Model reuse with metamodel-based transformations

Tihamer Levendovszky[1], Gabor Karsai[1], Miklos Maroti[1], Akos Ledeczi[1],
Hassan Charaf[2]

[1] Institute for Software Integrated Systems
Vanderbilt University
P.O. Box 36, Peabody
Nashville, TN 37203
{tihamer.levendovszky, gabor.karsai, miklos.maroti,
akos.ledeczi}@vanderbilt.edu
http://www.isis.vanderbilt.edu/
[2] Budapest University of Technology and Economics
Goldmann György tér. 1. V2 458
Budapest, XI.
Hungary
hassan@avalon.aut.bme.hu

Abstract. Metamodel-based transformations permit descriptions of mappings between models created using different concepts from possibly overlapping domains. This paper describes the basic algorithms used in matching metamodel constructs, and how this match is to be applied. The transformation process facilitates the reuse of models specified in one domain-specific modeling language in another context: another domain-specific modeling language. UML class diagrams are used as the language of the metamodels. The focus of the paper is on the matching and firing of transformation rules, and on finding efficient and generic algorithms. An illustrative case study is provided.

1 Introduction

Modeling has become fundamental in developing complex software systems where controlling and understanding every detail is beyond the capability of human comprehension using a general-purpose programming language. Presently, modeling can be considered as programming in a very high-level language that allows the designers to manage the complexity of the system. This approach has resulted in a number of different, yet related, modeling concepts that attempt to solve very similar representation problems.

Without the ability to perform model transformations, every existing model must be developed and understood separately, and/or has to be converted manually between the various modeling formalisms. This often requires as much effort as re-creating the models from scratch, in another modeling language. However, when automatic model transformations are used, the mapping between the different

concepts has to be developed only once for a pair of meta-models, not for each model instance.

To map models that have different metamodels is vital in software reuse, even for those software systems that are automatically generated directly from their models. This problem (and its solution) is the main concern of this paper.
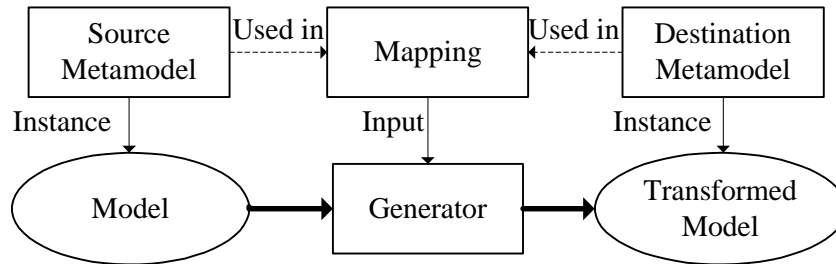

Fig. 1. Transformation process overview.

The overview of the concepts is depicted in Figure 1. The source and target models of the transformation are compliant with their corresponding metamodels. We create a representation of the mapping in terms of the metamodels. A generator, which is based on the mapping, is used to produce an output. The output from the generator is a transformation of the input model. The motivation for the automatic translation is that we wish to reuse the input model in an environment that supports the concepts of the output model.

To define the mapping between two conceptually different models requires a common basis that describes both the source and target domains of the transformation, and the transformation vocabulary. This common basis in our case is the metamodel [1]. We consider the metamodel as a set of metamodel elements, and we will use the widely adopted UML class diagram terminology [2] to describe the metamodel. Choosing the UML class diagram as a metamodel formalism implies that the two domains must share their meta-metamodel - namely, the UML class diagram metamodel (which is satisfied for a wide range of the practical applications). We regard a model as a set of model elements that are in correspondence with a metamodel element via the instantiation relationship. Metamodel-based transformations use only the elements of the metamodels, thus the transformation description is expressed in terms of the two metamodels.

Graph rewriting [3][5] is a powerful tool for graph transformations. The graph transformation is defined in the form of rewriting rules, where each such rule is a pair of graphs, called the LHS (left hand side) and RHS (right hand side). The rewriting rule operates as follows: the LHS of a rule is matched against the input graph, and the matching portion of that graph is replaced with the RHS of the rule. The replacement process is referred to as firing of the rewriting rule. In this work, we will use the

context elements [5] to describe the rewriting rules that change the edges between objects.

When we consider a match, we cannot match one model element in the input graph to two elements in the rewriting rule (identity condition) [5]. In our case, the complication we address is caused by the "schematic" nature of the metamodel. In a simple case, all models consist of objects with (typed) links between them. In the source graph we have to find exact matches to the left hand side of the rewriting rule, and replace it with precisely the right hand side of the rewriting rule. In the metamodel-based approach, the situation is more complex. The source and the destination model each consist of objects and their links. However, the left and right hand side of the rewriting rules are described in terms of the metamodel (i.e., classes, not instances). Under these circumstances the match is a subgraph of the source model that can be instantiated from the left hand side of the rewriting rules. If we have found a match, we can tell which model element (for instance a specific link) was instantiated from which metamodel element (in case of a link, which specific association). Then the right hand side of the rewriting rule is transformed into objects and links using these instantiation assignments. If an association names an instantiated link "l" during the matching phase, any occurrences on the right hand side of the rewriting rule will be replaced with "l." Afterwards, we will replace the match with these objects and the links based on the instantiation of the right hand side. Naturally, this may be a time-consuming, iterative process.

One of our basic objectives is to introduce a metamodel-based model transformation that is open to extension mechanisms that specialize the core algorithms. The reason why we need this extensibility is the algorithmic complexity of the matching and firing algorithms. There are practical reuse cases when a special property of the input graph can simplify the searching, and there are specific rules, where the firing algorithm can be simplified significantly by introducing new facilities in firing. Here, our main focus is on the core constructs and the related searching and firing algorithms. However, in our case study we present a sample extension to the firing rules. We have discovered in the literature many proven efficiency-increasing heuristics (e.g., [7]) that can be applied to extend the core concepts presented here.

2 A Case Study

Our example models contain processes and pipes. Suppose we have a modeling language that allows processes (P), containers (C) and connections between them. Containers may contain processes and other containers. Both containers and processes have ports (O), which can be connected to each other. For simplicity, processes must belong to a container. A sample model is depicted in Figure 6. This hierarchy is a very useful technique for organizing the models of complex systems, because higher-level containers hide the lower level details. Thus, the composite components can be regarded as a "black box" without the need to pay attention to the internal structure.

Suppose that the modeling language and tool of the target domain supports only processes and their connecting pipes. However, we want to reuse the hierarchical models in the target environment.
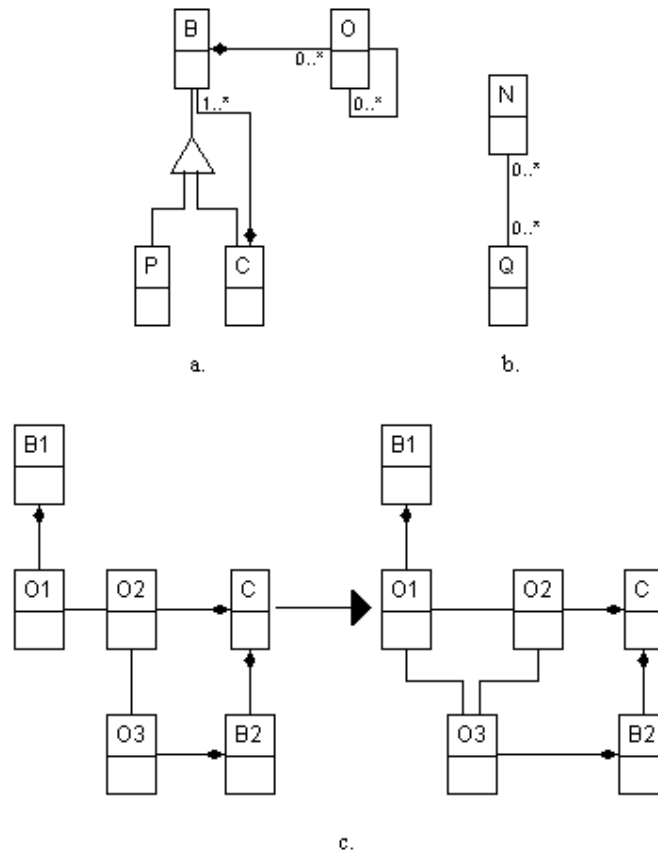

Fig. 2a. Metamodel of the source modeling language (uses containers). Containers are denoted by **C**, processes by **P** and ports by **O**. **2b.** Metamodel of the target modeling language (without containers). **N** denotes processes, **Q**s are pipes. **2c.** Transformation steps.

The input and output domain of the transformation can be seen on Figure 2ab. The textual description of the transformation is as follows:

> *We need to remove the containers. If an O_1 port in a C_1 container is connected to other O_t ports that are contained in other containers, the O_c ports of the elements in C_1 that are connected to O_1, should be connected to those O_t ports, and the port O_1 should be removed together with all its connections.*

First we can give a solution that is close to the textual transformation description. It can be seen in Figure 2c.

But, if we examine how this solution works, we can observe that rule 1 makes a fully connected graph between all the ports that are connected to a path in the input model. So this solution works and can be easily applied, but produces overhead.

Hence, the main problem in the solution proposed above is that, in this structure, it is hard to find a rule that "shortcuts" at least one port of a specific **C**, and deletes it in one step. If we try to shortcut a **C**-port in more than one step, we do not know if there is another match that will make another shortcut, and if the given port can be deleted. In order to accomplish this, we need an operation, which includes *all* the other connections of the port to each port.

We can identify a tradeoff that can be summarized as follows: On one hand, we use simple constructs to specify the LHS, but, as it was experienced in the example above, we cannot express complex patterns like "*all* the connections." In this case it is easy to match and fire the rewriting rule, but produce significant overhead (creating a fully connected graph in the example). This will be referred to as the basic case in the sequel. On the other hand, we apply more complex constructs to express the LHS, but these constructs require more complex matching and firing algorithms to support them.

In the next section we propose a set of complex constructs and their supporting algorithms.

3 Matching algorithms

One of the most crucial points of a graph rewriting-based transformation is the strategy for finding a match. As we introduced in Section 1, this is a particular problem in the case of a metamodel-based transformation, because there is an indirection in applying the left hand side of the rewriting rule: we want to find a match that can be generated from the left hand side of that rule. By allowing the multiplicity of objects to be different from the one in the metamodel, an increasing number of the model elements can be instantiated. To deal with this question, we consider the basic metamodel structures and how the multiplicity defined in the metamodel is related to the models.

3.1 The basic case

If we assume the multiplicity of the roles in an association to be one-to-one, we have an easy task because each of the metamodel elements can be instantiated to only one model element (except if the model uses inheritance). Without inheritance, we

can generate only one link for every association, and only one object for every class, in order to avoid the aforementioned indirection.

Because of this one-to-one mapping, the match can be considered as a form of type checking: if an object type is a specific class, then that object matches this class. The situation is exactly the same in the case of links ("instances") and associations (their "class"). If we have an inheritance relationship, we suggest a simple procedure: for every class that has a base class, a list should be maintained containing the type of all base classes. An object of any type in the list is regarded as a match. We assumed this basic case in Figure 2c.

Algorithm 1. Matching association of multiplicity one-to-one (basic case)
1. Form a type list for every derived class containing all of its base types and the derived class.
2. Assume there is an association **L** between classes **A** and **B**. Then, to match an object **a** having type **A** on the type list of its class, scan the neighbors of **a** to find an object **b** having **B** on its type list via a link of type **L**.

For the sake of simplicity, we use the term "type **A**" object instead of the precise description "having **A** on the type list of its class" in the sequel.

3.2 One-to-many case

The one-to-many case is depicted in Figure 3a. If we consider the example model in Figure 3b, we can see that the **A** class can only have one instance, which allows us to find it with the aforementioned type checking method.
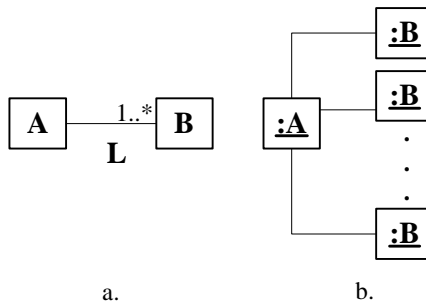

Fig. 3. One-to-many association: 3a.metamodel, 3b. example model.

After finding an object of type **A**, we can access all of its neighbors via links of type **L**. We also must check the multiplicity on the **B**-side of the association. The *x* denotes the number of found links. If this value (*x*) is in the multiplicity range, then there is a match and the rule can be fired.

Algorithm 2. Matching one-to-many associations.
1. Find an object of type **A**.
2. Find all of the **B** type neighbors of **A** type object via links type of **L**.
3. **N** denotes the set of integers containing the legal values for multiplicity. There is a match if and only if the number of links of type **L** from **A** to **B** is equal to n, and n∈**N**.

Effectively, Algorithm 2 is a breadth-first-search considering the type mechanism of the metamodel.

3.3 One-to-one case

The difference between the basic case and the one-to-one case is that the basic case considers one link of the several links of a many-to-many association as a match, and the one-to-many case does not. So the one-to-one is a specific case of the one-to-many.

Algorithm 3. Matching one-to-one associations.
1. Find an object of type **A**.
2. Find all of the **B** type neighbors of **A** type object via links type of **L**.
3. There is a match if and only if the number of links from **A** to **B** is precisely one.

3.4 Many-to-many case

The many-to-many case is the most complex construct. Matching many-to-many associations could be approached as follows:

1. Apply *Algorithm 2* to an arbitrary chosen **A** type object
2. Apply it again to the **B** type objects found in the previous steps.

This construction, however, could be substituted with multiple one-to-many relationships.

One may propose a definition as follows: the instantiated *N*-to-*M* association is a connected graph with vertices of type **A** and **B**, and links of type **L** (connecting only different object types). The degree *n* of an arbitrary **A** type vertex, and degree *m* of an arbitrary **B** type vertex must satisfy that $n \in N$, and $m \in M$.

The problem is that it can be hard to find a match for the many-to-many case (this is actually an arbitrary object net with type constraints), and even harder to apply the rule (see Section 4), so by default we do not use this construct to describe a transformation.

3.5 Iterative cases

In Figure 2c we can find a model element (in this specific case class **C**), which can be instantiated only once. We refer to these classes as *single classes* in the sequel. We can identify the single class easily: if there is no association end with multiplicity greater than 1 connected to a class, then it is a single class.

If a rule has a single class, we can find an object of the single class, and apply the basic search algorithms following the metamodel structure of the LHS of the rule. But, if we do not have single class, we have to iterate.
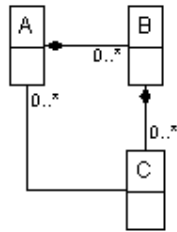

Fig. 4. Metamodel with no single classes

To trace the instantiation we use a set denoted by **O**A that contains the model elements matched by **A**. This notation is refined in Section 4.

Algorithm 4. Iteration from the "many" (> 1) multiplicity side.

1. Take a non-single class **A** as if it were single class.
2. Find all associated classes with **A** on the "many" side of the association. (On Figure 4. the result is **C**).
3. Find the **A** objects reachable via links from these **C** objects. Add these objects to the basic **O**A set, and iterate until there is no **A** object to add. (This can take significant time, so creating rules containing single classes is recommended.)

Another situation arises when we have to use iteration in the matching process and we reach a point where there is no unprocessed LHS association having multiplicity 1 at the processed object. In that case, the iteration scheme will be the same: we choose a link to go to the object on 1 side of the association, and search the other links.

3.6 The general search algorithm

We now outline the search algorithm needed to match the left sides of the rules using the concepts we described above.

Algorithm 5. Matching.

1. Find a single class. If we cannot find such a class, apply *Algorithm 4* for a class that participates in an association with the smallest (>1) multiplicity.
2. Following the left hand side of the transformations, match the associations using *Algorithm 2* or *Algorithm 3* based on the multiplicity values. When possible, follow the associations from the side with multiplicity 1 to the side with multiplicity >1 (many). When this is not possible, apply *Algorithm 4*.

Remarks. If the instantiation sets for the same class are different, we should consider only the intersection of the sets removing the connecting links.

After we successfully matched a class to an object, we mark it with the class name. To satisfy the identity condition, an already marked object can only instantiate the class it is marked with and nothing else.

3.7 Another solution to the case study

Using the concepts described in Section 3, we present another solution for our case study. One can see a more complex pattern in Figure 5. The lower case labels on the associations are for identification purposes only. The symbol **d+e** means the following: where there was a path via links of type **d** and **e** on the LHS, there must be a link along the association marked **d+e** on the RHS. This will be elaborated in Section 4. We concluded in Section 2 that there is a trade-off between the complexity of the rules and the efficiency of the application. Metamodel elements with the same name, but different numbers at the end of the name, refer to the same class, but to different instances of the class, because the identity condition has to be enforced. The matching algorithm checks all these classes regarding the multiplicity.
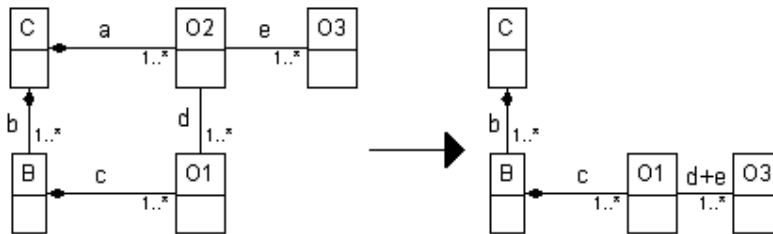

Fig. 5. A more efficient solution with more complicated search. This rule replaces the first rule in Figure 2c.

4 Applying rules

After finding a match to the left hand side of a rewriting rule, we have to fire it. But, in case of the metamodel-based transformation, the right hand side of the rule also contains an indirection that we have to resolve.
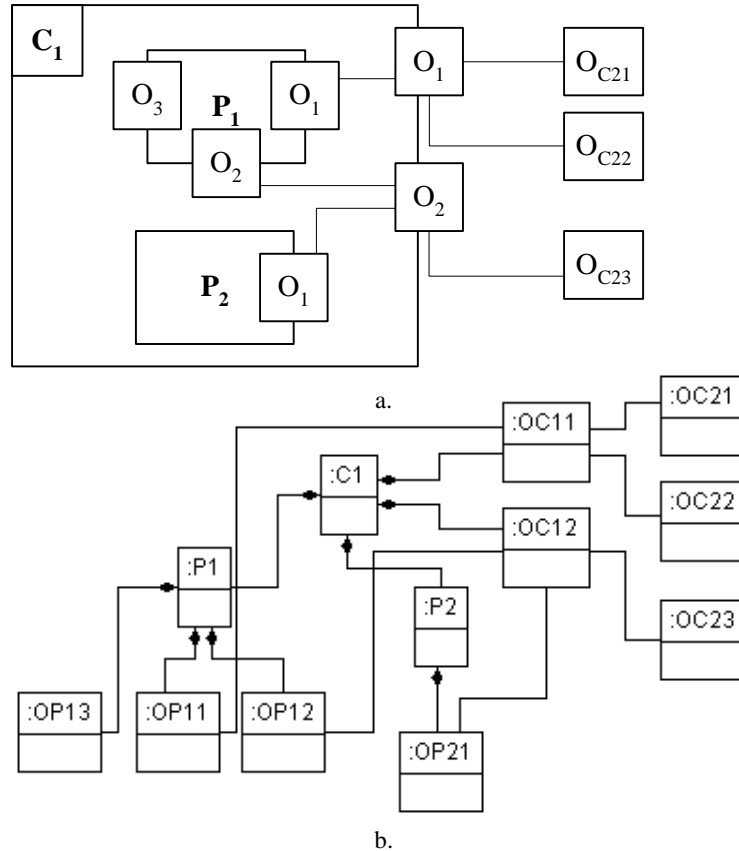

a.

b.

Fig. 6. A specific example. **6a.** Layout. **6b**. Model.

A notation suitable for handling the instantiation assignment is described below. This also facilitates the basic applicability checking on the right side of the rules.

An instantiation set, written as **OX,** means a collection of objects that are instances of class X of the LHS. An instantiation set **LY** denotes a collection of links, which are instances of the association Y of the LHS. For the rule on Figure 5, and the model on Figure 6b, the notation can be used as follows:

OC: $\{C_1\}$
La: $\{C_1/OC_{11}, C_1/OC_{12}\}$
OO$_2$: $\{OC_{11}, OC_{12}\}$
Ld: $\{\{OC_{11}/OP_{11}\}, \{OC_{12}/OP_{12}, OC_{12}/OP_{21}\}\}$
OO$_1$: $\{\{OP_{11}\}, \{OP_{12}, OP_{21}\}\}$

Lb: $\{C_1/P_1, C_1/P_2\}$
OB: $\{P_1, P_2\}$
Lc: $\{\{P_1/OP_{11}, P_1/OP_{12}\}, \{P_2/OP_{21}\}, \{P_1/OP_{13}\}\}$
OO$_1$': $\{\{OP_{11}, OP_{12}\}, \{OP_{21}\}, \{OP_{13}\}\}$

Le: $\{\{OC_{11}/OC_{21}, OC_{11}/OC_{22}\}, \{OC_{12}/OC_{23}\}\}$
OO$_3$: $\{\{OC_{21}, OC_{22}\}, \{OC_{23}\}\}$

We denoted the links using the object names they connect. As it has been mentioned in Section 3.6, to conform with the identity condition we mark all the objects and links assigned to a metamodel element on the right side of the rule as an instance, but we can assign the same right side metamodel element to a specific model element more than once, approaching from different directions (**O**O$_1$, and **O**O$_1$' in our example). This means that a metamodel element can have many different instantiation sets. In this case, we consider the intersection of the sets, but we keep all these sets around, because their structure can be different. In the example, this means removing OP_{13} from **O**O$_1$'. To easily trace the structure, we use nested sets to represent the tree structure of the instantiation sets. This redundancy facilitates checking mechanisms based on sets, without considering the topology.

Having completed the matching algorithm, we then build the instantiation set for the match. The instantiation set must be put together based on the right hand side of the rule. During the firing process we consider every right hand side metamodel element as a collection of model elements specified by the instantiation sets. First, we delete all the links and objects of right hand side classes and associations that do not appear on the left hand side of the rule. The notation **d**+**e** has the following semantics: *where there was a path via links of type **d** and **e** on the LHS, there must be a link along the association marked **d**+**e** on the RHS.* We define the extensions using the instantiation sets. The "**+**" operation defined below (for the sake of simplicity, we give the definition for only the case depth 1 here, but it can be easily generalized).

Algorithm 6. Computing the "+" operation between two subsequent associations.

1. Take the first operand, **d**. Replace every nested set with the Cartesian product of this set and the corresponding nested set in the second operand **e**.
2. The result will be in the first operand.

In our case the application of Algorithm 6 yields:

Ld+e: $\{\{OC_{11}/OP_{11}+ \ OC_{11}/OC_{21}, \ OC_{11}/OP_{11}+ \ OC_{11}/OC_{22}\}, \ \{OC_{12}/OP_{12}+ \ OC_{12}/OC_{23}, OC_{12}/OP_{21}+ OC_{12}/OC_{23}\}\}$

Simplifying:
Ld+e: $\{\{OP_{11}/OC_{21}, OP_{11}/OC_{22}\}, \{OP_{12}/OC_{23}, OP_{21}/OC_{23}\}\}$

After putting together the sets, the solution can be checked on Figure 6b. This case study illustrates the matching and firing mechanism, as well as the proposed notation and algorithms that support them in an extensible manner.

5 Related Work

With respect to metamodel-based transformation, an approach has been proposed that uses semantic nets (sNets) for metamodel descriptions [4]. This approach, however, does not deal with algorithmic, matching, and multiplicity issues. It uses mainly a textual representation for transformation rules.

The PROGRES graph rewriting system [6] also uses graph rewriting to transform, generate, and parse visual languages. The heuristic approaches to find a match in PROGRES reduces the algorithmic complexity using principles from database query systems. These heuristics can be easily integrated with those that are described in this paper.

Triple graph grammars [8] maintain a correspondence graph to specify the homomorphic mapping between the source and the target graph. This approach does not use object-oriented concepts.

An approach, similar to triple graph grammars is contributed by Akehurst [9], which is object-oriented and UML/OCL-based. This technique uses OCL and UML constructs to define the mapping between class diagrams, but it does not use graph replacement rules. Both the triple graph grammars and the UML/OCL techniques are bi-directional.

However, in several practical cases, the specification of a bi-directional mapping is hard or impossible, because information loss is inherent in the transformation. In our experience, the transformations need more than one pass to complete, thus more than one different, sequential transformation step is required. Direct mapping is not always possible in model reuse applications, because direct correspondence cannot be found between the different metamodeling concepts, but it can be specified between model constructs as it has been presented in our case study.

6 Conclusion

This paper presents an approach for reusing models based on different concepts that overlap somewhere in the domain. We require that these models have a common meta-metamodel: specifically a metamodel represented as UML class diagrams. We proposed a common language (the metamodel), defined transformations using the

elements of the metamodel, and offered algorithms to apply these transformations as graph rewriting rules. We intend this to serve as an algorithmic background for a forthcoming implementation where additional mechanisms will be added to control the application of the transformation rules.

A working case study has been presented showing the basic algorithms, formalisms and tradeoffs between the complex search algorithm and the size of the graph created during the intermediate transformation steps.

There are unsolved issues, however. We presented an extension in Section 4, but future work is needed to specify a general transformation system with the fewest number of extensions.

In this paper mainly the topological reuse issues have been covered. One can easily extend this transformation approach with simple constraints on the model elements (for instance, an attribute value must be greater than a specific value), but checking complex constraints, even those which are applied in the target domain, is the subject of ongoing research.

7 Acknowledgements

8 References

[1] Sztipanovits J., Karsai G., "Model-Integrated Computing," *IEEE Computer*, pp. 110-112, April, 1997.

[2] Rumbaugh, J., Booch, G., and Jacobson, I., "The Unified Modeling Language Reference Manual", Addison-Wesley, Reading, MA, 1999.

[3] D. Blostein, H. Fahmy, A. Grbavec, "Practical Use of Graph Rewriting", *Technical Report No. 95-373*, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, January, 1995.

[4] R. Lemesle, "Transformation Rules Based on Meta-Modeling", *EDOC '98*, La Jolla, California, 3-5 November 1998, pp.113-122.

[5] G. Rozenberg (ed.), "Handbook on Graph Grammars and Computing by Graph Transformation: Foundations", Vol.1-2. *World Scientific*, Singapore, 1997.

[6] The PROGRES system can be downloaded from http://www-i3.informatik.rwth-aachen.de

[7] A. Zündorf, "Graph Pattern Matching in PROGRES", In: "Graph Grammars and Their Applications in Computer Science", LNCS 1073, J. Cuny et al. (eds), Springer-Verlag, 1996, pp. 454-468.

[8] Schürr, A., "Specification of Graph Translators with Triple Graph Grammars", *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 903, Berlin: Springer-Verlag; June 1994; 151-163.

[9] Akehurst, D H, "Model Translation: A UML-based specification technique and active implementation approach", PhD Thesis, University of Kent at Canterbury, 2000.