

# Electronic Communications of the EASST Volume 42 (2011)



## Proceedings of the 4th International Workshop on Multi-Paradigm Modeling (MPM 2010)

### Automated Verification by Declarative Description of Graph Rewriting-Based Model Transformations

Márk Asztalos, Péter Ekler, László Lengyel, Tihamér Levendovszky, Gergely Mezei,  
Tamás Mészáros

14 pages

Guest Editors: Vasco Amaral, Hans Vangheluwe, Cécile Hardebolle, Lazlo Lengyel

Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer

ECEASST Home Page: <http://www.easst.org/eceasst/>

ISSN 1863-2122

# Automated Verification by Declarative Description of Graph Rewriting-Based Model Transformations \*

Márk Asztalos, Péter Ekler, László Lengyel, Tihamér Levendovszky,  
Gergely Mezei, Tamás Mészáros

Department of Automation and Applied Informatics  
Budapest University of Technology and Economics, Budapest 1111, Hungary  
{asztalos, peter.ekler, lengyel, tihamer, gmezei, mesztam}@aut.bme.hu

**Abstract:** Usually, verification of graph rewriting-based model transformations is performed manually, however, the industrial applications require automated methods. In several cases, transformation developers are interested in the offline analysis, when only the definition of the transformation and the specification of the modeling languages are taken into account. Hence, the analysis must be performed only once, and the results are independent from the concrete input models. For this purpose, transformations should be specified in a formalism that can be automatically analyzed. Based on our previous work that presented the mathematical background, this paper provides a platform-independent, declarative formalism for the specification of graph rewriting-based model transformations, and demonstrates its applicability on a case study of refactoring mobile-based social network models. Our results prove that several functional properties of the model transformations can be automatically verified, moreover, the capabilities of our methods can be extended in the future.

**Keywords:** graph rewriting-based transformation; automated verification

## 1 Introduction

With the increasing need of reliable systems, the verification of model processing programs has become a fundamental issue in model-based software engineering. In our terminology, a *model transformation* covers the definition of a model processing program that is based on graph rewriting systems [EEPT06] and is specified by a set of rewriting rules (based on the double-pushout approach, DPO) as well as an additional control structure that explicitly defines the execution order of the rules.

When they are feasible, *offline* verification methods are extremely useful in industrial applications. A verification technique is called *offline* if only the definition of the program and the specification of the languages that describe the models to be transformed

---

\* This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. This work is connected to the scientific program of the "Development of quality-oriented and harmonized R+D+I strategy and functional model at BUTE" project. This project is supported by the Hungarian Academy of Sciences - the Office for Subsidized Research Units and by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

are used during the analysis process. Therefore, the results of the analysis are general in the sense that they are independent from the concrete input models. Although the offline analysis is very complex (e.g. the termination of a graph transformation is undecidable in general [Plu98]), it must be performed only once. Several techniques can be found for the verification of graph rewriting-based model transformations, however, these methods usually lack generalization possibilities, since the analysis is performed manually or the methods can be applied only to a certain transformation class or to the analysis of a certain type of property only. Therefore, there is an increasing need for automated verification methods and tools. The goal of our research is to provide an offline, automated verification framework for the analysis of graph rewriting-based model transformations. For these purposes, model transformations need to be formalized in a way that their verification could be performed by automated methods. This paper outlines the concept of our verification approach, and we introduce a formalism for the declarative description of model transformations.

The rest of the paper is organized as follows: [Section 2](#) presents a case study of refactoring mobile-based social network models, which will be used to demonstrate our methods. [Section 3](#) introduces the main concepts of our verification methods informally. After the mathematical background is presented in [Section 4](#), we provide the contribution of this paper, the formalism for the specification of model transformations in [Section 5](#). We demonstrate the usability of our formalism on the case study in. Finally, we describe the related work in [Section 6](#) and summarize our results in [Section 7](#).

## 2 Application Domain of Mobile-Based Social Networks

In the mobile devices, phone books represent social relationships that can be integrated into social networks. *PhoneBookMark* is a mobile-based social network implementation by Nokia Siemens Networks [EL10], which provides methods for the automated synchronization of the data in the phone books of the members with the public data of other members. We took part in *PhoneBookMark* project and, before the public introduction, it was available for a group of general users (420 registered members with more than 72000 private contacts).

Visual Modeling and Transformation System (VMTS) [VMT] is a metamodeling and model transformation framework. We have created a domain-specific modeling environment for *PhoneBookMark* in VMTS. The entities of its metamodel are presented in [Figure 1a](#): a *member* is a user of the social network, a *phone* is a mobile device of a member, which can contain phone book entries, a *contact* corresponds to a phone book entry of a phone. Relations between the entities have also been defined: each member can own several phones (*PhoneOwnerConnection*), each phone can contain several contacts (*ContactContainment*). A contact can be connected to a member with a *CustomizedConnection* or a *SimilarityConnection* edge. A *CustomizedConnection*, or shortly customization edge, means that the current entry corresponds to the member of the social network. Whenever the owner member of the entry connects to the network, the data can be synchronized. *PhoneBookMark* provides a semi-automatic similarity detect-

ing and resolving mechanism, which detects similarities between phone book contacts and the members of the network. Similarity means that the algorithm suggest to the user that the contact and the member represent the same person. In this case, a *SimilarityConnection*, or shortly, a similarity edge is created between the contact and the appropriate member, later, the user has to decide the acceptance of this relation. For this purpose, *ApprovalState* attribute has been defined for similarity edges, whose value can be *approved*, *rejected*, or, the default value, *ignored*, which means that the user has not made a decision yet. During the refactoring of a model, approved edges will be converted to customization edges and rejected edges will be deleted from the model. In VMTS, the domain-specific environment for *PhoneBookMark* includes the metamodel and a concrete syntax for the instance models. A sample model is presented in Figure 1b.

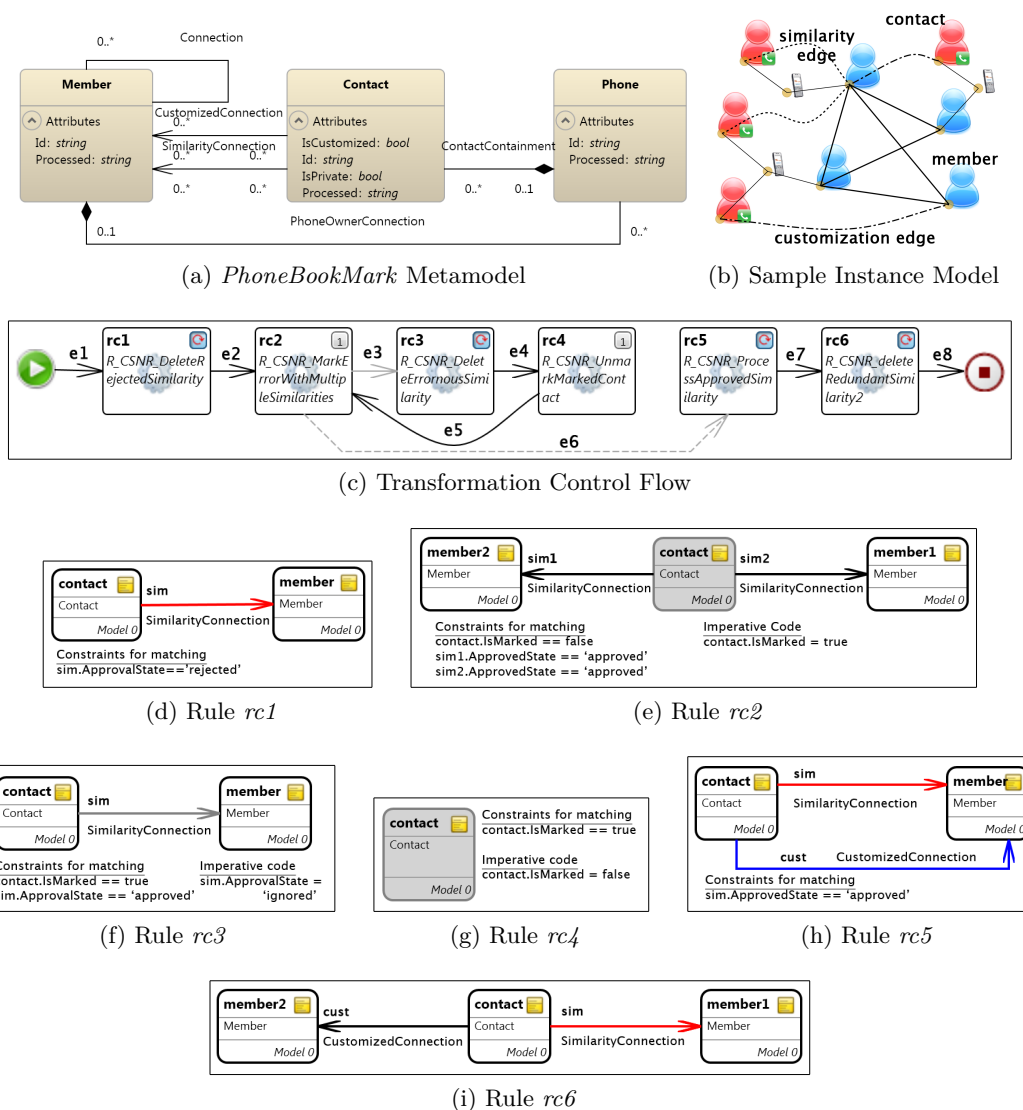


Figure 1: *PhoneBookMark* Domain and Similarity Refactoring Transformation in VMTS

In VMTS, the model transformations are based on graph rewriting and are defined with the use of two modeling languages: the Visual Control Flow Language (VCFL) and the Visual Transformation Definition Language (VTDL) [VMT]. The activity diagram-like VCFL models controls the execution order of the rewriting rules, while the rewriting rules are described with VTDL models. The application of the rules is based on the double pushout approach [EEPT06]. We have implemented a model transformation (*Similarity Handling Transformation*) that refactors *PhoneBookMark* models, more precisely, it processes the similarity edges, where processing means the deletion of the rejected edges and the conversion of the approved edges into customization edges. This refactoring step is needed to maintain *PhoneBookMark* models and to upgrade existing social network models to support mobile-based features as well.

The control flow graph of the transformation, as implemented in VMTS, is presented in Figure 1c. The dashed, gray control flow edges are followed if the application of the source rules was unsuccessful, which happens when no matches of the left-hand side can be found. The solid, gray edges are followed if the application of the previous rule was successful, while solid black edges are always followed. Rules with a circle in the top right bottom are executed exhaustively, which means that the rules are applied repeatedly, until they cannot be applied any more. Figure 1 contains the definition of the rules of the transformation. In VMTS, the left-hand side and right-hand side of the rules are merged, elements that are deleted by the rule are red, newly created are blue and the attributes of gray elements are modified. In Figure 1, we show the definition of the rules in VMTS along with the attached constraints and imperative code for the modification of the attributes. This model transformation will be used to demonstrate our verification concept.

### 3 Overview of Automated Verification

In this section, we provide the informal outline of our verification approach based on our previous work [ALL10]. As mentioned earlier, we restrict ourselves to the analysis of model processing programs based on graph rewriting systems, which are defined by a set of rewriting rules and an additional control structure. We also mentioned that we call such a program a *model transformation*. We assume that the control structure is a directed control flow graph, which conforms to certain conditions: (i) start node and end nodes are used to mark the starting point and possible end points of the transformations, other nodes of the transformation are the rewriting rules; (ii) the flow edges of the control flow graph define the execution order, branches can be defined. Moreover, we assume that the output of the execution is always the modified input model (in-place transformation). However, it is not a restriction of the generality of the model transformation, since assuming that we have multiple input and multiple output models, we can always compose their union and treat them as a single model.

Assume that we have a language (MCDL [AELL10b]) that is able to express the verifiable properties of the output models of the transformations. For example, given the Similarity Handling Transformation presented in Section 2, we may need to express the

following properties of the output models (the properties themselves are in *italic*): **(v1)** After the application of the transformation, *no approved similarity edge should be present in the model*, because each approved edge should be transformed to a customization edge, or should be deleted if there is more than one approved similarity edge from the same contact. **(v2)** After the application of the transformation, *no rejected similarity edge should be present in the model*, since all rejected similarity edges should be deleted. **(v3)** After the application of the transformation, *it is forbidden that a contact has a similarity and a customization edge at a time*, because, in this case, the similarity edge should have been deleted. **(v4)** *After the application of the transformation, it is forbidden that a contact has two customization edges at a time, provided that this pattern was also forbidden before that transformation started*. This would result in an inconsistent state. Assume that MCDL is able to express verifiable properties such as the ones presented above. We detail some simple MCDL expressions formally in [Section 5](#), but to illustrate the capabilities of this language, we mention that basically MCDL is able to express (i) static properties of models, e.g. that a certain pattern exists, or not exists in the model, (ii) dynamic properties of model transformations, e.g. that each instance of pattern  $P$  will be transformed to pattern  $P'$ . (iii) Moreover, the expression and analysis of non-functional properties, mainly the termination is subject of research.

Model Condition Inference Logic (MCIL) [[ALL10](#)] is an inference logic that is able to analyze logical implications such as  $\phi_1 \Rightarrow \phi_2$ , where Greek letters denote MCDL formulae. The result of the analysis can be the proof, the refutation, or the result that the implication is undecidable. For example, given an MCDL formula  $\varphi$  stating that after the application of the transformation, *the approval state of all similarity edges in the model will be ignored*, we can derive the first two verifiable formulae presented above from  $\varphi$ , because if all similarity edges are in ignored state, it implies that there are neither approved nor rejected edges. MCIL consists of several extensible deduction rules.

The main concept of our verification approach is the *assignments of formulae*, or *discovery algorithm*: given a model transformation, assume that we are able to assign MCDL formulae to each control flow edge such that given a flow edge  $f$ , the formula  $\phi_f$  assigned to  $f$  is a property that is satisfied by the model under transformation at its current state when the execution of the transformation reaches  $f$ . Assuming that we have only one end node in the control flow and it has only one incoming edge, and the formula  $\phi_{final}$  is assigned to this edge.  $\phi_{final}$  will be satisfied by all possible output models of the transformation. During the analysis of a model transformation, the goal of our methods is to produce these *assignments*. Its main benefit is as follows: given a property of the output models that should be validated, which is described as an MCDL expression  $\phi_{ver}$ , if we can prove  $\phi_{final} \Rightarrow \phi_{ver}$ , then the property is validated. Another benefit of the assignment is that we assign formulae not only to the edge before the end nodes, but to all flow edges, which helps locating the problematic points while debugging.

Obviously, the main question is how to produce such an assignment, i.e. how to *discover the formulae* on the control flow edges. We present the main concept of the recursive discovery algorithm in the following. The start node of the transformation has one outgoing edge. The formula assigned to this edge is called the initial formula. This is known, since this is the condition that must be satisfied by all possible input

models. Assume that we have a rule in the transformation with several incoming edges and one outgoing edge, and we have already assigned formulae to the incoming edges. In other words, we know some properties of the model under transformation when the execution reaches a rule. We have the formal definition of the rule, therefore, by its definition, we may derive certain properties that will be true after the application of the rule. These properties described in MCDL can be assigned to the outgoing edge. This method is called the *propagation of formulae through a rule*, which is a very complex task itself, it depends on the MCDL formulae, and the definition of the rule. The goal of our methods is to collect the most information in the formulae that are assigned to the edges. However, if nothing can be derived, it will not imply the failure of our algorithm, only that the assigned formulae will not contain relevant information, therefore, the verifiable properties could not be derived by MCIL.

The formalism to specify the model transformations largely influences the efficiency of the *propagation* and the *discovery* algorithms. The main contribution of this paper is a formalism for specifying the rules and the control flows in a declarative way such that the algorithms could be defined. The presentation of the algorithms themselves would exceed the limits of this paper, but we outline their operation on the case study in order to demonstrate the applicability of the presented formalism.

## 4 Mathematical Background

This section summarizes previously presented [AEL<sup>+</sup>10] definitions based on typed graphs [BELT04, EEPT06], which provides the mathematical background for the contributions of this paper. The main components what we formalize here are: (i) *metamodels*: types of entities and relations, with the names of the attributes; (ii) *models and patterns*: entities and relations typed over a metamodel along with abstract *attribute constraints*; (iii) *weakly typed morphisms*: formal mapping between patterns or between patterns and models.

**Definition 1** (type graph with inheritance and inheritance clans) A **type graph with inheritance** is a double  $(G_T, I)$  consisting of a type graph  $G_T = (N, E, s, t)$  (with a set  $N$  of nodes, a set  $E$  of edges, source and target functions  $s, t : E \rightarrow N$  for edges), and inheritance graph  $I$  sharing the same set of nodes  $N$ . For each node  $n$  in  $N$  ( $N \equiv N_I \equiv N_G$ ), the **inheritance clan** is defined by  $clan_I(n) = \{n' \in N \mid \exists \text{ path } n' \rightarrow^* n \text{ in } I\}$  where path of length 0 is included, i.e.  $n \in clan_I(n)$ .

**Definition 2** (clan morphism and instance graphs) Given a type graph with inheritance  $T = (G, I)$ , and a graph  $H$ , a **clan morphism**  $\tau : H \rightarrow T$  consists of two functions  $\tau_N : N_H \rightarrow N_G$ ,  $\tau_E : E_H \rightarrow E_G$  such that: (i)  $\forall e \in E_H : \tau_N \circ s_H(e) \in clan_I(s_G \circ \tau_E(e))$ , and (ii)  $\forall e \in E_H : \tau_N \circ t_H(e) \in clan_I(t_G \circ \tau_E(e))$ . Given a type graph (with inheritance)  $T$ , a double  $(G, \tau)$  of a graph  $G$  along with a clan morphism  $\tau : G \rightarrow T$  is called an **instance of  $T$** .  $G$  is said to be typed over  $T$ .

Given a node  $n$  in a type graph,  $clan_I(n)$  is the set of nodes that are inherited from  $n$ ,

moreover,  $\text{clan}_T(n)$  also contains  $n$ . An instance of a type graph (i.e. a graph typed over a concrete type graph) is defined by the instance graph itself and a special type morphism that assigns an element of the type graph to each element of the instance graph. The type morphism should take inheritance into account and is called clan morphism.

**Definition 3** (metamodel interface) A **metamodel interface**  $\mathfrak{M}$  is a triple  $(T, \mathcal{A}, \sigma)$ , where  $T$  is a type graph,  $\mathcal{A}$  is a set of attribute names that are defined on the elements of the type graph, and  $\sigma : L_T \rightarrow 2^{\mathcal{A}}$  is the attribute assignment function that assigns a set of attributes to each element in the type graph. Because of the inheritance of attributes, the following condition must hold:  $\forall n, n' : n, n' \in N_T, n' \in \text{clan}_T(n) \Rightarrow \sigma(n) \subseteq \sigma(n')$ .

Attributes are described in an abstract form: we specify only their names in the metamodel interfaces. Attribute constraints defined over models are abstracted as black box functions. The return value of such a function is the logical value *true* or *false*. Since a constraint function takes the values of the attributes as parameters, we need additional functions that map each parameter to a certain element of the model and to a certain attribute of the current element. To evaluate the logical value of a constraint, we need to assign values to the attributes first, which is described by a special function. In the next definitions, let  $[a, b]$  denote the interval of natural numbers from  $a$  to  $b$ , i.e.  $[a, b] = \{a, a + 1, \dots, b\}$ . Moreover, to facilitate formalization, we assume that  $\mathcal{V}$  is the set of all possible attribute values.

**Definition 4** (abstract attribute constraint) Given a metamodel interface  $\mathfrak{M} = (T, \mathcal{A}, \sigma)$  and an instance graph  $G$  typed over  $T$  by clan morphism  $\tau$ . An **abstract attribute constraint**  $c$  over  $G$  is defined by the triple  $(\varepsilon, \varrho, \omega)$ , where: (i)  $\varepsilon : \mathcal{V}^n \rightarrow \{\text{true}, \text{false}\}$  is the *evaluation function*,  $n > 0$ , *true* and *false* denote the logical constants; (ii)  $\varrho : [1, n] \rightarrow L_G$  is the *source function*; (iii)  $\omega : [1, n] \rightarrow \mathcal{A}$  is the *attribute selector function*, such that  $\forall i \in [1, n] \Rightarrow \omega(i) \in \sigma(\tau \circ \varrho(i))$ ; (iv)  $n$  is the number of the parameters of function  $\varepsilon$ , called the *arity* of  $c$  and is denoted by  $n = |c|$ .

**Definition 5** (attribute value assignment and constraint evaluation) Given a metamodel interface  $\mathfrak{M} = (T, \mathcal{A}, \sigma)$ , and a graph  $G$  typed over  $T$  by clan morphism  $\tau$ , an **attribute value assignment** is a function  $v : L_G \times \mathcal{A} \rightarrow \mathcal{V}$ .  $v$  is called *complete* if  $\forall l, a : l \in L_G, a \in \sigma(l) \Rightarrow \exists v(l, a)$ , otherwise, it is called *partial*. We say that  $v$  is complete with respect to constraint  $c$  if  $\forall i \in [1, |c|] \Rightarrow \exists v(\varrho(i), \omega(i))$ . Given a constraint  $c = (\varepsilon, \varrho, \omega)$  over a graph  $G$ , and an attribute value assignment  $v$  that is complete with respect to  $c$ , the **evaluation** of  $c$  is the evaluation of the function  $\varepsilon$  as follows:  $\varepsilon(v(\omega(1)), v(\omega(2)), \dots, v(\omega(n)))$ . We say that  $G$  satisfies  $c$  with respect to  $v$  (i.e.  $G \models_v c$ ) if the return value of the evaluation is *true*, otherwise  $c$  is not satisfied (denoted by  $G \not\models_v c$ ).

In the following, we define two types of abstract relations between constraints. Since the functions in the constraints are treated as black boxes, we will use these relations to compare constraints. Therefore, we assume that in several cases, we can determine the relation between constraints, for this purpose, we introduce two functions and assume



that both of them are available as global functions in our system.

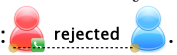
**Definition 6** (relations of constraints) Given two sets  $C_1, C_2$  of abstract attribute constraints over the same graph  $G$ . (i)  $C_1$  and  $C_2$  are **conflicting** (or are in conflict), denoted by  $C_1 \otimes C_2$ , if  $\nexists$  complete attribute value assignment  $v$  over  $G$  such that  $G \models_v C_1$  and  $G \models_v C_2$ . We can say shortly that the set  $C' = C_1 \cup C_2$  is conflicting.  $C_1$  and  $C_2$  are **not conflicting**, if  $\exists$  a complete attribute value assignment  $v$  over  $G$  such that  $G \models_v C_1, G \models_v C_2$ . (ii)  $C_{der}$  is **derivable** from  $C_{from}$ , denoted by  $C_{from} \vdash C_{der}$ , if for each possible complete attribute value assignment  $v$  over  $G$ :  $G \models_v C_{from} \Rightarrow G \models_v C_{der}$ .  $C_{der}$  is **not derivable** from  $C_{from}$ , if  $\exists$  a complete attribute value assignment  $v$  over  $G$  such that  $G \models_v C_{from}$ , but  $G \not\models_v C_{der}$ .

The function  $\text{ARECONFLICTING}(C, G)$  takes a set  $C$  of constraints, a graph  $G$ , on which the constraints are defined. Given any possible parameters, this function returns a value either *true*, *false*, or *unknown*. The value *true* means that it can be proved that the constraints in the set  $C$  are conflicting, *false* means that it can be proved that the constraints in  $C$  are not conflicting, and *unknown* means that neither can be proved, i.e. the system does not have enough information.

The function  $\text{ISDERIVABLE}(C_{from}, C_{der}, G)$  takes two sets  $C_{from}, C_{der}$  of constraints, a graph  $G$ , on which the constraints are defined. Given any possible parameters, this function returns a value either *true*, *false*, or *unknown*. The value *true* means that  $C_{der}$  can be proved to be derivable from  $C_{from}$ , *false* means that it can be proved that  $C_{der}$  is not derivable from  $C_{from}$ , and *unknown* means that neither can be proved, i.e. the system does not have enough information.

The assumption that we have these two functions seems to be very restrictive, since in complex attribute constraint description languages, it is really hard to determine the relation between arbitrary constraints. However, the implementation of the previous functions always have the possibility to return the value *unknown*.

**Definition 7** (pattern) A **pattern**  $P = (G, C)$  of a metamodel interface  $\mathfrak{M}$  is an instance graph  $G$  of  $\mathfrak{M}$  and a set  $C$  of abstract attribute constraints defined over  $G$ . The constraints of a pattern  $P$  are also denoted by  $\mathcal{C}@P$ , i.e.  $C = \mathcal{C}@P$ .

Patterns are instances of metamodel interfaces. A pattern defines the elements of a model part along with additional attribute constraints. A sample pattern described by the concrete syntax of the domain of mobile-based social networks is as follows: . This pattern contains a contact, a member, and a similarity edge between them. Moreover, we define a constraint which states that the similarity edge should be rejected, i.e. the value of the *ApprovalState* attribute must be *rejected*, which is denoted by the label 'rejected' on the similarity edge. Several other patterns will be presented in [Section 5](#).

In order to handle type inheritance in matches, we introduce the definition of weakly typed morphisms. It is easy to show that typed graphs along with weakly typed morphisms form a category. We also define the mapping of attribute constraints by morphisms.

**Definition 8** (weakly typed morphism and mapped constraint) Given typed graphs  $(G_1, \tau_1)$  and  $(G_2, \tau_2)$  typed over a type graph  $T$ . Let  $\tau_1 = (\tau_1^N, \tau_1^E)$  and  $\tau_2 = (\tau_2^N, \tau_2^E)$  be the clan morphisms of  $G_1$  and  $G_2$ . A weakly typed morphism  $m : G_1 \rightarrow G_2$  is a graph morphism such that: (i)  $\forall n \in N_{G_1} : \tau_2^N \circ m(n) \in \text{clan}_I(\tau_1^N(n))$ , (ii)  $\forall e \in E_{G_1} : \tau_2^E \circ m(e) = \tau_1^E(e)$ . Given a metamodel interface  $\mathfrak{M}$ , two patterns  $P_1 = (G_1, C_1)$ ,  $P_2 = (G_2, C_2)$  of  $\mathfrak{M}$ , and a weakly typed morphism  $m : G_1 \rightarrow G_2$ . For each  $c = (\varepsilon, \varrho, \omega) \in C_1$ , the mapping  $c' = m(c)$  is a constraint on  $P_2$ , which exists if  $\forall i \in [1, |c|] : \exists m(\varrho(i))$ . The mapped constraint  $c'$  is an abstract attribute constraint  $(\varepsilon', \varrho', \omega')$  on  $P_2$  such that  $\varepsilon' = \varepsilon$ ,  $\varrho' = m \circ \varrho$ , and  $\omega' = \omega$ .

## 5 Declarative Description of Model Transformations

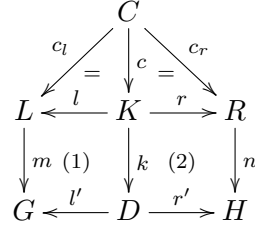
The declarative description of a model transformation contains the formal specification of the rewriting rules and the control flow graph.

**Definition 9** (rewriting rule interface) A **rewriting rule interface**  $p$  is defined by four patterns  $L$ ,  $C$ ,  $K$ , and  $R$ , called the left hand side (LHS), the context, the interface (or gluing), and the right hand side (RHS) patterns respectively, and three injective, total weakly typed morphisms  $l : K \rightarrow L$  and  $r : K \rightarrow R$  and  $i : I \rightarrow K$ . No constraints are allowed to be defined in  $C$  and  $K$ , i.e.  $\mathcal{C}@C = \emptyset$  and  $\mathcal{C}@K = \emptyset$ . Moreover, let  $c_l = l \circ c$  and  $c_r = r \circ c$ .

$L$  is the pattern that needs to be matched before the application of the rule. In the match, the constraints of  $L$  must be satisfied. During the application of the rule,  $L$  will be replaced by a pattern that is isomorphic to  $R$  such that the constraints of  $R$  will be satisfied. As in the case of traditional rewriting rules,  $K$  represents the pattern that is matched, but not modified. However, in this case, modification concerns only the structure, i.e. deletion of the elements of the graph, therefore, the attributes of these elements may be modified. We introduce a new component, pattern  $C$ , for this purpose, which specifies the part of  $K$  that is completely left intact, i.e. the attributes of these elements are also not modified. In the following, we assume that given a total, injective weakly typed morphism  $q$ ,  $q^{-1}$  denotes the inverse morphism, which is also injective, but may not be total. The application of a rule can also be formalized as an interface:

**Definition 10** (direct model transformation interface) Given a rewriting rule interface  $p$  and a pattern  $G$  with a total, injective, weakly typed morphism  $m : L \rightarrow G$  called match such that  $\text{ARECONFLICTING}(\mathcal{C}@G \cup m(\mathcal{C}@L), G) \neq \text{true}$ , a **direct model transformation interface** is given by the following double pushout diagram, where (1) and (2) are pushouts,  $m$ ,  $k$ ,  $n$  are injective, total weakly typed morphisms and  $G$ ,  $D$ ,  $H$  are model patterns. During the composition of the patterns  $D$  and  $M$ , their structure (i.e. their elements) are computed by the traditional DPO approach. Moreover, the constraints attached to these patterns are as follows: (i) For each constraint  $c \in \mathcal{C}@G$  such that  $\forall i \in [1, |c|] \Rightarrow \nexists l \in L : m(l) = \varrho(i) \vee \exists l \in C : m \circ c_l(l) = \varrho(i)$ , we compose  $c' = l'^{-1}(c)$ .  $\mathcal{C}@D$  is the set of all  $c'$  constraints. (ii)  $\mathcal{C}@H = r'(\mathcal{C}@D) \cup n(\mathcal{C}@R)$ . Moreover, we require that

$\text{ARECONFLICTING}(C@H, H) = \text{false}$ .



In the following, we present the definition of control flow graphs to formally describe the control structure of model transformations.

**Definition 11** A control graph  $C_T$  is defined by the tuple  $(N_T, F_T, source_T, target_T, start_T, S_T, ES_T, E_T, step_T, R_T, action_T)$ , where:

- $N_T$  is the set of the nodes of the graph, which is composed by the three disjoint sets  $\{start_T\}, S_T, E_T$ , i.e.  $N_T = \{start_T\} \cup S_T \cup E_T$ ,  $S_T \cap E_T = \emptyset$ , and  $start_T \notin S_T, start_T \notin E_T$ .  $ES_T$  is a subset of  $S_T$  (i.e.  $ES_T \subseteq S_T$ ) called exhaustive steps.
- $start_T$  is the starting node of the control flow, this is the point where the execution starts, and  $E_T$  contains the end nodes, these are the points of the control flow, where the execution finishes.
- $R_T$  is the set of rewriting rule interfaces that are referenced by the control flow graph.  $S_T$  is the set of rule nodes that are called steps in this context. Function  $step_T : S_T \rightarrow R_T$  assigns a rewriting rule to each step of the control flow graph.
- $F_T$  is the set of the edges called flow edges.  $source_T : F_T \rightarrow N_T$  and  $target_T : F_T \rightarrow N_T$  are the source and target functions for the flow edges respectively. Function  $action_T : F_T \rightarrow \{s, f, x\}$  assigns one of the values  $s, f$ , or  $x$  to each flow edge.

The separate building elements of the transformations are called *steps*. A step is a traditional rewriting rule. The application of a rewriting rule can be successful, or unsuccessful, this property can be used in the control flow to define branches. Therefore, to each flow edge an action value is assigned by function  $action_T$ . (i) Value  $s$  ('success') means that the flow edge is followed if the application of the source step was successful. (ii) Value  $f$  ('failure') means that the flow edge is followed if the application of the source step was unsuccessful. (iii) Value  $x$  ('dontcare') means that the edge is followed in both cases. If there are multiple edges that can be followed after the application of a rule, one is chosen nondeterministically. Start node and end nodes are used to mark the starting point and possible end points of the transformations. Moreover, exhaustive steps are rules that are applied exhaustively. It may happen that a step does not have an outgoing edge to be followed, in this case the transformation terminates without reaching an end node, which is a irregular operation. Because of this and other restrictions on the graphs, we provide the definition of *regular* control flow graphs as follows:

**Definition 12** (regular control flow graph) A control graph  $C_T$  is **regular** if the following conditions hold: (i)  $\exists$  unique  $f \in F_T : source_T(f) = start_T$ , (ii) the graph is connected, i.e.  $\forall n \in N_T : \exists$  path along directed flow edges from  $start_T$  to  $n$ , (iii)  $\forall s \in S_T : s$  has

either at least one outgoing flow edge with the action value  $x$  or has at least two outgoing edges with the action values  $s$  and  $f$  respectively.

The definition of Similarity Handling Transformation can be transformed into the presented formalism trivially. To illustrate our definitions, we present the declarative description of rule  $rc2$  in Figure 2.

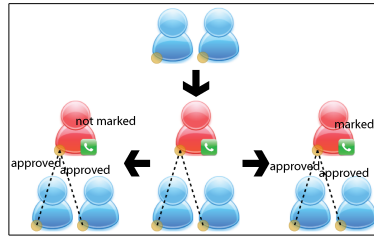





Figure 2: Formal Description of Rule  $rc2$  of Similarity Handling Transformation

The formal, complete presentation of the verification of Similarity Handling Transformation would exceed the limits of this paper. However, the main benefit of the previously proposed declarative description is the ability to support the automated offline analysis. Therefore, we selected the verifiable property (v2), and in the following, we present its verification informally to illustrate how our formalism makes the automated analysis possible. Note that we informally explain the operation of the completely automated algorithms implemented in VMTS. For deeper details about the formal presentation of the analysis, see [AELL10a].

Before starting the analysis, we need to introduce the syntax and semantics of a concrete type of MCDL formula. A *simple MCDL formula* [AELL10b] is defined as  $\exists P$ , where  $P$  is a pattern. Given a pattern  $M$ ,  $M$  satisfies  $\exists P$  if and only if there exists a total, injective, weakly typed morphism  $m : P \rightarrow M$  such that  $\text{ISDERIVABLE}(\mathcal{C}@M, m(\mathcal{C}@P), M)$ . Moreover,  $M$  satisfies  $\nexists P$  if  $M$  does not satisfy  $\exists P$ . We use a simple MCDL formula to express the verifiable property  $\varphi_{v2}$ .

Let  $\varphi_{v2} = \nexists$     =  $\nexists P_{v2}$ , this property must be true for the output models, therefore, the verification means formally proving that for each possible input model, the output model satisfies  $\varphi_{v2}$ . The discovery algorithm processes the rules of the control flow graph in the following order:  $rc1$ ,  $rc2$ ,  $rc3$ ,  $rc4$ ,  $rc5$ , and finally  $rc6$ . It is easy to show that the exhaustive application of  $rc1$  terminates, because in each application it deletes one similarity edge, but does not create new similarity edges. Obviously, after the exhaustive application of  $rc1$ , LHS of  $rc1$  will not be present in the model, otherwise the rule could have been applied again. The discovery algorithm can infer this property, therefore, assuming that the MCDL formula  $\varphi_{e2}$  is assigned to edge  $e2$ , we can say that  $\varphi_{e2} \Rightarrow \varphi_{v2}$ , since LHS of  $rc1$  is the pattern that is present in  $\varphi_{v2}$ . Here, we skip some steps of the analysis, which would require the analysis of the control flow graph itself, but we assume that the discovery algorithm has computed formula  $\varphi_{e6}$  assigned to edge  $e6$ . We also assume that we can derive from  $\varphi_{e6}$  that  $\varphi_{v2}$  is true, i.e.  $\varphi_{e6} \Rightarrow \varphi_{v2}$ . Given the rule interface of  $rc5$ , it is easy to prove that if  $\varphi_{v2}$  is true before the application

of  $rc5$ , it will be so after its application. We illustrate how this information can be inferred from the declarative rule interface: assume that after the application of  $rc5$ , the pattern  $P_{v2}$  is present. We can enumerate all possible patterns that describe the part of the modified model, which part contains RHS of  $rc5$  and pattern  $P_{v2}$ . For each possible pattern, we can compute what was present before the application of the rule by applying [Definition 10](#) in the reverse direction. In these computed patterns, it is easy to prove, that pattern  $P_{v2}$  exists. Therefore, we can prove that if  $P_{v2}$  exists after the application of  $rc5$ , it must exist before its application as well. Similarly, we can prove that  $\varphi_{e7} \Rightarrow \varphi_{v2}$  and  $\varphi_{e8} \Rightarrow \varphi_{v2}$ , therefore  $\varphi_{v2}$  will be true for all possible output models of the transformation. Once again, because of lack of space, this informal explanation only illustrates how the rule interfaces are used by the propagation algorithm.

Although the verification of only the property (v2) is outlined informally for demonstration purposes, we emphasize that the analysis of all properties could be performed automatically in VMTS.

## 6 Related Work

In this section, we present the work related to the formalism that constitutes the contribution of this paper. For a detailed discussion on offline verification methods for graph rewriting-based model transformations in general, see our previous paper [\[ALL10\]](#).

[\[Str08\]](#) presents an approach to formally describe certain parts of graph transformations in order to reason about the transformations in a proof assistant. However, this approach is limited to the structural aspects of graph rewriting.

In [\[Sch09\]](#), [\[Sch10\]](#), a verification method for graph rewriting-based model transformations are presented. The approach provides a formal representation to describe model transformations as declarative relations in Prolog style. The specification of the transformations is not based on rule-based graph grammars, but uses a textual description based on a relational, declarative calculus. [\[Sch09\]](#) shows that the presented representation can be directly translated into representations for theorem provers. One of the key differences between this approach and that presented in our paper is the handling of attributes. In [\[Sch09\]](#), attribute values are explicitly defined in the declarative description of the rules, while our approach can contain arbitrary constraints by the definition of abstract attribute constraints.

[\[Pen09\]](#) presents a notation of structural transformations, namely programs with interface, that are a generalization of programs over transformation rules. The presented formalism makes it possible to analyze and verify the programs. Nested conditions that can express the verifiable properties are also defined. Similarly to the concept presented in [\[Ore08\]](#), [\[Pen09\]](#) assumes that the language of attribute constraints is defined by a data signature and algebra. In our approach, we do not rely on these formalisms, the definition of abstract attribute constraints make it possible to work with any type of imperative and constraint description language (e.g.  $C\#$  in VMTS) during the implementation of the verification system.

We also need to mention the approach presented in [\[CCGL10, CCGL08\]](#). The authors

translate the definition of graph rewriting rules into an OCL-based representation, which can be combined with constraints defined in the metamodels. Moreover, combining individual rules, a transformation model can also be defined, which can be analyzed by several tools. One main difference between this approach and the one discussed in our paper is that we analyze the control structure of model transformations, which can be an arbitrary directed graph.

## 7 Conclusions

The verification of model processing programs is a fundamental issue, in our research, we concentrate on the verification of graph rewriting-based model transformations. Moreover, we are interested in the automated, offline verification of such programs. In this paper, we have summarized the operation of our framework for the automated verification of model transformations. The main element of the concept is the declarative, formal, platform-independent description of model transformations. Based on a solid mathematical background, we have provided a formalism for the specification of model transformations. On a case study of refactoring mobile-based social network models, we demonstrated how the formal description can be generated from the implementation of a model transformation in a modeling framework VMTS. The complete formal verification of the presented model transformation would exceed the limits of this paper, therefore, we are not able to formally present how the provided formalism can be used by automated methods. However, we informally explained how these methods work to illustrate the importance of our results.

Our results show that several functional properties of the model transformations can be automatically verified. Moreover, the capabilities of our methods can be extended in the future. In future work, our goal is to test our verification framework in more complex industrial case studies.

## Bibliography

- [AEL<sup>+</sup>10] M. Asztalos, P. Ekler, L. Lengyel, T. Levendovszky, T. Mészáros. Formalizing Models with Abstract Attribute Constraints. In *GCM*. Enschede, The Netherlands, October 2010. (submitted, under review).
- [AELL10a] M. Asztalos, P. Ekler, L. Lengyel, T. Levendovszky. Applying Offline Verification of Model Transformations to Mobile Social Networks. In *Pre-Proceedings of GraBaTs*. ECEASST, Enschede, The Netherlands, September 2010.
- [AELL10b] M. Asztalos, P. Ekler, L. Lengyel, T. Levendovszky. MCDL: A Language for Specifying Graph Conditions with Attribute Constraints. In *MoDeVVa*. Oslo, Norway, October 2010. (submitted, under review).

- [ALL10] M. Asztalos, L. Lengyel, T. Levendovszky. Towards Automated, Formal Verification of Model Transformations. In *ICST*. Paris, France, 2010.
- [BELT04] R. Bardohl, H. Ehrig, J. de Lara, G. Taentzer. Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In *FASE*. Pp. 214–228. 2004.
- [CCGL08] J. Cabot, R. Clarisó, E. Guerra, J. Lara. Analysing Graph Transformation Rules through OCL. In *ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations*. Pp. 229–244. Springer-Verlag, Berlin, Heidelberg, 2008.
- [CCGL10] J. Cabot, R. Clarisó, E. Guerra, J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *J. Syst. Softw.* 83(2):283–302, 2010.
- [EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series XIV. Springer, 2006.
- [EL10] P. Ekler, T. Lukovszki. Experiences with phonebook-centric social networks. In *CCNC*. Las Vegas, USA, 2010.
- [Ore08] F. Orejas. Attributed Graph Constraints. In *ICGT*. Pp. 274–288. Springer-Verlag, Berlin, Heidelberg, 2008.
- [Pen09] K.-H. Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, University of Oldenburg, 2009.
- [Plu98] D. Plump. Termination of graph rewriting is undecidable. *Fundam. Inf.* 33(2):201–209, 1998.
- [Sch09] B. Schätz. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. *SLE, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, pp. 227–244, 2009.
- [Sch10] B. Schätz. Verification of Model Transformations. In *Pre-Proc. of GT-VMT*. Pp. 129–142. ECEASST, Paphos, Cyprus, March 2010.
- [Str08] M. Strecker. Modeling and Verifying Graph Transformations in Proof Assistants. *Electr. Notes Theor. Comput. Sci.* 203(1):135–148, 2008.
- [VMT] VMTS website. <http://vmts.aut.bme.hu/>.