Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 211 (2008) 63–74
www.elsevier.com/locate/entcs

# Transformational Pattern System - Some Assembly Required

Mika Siikarla[1] and Tarja Systä[2]

*Institute of Software Systems*
*Tampere University of Technology*
*Tampere, Finland*

**Abstract**

In the context of Model Driven Architecture (MDA), most model transformation mechanisms aim for rigorously and unambiguously defined, fully automatic transformations. We argue that such techniques, even when fully mature, are not applicable in all cases of software development. These difficult cases would benefit from flexible and semi-automatic open transformations. We present a mechanism, so called transformational pattern system, and show how it can combine human made decisions and intentionally vague and incomplete rules to perform a transformation.

*Keywords:* graph transformation, open model transformation, pattern, mda, pattern system

## 1 Introduction

Model Driven Architecture [7] (MDA) is the most recent and most prominent attempt to raise the level of abstraction used in defining software. The level has previously been successfully raised from machine code, symbolic assembler, and primitive programming languages to modern high level programming languages and in some cases even generating code from models. Now the goal is to use models from earlier and earlier design and perhaps even requirements capture phases and derive implementation from them.

The benefits of achieving the MDA vision would of course be significant. Production efficiency would rise due to higher abstraction level. Maintainability would be improved when design models would always be up-to-date. Because rising the abstraction level has been so successful previously, some believe this next step will be just as successful, as soon as good enough methods and tools have been developed. We argue that such expectations are reasonable only when certain restrictions

[1] Email: mika.siikarla@tut.fi
[2] Email: tarja.systa@tut.fi


doi:10.1016/j.entcs.2008.04.030

apply. When the level of abstraction gets higher, automatic transformations get more complicated, their cost goes up, and they have to make decisions with greater consequences all leading to fewer cases where the transformation is usable.

Models from early design phases have less details than ones from later phases. They do not just show less details, they actually have less details. After all, an important reason for using high abstract level is to avoid committing to details too early. Details are added later, refining the model. Some of them are inconsequential, but some are important design decisions. The more abstract the model is, the bigger impact the decisions have on the end result. A guess can be made at source code level, knowing that at worst it will be off an opcode or two. A guess at the architecture level can go wrong a subsystem or two.

An automatic transformation can only succeed, if it knows what the design decisions should be. This is more likely in the context of, e.g. a single problem domain, company or product line or versions of a product, where the situation is well understood and rather stable. For example, C++ has standard fixed semantics, so a C++ compiler does not need to (must not!) make behaviour affecting decisions. If the context is not limited in any way, there are infinitely many possibilities, too many to take into account beforehand.

Automatic transformations do not get rid of complexity. Instead of relying on the expertise and wisdom of a designer to create a target model, we rely on the transformation engineer to create a transformation. The transformation must solve a more generic problem and apply to more cases than one, and is therefore more difficult to build. The relative development cost is reduced, if the transformation is applied to several products. For a one-of-a-kind product or for a small organization, it might not be cost-efficient to develop (and maintain!) another piece of software, i.e. the transformation itself.

We argue that in some cases where an automatic transformation is not feasible or even possible, some of the MDA benefits can still be achieved. Dropping the requirement for full automation and instead incorporating a human in the transformation process, by interacting with him and allowing manual changes, enables more flexible transformation mechanisms. In order for the human to be able to make a difficult design decision, he needs to understand its context. There is need for open transformation mechanisms, i.e. ones that are transparent, accessible, interactive, and flexible.

We present an experiemental semi-automatic transformation mechanism based on so called *transformational patterns*. This paper extends our previous work [10], where transformational patterns were used alone, by adding a method for joining several patterns together. The mechanism is fully transparent and allows the user to choose the order of tasks and make manual changes to the models. At this time, we do not attempt to tackle problems caused by incremental changes to the source model. We illustrate the use of the mechanism with an example.

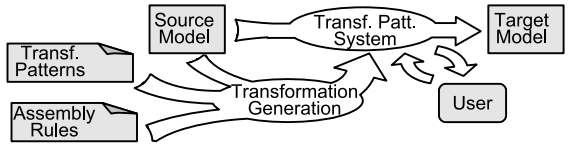



Fig. 1. Generating and applying a transformational pattern system

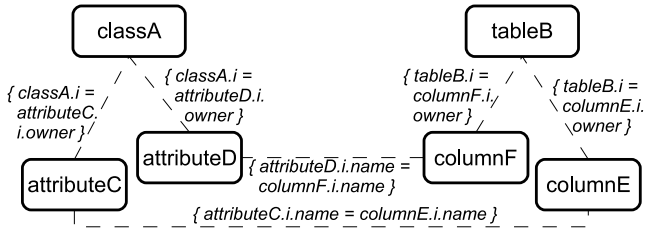
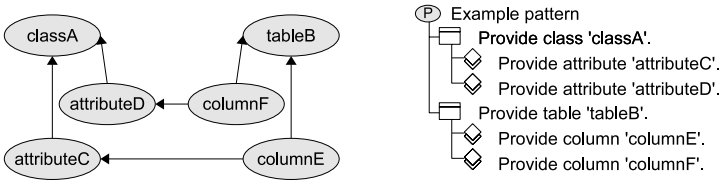



Fig. 2. A pattern with six roles and six constraints




(a) A task graph (b) A task hierarchy

Fig. 3. A task graph and the corresponding task hierarchy

## 2 Transformation Mechanism

In this paper, a transformation specification consists of so called *(transformational) patterns* and *assembly rules*. A transformational pattern describes how a transformation rule, e.g. `Transform a UML Class into a Java class`, is implemented. The assembly rules describe how the individual patterns relate and which patterns are applied to which source model elements. We call such a collection of interrelated patterns a *(transformational) pattern system*. It is an implementation of an interactive transformation for a specific source model. These different components of a transformation are presented in Figure 1.

A transformational pattern describes a configuration of model elements, which must exist after the corresponding transformation rule has been applied. A pattern is given as a set of roles and constraints. Each role of a pattern instance is attached, i.e. *bound*, to a model element. The constraints restrict to which elements a role can be bound. A small pattern is depicted in Figure 2. The constraints state, for example, that (the attributes bound to) roles *attributeB* and *attributeC* belong to (the class bound to) role *classA*. They also require *columnE* to have the same name as *attributeC*. If the constraints permit, multiple roles can be bound to the same element.

Applying a pattern can also be viewed as a set of tasks; "bind *classA*", "bind *tableB*", etc. When all the tasks have been performed either by selecting an ex-

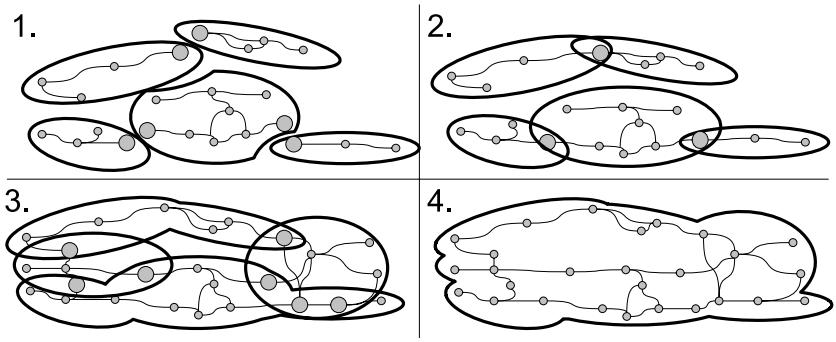



Fig. 4. An example of joining task graphs (patterns) together

isting model element or by generating a new one (while observing the constraints), the pattern has been applied. To make performing tasks easier, each constraint concerning two or more roles is directed. That is, one role (binding) is considered to be "correct" and the other(s) must be bound to conforming element(s). This implies a partial ordering of tasks, which can be presented as a directed acyclic graph. Figure 3(a) depicts a task graph for the pattern in Figure 2. A task graph resembles a function or a program that derives new bindings based on existing ones. For every task, a new role is bound to a model element. If there is only one option the task can be performed automatically, otherwise human interaction is required.

We use MADE [5] to apply patterns. For easier task selection, MADE presents a task graph as a hierarchy of roles/tasks. Figure 3(b) shows the task hierarchy for the task graph in Figure 3(a). The hierarchy criteria is currently fixed and is based on containment. For example, the task for *attributeC* is under *classA* because the constraints demand that the class bound to *classA* contains the attribute bound to *attributeC*. The user can browse this hierarchy by selecting a task. The tool will then show the list of tasks directly underneath the selected task. Tasks with unbound dependencies will be hidden. For example, the task for *columnE* will not appear before tasks for *tableB* and *attributeC* have been performed. MADE also offers some shorthand commands, for example to perform all automatic tasks in a task list.

Task graphs can be connected together in sequence and in parallel by merging some of their nodes. This is equivalent to merging the roles, where the new role has all the dependencies and constraints of the merged roles. Such a pattern system is a more complicated function, assembled from simpler ones, and fulfills a more complicated purpose. Since a pattern system is itself a pattern, MADE can be used to apply pattern systems, too.

The example in Figure 4 contains five task graphs (1.). A pattern system is assembled from the two top patterns by merging one node from each task graph. Likewise, the three patterns on the bottom are assembled into a second pattern system (2.). Two new pattern instances are created and joined with the old ones (3.) creating the task graph for the complete pattern system (4.).

The pattern assembly mechanism parses the source model and as a side-effect

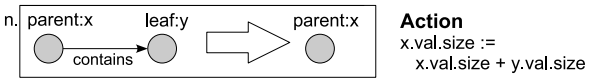



Fig. 5. An example of an assembly rule; a graph production and an action

forms a pattern system by creating and joining pattern instances. The mechanism is essentially a graph rewrite system (GRS). However, each graph production $p_i$ is associated with an action $a_i$. A production-action pair $\langle p_i, a_i \rangle$ is called an *assembly rule*. The productions are applied to a directed labeled graph representing the source model, where each node has a type and can have named values attached. Whenever a production is used, the associated action is triggered. The productions reduce the input graph step by step, while the actions construct the resulting pattern system. In other words, the GRS is used to recognize or parse the source graph.

The assembly rules are ordered and the first with an applicable production is always used. When no more productions apply, the mechanism stops regardless of how many nodes or edges remain in the graph.

In the beginning each node corresponds to one source model element and the node's name-values come from the element, e.g. the name and id of a UML class. Later on the values are usually roles or pattern instances created by actions. When a production triggers an action, it has access to the values of the nodes matching to the production's left hand side (LHS) and right hand side (RHS). A typical action fetches the patterns attached to two nodes in the LHS and joins them together. The concept is analoguous to the grammar rules (productions) and actions in the common textual parser generator yacc.

For a simple example, consider a graph consisting of directed trees, i.e. a directed forest, and that we want to know the amount of nodes in each of the trees. Let us assume that in the beginning the leaf nodes are of type $leaf$ and the other nodes are $parent$. Let us also assume that each node starts with a single named value; $size = 1$. The assembly rule in Figure 5 could be used as part of the solution. It is applicable whenever there are two nodes, $x$ and $y$, such that $x$ is of type $parent$, and $y$ is of type $leaf$, and $y$ is a child node of $x$. When the production is applied its action increments the value of $size$ in $x$ by the value of $size$ in $y$. The leaf $y$ is then removed from the graph. A few more assembly rules are needed to complete the example. One changes a $parent$ with no children into a $leaf$. Another collects the $size$ from a one-node tree into some global stack and removes the tree.

It is important to note that the assembly rules do not perform the actual model transformation. They only assemble the pattern system, which is then used to transform the model, guided by the user.

In the implementation, the productions are given using Object Constraint Language [8] (OCL) expressions and Python code. For this reason, a graphical notation (Figure 5, 7) is used in this paper for presenting productions. The notation is used solely for visualization, and is not formally defined. The production rule implementation is currently not automatically derived from the description. Actions are expressed in Python.

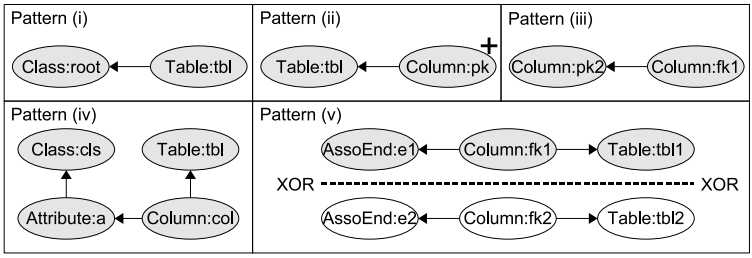



Fig. 6. Transformational patterns corresponding to the informal rules

# 3 Example of Constructing a Transformation

As an example, consider the seemingly simple transformation from a structure model (a UML Class Diagram) into a relational database schema. It seems quite straight forward, but there are details, options and exceptions that add complexity. For example, there are different ways to interpret and transform composition, inheritance and other relations between classes, and there is not always enough information in the source model to make the decision. It is in managing these details and variations where the real challenge for a transformation mechanism lies. With transformational patterns, their inherent flexibility and interactive nature helps overcome some of these difficulties.

A rough natural language description of the transformation might be:

(i) Each class inheritance hierarchy is transformed into a single table. The table is named after the root class.

(ii) At least one column in each table belongs to its primary key.

(iii) Foreign key should reflect the primary key selected for the target table.

(iv) Each attribute is transformed into a column in the table corresponding to the attribute's class. The column is named after the attribute.

(v) Each association is transformed into a table reference. The designer decides which table holds the foreign keys. The foreign keys are named after the primary keys and the association role chosen.

The (task graphs for) transformational patterns in Figure 6, one for each informal rule, describe how the rules are implemented. The constraints have been omitted for clarity. The patterns could be read as `"a table is created based on some class"` (pattern i), `"some columns are chosen from some table"` (pattern ii), and so on. When an instance of such pattern is partially bound, it gets a more precise meaning, e.g. `"a table is created based on class` *Show*`"`. The flexibility in patterns and the choices the user will make eventually decide how *exactly* the rule is applied.

The `+` in pattern (ii) and the `XOR` in pattern (v) are details of the notation for MADE, the tool used for applying patterns. The markings mean that the user decides at runtime how many *pk* roles pattern (ii) has and which of the alternative structures is used for pattern (v).

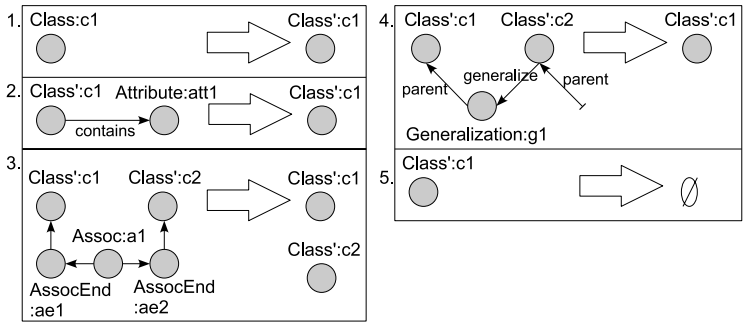



Fig. 7. GRS productions for the pattern assembly rules

There are five assembly rules and their productions are in Figure 7. It is a coincidence, that there are as many rules as there are patterns. The first rule is used for initializing a *Class* node. Production (1) marks an initialized node by changing its type to *Class′*. The rest of the productions parse the source graph. Productions (2) and (3) remove attributes and associations. Production (4) removes a leaf *Class′* in an inheritance hierarchy. When the hierarchy has been reduced to a single node, production (5) removes that node. The actions for productions (1) and (2) (in pseudo-code) are:

```
1 patt = new Pattern_ii #For the patterns, see Figure 6
  c1.val = {class:  new ClassRole, table:  patt.tbl, pk:  patt.pk}
  bind_role(c1.val.class, c1.id)
```

```
2 patt = new Pattern_iv
  merge_role(c1.val.class, patt.cls)
  merge_role(c1.val.table, patt.tbl)
  bind_role(patt.a, att1.id)
```

When the pattern assembly rules are used on, e.g. the diagram in Figure 9, the first production applies and is used. The action (1) is triggered and variable `c1` points to one of the graph nodes representing a class. The action attaches three roles as named values to the node; *class*, *table*, and *pk*. In addition, it binds the role *class* to the source model class the node corresponds to. The production changes the type of the node from *Class* to *Class′*, so that the first production will not be used on the node again. This is repeated on each node of type *Class*. So, the first assembly rule does not change the structure of the graph, it merely initializes the class nodes' values.

The second production is used when the first no longer applies. It finds attribute nodes and removes them. The action (2) creates a new instance of pattern (iv), binds the attribute role to the source model attribute the attribute node corresponds to, and finally merges the pattern's *cls* and *tbl* roles with the roles *class* and *table* associated with the class node. The left side of Figure 8 shows the pattern associated with a class node after one of its attributes has been removed. The right side shows the pattern after another attribute has been removed. The stacked tasks represent merged tasks. In reality, it is not possible to tell after the fact, whether a task has

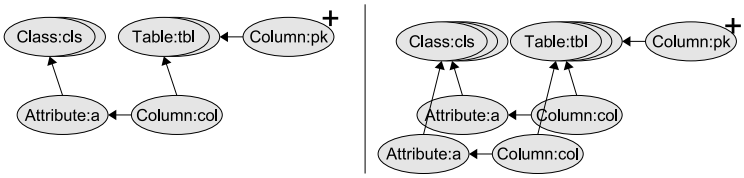



Fig. 8. Rule (iv) pattern joined once (left) and twice (right)

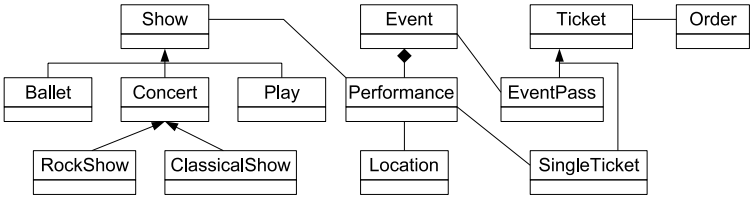



Fig. 9. Structure model used in the example

been merged.

After the second production rule no longer applies, the third one is used, then the fourth, and so on, until no production rules apply. At that point, there is a complete transformational pattern system created by the actions.

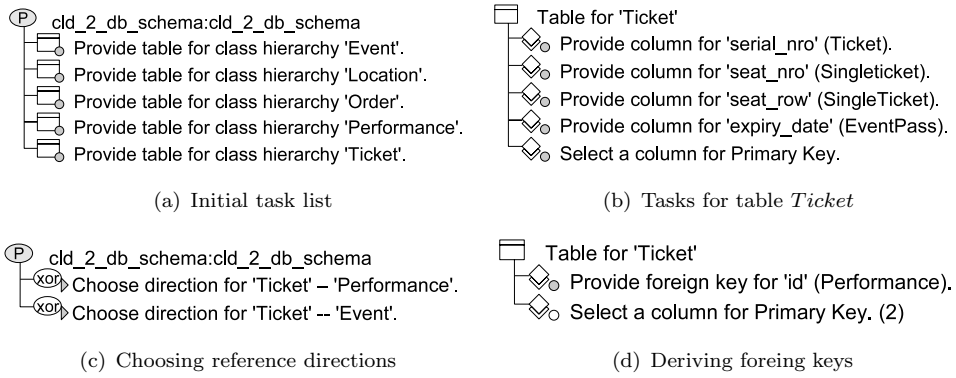
## 4 Example of Applying a Transformation

To demonstrate applying a transformational pattern system, a possible user session is presented step by step. The transformation itself is the structure model to database schema presented in Section 3 and it will be applied to a ticket service structure model (Figure 9). Bob is assigned with the task of creating the database schema. A CASE-tool is used for visualizing the structure model and the schema (both as UML Class Diagrams) and MADE is used for applying the pattern system.

Bob starts the CASE-tool and loads the source model. He executes the assembly rules from the command line, starts MADE and imports the pattern system. A list of tasks appears, one `Provide table for class hierarchy <name>` task for each class hierarchy (Figure 10(a) [3] ). Bob selects the task for *Performance* and tells MADE to generate a new table. A new class representing the table is generated and appears in the CASE-tool. New tasks become available and are listed under the old, now inactive, task; one for selecting primary keys and one for transforming attributes to columns. Bob ignores them for now, and instead instructs MADE to create tables *Show*, *Event*, and *Ticket*.

Bob looks at the tasks (Figure 10(b)) listed under the *Ticket* table; creating columns and selecting primary keys. He selects `Perform all automatic tasks` and a column (represented by an attribute) is created and appears in the CASE-tool for each attribute in the classes *Ticket*, *EventPass*, and *SingleTicket*. Primary keys are not selected, because that task is not automatic. Bob performs the task

[3] For better image scaling, bitmap screen captures in Figure 10(a)–10(d) have been manually redrawn in a vector format.




(a) Initial task list (b) Tasks for table *Ticket*

(c) Choosing reference directions (d) Deriving foreing keys

Fig. 10. Binding a pattern and the representation as a task graph

manually by selecting the column *serial_no* which he knows uniquely identifies a ticket. The selection is visualized by stereotyping the column as ≪`PrimaryKey`≫. There are no more mandatory tasks for this class, but he could select more primary keys if he wanted.

There is nothing special about the attributes of *Event* and *Performance*, so Bob tells MADE to generate columns for those, too. When looking for primary key candidates, he realizes none of the columns will work. He switches to the CASE-tool and manually adds a column *id* in both tables. Then, in MADE, he selects them as primary keys for their tables.

When primary keys have been selected for some tables, choosing the directions for table references between those tables is enabled. The task list now includes tasks for the associations from *Ticket* to *Event* and *Performance* (Figure 10(c)). Bob is able to reason that there will be many *SingleTickets* for each performance, and that one ticket can be used for one show only. He therefore selects the task for the association between *SingleTicket* and *Performance* and chooses *SingleTicket* to hold the foreign keys. When the choice is made, tasks for deriving the actual foreign key columns from the primary key of *Performance* appear (Figure 10(d)). Bob tells MADE to generate the foreign keys, and the column *performance_id* stereotyped ≪`ForeignKey`≫ appears in *Ticket*. Bob applies the same reasoning for the other table reference and generates the column *event_id* under *Ticket*, too.

The user can always choose the next tasks freely, as long as the tasks it depends on have been performed first. Bob utilizes this freedom fully, when he, in this order, generates the tables for *Show* and *Order*, manually adds a column in *Show*, generates the derived columns for *Order*, selects one primary key column for *Show*, one for *Order*, then another for *show*, generates the table *Location*, and chooses a direction for the reference between *Show* and *Event*. This order may seem random from the outside, but Bob is working according to some personal internal logic, probably inexplicable even to himself. When he, not the tool, chooses what to do an in what order, he keeps better track of the context and is therefore more capable of making the right design decisions when the tool needs the human plug-in.

When looking at the relationship between *Location* and *Performance*, Bob con-

cludes it is more complex than the previous ones. He decides there needs to be a third table to map the other two. There is no task for it, because such a possibility was not taken into account when designing the transformation. Still, Bob can manually create the mapping table and all required columns in the three tables. There is currently no way of marking a task obsolete, so he has to remember to ignore the task for choosing the direction for the *Location - Performance* reference. Although the purpose of the new table is not "understood" by the transformation, that does not affect the rest of the model and the rest of the transformation.

Bob started working on the transformation so late in the day, that he is not able to finish it before leaving work. So, he saves his work in the CASE-tool and MADE, knowing he can load the structure model, database schema and the transformation the next day and continue right where he left off.

## 5 Related work

There are many model transformation approaches, but few attempt interaction or manual editing of models beyond pre-determined choices or parameters.

Triple graph grammars [9] are grammars spanning three related graphs; one for the source model, one for the target model, and one for the relationships between the models. Each production alters all the graphs (models) at the same time, keeping them always synchronized and confirmant with their schemas (metamodels). A transformational pattern system contains elements for the source and target models and their relations. In that sense, a pattern system is an abstract triple graph. Due to the flexibility in binding, it represents a group of triple graphs.

With triple graph grammars, additions to the source or target models can be dealt with simply by applying further productions. We have not yet addressed the problem of incrementality for pattern systems. Triple graph grammars are also bidirectional. Although a transformational pattern itself is not directed, a derived task graph always is. The assembly rules, too, create a bias towards a direction.

Some graph transformation tools provide interaction, e.g. AGG [11], and AToM$^3$ [4]. The user can perform stepwise transformations and to choose the next production to apply. In AGG the user can even choose on which graph elements the production is applied, which resembles binding a pattern. Allowing the user to choose productions is powerful and enables ambiguous rules. But in order to make a decision, the user has to thoroughly understand the grammar in addition to understanding the transformation semantics, e.g. classes to tables, attributes to columns. We try to put the decisions more in terms of the semantics by placing the interaction in the pattern system. The user still has to work with a tool's process, but we believe it to be more similar to the user's view of the transformation process. Perhaps the power of interactive grammars can somehow be combined with the intuitiveness of pattern systems.

GREaT [2] is a graph transformation tool, which produces a Java program that can be run to perform the model transformation. We use assembly rules to produce a pattern system, which is then applied with MADE. However, the motivation with

GREaT seems to be integration into Java applications and possibly efficiency. User interaction does not seem to be considered.

A transformational pattern system, once all the roles are bound, is also a mapping between the source and target models. So, model mapping techniques [6] are in some way similar. However, they are typically bidirectional, whereas transformational patterns are not.

ATL [1], among others, approaches the problem of too strict transformation definitions by enabling specialization of transformations. This, in effect, allows vague or general rules, which are then refined for a more specific situation.

## 6 Conclusions and future work

Transformational patterns (and thus also pattern systems) are rather flexible in describing structures. They can be viewed as task graphs, which are executable and give an implementation for applying the patterns. Tasks also have a natural interpretation as user choices, making task graphs interactive. Adding assembly rules gives the approach some of the benefits of the fully automatic approaches without removing the built-in user interaction.

Although incrementality was not considered in this work, it is very important for open transformations. As it is now, any significant change to the source model demands a reassembly of the pattern system, effectively forgetting the previous user decisions. Supporting incremental transformations needs to be researched. The pattern assembly mechanism also has to be better integrated with the pattern tool, to improve the user experience. For the same reason, the production rules need a well-defined and intuitive notation.

We also intend to strengthen the theoretical foundation of our approach with, e.g. graph grammars. For example, it has been pointed out to us that transformation pattern systems might bear resemblance to graph processes [3]. This is an interesting connection we intend to explore further.

## References

[1] Bézivin, J. and F. Jouault, *Using ATL for checking models*, in: *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT)* (2003).

[2] Christoph, A., *Graph rewrite systems for software design transformations*, in: *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World* (2003), pp. 76–86.

[3] Corradini, A., U. Montanari and F. Rossi, *Graph processes*, Fundamenta Informaticae **26** (1996), pp. 241–265.

[4] de Lara, J. and H. Vangheluwe, *AToM3: A tool for multi-formalism and meta-modelling*, in: *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering* (2002), pp. 174–188.

[5] Hammouda, I., J. Koskinen, M. Pussinen, M. Katara and T. Mikkonen, *Adaptable concern-based framework specialization in UML*, in: *Proceedings of ASE 2004* (2004), pp. 78–87.

[6] Hausmann, J. H. and S. Kent, *Visualizing model mappings in UML*, in: *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization* (2003), pp. 169–178.

[7] OMG, *Model driven architecture (MDA)* (2001).
URL http://www.omg.org/cgi-bin/apps/doc?ormsc/01-07-01.pdf

[8] OMG, *UML 2 OCL available specification* (2005).
URL http://www.omg.org/cgi-bin/doc?ptc/2005-06-06

[9] Schürr, A., *Specification of graph translators with triple graph grammars.*, in: E. W. Mayr, G. Schmidt and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*, Lecture Notes in Computer Science **903** (1995), pp. 151–163.

[10] Siikarla, M., K. Koskimies and T. Systä, *Open MDA using transformational patterns*, in: U. Aßmann, M. Aksit and A. Rensink, editors, *Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers*, Lecture Notes in Computer Science **3599** (2005), pp. 108–122.

[11] Taentzer, G., *AGG: A graph transformation environment for modeling and validation of software.*, in: J. L. Pfaltz, M. Nagl and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, Lecture Notes in Computer Science **3062** (2004), pp. 446–453.