Towards an Alignment of Declarative Modeling and Model-to-Model Transformation Languages

Andreas Petter, Alexander Behring,

{a_petter,behring}@tk.informatik.tu-darmstadt.de

Abstract: Declarative modeling and model driven software engineering seem to be two fields of research with completely different focus. However, the term “modeling” is used by both communities and both communities claim to use “declarative” techniques. In this paper we give a small overview on some of the differences of both fields and present our model-to-model transformation language which brings together constraint solving, optimization, and model-to-model transformation. The language can be seen as an example how intuitively both communities can integrate to explore new applications.

1 Introduction

In the declarative modeling community the term model describes a set of facts about the domain of possible solutions. In model driven software engineering models are used to create software from them. However, most software engineering models are based on graphs, while most declarative models are based on variables and constraints. Model-to-model transformations transform software engineering models. Similar to declarative modeling techniques they may be described using declarative model-to-model transformation languages. When it comes to finding similarities between both communities, these languages therefore classify as interesting candidates. Our model-to-model transformation language, called “Solverational” can be used to define constraint problems in model-to-model transformations. In fact our model transformation engine transforms model-to-model transformations written in “Solverational” into constraint solving problems (see Section 4). These similarities can be used to explore the theory behind both approaches, to use similar mathematical tools to compare the complexity and cardinality of model-to-model transformation languages and declarative modeling languages. We demonstrate how our language maps to a CSP and believe that this approach could be used to compare many other declarative model transformation languages with delcarative modeling languages as well.

The contributions in this paper are:

- Examining the differences of models used for model-to-model transformation and declarative models
- Presenting model-to-model transformation as a use case for declarative modeling

- Mapping Solverational to a CSP as an example for a larger set of graph grammars

1.1 Terminology

[Sta73] gives an introduction into the term "model" in general, while [Lud03] specializes the term for models used in software engineering. We use the term as proposed by them by referring to *software engineering model*, but also use "model" for *constraint based models*. Software engineering models are graphs. The nodes are called model elements and edges are called associations. Model elements adhere to a specification, called a class, and are also called objects. They may have attributes. One major difference is that models from model-driven software engineering better describe static aspects while constraint programs (declarative models) describe the outputs of algorithms. This is similar to declarative model-to-model transformations, which also describe algorithms. Therefore, in the following, we focus on model-to-model transformation.

In fact, the term "declarative" is used very similar in both communities. "Declarative" means that developers do not specify how s.th. is executed but facts about the result. However, in the model-driven software engineering community there are many cases where visual workflow languages are called declarative even though the language itself is more in form of a procedural language (e.g. Microsoft calls the Windows Workflow Foundation language a declarative language [BS06]). We conclude that the term declarative may be used for workflows as well, as long as the tasks in the workflow are of very abstract nature.

1.2 Paper Outline

The next Section introduces the state of the art of model-to-model transformation which in some way makes use of constraint solving. Section 3 gives a small introduction into the background of model-to-model transformation. The following three Sections demonstrate how to map model transformations to CSPs and present the implementation of our language called "Solverational". Section 5 presents the differences of our approach to a classical well-known declarative modeling language. Section 6 presents an example to provide an intuitive way of understanding Solverational. The last Section summarizes the paper and provides opportunities for further research.

2 Related Work

Our work on mapping model-to-model transformations to declarative modeling is related to a diversity of works in model driven software engineering.

Software engineering models often are enhanced by constraints using OCL [OMG06]. These constraints may restrict almost any aspect of the models for which they have been specified. Cabot et al. explain in [CCR08] how UML class diagrams having constraints can be checked by transforming the models into a CSP. It is therefore related to our work as it also maps constraints used in the software engineering community into a CSP. However, this work does not transform models and therefore cannot be seen as a declarative description of model-to-model transformation and addresses the validation of models rather than their transformation.

Merging several models into one model is called model weaving, especially in the case of the adoption of aspect oriented programming for MDSD. If constraints are used in several target models, the model merging process needs to satisfy all constraints in the models. Therefore, White et al. ([WGS09]) use mappings to CSPs to solve the constraints during the weaving process automatically. However, the work of White is not a generic model-to-model transformation language.

Rudolf [Rud00] uses constraint solving to search for patterns in graphs. This concept is applied to graph transformations using Attributed Graph Grammars. The technique is explained in Section 3 in more detail. Based on this approach, El-Boussaidi and Mili present an approach that is able to search for model patterns [EBM08]. This work does not solve constraints on target model elements, as is natural to our approach. Still this work presents how to map the process of pattern matching to CSPs, which can be used as a basis for mapping the complete process of model-to-model transformation to CSPs - c.f. Section 3.

Kessentini et al. treat the generation of a model-to-model transformation program as an optimization problem [KSB08]. They use particle swarm optimization to transform models. Thereby, a model-to-model transformation is being constructed by providing an example, and not by a declarative model-to-model transformation language. Thereby, we take a different approach to ease the declaration of model-to-model transformation.

Chenouard et al. present an approach which models different declarative constraint modeling languages using meta-modeling from MDSD [CGS08]. A CSP then is an instance of the meta-model and therefore expressed in a visual language. These models can then be transformed into a different declarative programming language using standard model-to-model transformation tools (ATL). However, the approach transforms declarative models into different declarative models and therefore does not solve a CSP but enables a form of solver independence. Our model-to-model transformation with constraints therefore is only related concerning the use of models in connection with declarative modelling languages.

3 Theoretic Background

Every graph transformation is composed of several transformation rules, each having a right hand side (RHS) and a left hand side (LHS). The LHS is used to search for a subgraph in the source graph, while the RHS is used to search and enforce target elements which are again nodes or edges of a target graph. Model-to-model transformations are a specific class of graph grammars, which perform transformations over software engineering models. A small description of QVT Relations and Solverational can be found in Section 4, but for better readability an example of LHS and RHS can be found in Figure 3. Each “domain” represents one side of the rule. If the rule is executed from “abstractuimodel” to “concreteuimodel”, then the domain connected to “a” (i.e. it matches patterns in source model a) will be the LHS and the domain connected to “b” will be the RHS (i.e. it performs the rule action on target model b).

Rudolf demonstrated in [Rud00] that searching (“pattern matching”) for model elements can be implemented using constraint solving problems. Each model element from the subgraph that is being searched for, i.e. each graph node, and each association (graph edge) is mapped to a CSP variable. The domains are elements and edges from the source graph. Constraints represent the edges and their direction. When the problem has been solved, the variables are instantiated to values (from the source graph) representing appropriate model elements and edges.

Our language called “Solverational”, which is presented in Section 4, maps the RHS of model-to-model transformation to non-linear CSPs. Attributes of objects of model elements are mapped to CSP variables, while the allowed classes of the objects will be mapped to variables, if there are a multitude of different objects allowed.

To show how model transformations and constraint solving relate to each other, a complete mapping from Solverational to a CSP is needed. However, currently it does not use constraint solving for pattern matching, but is using simpler search methods which are not implemented using a declarative programming approach. It is heavily based on concepts from the QVT Relations standard [OMG07], but enhances it with constraint solving and optimization. Therefore it is able to map attributes of target model elements to variables in a CSP.

3.1 Mapping Graph Transformations to CSPs

Solverational has been implemented by providing a mapping to constraint solving and constraint optimization problems, but only concerning the target model (RHS).

By using the ideas from [Rud00] pattern matching can be expressed as a constraint solving problem. Therefore, the pattern matching algorithm of Solverational could also be expressed as a CSP. By refferring to this (currently unimplemented) abstract concepts, both the LHS and the RHS are based on constraint solving. This means that they can both be expressed as a CSP. Pattern matching is done by the first CSP, enforcement is done by the second CSP from our language (which is implemented).

In model-to-model transformation languages LHS and RHS can be connected using variables (from the language), which contain information from the source model. This is also a missing link between the two CSPs which also needs to be expressed in CSP terms. Initial investigation results indicate that this could be expressed using “and” between the constraints (usually implemented as reified constraints), such that the LHS constraints and the RHS constraints match, which would normally be used in the same transformation rule.

We believe that many other model-to-model transformation languages can be seen as special cases of this concept. The first CSP could be used for most model-to-model transformation languages, as it is a generic pattern matching algorithm for model-to-model transformation languages. The second is not needed for most languages, because the vast majority uses simple assignments based on equality. However, equality can be seen as a constraint as well (although the most basic one). Therefore, most declarative relational transformation languages can also be mapped to a CSP with equality constraints.

Note that this is not really valid for in-place transformations, which need the process to be re-executed, which is not handled by our model-to-model transformation engine. However, since each possible transformation may be seen as a transitive application of different transformation rules, the concepts may still be valid, but we did not investigate this matter.

4 Model-to-Model Transformation using Constraint Solving

In the following we present our implementation, a language called Solverational. It has been implemented in a transformation engine which maps the process of using constraints on attributes of target model elements to a CSP.

Our model-to-model transformation language "Solverational" is based on the QVT Relation language [OMG07] which is part of a family of standardized model-to-model transformation languages. The standard is composed of an imperative language (QVT Operational), a low-level declarative language (QVT Core) and a high-level declarative language - QVT Relations. We chose QVT Relations as the basis for our language as it is already known to be rather declarative. This declarativity results from a relational programming approach to model transformation. However, instead of the logic programming based approach which is commonly used to implement declarative model-to-model transformation engines, it maps the model-to-model transformation to a CSP. As our language extends the current language with constraints it introduces an even greater level of declarativity into model-to-model transformation.

Other model-to-model transformation approaches (e.g. [LLC05]) also try to incorporate constraints in the transformation process, but the process is not able to actually solve the constraints.

Each QVT Relations transformation is composed of several transformation rules, called "relations". A relation is a mapping from a domain of source model elements (also called domain in QVT) and associations (called "ObjectTemplateExpressions") to another domain which also consists of model elements and associations. Each domain may have several "PropertyTemplateItems", which selects and sets properties of model elements by using equations. PropertyTemplateItems consist of a property name, an equality sign and an expression. Variables may be used to store attribute values from the source domain in property values of the target domain by using them in expressions.

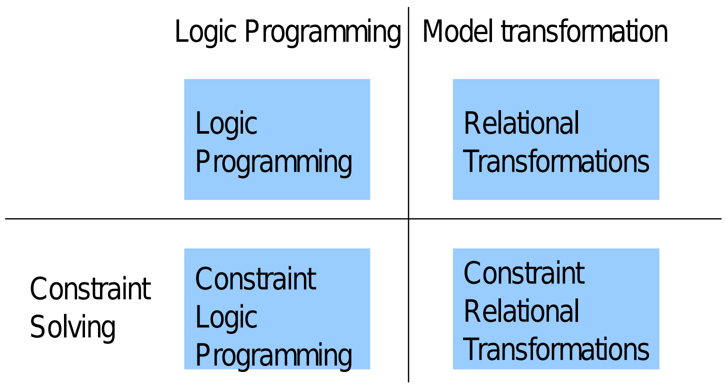
Logic Programming
Model transformation
Logic Programming
Relational Transformations
Constraint Solving
Constraint Logic Programming
Constraint Relational Transformations

Figure 1: Comparison of language-types.

4.1 Solverational and Constraint Relational Transformations

In the case of our language, Solverational, it is possible to use inequalities instead of equalities in PropertyTemplateItems [PBM09]. The equality sign therefore can be replaced by smaller and greater (smaller equal, greater equal) signs.

Therefore, model-to-model transformation is a constraint satisfaction problem and not only a logic programming approach any more. We were surprised how intuitive this concept is, as it just requires a simple change: allowing inequalities instead of equalities, only. We call this type of model-to-model transformation languages "Constraint Relational Transformations" [PBM09]. Figure 1 shows the relation of relational model-to-model transformations to constraint relational model-to-model transformations. This relation seems to be very similar to the one encountered when logic programming is enhanced to constraint logic programming.

Our language is implemented as a plugin for Eclipse (the Eclipse JAVA IDE) using the ECLiPSe system (Eclipse Constraint Logic Programming System - a Prolog based constraint logic programming system based on many improvements to the CHIP solver, Figure 2). The transformation engine maps the input model-to-model program written in Solverational as well as the meta-models to an ECLiPSe program. This program can be fed with instances of these meta-models (to be more specific: the input meta-models), the concrete input models. It transforms these input models into the target model by executing an algorithm also presented in [PBM09], which basically constructs a CSP and executes it.

The model-to-model transformation engine can also use an optimization function to select a possible target model from the space of all target models. This optimization function will choose the optimal target model according to an expression given in OCL in the Solverational transformation description. This shows that model-to-model transformation

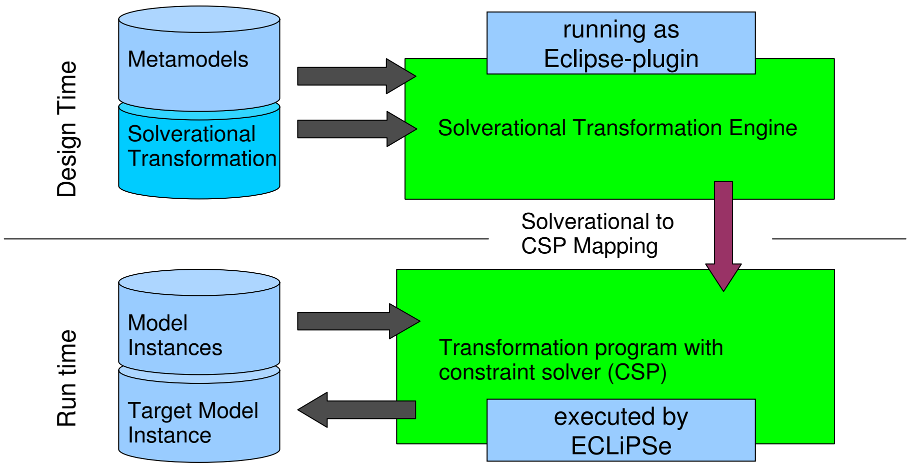
Design Time
Metamodels
Solverational Transformation
running as Eclipse-plugin
Solverational Transformation Engine
Solverational to CSP Mapping
Run time
Model Instances
Target Model Instance
Transformation program with constraint solver (CSP)
executed by ECLiPSe

Figure 2: General outline of the architecture of the model transformation engine.

problems can also generate constraint optimization problems, which are also important for the declarative modeling community.

5 Differences to Classical Declarative modeling Languages

Classical declarative modeling languages, like OPL [VH99], are working on variables. Variables may be used in constraints, which relate them. To start with a good example involving constraints, we originally tried to implement model-to-model transformations for user interfaces directly in OPL. However, the nature of OPL allows only to work on instances of model elements. This means that each attribute of a model element needs to be explicitly integrated into the OPL program by explicitly naming a new variable. Therefore OPL programs need to be changed when introducing new instances of model elements.

Our language enables a way to declaratively control the constraint problem generation. In fact the language does not allow to explicitly add constraints for each single model element, but requires constraints to be defined over meta-model elements. This is called a “model type mapping” approach to model transformation, in contrast to a “model instance mapping” approach. The original OPL based way can in some way be seen as a “model instance mapping” approach, because every instance of an attribute needs explicit markings (i.e. an explicit variable).

Additionally, the most obvious difference is that nodes of graphs need to be mapped to variables of a constraint problem, which is not the case for Solverational, as it is able to work directly on types of nodes. This makes handling graphical modeling languages less cumbersome.

5.1 Advantages of Implementing Engines Using CSPs

There are several advantages, if constraint relational transformation languages are mapped to CSPs.

Using these mappings can save development efforts and eases the implementation, especially, if the developers are not familiar with implementing constraint solvers. In that case, the developers of the transformation engine are able to use the capabilities of solvers, which are available on the market. Modern constraint solvers are able to perform numerous optimizations, which are hard to implement, especially when it comes to sophisticated techniques to perform constraint propagation. This is similar to the investigation for pattern matching made by Rudolf ([Rud00]).

Another advantage adresses debugging. While it is hard to develop debuggers and debug declarative transformation programs it is possible to debug constraint logic programs - at least the CSP generation part. A debugger exists for these while development effort needs to be invested for a specific one for the constraint relational transformation language.

Furthermore, changing the transformation under certain aspects can be easier when editing the constraint logic code. The model transformation language is fixed to a certain set of constraints which will probably be a bit less powerful than the constraint solver. The reason is, that the approach to generate the CSPs via the transformation language is not able to generate every possible constraint, but only these that are handled in the language (which can only use constraints handled by the constraint solving algorithm). But it is possible to change the generated constraint logic program to implement the constraint. Although this sounds cumbersome, our expierences show that this is possible and is done often, especially when the transformation engine is being improved by further development.

It is also important to note, that developers can start the development process of the transformation engine with a simple transformation from the constraint relational transformation language to the constraint logic program. This helps in understanding how the mapping and the constraints should look like and when options to follow are not clear (because every documention of constraint solvers is limited in some way) developers are able to test the constraint logic program before implementing the option in the transformation engine.

6 Example

To demonstrate how constraints are used in the model-to-model transformation language and how they are reflected in ECLiPSe and therefore in the CSP, we present an example derived from model driven development of user interfaces.

Abstract user interface models are transformed - which abstract from device specific prop-

```
top relation Container2ConcreteContainers {

        domain a i:abstractsoknosuimodel::Container {
                children = d:abstractsoknosuimodel::Interactor { }
        };

        domain c o:concretesoknosuimodel::Panel {
                height >= sum(children.height),
                width >= max(children.width),
                children = e:concretesoknosuimodel::Component {
                   x = parent.x,
                   y >= parent.y,
                   y <= parent.y + parent.height,
                   y < nextSibling.y - height,
                   width < parent.width,
                   height < parent.height
                }
        };

}
```

Figure 3: Example of a transformation rule with constraints.

erties - into concrete user interface models. In the example, a model for a graphical user interface is being created by the transformation. These models in turn can be transformed by a model to code transformation into code for graphical user interfaces or interpreted by a model interpreter (e.g. [BPM09]). Due to space constraints we can not present the complete transformation, but only a single transformation rule (Figure 3), because the complete transformation is composed of a large set of rules and its corresponding constraint logic program is over a thousand lines of code long. Of course, the transformation language (and the engine) is able to transform models based on various meta-models and is not fixed to user interface models, but still there is a long tradition in developing user interfaces based on CSPs ([Sut64]) and also on developing user interfaces using model driven engineering ([BDRS96]). Therefore, our transformation language can be seen as a solution to combine both ways to develop user interfaces.

In our example, we call model elements in the abstract model “Interactors” while their more concrete counterparts are called “Components”. The transformation rule presented will therefore produce Components from Interactors. The Interactors are contained in Containers, which are called Panels in the concrete case, thereby Panels will be produced from Containers. While Interactors do not have sizes and positions, because they are not needed in abstract models, Components do have a position ("x" and "y") as well as a height and width in pixels. In Figure 3 (a simplified example) two domains are presented, which reflect the Containers and Panels. They refer to their "children" by an association, which are either Interactors or Components, respectively.

The transformation shall check that the user interface components do not overlap and that they are contained within their parent component, the Panel. This is done by introducing several constraints. The first constraint requests that the height of the Panel is equal to the sum of the heights of the children components while the second requests the width to be greater or equal to the maximum of the widths of the children:

```
...
height = sum(children.height),
width >= max(children.width),
...
```

There are two different types of constraints on attributes of model elements:

- "local" constraints, which only affect attributes on the model element being processed
- "association" constraints, which affect attributes of model elements which are not the model elements being processed

Constraints on model elements, which are only using constraints which are "local" to the model element currently being processed can directly be inserted into the CSP. This is done straight forward by transforming the constraint directly into ECLiPSe-code.

Constraints over associations (or references to other model elements) cannot immediatly be inserted into the CSP. E.g. children.height must be resolved before the whole CSP can be generated, because it is not known how many associated Components will be generated in the process or if any more associations will be set by different relations. Because all model elements in the target model must be generated, first, these associations require delayed execution. According to "delayed goals" from constraint logic programming we called these goals "semi-delayed goals". These semi-delayed goals need to be stored. The following ECLiPSe-code shows how these constraints are being reflected in the CSP (the example shows the constraint on height from above):

```
...
get_var("height" ... panelattribute ... VarValue0),
make_ref_var(... c_Panel_0 ... VarValue1_sum0),
C12=(VarValue0#=VarValue1_sum0),
...
```

```
c_Panel_0(...) :-
     ...
     C1=(VarValue#=sum(VarValues)),
     ...
```

These “semi-delayed goals” are being added to a heap and executed right before the constraint solver. In our example we store the reference to the term “c_Panel_0” and the constraint on it and execute it later. The term $make\_ref\_var$ stores the semi-delayed goal on the heap and produces a reference to a variable which can then be used within a constraint ($VarValue1\_sum0$). The constraint is a simple equality constraint. As can be seen in the code it is also stored for later execution, but the variable reference which has been produced by $make\_ref\_var$ survives storing on the heap and therefore the variable can be used in the constraint solving process afterwards. This reference is exactly the same as $VarValue$ in the c_Panel_0 term. Therefore, the delayed execution of the semi-delayed goal binds the sum constraint to the reference. In ECLiPSe this enforces the variable to be part of the CSP and therefore the constraints get connected by their variable reference. At the point where c_Panel_0 gets executed, the structure of the graph is fixed, already (directly before the solver gets executed). Therefore, the associations can be resolved completely and the parameters for sum (attribute“height” of all children elements) can be replaced by variables from the heap.

As mentioned in the previous Section, the constraints are attached to each single model element and therefore the same number of constraints are being inserted into the CSP for each model element of the same type. The whole CSP is composed of all constraints of all model elements and therefore the complexity of the CSP depends on the size of the input model, the number of constraints on each type of input model (the number of constraints in the transformation definition) and the number and complexity of the transformation rules.

The execution of the CSP is done by a standard constraint solving algorithm. Most constraint solvers commonly used can be employed. Because we are mapping to ECLiPSe it was straight forward for us to use the ic library shipped with ECLiPSe, which is a highly optimized version of the CHIP solver.

It is obvious that the generated Panels will be Panels where all the Components are aligned vertically. This example is simplified as Solverational is also able to automatically choose between alternative domains. In our case this alternative could be two different Panel: one for vertical and one for horizontal alignment. As optimization functions can be defined in Solverational one can use an optimization function to compute the best possible layout (e.g. use an objective function from HCI papers like [GW04] or [Sea93]).

6.1 Results from the example

As can be derived from the example the mapping of Solverational constraints to constraints of the CSP is done on a per constraint basis. During this mapping process constraints will either be directly mapped to CSP constraints, if the constraints are “local” to the current model element or they need to be delayed and executed afterwards. This is done by introducing “semi-delayed goals” which apply the constraints after the generation of the model elements.

In general, this concept can be applied to many relational transformation approaches. The major benefit of using CSPs to implement model-to-model transformations seems to be the implementation, which can be done much faster than implementing new solvers. However, our implementation can only cope with Solverational transformations. It remains an interesting research question which type of transformation approaches can effectively benefit from the approach.

7 Conclusions and Future Work

By providing a mapping from a model-to-model transformation language to a constraint solving problem we show that both fields will be closely related, if model-to-model transformation languages and engines implement constraints and constraint solvers. We demonstrated how it is possible to map transformation rules (LHS and RHS) to constraint solving problems in a generic way that can be applied to the QVT Relations language. It is likely that this approach can be used for many declarative model-to-model transformation languages.

Our language and our example show that model-to-model transformations are an interesting case study for constraint solving problems. Also, for implementing model-to-model transformations with constraints our approach offers opportunities for faster implementation.

In the near future we will further increase the functionality of our prototype and explore the practical applications offered by our approach. Our research is focused on model-to-model transformation of user interface models and we will apply it within the SoKNOS project, which is a project on crisis management systems funded by the German Ministry of Education and Research (BMBF).

References

[BDRS96] Thomas Browne, David Davila, Spencer Rugaber, and R. E. Kurt Stirewalt. The MASTERMIND User Interface Generation Project. Technical report, Georgia Institute of Technology, 1996.

[BPM09] Alexander Behring, Andreas Petter, and Max Mühlhäuser. Rapidly Modifying Multiple User Interfaces of one Application. In *ICSOFT (SE)*. INSTICC Press, 2009. to appear.

[BS06] Don Box and Dharma Shukla. Simplify Development With The Declarative Model Of Windows Workflow Foundation. Internet, January 2006. http://msdn.microsoft.com/en-us/magazine/cc163661.aspx.

[CCR08] Jordi Cabot, Robert Clarisó, and Daniel Riera. 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.

[CGS08] Raphaël Chenouard, Laurent Granvilliers, and Ricardo Soto. Model-driven constraint programming. In *PPDP '08: Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 236–246, New York, NY, USA, 2008. ACM.

[EBM08] Ghizlane El-Boussaidi and Hafedh Mili. 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*, pages 189–203, Berlin, Heidelberg, 2008. Springer-Verlag.

[GW04] Krzysztof Gajos and Daniel S. Weld. SUPPLE: automatically generating user interfaces. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, pages 93–100, New York, NY, USA, 2004. ACM Press.

[KSB08] Marouane Kessentini, Houari A. Sahraoui, and Mounir Boukadoum. Model Transformation as an Optimization Problem. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Proceedings of Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008*, volume 5301 of *Lecture Notes in Computer Science*, pages 159–173. Springer, September 2008.

[LLC05] László Lengyel, Tihamér Levendovszky, and Hassan Charaf. Constraint Validation Support in Visual Model Transformation Systems. *Acta Cybernetica*, 17(2):339–357, 2005.

[Lud03] Jochen Ludewig. Models in software engineering - an introduction. *Models in software engineering - an introduction*, 2:5–14, 2003.

[OMG06] OMG. Object Constraint Language OMG Available Specification Version 2.0. OMG, May 2006.

[OMG07] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. OMG, July 2007. ptc/07-07-07.

[PBM09] Andreas Petter, Alexander Behring, and Max Mühlhäuser. Constraint Solving in Model Transformations. In Richard F. Paige, editor, *International Conference on Model Transformation, ICMT 2009*. Springer, 2009. to appear.

[Rud00] Michael Rudolf. *Theory and Application of Graph Transformations*, volume 1764/2000 of *Lecture Notes in Computer Science*, chapter Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching, pages 381 –394. Springer, 2000.

[Sea93] A. Sears. Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout. *IEEE Trans. Softw. Eng.*, 19(7):707–719, 1993.

[Sta73] Herbert Stachowiak. Allgemeine Modelltheorie. Book, 1973.

[Sut64] Ivan E. Sutherland. Sketch pad a man-machine graphical communication system. In *DAC '64: Proceedings of the SHARE design automation workshop*, pages 6.329–6.346, New York, NY, USA, 1964. ACM Press.

[VH99] Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

[WGS09] Jules White, Jeff Gray, and Douglas C. Schmidt. Constraint-based Model Weaving. *Transactions on Aspect-Oriented Software Development*, open:open, 2009. to appear.