contained in the models (or meta-models) directly. Thus, if developers of transformations use model-to-model transformation languages and want a constraint based problem to be satisfied in target models, they will need to solve the constraints by hand after the transformation has finished.

Our contribution therefore is the *combination of constraint programming and model- to-model transformations*. Hereby we fill the gap how constraints can be specified in model transformations and how they can be satisfied in target models. Due to the generality of the basic approach, called “Constraint Relational Transformations” many model-to-model transformation languages can benefit from our contribution.

The contributions in this paper are:

- a model transformation language to define model transformations with constraint satisfaction problems,
- using constraint programming for the specification of attribute values in target models,
- a qualitative evaluation how well constraints integrate with declarative model transformation languages, investigated on the example of QVT Relations, and
- an implementation of a model transformation engine which is able to solve constraints.

1.1 Overview

The third section defines “Constraint Relational Transformations” which allow for constraint solving to be used in transformations. Section 4 presents how we changed QVT Relations and OCL to support for constraint solving. Section 5 presents our implementation of the transformation engine while section 6 discusses the benefits and 7 the state of the art. Section 8 summarizes the findings and presents opportunities for further research.

2 Constraints and Running Examples

It is important to note that the complexity of the constraint language greatly affects the complexity of the algorithms needed to satisfy the constraints. For example it is comparatively easy to solve continuous linear constraints, but difficult to solve discrete, non-linear constraints. However, in model driven engineering the most frequently used constraint language is the Object Constraint Language (OCL) [4]. It allows for the definition of constraints on almost any facet of common model elements. The language has been found to be non-computable in theory [6]. Therefore, it imposes a problem to do constraint solving over all aspects of the OCL language. Constraint solving over OCL constraints can only be done efficiently on a subset of the language. While it remains an interesting research task to formally identify this subset, we restrict our constraints to the subset of non-linear numeric constraints. This type of constraints is needed in many domains where constraint programming can be applied [7].

Constraints can be divided into *two different types: constraints over attribute values and associations in transformations versus global graph constraints*. The first type of constraints can be attached to model elements and therefore are

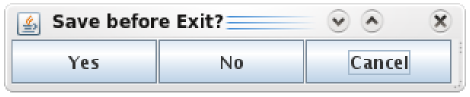

Fig. 1. Outline of the desired target window for the example (JAVA Swing)

"local" to them. Nevertheless, they can span several model elements through transitive use of associations. Constraints used in common modeling languages are local (e.g. UML and OCL). We assume that most constraints used in modelling fall into this category. The second type is global to a transformation and requires the transformation engine to satisfy them even across all model elements. Scheduling and planning problems fall into this category. We only take into account constraints of the first type. The second type would require extensive research how the concept of graph constraints can be combined with a model transformation language and integrated with constraints over attributes and associations.

This paper focuses on numeric, non-linear, local constraints. Hereby, problems like generating user interface models or problems from operations research can be addressed using a declarative model-to-model transformation language with constraint solving.

Our constraint based approach can also be used to simultaneously span constraints over source and target models at once. This is not our research focus as the main application of such a setup would be model synchronization, which we are not heading for. Therefore, the language presented can only use variables to retrieve information from the source model and not use constraints which simultaneously enforces constraints on source and target model elements.

2.1 Transformation Examples

Even in very simple model transformations numeric constraints can be a requirement. Our illustrative example is borrowed from the user interface community where constraints have been widely used to define user interfaces. The developer wants to develop a model-to-model transformation that is able to account for adaption to the display of the target platform. The target platform will show a window (c.f., figure 1), which contains three standard buttons ("Ok", "Cancel", and "Ignore"). The suitable meta-model of the abstract user interface model is shown in figure 2 while the platform specific one is shown in figure 3. The sizes of the components used when displaying are retrieved from the model (e.g. because they can be altered in a graphical model editor). Therefore, the model-to-model transformation should set them.

Hereby the following constraints must be satisfied: The sum of the width of the buttons should not be larger than the window and the window should fit on the screen (800 pixels). Furthermore the buttons must have minimum and maximum sizes (e.g. $30 \leq \text{width} \leq 60$). The resulting model transformation will

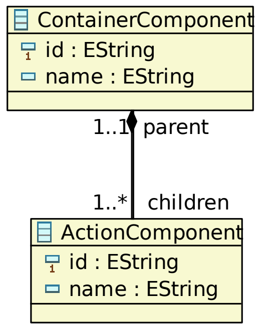

Fig. 2. Meta-model of the abstract user interface model

be a constraint solving problem that transforms an abstract user interface model into a platform specific user interface model.

This example currently can not be executed using a model-to-model transformation language without investing significant effort.

To implement the transformation with a common model-to-model transformation language, the developer would need to either write a specialized constraint solver using imperative language constructs, which is cumbersome, or use a constraint solver library, pass the constraints to the solver using a special language and call it from the transformation. Obviously, both approaches require efforts, which are not to be underestimated and may be different for each type of transformation being implemented.

3 Constraint Relational Transformations

Depending on the definition of the term “constraints”, different mechanism to reflect and solve the constraints need to be applied in transformation languages. Especially, it must be made clear what type of constraints must be supported. If constraints are restricted to equations, then QVT Relations will classify as a constraint solving approach [8]. While one can arguably call an equation a constraint (e.g. PropertyTemplates in QVT Relations are equations), in fact no constraint solving is performed. Instead, concepts from logic programming are applied. Czarnecki mentions logic programming [8] as the standard way to implement such approaches. Hereby, the main concept is “unification” which is able to cope with variables used in predicates. However, logic programming has been extended to constraint logic programming to support constraints used by constraint solvers (e.g. numeric constraints). In analogy, we extend relational transformations to “constraint relational transformations”.

Constraint relational transformations support local or global constraints, as has been motivated in the last section. Hereby, local constraints can be applied on values of model elements of the target model to enforce relations between

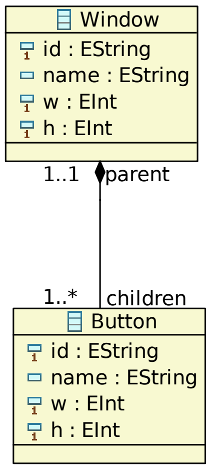

Fig. 3. Meta-model of the platform specific user interface model

them. Additionally, variables may be used to relate source model elements to target model constraints. A set of constraints forms a constraint system, called CSP. This is a constraint program, which needs to be solved by a constraint solver. For example, two model elements representing user interface components are contained in a model element representing a container component. The model elements are related, because they share the same parent space on the screen but may not overlap. Therefore their sizes add to the size of the container component (see section 2.1). This is normally done by using numeric constraints. Satisfaction of these numeric constraints in target models can be established using constraint solving and in the case at hand should be done in the model-to-model transformation engine.

4 Extending the Syntax of QVT and OCL

Defining a completely new transformation language is difficult, especially when it should be declarative. Therefore this section enhances a concrete transformation language to become a constraint relational transformation language. QVT Relations is known to have a very declarative and abstract notation [9]. Despite commonly known problems (e.g. [10]) we chose QVT Relations because of its declarativity, which we believe is a good start to add concepts from constraint programming. However, extending QVT Relations is only an example and the main concepts presented here, can be used to extend other relational model transformation approaches, e.g. Tefkat [2] or MTF [11], as well.

Our QVT Relations dialect is called “Solverational”. To implement Solverational we needed to change the syntax of QVT Relations, as well as the semantics of OCL.

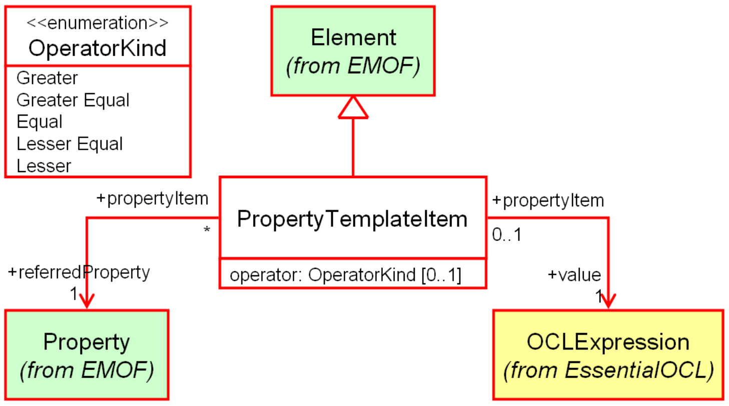

Fig. 4. QVT Relations abstract syntax of PropertyTemplateItem. The PropertyTemplateItem is extended with the OperatorKind attribute to support inequalities.

4.1 QVT Relations

Transformations written in a QVT language are defined over MOF [12] based meta-models. MOF may be seen as to be a small subset of the well known UML, which is usually used to define meta-models. A transformation rule in QVT Relations is called a "relation". Simplified, a relation is a mapping between several sets of model elements, called "domains". Each domain is based on a meta-model class. The domains of the model elements are defined by types from the source and target meta-models.

Every domain may contain "PropertyTemplateItems". If applied to the source domain, the PropertyTemplateItems will select instances from the model. In the target domain they set the properties or associations of the target instance. These template items will use values from other domains, if their variable names are declared within the relation. *PropertyTemplates in traditional QVT Relations cannot contain inequalities or multiple constraints to assign values*, instead equations are used.

We defined a simple extension to the QVT Relations textual concrete syntax [1]: *The equal sign in "propertyTemplate" may be replaced by smaller, greater, smaller or equal, or greater or equal.* The following grammar rule is a replacement for the one given in the QVT Relations definition. This simple extension already has significant implications on the model transformation process. Current model-to-model transformation engines are not able to execute a transformation based on this grammar.

```
<propertyTemplate> ::=
    <identifier> ('='|'<'|'<='|'>='|'>'|'<>') <OclExpressionCS>
```

The complete QVT Relations abstract syntax meta-model can also be found in [1]. In figure 4 the meta-model element PropertyTemplateItem is depicted, a small part of the QVT Relations meta-model. A PropertyTemplateItem has a property (which is actually the left side of the equation in the traditional concrete syntax) and an OCLExpression. To allow for more than equality in PropertyTemplateItems we introduce comparator types (the name ”operator” is being used in the OCL description, so we used it in the model as well).

Due to missing definitions in the original QVT Relations document, the semantics need to be made more precise, only. The QVT Relations specification states that the “property value must match the . . . expression”. As the word “match” is not defined we assume that it is understood as “equality in at least one item of a set”. We refined the meaning of “match” and suggest that a match also occurs in the case the property value is really unequal (“$<$”, “$>$”, “$\leq$”, “$\geq$”, “$\neq$”).

4.2 OCL

Our constraints heavily rely on aggregation functions to evaluate constraints over associations. In OCL aggregation functions (e.g. sum, max, min, product) are used on collections. To support most intuitive usage in the transformation language these aggregation functions were derived from the ones used by OCL but may be used directly in OCL expressions. Because they are not used on collections, their signatures have changed. We implemented only a subset of the OCL, that is most interesting for constraint solving (numeric constraints).

The constraints may have arguments, which are evaluated by identifying appropriate model elements and their values in the model. They may refer to associations to access other model elements. Results from the aggregation functions are not directly computed, but used in the constraint solving process to determine the values of all model elements and their attributes involved in the constraints.

This can best be explained by referring to the Solverational description of the example, which is shown in figures 5 and 1. The function “sum” sums up all “x” attributes of the model elements associated by the “childs” association. However, the value of the sum is not pre-determined. In fact, the transformation will calculate appropriate values for the x attribute by relating the values of the children. We call this form of transformation containing smaller equal constraints “smaller-equal-transformation”.

We also wrote a different version of this transformation with a pre-determined sum. The width “w” of the model element called “Window” is fixed by setting it to $w = 800$. This forces the constraint solver to determine values for the “w” attributes of the child elements. This form is referred to by “equal-transformation”.

4.3 Additional Solverational features

The most obvious strength of Solverational is its ability to use constraints instead of using attribute assignments. Therefore the transformation developer is freed from the cumbersome task of solving the constraints by hand afterwards. In Solverational this can be noted in an intuitive way. The developer simply

```
transformation trafo (a:SemiAbstractUIMetaModel, b:ConcreteUIMetaModel) {

top relation SACopm2CComp  {
    theID : String;
    theName :String;
    theChID : String;
    theChName :String;
    domain a c:SemiAbstractUIMetaModel::ContainerComponent {
        id = theID,
        name = theName,
        children = f:SemiAbstractUIMetaModel::ActionComponent {
            id = theChID,
            name = theChName
        }
    };
    domain b d:ConcreteUIMetaModel::Window {
        id = theID,
        name = theName,
        w = sum(children.w) + 2*(count(children) - 1) + 6,
        w <= 800,
        h = max(children.h) + 13,
        h <= 600,
        children=g:ConcreteUIMetaModel::Button {
            id = theChID,
            name = theChName,
            w <= 60,
            w >= 30,
            h <= 20,
            h >= 10
        }
    };
}
}
```

Fig. 5. Transformation example noted in Solverational

uses inequality comperators to get a solved target model. Every attribute can be subject to a multitude of constraints. Therefore - in contrast to QVT Relations, where this does not make sense (ironically it is not forbidden in the specification), the developer can use the same attribute multiple times in the same rule (the same “ObjectTemplateExp”). Each time he may specify a different constraint on the attribute. The attribute may also take part in a computation which will implicitly enforce a constraint on the attribute value, if the result of the calculation is set by a different constraint. If the developer over-specifies a property, such that the constraints are not solveable at all - the transformation engine will not transform the transformation at all and report failure to do so.

Solverational allows for usage of aggregation functions to sum, multiply or select elements in a set (OCL terminology:“collection”). As Solverational is able to do work on target models and not just copies values, this adds a new form of relationship to transformation processes: attributes of target model elements can be related in the transformation rules. Even though this is not possible with normal programming languages, it is intuitive as it is a completely declarative way to express this class of complex relationships.

5 Implementation

After introducing the concept of constraint relational transformations in section 2 and an extension to QVT Relations, this section presents a prototype implementation. The architecture presented in figure 6 can be devided into two parts. *The upper half of the figure shows the compilation step.* From the meta-models of the source and target models together with the Solverational transformation, an ECLiPSe form is produced (transformation compiler). This program can be used for an arbitrary number of transformations of different model instances (transformation compiler). The form depends on the direction of the transformation and must be regenerated in case the direction changes. Then, *the compiled transformation is executed as depicted in the lower half of the figure.* For the execution the models are read-in and transformed by the ECLiPSe form.

Compilation step. The language for the compiled transformation has been chosen to best support the mapping process from Solverational (this was inspired by the verification from Cabot et al. [13]). As a result from the thoughts given in [14] we chose a logic programming language. This language needs to support constraint solving, as is done by the chosen ECLiPSe[1] PROLOG dialect ([15]). ECLiPSe is a constraint logic programming language, which supports common constraint solvers and search algorithms. It allows for the definition of heuristics and the search order of domain values.

The meta-model and the Solverational description is used to produce the ECLiPSe representation of the Solverational transformation definition. Several challenges need to be addressed in this compilation step.

For example, the execution order of the relations can be seen as a scheduling problem (a Solverational transformation can contain multiple relations). This problem can best be addressed with a constraint solver. Using the constraint solver “Choco” ([16]), the Solverational model transformation engine orders the relations according to “when” and “where” clauses. The result is used to write the relations in a correct execution order into the ECLiPSe PROLOG file. Thus, at runtime, relations will automatically be executed in correct order. However, we never tested this approach with a very large number of transformation rules.

Execution step. The EMF to ECLiPSe mapping transforms EMF models into a term based representation in ECLiPSe. Every meta-model element is identified by a term. The terms of the nodes, attributes, and references have special notations which can be distinguished according to their type. When transforming an EMF model, the mapping component creates identifiers and stores them until the transformation process is completed. Afterwards it can map instances

[1] The ECLiPSe PROLOG dialect is used for constraint programming. The Eclipse IDE is used for developing JAVA applications. We used both and created a plugin for Eclipse to generate ECLiPSe model-transformation-programs.

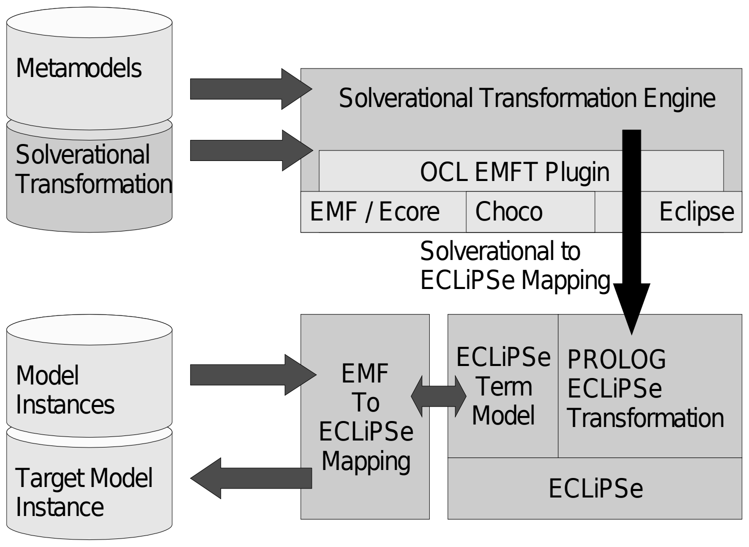

Fig. 6. Architecture of the transformation engine.

of model elements back onto their corresponding EMF representation or create new ones. The transformation of the models to its ECLiPSe form as well as the execution of the transformation in ECLiPSe are performed automatically from JAVA using embedded ECLiPSe programs. During the execution the algorithm given in section 5.1 is executed.

The transformation engine has been realized as an Eclipse plugin, because the Eclipse IDE provides support for many modeling tasks, such as loading and saving models which are used by the engine. The developer can therefore combine the transformation plugin with many of the modeling tools developed by many of the Eclipse projects.

5.1 Algorithm

A generated program that implements a mapping from Solverational to ECLiPSe is a sequence of 6 steps. Their purpose is to collect constraints which need to be satisfied and solve the generated CSP (constraint solving problem). Each CSP variable maps to an attribute in the target model. As soon as the CSP finds a solution, the values of the attributes will be assigned from the CSP variable.

An important aspect is the question at what point a constraint can be inserted in to the CSP. If associations are used in a constraint it can only evaluated after all relevant model elements have been created. Therefore, our generation is composed of 6 steps:

1. First, target model elements are being created or searched for.

2. When a model element is created or found, the algorithm creates a variable for each attribute of the model element on the heap, so they can be resolved during later steps.
3. If there are constraints for the attributes of the model element they are inserted directly into the CSP.
4. After all model elements have been created, constraints for attribute values spanning associations are inserted.
5. Then, all variables of the attributes are retrieved from the heap and inserted into the CSP.
6. Finally, the CSP is going to be solved by executing the constraint solver and results are calculated.

Our algorithm can also cope with the selection of a specific target model out of the set of possible target models in an intuitive way by defining a target function over the set of model elements. However, this is a whole work on its own and due to space constraints we must restrict ourselfs to the contribution of constraint programming in model transformations.

6 Discussion

Our example shows that developers can benefit from the declarative style of constraint programming. We assume that the size of the effect is domain dependent. Nevertheless, as this seems to be the case for many domains [7], we conclude that constraint programming may be a valuable asset to model transformation.

Implementing even the very simple example using the plain QVT Relations language, instead of Solverational, is very difficult. The only option would be to specify values by hand after the transformation has been performed. However, this can only be done for a small set of model elements and it has to be done every time the transformation is executed on a different set of input models. This is cumbersome and a clear indication that this process should be automated. The concepts presented in this paper allow for automating transformations using local constraints.

The implementation shows that it is feasible to do constraint solving on smaller models. To determine the size of models that can be handled efficiently, we evaluated three transformations on models with increasing numbers of model elements. As explained in section 4.2, the three Solverational transformations use different constraints. The “smaller-equal-transformation” constraints the maximum width “w” of “Window” model elements, while the “equal-transformation” sets the size to a specific value and enforces “w” of children to be calculated by the constraint solver. Of course, we had to alter the maximum size constraint for “w” of “Window”, such that it still fits the sum of model elements of the “w” of the “children”, depending on the number of model elements we wanted to transform.

Additionally, we tried to run the transformation without any constraints (because plain QVT is not able to solve constraints) with an open source QVT Relations transformation engine (“QVT medini” [17]). We expected it to be

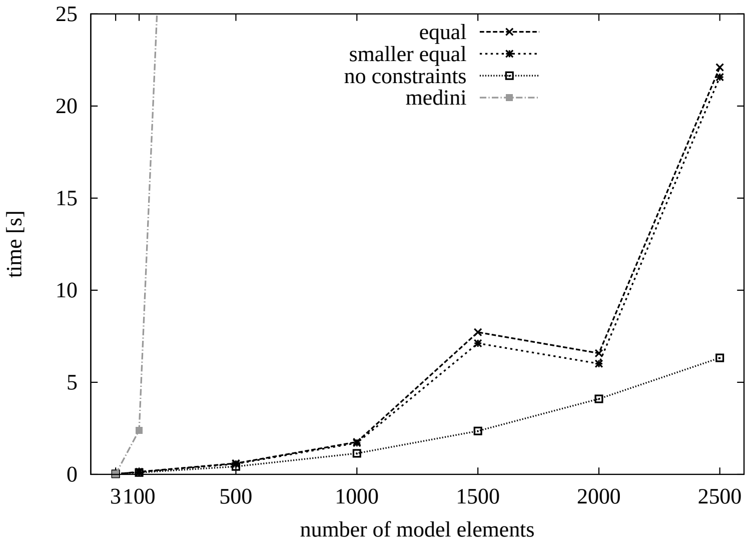

Fig. 7. Number of model elements versus time in seconds for different transformations

much faster, as it does not use constraint solving and is an industrial product. It is noteworthy that Solverational is a QVT Relations dialect and therefore is compatible with a large subset of QVT Relations transformations.

We measured the CPU time for the different transformations. We did not add the time needed to save and load the models, but only the time needed to perform the transformation. All results were taken on a computer with a 1.8GHz Intel CPU and 1GB RAM, which was not exceeded during the test runs. Reperforming several transformations on other CPUs indicate that the time needed to perform the transformations is proportional to the speed of the CPU. The results are illustrated in figure 7.

Execution times of the Solverational transformation engine increased as expected, with a slight decrease of calculation time by enforcing constraints for 2000 model elements. The constraints which were more difficult to solve took more CPU time. The "equal-transformation" took longest to compute, because the constraint solver needed to set the values of the "children" model elements. Although, time increased only slightly compared to the one needed for the "smaller-equal-transformation".

Both transformation engines seem to have sub-exponential execution behaviour, but to our great surprise our transformation engine seems to be comparatively fast. Note, that in the case of 2500 model elements, the constraint solver needs to solve more than 10000 constraints with more than 5000 variables. But still the medini QVT engine had considerable larger execution times.

These results indicate that using constraint solving in model transformations is not the time-killing factor. The medini QVT engine implements other features, which seem to be far more time consuming than constraint solving in the model transformation process. However, our examples used very simple constraints and solving time is highly dependent on them. Using e.g. non-linear constraints can arguably produce completely different results. However, we argued in section 1 that many modeling constraints are similar to the ones we were using.

7 Related Work

There are two categories of related work: work on model or graph transformation and work on models and constraint solving.

Ehrig et al. outline that constraint solving is a standard way to optimize pattern matching to search for model elements in model transformations [18]. This has been used in Attributed Graph Grammars (AGG) [19]. However, searching differs from our constraint solving for target models, because our approach enforces constraints in target models.

In different work Ehrig provides a graph grammar based theory which shows how graph constraints can be transformed into application conditions of transformation rules [20]. Then the application conditions support the consistency of graphs involved. The graph transformation rules may not be fired when an application condition (and therefore a graph constraint) is violated. This is not equal to constraint solving, where the solver enforces attribute values.

El-Boussaidi and Mili present an approach that is able to search for model patterns [21]. It is based on the ILOG JSlover constraint solver library and searches for patterns by solving a constraint solving problem over graphs, similar to the approach mentioned above [19].

A proposal called "xMOF" [22] to the OMG RFP for the QVT [23] transformation language has not been accepted as such, but the approach presented uses constraints within the definition of the model transformation. The approach recommends to use a minimalistic extension to OCL [4] to develop model transformations based on meta-models written in MOF [12]. The proposal has not been implemented, so it does not use constraint solving (OptimalJ, for which xMOF had been designed, implements QVT Core instead). From our experiences with our current implementation of the transformation language we suspect that an efficient implementation of the proposal would be rather difficult.

Lengyel et al. allow for the definition of transformation rules and the attachment of constraints [24]. They define "constraint validation", "constraint preservation" and "constraint guarantee" for model transformations. However, the OCL constraints are not enforced by their implementation and therefore does not use constraint solving. As they do not use backtracking the approach cannot produce all possible solutions. Furthermore, it uses XSLT which cannot cope with constraint satisfaction problems.

White et al. focus on using constraint solving to weave models [25]. Model weaving is similar to model transformation in that it is able to weave (or transform) several models into a single model. However, this model must then be transformed by model transformation to be in accordance with the output target platform meta-model, which is a concrete weaving model. However, White et al. do not provide a transformation language that performs constraint solving, but a model weaver which is integrated in the GME environment. Therefore, the constraints of the target platforms would need to be integrated into the constraints of the source model or the model weaver, which would not be very useful, since they should be abstract to produce an abstract weaving solution. Our model transformation language can be applied to complement the solution in the transformation process.

Constraints have also been used to transform models of user interfaces. SUPPLE [26] uses constraints to define the properties of target platforms. The transformation does not transform the user interface descriptions into models, but the user interface is directly displayed after the constraints are solved. Furthermore SUPPLE does not use a model-to-model transformation language to define the mapping between model elements and the user interface. Therefore, the model can not be modified by the developer, before it is being displayed.

MASTERMIND uses declarative models to construct software based on generative constraints to enforce dependencies between user interface components [27, 28]. Starting from the MASTERMIND textual format, model-to-code transformations can be executed, which take into account layout constraints given in a presentation model. MASTERMIND does not use a model-to-model transformation language and therefore is not able to handle constraints in model-to-model transformations.

In a limited scope our approach is also well suited for round-trip engineering as it is able to produce a set of possible source models when it is used as a bidirectional transformation. A definition of round trip transformations and how they relate to the problem of multiple possible source models is given in [29]. In that sense our approach is therefore similar to the one presented in [30], which also uses a logic programming language to implement the generation of multiple source models. However, while Cicchetti's approach is used mainly for round trip transformations and does not use constraint solving but answer set programming to produce possible target models, our approach focuses on the transformation of consraints on attribute values.

8 Conclusions and Future Work

In this paper we presented how constraint solving and model transformations are integrated to constraint relational transformations. Constraint relational transformations allow for the first time to solve local numeric constraints during model-to-model transformations. It was shown that constraints integrate, almost naturally, in QVT Relations transformations. The new dialect is called Solverational.

We presented our implementation of a model transformation engine and showed how a model transformation engine can be extended to support constraint relational transformations. We showed that our concept, when applied, is not the main factor for transformation times.

Our goal is to use the transformation language on user interface models. This will allow for optimization of the user interface models during model-to-model transformation. This work will be applied in the domain of crisis management within the SoKNOS project.

Further, it is planned to work on the integration on graph constraints that can be used for planning and solve problems known from operations research. This will open up more complex application scenarios to model-to-model transformations suggesting constraint solving, e.g. planning and scheduling.

Acknowledgements The work presented was partly funded by the SoKNOS project. The funding is provided by the German Ministry of Education and Research (BMBF).

References

1. OMG: Meta object facility (mof) 2.0 query/view/transformation specification. OMG (July 2007) ptc/07-07-07.
2. Lawley, M., Raymond, K.: Implementing a practical declarative logic-based model transformation engine. In: SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, New York, NY, USA, ACM (2007) 971–977
3. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Object Technology Series. Addison-Wesley, Reading/MA (August 2003)
4. OMG: Object constraint language omg available specification version 2.0. OMG (May 2006)
5. Freuder, E.C.: In pursuit of the holy grail. Constraints **2**(1) (1997) 57 –61
6. Brucker, A.D., Doser, J., Wolff, B.: Semantic issues of OCL: Past, present, and future. Electronic Communications of the EASST **5** (2006) 213–228
7. Ratschan, S.: Applications of quantified constraint solving over the reals. Internet (January 2008) http://www2.cs.cas.cz/ ratschan/appqcs.html, visited 01/09.
8. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3) (2006) 621–646
9. Jouault, F., Kurtev, I.: On the architectural alignment of atl and qvt. In: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, New York, NY, USA, ACM Press (2006) 1188–1195
10. Stevens, P.: Bidirectional model transformations in qvt: Semantic issues and open questions. In: Model Driven Engineering Languages and Systems. Volume 4735/2007., Springer Berlin / Heidelberg (2007) 1 –15
11. IBM United Kingdom Laboratories Ltd., I.a.: Model transformation framework (mtf). IBM alphaWorks (2004) http://www.alphaworks.ibm.com/tech/mtf.
12. OMG: Meta object facility 2.0 core final adopted specification. OMG (October 2003)
13. Cabot, J., Clariso, R., Riera, D.: Verification of uml/ocl class diagrams using constraint programming. In: Model Driven Engineering, Verification, And Validation: Integrating Verification And Validation in MDE (MoDeVVA 2008). (2008)

14. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of mda. In: ICGT '02: Proceedings of the First International Conference on Graph Transformation, London, UK, Springer-Verlag (2002) 90–105
15. Apt, K.R., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge University Press, New York, NY, USA (2007)
16. Jussien, N., Rochart, G., Lorca, X.: The choco constraint programming solver. In: CPAIOR'08 workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08), Paris, France (June 2008)
17. ikv++ technologies AG: Qvt medini. Internet http://www.ikv.de/ikv_movies/mediniQVT.swf.
18. Ehrig, K., Taentzer, G., Varro, D.: Tool integration by model transformations based on the eclipse modeling framework. Technical report, EASST Newsletter (2006)
19. Rudolf, M.: Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: Theory and Application of Graph Transformations. Volume 1764/2000 of Lecture Notes in Computer Science. Springer (2000) 381 –394
20. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Constraints and application conditions: From graphs to high-level structures. In: ICGT. Volume 3256., Springer (2004) 287–303
21. El-Boussaidi, G., Mili, H.: Detecting patterns of poor design solutions using constraint propagation. In: MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, Berlin, Heidelberg, Springer-Verlag (2008) 189–203
22. Compuware-Corporation, SUN-Microsystems: Xmof queries, views and transformations on models using mof, ocl and patterns. OMG (August 2003) OMG Document ad/2003-08-07.
23. OMG: Mof 2.0 query / views / transformations rfp (April 2004)
24. Lengyel, L., Levendovszky, T., Charaf, H.: Constraint Validation Support in Visual Model Transformation Systems. Acta Cybernetica **17**(2) (2005) 339–357
25. White, J., Gray, J., Schmidt, D.C.: Constraint-based model weaving. Transactions on Aspect-Oriented Software Development (2009) to appear.
26. Gajos, K., Weld, D.S.: Supple: automatically generating user interfaces. In: IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces, New York, NY, USA, ACM Press (2004) 93–100
27. Palanque, P., Paterno, F.: Formal Methods in Human-Computer Interaction. Springer, Berlin (1998) ISBN 978-3540761587.
28. Browne, T., Davila, D., Rugaber, S., Stirewalt, R.E.K.: The mastermind user interface generation project. Technical report, Georgia Institute of Technology (1996)
29. Hettel, T., Lawley, M., Raymond, K.: Theory and practice of model transformations. In: Theory and Practice of Model Transformations, ICMT2008. Volume 5063/2008 of Lecture Notes in Computer Science., Springer (2008) 31 –45
30. Cicchetti, A., Di Ruscio, D., Eramo, R.: Towards propagation of changes by model approximations. In: EDOCW '06: Proceedings of the 10th IEEE on International Enterprise Distributed Object Computing Conference Workshops, Washington, DC, USA, IEEE Computer Society (2006) 24