PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 49, NO. 1–2, PP. 87–107 (2005)

PATTERN MATCHING IN METAMODEL-BASED MODEL TRANSFORMATION SYSTEMS

Tihamér LEVENDOVSZKY and Hassan CHARAF

Budapest University of Technology and Economics
H–1521 Budapest, Hungary
e-mail: tihamer@aut.bme.hu, hassan@aut.bme.hu

Received: Mai 11, 2005

Abstract

The vision of the OMG's Model-Driven Architecture (MDA) has necessitated the extensive research of model compilers, which are able to process graph-based visual models specified mainly in the Unified Modeling Language (UML). A possible mechanism for the realization of MDA model compilers can be graph rewriting-based transformation approach. Previous work has introduced the tool Visual Modeling and Transformation System, which uses graph rewriting as transformation mechanism, but the pattern language of the rewriting rules consists of UML class diagram elements instead of object diagram level patterns. This paper provides the algorithmic background for the application of these rules specified by the class diagram elements. To achieve that, it examines the allowed instantiation configuration based on the UML standard, and supplies a constructive algorithm to compute the allowed number of the objects participating in a valid instantiation of a class model. Furthermore, starting from the VF2 algorithm, the pattern matching algorithm for the left hand side of the metamodel-based rewriting rule is provided via several optimization steps examined.

Keywords: Pattern matching, UML, Graph Rewriting, Model Transformation.

1. Introduction

Analysing the development of the software engineering discipline a continuous increase of the abstraction level can be observed with respect to the implementation approaches. As the structured paradigm has been extended with object-oriented approach, OMG's Model-Driven Architecture [16] and the related research aim at raising the abstraction level to the visual model level. The MDA vision divides the development artifacts into two parts: (i) the Platform Independent Model (PIM), which is specified by the developers with the help of other stakeholders, (ii) then from PIM and the platform-specific information a so-called Platform Specific Model (PSM) is generated automatically by model compilers. Consequently, model compilers play a major role in an MDA environment.

Pattern matching lies at the heart of the model transformation systems which use graph rewriting techniques as the core mechanism of the transformation engine. The traditional algorithms take into account only topological considerations, ignoring types and attribute constraints. A graph rewriting-based transformation is a sequence of rewriting rules [7] that contain a left hand side graph (LHS) and a right hand side graph (RHS). The graph which the rule is applied to is referred to as

a host graph. Applying a rewriting rule means to find a subgraph in the host graph which is isomorphic to LHS.

The host graph $G^{\mathrm{H}}(V^{\mathrm{H}}, E^{\mathrm{H}})$ and the LHS graph $G^{\mathrm{LHS}}(V^{\mathrm{LHS}}, E^{\mathrm{LHS}})$ are given by their edge (E) and vertix (V) sets. In case of models E^{LHS} is modelled as a multiset. The labelling functions $lv^{\mathrm{H}} : V^{\mathrm{H}} \mapsto \Omega$, $le^{\mathrm{H}} : E^{\mathrm{H}} \mapsto \Omega$ assign the appropriate set to an arbitrary alphabet (Ω).

In case of graphs subgraph isomorphism and pattern matching are equivalent, but considering software models there can be more complex constructions in pattern matching. Our basic assumption is that the metamodel is always available containing the type information of the input model. This directly follows from the representation of the models in the metamodel-based environments. The metamodel-based pattern matching raises an open issue: there is no algorithmic background to find an instantiation instead of the traditional isomorphic subgraph in the host graph.

There are two main approaches to the subgraph isomorphism problem: (i) search with local heuristics, and (ii) perceiving it as a constraint satisfaction problem (CSP). Because the high level algorithms described in [4] have strong local nature, we have decided to apply the concept of the algorithms based on local heuristics.

This paper contributes to a technique and related algorithms, which consider types in the pattern matching as an efficient heuristics, and the algorithmic background to a metamodel-based model transformation system elaborated in [12]. The proof-concept implementation of the theoretical results has been implemented in the Visual Modeling and Transformation System (VMTS) tool. [12].

The rest of the paper is organized as follows: Section 2 elaborates the related work, focusing on the VF2 subgraph isomorphism algorithm and the metamodelling backgrounds. Section 3 contributes heuristics for the case when the metamodel of the host graph is available. Section 4 presents a novel algorithm for the pattern matching of the metamodel-based LHS graphs, where an instantiation needs to be found instead of an isomorphic subgraph. We delineate future work and conclude in Section 5.

2. Related Work

Transformation systems using graph rewriting are applied extensively in model transformation systems. These systems are built on wide variety of pattern matching algorithms. The Visual Automated Model Transformations framework (VIATRA) [5] facilitates model checking using precise mathematical background and metamodeling techniques. This tool applies the facilities of the PROLOG environment.

In the implementation of graph rewriting systems, PROGRES [6] provides constructs in rule firing and in sequencing the rules to form a controllable transformation process. To solve the subgraph isomorphism it creates an action graph used to establish a valid search plan, which is executed in an order given by special action priority heuristics.

The following two subsections introduce the concepts of the contribution related algorithms. Since later in this paper we consider software models, the existing algorithms have been modified to work on multigraphs as well.

2.1. The VF2 Algorithm

A graph and subgraph isomorphism algorithm, which is known as VF2 is presented in [1]. Similarly to Ullmann's subgraph ismomorphism algorithm [14] VF2 is general in the sense, that it does not impose any constraints on the input graphs. We will consider only the subgraph isomorphism part of VF2 in the sequel. A high-level description of VF2 is depicted in *Fig. 1*.

```
MATCH (Mapping M)
 1 if M covers all the nodes in G^LHS
 2     then M is a match, save it, if only one match had to be found, terminate
 3 else
 4     P=COMPUTE_CANDIDATE_PAIRS()
 5     for each (n, m) ∈ P
 6         do if FEASIBLE(M, n, m) then
 7             add (n, m) to M
 8             MATCH(M)
 9             remove (n, m) from M
10         end if
11     end for
12 end if
```

Fig. 1. The outline of the VF2 algorithm

In the procedure detailed above the following notations are used: two node variables $n \in G^{\mathrm{H}}, m \in G^{\mathrm{LHS}}$, the set of the candidate pairs $P \subseteq V^{\mathrm{H}} \times V^{\mathrm{LHS}}$, and $M \subseteq V^{\mathrm{H}} \times V^{\mathrm{LHS}}$ for the mapping between a subset of V^{H} and V^{LHS}.

For reasons becoming apparent later we introduce here a simpler version of the FEASIBLE procedure. This is basically a consistency check for the mapping: if $m \in G^{\mathrm{LHS}}$ has at most the same number of outgoing and incoming edges to or from the nodes already in the mapping as the corresponding node $n \in G^{\mathrm{H}}$, then the mapping M is consistent. The notations pred (G, x) and succ (G, x) refer to the set of nodes having incoming/outgoing edges to/from the node x in graph G. The multiplicity of an x, y element of a multiset MS is denoted with $\mu^{\mathrm{MS}}(x, y)$, and in Formulas (1) and (2) it corresponds to the number of edges between two vertices.

$$m' \in M^{\mathrm{LHS}} \wedge \exists n' | (n', m') \in M, n' \in \mathrm{pred}\,(G^{\mathrm{H}}, n) \\ \wedge \mu^{E^{\mathrm{LHS}}}(m', m) \leq \mu^{E^{\mathrm{H}}}(n', n) \tag{1}$$

$$m' \in M^{\mathrm{LHS}} \wedge \exists n' | (n', m') \in M, n' \in \mathrm{succ}\ (G^{\mathrm{H}}, n)$$
$$\wedge\ \mu^{E^{\mathrm{LHS}}}(m, m') \leq \mu^{E^{\mathrm{H}}}(n, n') \tag{2}$$

The validation is performed by the SIMPLE_FEASIBLE procedure (*Fig. 2*).

```
SIMPLE_FEASIBLE (Mapping M, Node n, Node m)
 1 for each predecessor m' of m
 2       do if not (1)
 3             return false
 4       end if
 5 end for
 6 for each successor m' of m
 7       do if not (2)
 8             return false
 9       end if
10 end for
11 return true
```

Fig. 2. A simple feasible procedure

The validation checked by the procedure SIMPLE_FEASIBLE is necessary if one wants to have correct matches. There are, however, optional conditions which can accelerate the algorithm in case of the vast majority of the input graphs, but a correct and consistent match can be achieved without them. VF2 algorithm enforces two more types of such constraints described in [2]. These heuristics are illustrated by the FEASIBLE procedure. The terminal sets (*Eq.* (3)-(6)) contain the nodes adjacent to the ones already included in the mach, but themselves are not included in the match. Then we check whether the number of the nodes adjacent to the new candidate pair (n, m) is consistent in both graphs. The notation of the sets is parametrized by the incoming/outgoing property of the edges, and the graph which the nodes in the set belong to. The notations M^{H} and M^{LHS} are the projections of M onto V^{H} and V^{LHS}, respectively.

$$T_{\mathrm{H}}^{\mathrm{in}} = \{x \mid x \in \mathrm{pred}\ (G^{\mathrm{H}}, y), y \in M^{\mathrm{H}}, x \notin M^{\mathrm{H}}, x, y \in G^{\mathrm{H}}\} \tag{3}$$
$$T_{\mathrm{LHS}}^{in} = \{x \mid x \in \mathrm{pred}\ (G^{\mathrm{LHS}}, y), y \in M^{\mathrm{LHS}}, x \notin M^{\mathrm{LHS}}, x, y \in G^{\mathrm{LHS}}\} \tag{4}$$
$$T_{\mathrm{H}}^{\mathrm{out}} = \{x \mid x \in \mathrm{succ}\ (G^{\mathrm{H}}, y), y \in M^{\mathrm{H}}, x \notin M^{\mathrm{H}}, x, y \in G^{\mathrm{H}}\} \tag{5}$$
$$T_{\mathrm{LHS}}^{\mathrm{out}} = \{x \mid x \in \mathrm{succ}\ (G^{\mathrm{LHS}}, y), y \in M^{\mathrm{LHS}}, x \notin M^{LHS}, x, y \in G^{\mathrm{LHS}}\} \tag{6}$$

The first type of the conditions can be expressed in the following forms:

$$\#\{\mathrm{pred}\ (G^{\mathrm{LHS}}, m) \cap T_{\mathrm{LHS}}^{\mathrm{in}}\} \leq \#\{\mathrm{pred}\ (G^{\mathrm{H}}, n) \cap T_{\mathrm{H}}^{\mathrm{in}}\} \tag{7}$$
$$\#\{\mathrm{succ}\ (G^{\mathrm{LHS}}, m) \cap T_{\mathrm{LHS}}^{\mathrm{in}}\} \leq \#\{\mathrm{succ}\ (G^{\mathrm{H}}, n) \cap T_{\mathrm{H}}^{\mathrm{in}}\} \tag{8}$$

$$\#\{\text{pred}\,(G^{\text{LHS}}, m) \cap T^{\text{out}}_{\text{LHS}}\} \leq \#\{\text{pred}\,(G^{\text{H}}, n) \cap T^{\text{out}}_{\text{H}}\} \tag{9}$$

$$\#\{\text{succ}\,(G^{\text{LHS}}, m) \cap T^{\text{out}}_{\text{LHS}}\} \leq \#\{\text{succ}\,(G^{\text{H}}, n) \cap T^{\text{out}}_{\text{H}}\} \tag{10}$$

The second type of constraints examines the nodes either in M or the sets (*Eq.* (3)–(6)) in both input graphs.

$$\begin{aligned}&\#\{x \mid x \in \text{pred}\,(G^{\text{LHS}}, m), x \notin M^{\text{LHS}}, (w, x) \notin (T^{\text{out}}_{\text{H}} \cup T^{\text{in}}_{\text{H}})\} \leq \\ &\#\{y \mid y \in \text{pred}\,(G^{\text{H}}, n), y \notin M^{\text{H}}, (y, z) \notin (T^{\text{out}}_{\text{LHS}} \cup T^{\text{in}}_{\text{LHS}})\} \\ &w \in G^{\text{H}}, z \in G^{\text{LHS}}\end{aligned} \tag{11}$$

$$\begin{aligned}&\#\{x \mid x \in \text{succ}\,(G^{\text{LHS}}, m), x \notin M^{\text{LHS}}, (w, x) \notin (T^{\text{out}}_{\text{H}} \cup T^{\text{in}}_{\text{H}})\} \leq \\ &\#\{y \mid y \in \text{succ}\,(G^{\text{H}}, n), y \notin M^{\text{H}}, (y, z) \notin (T^{\text{out}}_{\text{LHS}} \cup T^{\text{in}}_{\text{LHS}})\} \\ &z \in G^{\text{H}}, w \in G^{\text{LHS}}\end{aligned} \tag{12}$$

The validation suggested by VF2 is executed by the FEASIBLE procedure (*Fig. 3*).

```
FEASIBLE (Mapping M, Node n, Node m)

1 if not SIMPLE_FEASIBLE (Mapping M, Node n, Node m) return false
2 if not (7)-(10) then return false
3 return true
```

Fig. 3. The feasibility check of the VF2 algorithm

In the general MATCH algorithm (*Fig. 1*) line 4 computes the set P of the pairs which are candidate for inclusion in M. The procedure for this computation is shown in *Fig. 4*.

```
COMPUTE_CANDIDATE_PAIRS()

1 if T_H^out ≠ ∅ ∧ T_LHS^out ≠ ∅ then P = T_H^out × {min T_LHS^out}
2 else if T_H^in ≠ ∅ ∧ T_LHS^in ≠ ∅ then P = T_H^in × {min T_LHS^in}
3 else P = (V^H − M^H) × {min(V^LHS − M^LHS)}
```

Fig. 4. Computing the candidates in the VF2 algorithm

2.2. Metamodeling Backgrounds

Software (and often hardware) models are typed graphs: type information is assigned to the vertices and edges. Formally, this type information can be assigned as a part of the labels, but another way is to store it in another graph creating a mapping

from a node or edge and their types. This $G^M(V^M, E^M)$ graph is called metamodel. The edge to edge and the node to node mappings are performed by functions using the labelling mechanism. We introduce the functions *typeof:* $V \mapsto V^M$ and *typeof:* $E \mapsto E^M$, which are uniquely identified by their arguments, thus the same name does not result in confusion. For instance, *typeof (x)*, where x is a node in the V vertex set retrieves the type of the node x, and *typeof (x, y)* returns the type of the x, y edge, where x, y is a node pair in the E edge multiset of a graph. Similarly, we use the functions *instanceof:* $V^M \mapsto \{V\}$, and *instanceof:* $E^M \mapsto \{E\}$ to retrieve the node or edge sets corresponding to specific metaelement. The type information originates from the G^M metagraph with the help of the labelling functions. It is assumed that each edge and node label contains information uniquely identifying the particular element, because the type function needs to distinguish between the identical elements of the edge multiset.

In practice the elements of the metamodel are closely attached to the model elements, that is, both directions can be traversed at a cost of O(1) complexity. Furthermore the metamodel is often made available before the transformation, because the environment facilitating the creation of the model uses the metamodel as its internal data representation and maintains the model-metamodel mapping automatically.

Metamodeling and metamodeling environments are widely used techniques and tools creating software and hardware models. Generic Modeling Environment (GME) [8] is the metamodeling tool from which our system has borrowed several concepts of a metamodel-based modelling tool. GME on its own is not a transformation system, although the underlying MultiGraph Architecture (MGA) can be reached from the GReAT [9] transformation system. The GReAT framework is a transformation system for domain specific languages (DSL) built on metamodeling and graph rewriting concepts; it uses a proprietary notation and interpretation instead of instantiation between the rules expressed with meta elements.

AGSI is a metamodel-based storage and model transformation system introduced by [10]. It is the core part of the Visual Modeling and Transformation System (VMTS) [12], which supports domain specific visual languages and the Unified Modeling Language (UML)[OMG UML]. VMTS has a built-in transformation engine facilitating the specification of rewriting rules based on metamodel elements. The results discussed in this paper have been validated in VMTS as a proof-of concept implementation.

3. Type-aware Matching with Heuristics

The previous sections have summarized two algorithms applying different graph representations. In this section both methods are extended with metamodel-related data structures and heuristics. These methods often perform better in time than their original counterparts, but their storage space requirements are higher, because of the additional meta-information.

First of all the VF2 algorithm must be capable of checking the type information during the matching process. Since this is a consistency check, it is implemented in the SIMPLE_FEASIBLE procedure.

```
SIMPLE_FEASIBLE (Mapping M, Node n, Node m)
 1 if not typeof(n)=typeof(m) return false;
 2 for each predecessor m' of m
 3       do if not (13)
 4             return false
 5       end if
 6 end for
 7 for each successor m' of m
 8       do if not (14)
 9             return false
10 end if
11 end for
12 return true
```

Fig. 5. The modified Simple_Feasible procedure

The conditions have to check the edge types as well:

$$m' \in M^{\mathrm{LHS}} \wedge \exists n' | (n', m') \in M, n' \in \mathrm{pred}\,(G^{\mathrm{H}}, n) \wedge$$
$$\forall (typeof\,(n, n') \mu^{E^{\mathrm{LHS}}}(m', m) \leq \mu^{E^{\mathrm{H}}}(n', n) \wedge typeof\,(m', m) = typeof\,(n'n) \quad (13)$$
$$m' \in M^{\mathrm{LHS}} \wedge \exists n' | (n', m') \in M, n' \in \mathrm{succ}\,(G^{\mathrm{H}}, n) \wedge$$
$$\forall (typeof\,(n, n') \mu^{E^{\mathrm{LHS}}}(m, m') \leq \mu^{E^{\mathrm{H}}}(n, n') \wedge typeof\,(m, m') = typeof\,(n, n') \quad (14)$$

The conditions (7)-(12) must be extended in a very similar way. Now we have a variant of the VF2 algorithm which is capable of working with metamodels, and will be referred to as MetaVF2 in the sequel. Then it is possible to add modifications that benefit from the availability of the metamodel and the metamodel-model mapping.

The first step of computing the candidate pairs (when all the terminal sets are empty since the mapping itself is empty) is to match a node from the LHS graph to all the nodes in the host graph, and the type equivalence of these nodes are checked by calling SIMPLE_FEASIBLE. We can reduce the cardinality of this set, if after selecting an element form LHS we walk down to the metamodel element representing its type and retrieve the nodes of this type that reside in the host graph.

PROPOSITION 1 *The worst case of the* COMPUTE_CANDIDATE_PAIRS*() procedure (Fig. 6) is the one provided by the VF2 algorithm (Fig. 4) assuming connected pattern graph.*

Proof. The only difference between the MetaVF2 and this modified version is that in case of the first step (or the first step when processing a new connected block of a disconnected pattern graph) does not compute a Cartesian product of the given minimal LHS element and all host graph vertices ($P = (V^{\mathrm{H}} - M^{\mathrm{H}}) \times \{\min(V^{\mathrm{LHS}} - M^{\mathrm{LHS}})\}$), but only those of them that are instances of the minimal LHS element ($P = \mathit{Instanceof}\,(\min(V^{\mathrm{LHS}} - M^{\mathrm{LHS}})) \times \{\min(V^{\mathrm{LHS}} - M^{\mathrm{LHS}})\}$). It has been assumed in our model that the *Instanceof* operation takes $O(1)$ time complexity, thus retrieving $(V^{\mathrm{H}} - M^{\mathrm{H}})$ takes the same time as computing *Instanceof* $(\min(V^{\mathrm{LHS}} - M^{\mathrm{LHS}})$. This shortcut, however, is only beneficial, when $\#(V^{\mathrm{H}} - M^{\mathrm{H}}) > \#\mathit{Instanceof}\,(\min(V^{\mathrm{LHS}} - M^{\mathrm{LHS}})$. If the pattern graph is connected, line 3 in *Fig. 6* can only be executed for the first time of the function call. Hence for connected pattern graphs $M^{\mathrm{H}} = \emptyset$, $M^{\mathrm{LHS}} = \emptyset$ holds. Then $\#V^{\mathrm{H}} < \#\mathit{Instanceof}\,(\min(V^{\mathrm{LHS}}))$ is impossible, $\#V^{\mathrm{H}} = \#\mathit{Instanceof}\,(\min(V^{\mathrm{LHS}}))$ occurs only when every instance graph element has the same metaelement: therefore this is the worst case of the algorithm (same as the MetaVF2), for all other graphs $\#V^{\mathrm{H}} > \#\mathit{Instanceof}\,(\min(V^{\mathrm{LHS}}))$.

COMPUTE_CANDIDATE_PAIRS()

1 **if** $T^{\mathrm{out}}_{\mathrm{H}} \neq \emptyset \wedge T^{\mathrm{out}}_{\mathrm{LHS}} \neq \emptyset$ **then** $P = T^{\mathrm{out}}_{\mathrm{H}} \times \{\min T^{\mathrm{out}}_{\mathrm{LHS}}\}$
2 **else if** $T^{\mathrm{in}}_{\mathrm{H}} \neq \emptyset \wedge T^{\mathrm{in}}_{\mathrm{LHS}} \neq \emptyset$ **then** $P = T^{\mathrm{in}}_{\mathrm{H}} \times \{\min T^{\mathrm{in}}_{\mathrm{LHS}}\}$
3 **else** P = *Instanceof* $(\min(V^{\mathrm{LHS}} - M^{\mathrm{LHS}})) \times \{min(V^{\mathrm{LHS}} - M^{\mathrm{LHS}})\}$

Fig. 6. Computing candidates in MetaVF2 algorithm

Another offline heuristics assumes that some simple statistical information is available about the model-metamodel relationship. If the first selected node is the instance of the metanode which has the fewest instance in the host graph, we minimize the $\#\mathit{Instanceof}\,(\min(V^{\mathrm{LHS}}))$ expression by choosing the V^{LHS} vertex having the fewest instance. Unfortunately, this heuristics is not appropriate in every case. By choosing the V^{LHS} vertex having the fewest instance in every step, it is possible that more steps are necessary than in case of the algorithm developed in Proposition 1. In order to contradict, we assume that by choosing the V^{LHS} vertex having the fewest instance, less or the same number of steps are necessary than in case of the algorithm developed in Proposition 1. A simple counterexample can be seen in *Fig. 7* where starting with the fewest instance we need two steps to realize that no match exists, but only one step is enough if we start with a : B node.

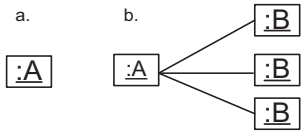


Fig. 7. A simple counterexample. a. Host graph, b. LHS

We assume that during the model building a vertex list is maintained in the metamodel, which is ordered by the number of instance belonging to a vertex. Gathering the statistical information might be performed online (in matching time), but it requires proof by simulation, which is the subject of future work.

4. Patterns Specified by Metamodel Elements

This section is devoted to the algorithmic background of the metamodel-based pattern matching. Two results are presented: (i) a proven algorithm to calculate the number of objects in a valid instantiation of a UML class diagram, (ii) a matching algorithm which finds an instantiation of the LHS graph in the host graph.

4.1. Feasibility of the Instantiation

If we want to use UML class diagram formalism for describing patterns, we have to examine the instantiation process on a mathematical basis. For instance there are patterns for which no valid instantiation exists.

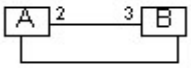


Fig. 8. A UML class diagram having no valid instances

As it is depicted in *Fig. 8*, each object of class *A* is required to be connected to three number of *B* type objects via an association, and required to be connected with exactly one *B* via another association, which is not feasible.

Fig. 9. A general binary association

In order to deal with this problem, we need to determine the number of objects participating in a valid instantiation and how it depends on the multiplicity values. Previous work [15] has proven the following instantiation equation. If no multiple edges for associations not tied to an association class are allowed in the object diagram, the class diagram in *Fig. 9* can be instantiated by na objects of type A and nb objects of type B, where n is an arbitrary positive integer.

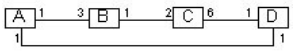
For computing the allowed numbers of the participating objects in the whole instantiation model graph we consider a specific representation derived from the metamodel, which is referred to as Incidence Matrix with Multiplicity (IMM) [15].

An example for the algorithm creating IMM (*Fig. 10*) is illustrated in *Fig. 11*.

CREATEIMM( )

1 Traverse the model graph
2 Take each edge e_j.
3 If the v_k and v_l are the vertices incident upon e_j, then set IMM_{kj} to the v_k side multiplicity and IMM_{lj} to the v_l side multiplicity of e_j.

Fig. 10. Algorithm for creating the IMM matrix



$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 3 & 1 & 0 & 0 \\ 0 & 2 & 6 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Fig. 11. An example for creating IMM

IMM can be considered as a short representation of the equations established for each node based on the instantiation equation. Our example (*Fig. 11*) can be written as follows:

$$\begin{aligned} A &: x_0 = x_3 \\ B &: 3x_0 = x_1 \\ C &: 2x_1 = 6x_2 \\ D &: x_2 = x_3 \end{aligned} \tag{15}$$

Obviously, in (15) above, all the x_i variables are nonzero integers. To solve this sort of equations an elimination algorithm is available as it is depicted in *Fig. 12*.

After the elimination the 0th row of the IMM contains a factor f_i for each edge e_i. Each m_1, m_2 multiplicity on an edge e_i must be multiplied by factor f_i. The number of the m_1 side node can be $m_1 f_i k$, where k is an arbitrary nonzero integer and the cardinality of the other node can be computed similarly. An example for the elimination algorithm is shown in *Fig. 13*

It can be seen that the rows of the IMM matrix represent the equations to be solved. In order to prove that the IMM algorithm provides all the correct solutions to the problem, we have to examine how the steps of the algorithm influence the equations.

PROPOSITION 2 *The matrix resulted by an arbitrary elimination step represents an equation set which is equivalent to the multiplicity equations established according to the instantiation equation. Furthermore, there is no solution which can form a valid instantiation, but it is not among the results of the IMM algorithm.*

Proof. The IMM constructed by the *CreateIMM* procedure is a representation of the equation established according to the instantiation equation such that each row of

```
ELIMINATION(IMM imm)
 1 for each j column index
 2       r0=index of the row containing the first nonzero element
 3       r1=index of the row containing the second nonzero element
 4       if there is no r0 and r1 then break
 5       lcm=LCM(imm[r0,j],imm[r1,j])
 6       MultiplyRow(r0, lcm/imm[r0,j])
 7       MultiplyRow(r1, lcm/imm[r1,j])
 8       for each j2 column index
 9            if imm[r0,j2]= = imm[r1,j2] then
10                 imm[r1,j2]=0
11            else if imm[r0,j2]!=0 and imm[r1,j2]!=0 then
12                 Error: Inconsistent parallel paths. No instantiation exists.
13            else if imm[r1,j2]!=0
14                 imm[r0,j2]=imm[r1,j2];
15                 imm[r1,j2]=0;
16            end if
17       end for
18 end for
19 rowlcm=LCM of the 0th row of imm
20 for each j column index
21       imm[0,j]=rowlcm/ imm[0,j]
22 end for
```

Fig. 12. The IMM elimination algorithm

the IMM contains an equation. Because the distinction between the set of equations and the IMM lies with the representation only, this is equivalent to the original set of equations. Line 6 and 7 of the elimination algorithm multiplies two equations by an integer, which results in an equivalent set of equations. Lines 8–17 merge two equations

$$ax_1 = a_2x_2 = ... = a_nx_n \tag{16}$$

$$ax_1 = b_2x_2 = ... = b_nx_n \tag{17}$$

into one equation:

$$ax_1 = c_2x_2 = ... = c_nx_n. \tag{18}$$

This merging step can be accomplished if there is no contradiction between the individual coefficients. The value zero means that the variable is not concerned, so zero can always be replaced during the merging process. But if there are two coefficients different form zero at the same position in the two matrix rows, merging cannot be completed because of contradiction. But if merging can be performed, the new equation remains equivalent to the unmerged ones, which follows from the

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 3 & 1 & 0 & 0 \\ 0 & 2 & 6 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & 0 & 0 & 3 \\ 3 & 1 & 0 & 0 \\ 0 & 2 & 6 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}, \quad \begin{bmatrix} 3 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 6 & 2 & 0 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix},$$

$$\begin{bmatrix} 6 & 2 & 6 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 6 & 2 & 6 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 6 \end{bmatrix}, \quad \begin{bmatrix} 6 & 2 & 6 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \Rightarrow (\text{LCM} = 6),$$

$$\begin{bmatrix} 1 & 3 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \Rightarrow \begin{matrix} x_0 = 1 \\ x_1 = 3 \\ x_2 = 1 \\ x_3 = 1 \end{matrix} \Rightarrow \begin{matrix} A : 1n \\ B : 3n \\ C : 6n \\ D : 1n \end{matrix}$$

Fig. 13. An example for the IMM elimination algorithm

transitive property of the equality. Lines 20–22 replace the coefficients with the solution based on (19).

$$d_1 x_1 = d_2 x_2 = ... = d_n x_n = nLCM(d_1, d_2, ..., d_n) \tag{19}$$

To prove the second part of the proposition, we assume that there is a solution vector X which forms a valid instantiation, but it is not a part of the solution provided by the IMM elimination algorithm. If X can occur in a valid instantiation it has to satisfy the equations appearing in the instantiation equation. As it has been proven above, the set of equations remain equivalent during the elimination process, it has to be the solution of the equation set resulted by the elimination process as well.

4.2. The Metamodel-Based Matching Algorithm

In *Fig. 14* two alternatives are depicted as valid matches for a pattern containing multiplicities more than one on both side of the association. This inherent non-determinism follows from the UML standard itself, where both cases are a valid instantiation of the class diagram.

Our experience has shown that it is enough to examine only one block (the case where $n = 1$ in the previous section) for practical applications. Thus the algorithm provided in this section considers that case only.

In order to get closer to metamodel-based matching we analyse the VF2 algorithm. It can be observed that each recursive level deals with an LHS element: it attempts to match all the possible pairs including the LHS element in a particular matching situation. For a certain level the COMPUTECANDIDATEPAIRS() function generates the pairs to be tested. The main difference between the metamodel-based

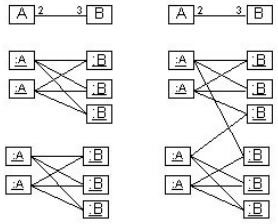


Fig. 14. Two instantiation possibilities

and the basic case is that metamodel elements specify more than one input model node to be matched, and this number can be dynamic (i.e. depending on the host graph topology). This issue is resolved by placing virtual nodes in the candidate list in accordance with the results of the previous section. We define a so-called precedence list for the nodes connected to the current pattern element. The precedence list consists of nodes and an integer value assigned to each node, which denotes the multiplicity. Each node is placed in the list with the maximum multiplicity. If the match algorithm realizes that there is no way further, the IS_FULL_MATCH function checks whether the current assignment is a valid match (*Fig. 15*).

```
UMLMATCH(Hashtable out hLHSHost, ArrayList out precedenceList, int
modelID, int ruleID): bool
 1 if IS_FULL_MATCH(hLHSHost, ruleID) then
 2      SAVE_MATCH(hLHSHost)
 3      return true
 4 end if
 5 if precedenceList.Count > 0 then
 6       if not HANDLE_PRECEDENCE_ITEM(precedenceList[0]) then return
        false
 7      precedenceList[0].MaxMultiplicity = precedenceList[0].MaxMultiplicity-1
 8      if precedenceList[0].MaxMultiplicity == 0 and establish multiplicities with
        IMM then
 9           hLHSHost.Add(precedenceList[0].LHSNodeID, precedenceList[0].
             HostNodeIDs)
10           precedenceList.Remove(precedenceList[0])
11      end if
12      if (UMLMATCH (hLHSHost, precedenceList, modelID, ruleID)) then
        return true
13      else
14           hLHSHost.Remove(precedenceList.LHSNodeID)
15           return false
16      end if
17 else
```

```
18      VMTSCandidatePair[] candidatePairs = UML
        COMPUTE_CANDIDATEPAIRS(hLHSHost, modelID, ruleID)
19      foreach VMTSCandidatePair candidatePair in candidatePairs
20          if UMLFEASIBLE(hLHSHost, candidatePair) then
21              if candidatePair.MaxMultiplicity == 0 then
22                  hLHSHost.Add(candidatePair.LHSNodeID,
                    candidatePair.HostNodeID)
23              else
24                  precedenceList.Add(candidatePair, candidatePairs)
25              end if
26              if (UMLMATCH (hLHSHost, precedenceList, modelID,
                ruleID)) then return true
27              else
28                  hLHSHost.Remove(candidatePair.LHSNodeID)
29                  return false
30              end if
31          end if
32      end foreach
33  end if
```

bool UMLFEASIBLE (Match M, HostGraphNode n, PatternNode m)

```
 1 foreach matchPair in M
 2      foreach hostNode in matchPair
 3          if not CHECKTWOPOINT(hostNode, matchPair.LHSNode, n, m)
            then return false
 4      end foreach
 5 end foreach
 6 foreach matchPair in M
 7      foreach hostNode in matchPair
 8          foreach lhsNeighbourNode in GETLHSNEIGHBOURNODES
            (matchPair.LHSNode)
 9              if ISEDGEBETWEENRULENODES(lhsNeighbourNode, m) then
10                  bHasProperNode = false
11                      foreach hostNeighbourNode in
                        GETHOSTNEIGHBOURNODES (matchPair.HostNode)
12                      if HASPROPEREDGEBETWEENNODES
                        hostNeighbourNode, lhsNeighbourNode, n, m)
                        then bHasProperNode = true
13                  end foreach
14                  if not bHasProperNode return false
15              end if
16          end foreach
17      end foreach
18 end foreach
```

```
19 foreach lhsNeighbourNode in GETLHSNEIGHBOURNODES (m)
20     if ISEDGEBETWEENRULENODES(lhsNeighbourNode, m) then
21     bHasProperNode = false
22         foreach hostNeighbourNode in GETHOSTNEIGHBOURNODES(n)
23             if HASPROPEREDGEBETWEENNODES(hostNeighbourNode,
               lhsNeighbourNode, n, m) then bHasProperNode = true
24         end foreach
25         if not bHasProperNode return false
26     end if
27 end foreach
28 return true
```

UMLCOMPUTE_CANDIDATEPAIRS()

if $T_{\mathrm{H}}^{\mathrm{out}} \neq \emptyset \wedge PL^{\mathrm{out}} \neq \emptyset$ **then** $P = T_{\mathrm{H}}^{\mathrm{out}} \times \{\min PL^{\mathrm{out}}\}$
else if $T_{\mathrm{H}}^{\mathrm{in}} \neq \emptyset \wedge PL^{\mathrm{in}} \neq \emptyset$ **then** $P = T_{\mathrm{H}}^{\mathrm{in}} \times \{\min PL^{\mathrm{in}}\}$
else if $T_{\mathrm{H}}^{\mathrm{out}} \neq \emptyset \wedge T_{\mathrm{LHS}}^{\mathrm{out}} \neq \emptyset$ **then** $P = T_{\mathrm{H}}^{\mathrm{out}} \times \{\min T_{LHS}^{\mathrm{out}}\}$
else if $T_{\mathrm{H}}^{in} \neq \emptyset \wedge T_{\mathrm{LHS}}^{\mathrm{in}} \neq \emptyset$ **then** $P = T_{\mathrm{H}}^{\mathrm{in}} \times \{\min T_{\mathrm{LHS}}^{\mathrm{in}}\}$
else $P = \mathit{Instanceof}\,(\min(V^{\mathrm{LHS}} - M^{\mathrm{LHS}})) \times \{\min(V^{\mathrm{LHS}} - M^{\mathrm{LHS}})\}$
end if

Fig. 15. The UML-based pattern matching algorithm

The final algorithm has another optimization step compared to the original VF2 philosophy: only the connected elements are considered among the neighbour nodes when the feasibility of the algorithm is investigated.

The matching algorithm and the role of the precedence list in this process are illustrated via an example. In *Fig. 16* a metamodel and the related instance model is depicted. With the help of this construct we present how the matching algorithm parses this instance model, assuming that the LHS graph of the rewriting rule is the same as the metamodel.

The transformation contains the rewriting rules, the ID of the model to which the transformation has to be applied and a list with the pivot nodes which select the nodes in the LHS graphs, where the algorithm has to start the pattern matching. These parameters are passed to the matching algorithm. In the present case the transformation contains only one rewriting rule, and it is assumed that there is no passed pivot node, therefore the algorithm selects the pivot node automatically based on the statistics information. The node A will be the pivot node, because it has the fewest instances. In the first step the algorithm selects the instance nodes with type$A$ ($A1$), it adds the $A - A1$ pair to the match and calls recursively the matching method. In the second step the algorithm searches for suitable B type nodes in the instance model based on the LHS graph, therefore it selects the neighbour nodes of the $A1$ node with B type: $B1$, $B2$ and $B3$. It means that the algorithm has

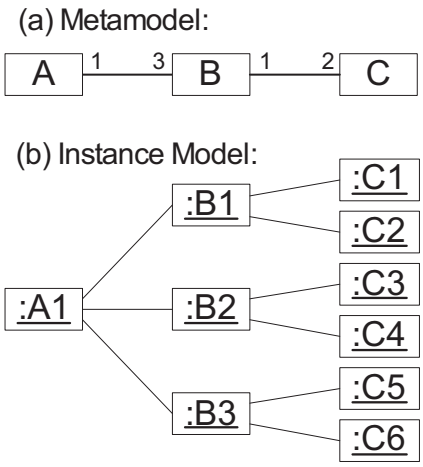


Fig. 16. (a) Metamodel, (b) Instance Model of the Precedence List example

three possible pairs: $B - B1$, $B - B2$ and $B - B3$. The algorithm selects e.g. the $B - B1$ pair and validates the following conditions for this possible pair: checks the actual pair (i) against the actual match, (ii) against the nodes which are adjacent the match and (iii) against the nodes which are adjacent to the actual pair ($B - B1$). The actual pair is feasible, hence the algorithm checks the multiplicity, which is 3, thus it does not add the actual pair to the match but creates a precedence item and adds it to the precedence list. The precedence item in the current case contains the following information: the LHS node which is already in the match (A), the actual LHS node (B), the edge between them (the edge in metamodel between the A and B nodes), the already checked (feasible) instance nodes ($B1$), the possible instance nodes ($B2$, $B3$) and the number of the instance nodes the algorithm has to find to satisfy the expected multiplicity ($3 - 1 = 2$). After the precedence item creation the algorithm calls itself recursively again. In this call the algorithm notices that the precedence list is not empty, therefore it tries to match the elements in the list: the algorithm finds the feasible instance node (e.g. $B2$) in the list of possible instance nodes, and decreases the number of the searched instance nodes to 1. In the fourth step the $B3$ instance node is the next feasible node, and the algorithm decreases the number of the searched instance nodes again, which in this step becomes 0, hence the algorithm removes the precedence item from the precedence list and adds the following pair to the actual match: $B - B1$, $B2$, $B3$. In the fifth step the algorithm searches for suitable C type nodes in the instance model: $C1$, $C2$, $C3$, $C4$, $C5$ and $C6$. The algorithm creates precedence item again, and it processes the C type instance nodes step by step, and finally adds the last pair to the match: $C - C1$, $C2$, $C3$, $C4$, $C5$, $C6$. *Table 1* presents the process of the matching step by step.

Table 1. The matching process

	Actual LHS Node	Actual Match	Actual Item of the Precedence List		
			Already Checked Nodes	Possible Nodes	Number of Necessary Nodes
Step 1	A	A – A1	–	–	–
Step 2	B	A – A1	B1	B2, B3	2
Step 3	B	A – A1	B1, B2	B3	1
Step 4	B	A – A1 B – B1, B2, B3	–	–	–
Step 5	C	A – A1 B – B1, B2, B3	C1	C2, C3, C4, C5, C6	5
Step 6	C	A – A1 B – B1, B2, B3	C1, C2	C3, C4, C5, C6	4
Step 7	C	A – A1 B – B1, B2, B3	C1, C2, C3	C4, C5, C6	3
Step 8	C	A – A1 B – B1, B2, B3	C1, C2, C3, C4	C5, C6	2
Step 9	C	A – A1 B – B1, B2, B3	C1, C2, C3, C4, C5	C6	1
Step 10	C	A – A1 B – B1, B2, B3 C – C1, C2, C3, C4, C5, C6	–	–	–

5. A Case Study

To illustrate the metamodel-based matching a case study is provided. In *Fig. 17* a sample rewriting rule and a statechart model are depicted, which are the input models of the transformation.

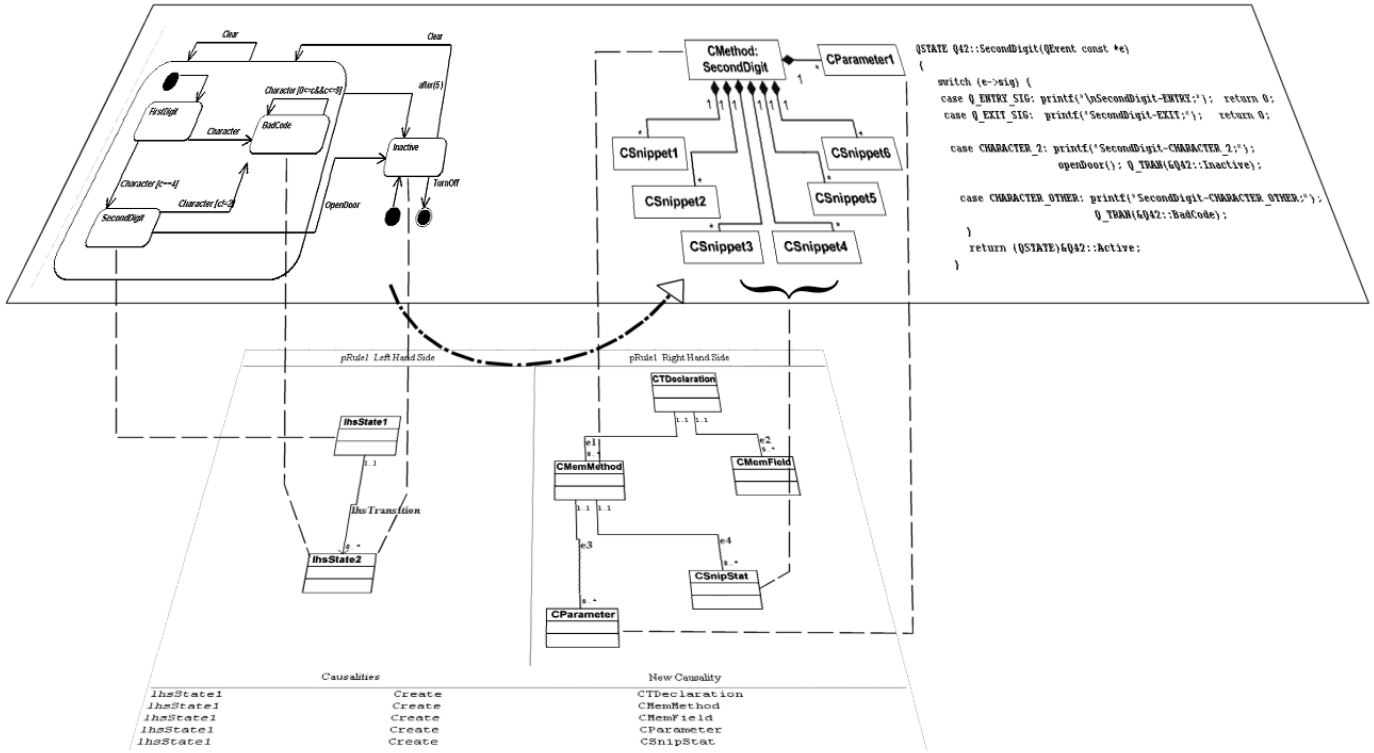


Fig. 17. Metamodel-based matching

Recall that in VMTS it is possible that the LHS and the RHS graphs of a rewriting rule have different metamodels. In the rewriting rule of the case study the metamodel of the LHS is the Statechart metamodel [VMTS] [OMG UML], and the metamodel of the RHS is the CodeDOM metamodel [VMTS] [12], which is a high level source code representation for the C++ language. On the LHS of the rewriting rule there are two states whose meta type is the statechart state, and there is a transition between them with a 0..* multiplicity on the side of the target state. It means that exhaustively applying this rewriting rule for a statechart model, it will match all the states with their adjacent target states. The rule has to match the accessible adjacent states, because we need them to generate the state-transitions in the source code. Of course it is possible that a state has no outgoing transition, and the reason why we enable the 0 multiplicity is that we want to match states having only incoming transitions to generate CodeDOM tree for them. On the RHS of the rewriting rule *CTypeDeclaration* represents a type declaration for a class, structure, interface or enumeration. *CMemberField* can be used to denote the declaration for a field of a type, and *CMemberMethod* to phrase the declaration for a method. *CParameter* represents a parameter declaration for a method, property, or constructor, and *CSnippetStatement* means a statement using a literal code fragment.

The code generation is a syntax tree generation from which the .NET Framework generates the source code. We generate C++ code for the Quantum Framework (qF) [QF], which is a real-time, event-driven, state machine-based application framework for embedded systems.

In *Fig. 17* the dashed lines denote a sample match. The *Second digit* state is matched against the *lhsState1* and the *Bad code* and *Inactive* states against the *lhsState2*. Based on the actual match, the transformation creates the CodeDOM tree (*Fig. 17*) and with the help of the .NET Framework it generates the source code (*Fig. 18*).

```
QSTATE Q42::SecondDigit(QEvent const *e) {
    switch (e->sig) {
    case Q_ENTRY_SIG: printf("\nSecondDigit-ENTRY;"); return 0;
    case Q_EXIT_SIG: printf("SecondDigit-EXIT;"); return 0;


    case CHARACTER_2:  printf("SecondDigit-CHARACTER_2;");
    openDoor();
    Q_TRAN(&Q42::Inactive);
    case CHARACTER_OTHER: printf("SecondDigit-CHARACTER_OTHER;");
    Q_TRAN(&Q42::BadCode);
    }
    return (QSTATE)&Q42::Active;
}
```

Fig. 18. Part of the generated source code for the example

6. Conclusions and Future Work

An algorithmic background for metamodel-based model transformation has been contributed in this paper. The matching process has been accelerated with type-aware heuristics, which uses the naturally available metamodel of the host graph. Then the IMM elimination algorithm was provided which is suitable for predicting the possible number of objects participating in a valid instantiation of a specific class diagram. The correctness of the algorithm has also been proven. In order to find a match for LHS consisting of metamodel elements, a pattern matching algorithm has been provided, which is an enhanced version of the VF2 subgraph isomorphism algorithm.

These algorithms take the advantage of the availability of the metamodels in environments like GME, GReAT and VMTS. In the proposed solution the memory requirements are higher, but the metamodels are naturally available because of the technique used in the transformation. Considering the performance issues, Ullmann's algorithm is very popular, but performance measurements [3] have shown that better performance can be achieved using VF2. These evaluation measurements, however, have taken place on randomly generated graphs. The examination of UML and DSL models is subject of future research. Further work includes devising new heuristics based on metamodel parameters and instantiation statistics.

7. Acknowledgements

The fund of Microsoft Research (MSR-27334) has supported, in part, the activities described in this paper. The authors are grateful to László Lengyel for coding and testing the proof-of-concept implementation of the discussed algorithms, and for the valuable feedbacks.

References

[1] CORDELLA, L. P. – FOGGIA, P. – SANSONE, C. – VENTO, M., An Improved Algorithm for Matching Large Graphs, *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, Ischia, May 23–25, pp. 149–159, 2001.

[2] CORDELLA, L. P. – FOGGIA, P. – SANSONE, C. – VENTO, M., Performance Evaluation of the VF Graph Matching Algorithm, *Proc. of the 10th ICIAP*, IEEE Computer Society Press, **2** (1999), pp. 1038–1041.

[3] FOGGIA, P. – SANSONE, C. – VENTO, M., A Performance Comparison of Five Algorithms for Graph Isomorphism, *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, Ischia, May 23–25, pp. 188–199, 2001.

[4] LEVENDOVSZKY, T. – KARSAI, G. – MAROTI, M. – LEDECZI, A. – CHARAF, H., Model Reuse with Metamodel-Based Transformations, *Lecture Notes in Computer Science – ICSR7*, Austin, TX, April 18, 2002.

[5] VARRÓ, D. – VARRÓ, G. – PATARICZA, A., Designing the Automatic Transformation of Visual Languages, *Science of Computer Programming*, **44** (2002), pp. 205–227.

[6] ZÜNDORF, A., 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.

[7] ROZENBERG, G. (ed.), *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, Vol.1 World Scientific, Singapore, 1997.
[8] SZTIPANOVITS, J. – KARSAI, G., Model-Integrated Computing, *IEEE Computer*, pp. 110–112, April, 1997.
[9] KARSAI, G. – AGRAWAL, A. – SHI, F., On the Use of Graph Transformation in the Formal Specification of Model Interpreters, *Journal of Universal Computer Science*, Special issue on Formal Specification of CBS, 2003.
[10] LEVENDOVSZKY, T. – LENGYEL, L. – CHARAF, H., *Software Composition with a Multipurpose Modeling and Model Transformation Framework*, IASTED 2004, Innsbruck, 2004, pp. 590–594.
[11] The VMTS Homepage. http://avalon.aut.bme.hu/∼tihamer/research/vmts/
[12] LEVENDOVSZKY, T. – LENGYEL, L. – MEZEI, G. – CHARAF, H., A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, *Electronic Notes in Theoretical Computer Science*, International Workshop on Graph-Based Tools (GraBaTs) Rome, 2004.
[13] Object Management Group, Unified Modeling Language Specification, v1.5, www.uml.org.
[14] ULLMANN, J. R., An Algorithm for Subgraph Isomorphism, *Journal of the ACM*, **23** No. 1 (1976), pp. 31–42.
[15] LEVENDOVSZKY, T. – LENGYEL, L. – CHARAF, H., A UML Class Diagram-Based Pattern Language for Model Transformation Systems, *WSEAS Transactions on Computers*, 2005 ISSN: 110-92750 **4** (2) (2005), pp. 190–195.
[16] MDA Guide Version 1.0.1, OMG, document number: omg/2003-06-01, 12th June 2003, www.omg.org/docs/omg/03-06-01.pdf
[17] SAMEK, M., Practical Statecharts in C/C++", CMP Books, 2002, http://www.quantum-leaps.com/qf.htm