A Formalism for Automated Verification of Model Transformations

Márk Asztalos, László Lengyel, Tihamér Levendovszky

Budapest University of Technology and Economics, Hungary
{asztalos, lengyel, tihamer}@aut.bme.hu

Abstract: Verification of models and model processing programs are fundamental issues and are inevitable in model-based software development in order to apply them in real-world solutions. Verification concerns the analysis of non-functional and functional properties as well. Model transformation developers are interested in offline methods for the verification process. Offline analysis means that only the definition of the model transformation and the metmodels of the source and target languages are used to analyze the properties and no concrete input models are taken into account. Therefore, the results of the analysis hold for each output model not just particular ones, and we have to perform the analysis only once. Most often, formal verification of model transformations is performed manually, but automated or semi-automated approaches have gained focus recently. We have previously presented a method to formally describe the main characteristics of model transformations. Our concept consists of two steps: (i) The automatic generation of a formal description from a concrete transformation, which is manually extended by formal assertions by transformation experts. (ii) A reasoning system is used to automatically derive the proof of certain properties from the previous formal description. In this paper, we show how deduction rules of the reasoning system can be defined.

1 Introduction

In model-based approaches, models are primary artifacts. Verification of models means proving some properties of the models, which is a fundamental issue in industrial solutions. We usually need to convert different models to others, for example when generating source code from a UML [1] class diagram. Model transformation is an often used model processing technique for this purpose [2]. Verification of model transformations means proving some properties of the model transformations [3], functional and non-functional properties, and some properties of the generated models as well. If model transformations are verified completely, the generated models do not need to be analyzed separately after each application of the transformation. The analysis of a transformation is called *offline*, when only the definition of the transformation and the language of the input and output models are used during the analysis process and no concrete

input models are taken into account. Therefore, all results are independent from the input models and the analysis has to be performed only once for the transformation. In this case, it is not critical if manual work is needed.

2 Background

2.1 Formal Background of Model Transformations

Graph rewriting-based model transformations offer a strong mathematical background for the formalization and analysis of model transformations [5, 6]. Graph transformations are used as a modeling technique in software engineering and to process visual models created in various modeling languages such as DSMLs and the Unified Modeling Language. We do not want to formally cite all definitions of graph rewriting systems, we refer only to the main definitions informally.

Graph transformations consists of separate graph rewriting rules, each rule is defined with its left-hand-side graph (LHS) and right-hand-side graph (RHS). A transformation is the application of a sequence of rules on an input graph, where the application of a rule means finding an isomorphic occurrence of LHS in the input model and replacing it with RHS (Double Pushout (DPO) approach). An application of a rule is successful if a match has been found.

In model transformation frameworks, transformations consists of separate rules and an additional control structure (control flow) that defines the execution order of the rules. Models can be considered special graphs, for these applications, attributed typed graphs have been introduced. A model transformation typically has one input and one output model, but may process multiple models as well. In-place transformations are performed on a single model. For the better understandability, in this paper, we consider in-place transformations, which does not restrict the capabilities of the approach to be presented.

Metamodels are special models the instances of which are instance models. A meatmodel defines the type of entities and the relations between them that can appear in the instance models. In metamodeling frameworks, we can usually define additional restrictions (constraints) in the metamodels that have to be satisfied on the instances.

2.2 Introduction to the Concept of Formal Description of Transformations

Completely general methods for verification of all properties are unreachable, since, for example, the termination of a model transformation in general is an

algorithmically undecidable problem [7]. Our goal is to develop an automated or semi-automated reasoning system that can prove some properties of transformations in an offline way. Previously [8], we have introduced the term *assertion*: an assertion is a formal expression that describes some characteristics of a model under transformation or a model transformation. In other words, an assertion states something about the model or the model transformation. We want the system to prove if these assertions are true or prove if they are false. We have proposes the formal definition language for describing the assertions. We can define deduction rules for the reasoning system. Using an initial assertion set that describe a model transformation and applying the deduction rules, the reasoning system may derive some more assertions.

We expect our system to be able to handle many types of assertions, and it should be extendable with new ones in the future. In other words, we do not want to make restrictions on the types of the analyzed properties. We want our system to be able to investigate functional and non-functional properties as well. The set of deduction rules should be extendable as well. The expressiveness of the first-order logic extended by handling of attributes and topological queries is satisfying for this purpose. To make this paper mor understandable and self contained, in the following, we summarize the formal description presented in [8] for describing model transformations.

Formal Transformation Descriptions

Definition 1. A *transformation description* $D_T = (C_T, action_T, start_T, A_T)$ of a transformation *T* consists of

- A control graph $C_T = (S_T, F_T, source_T, target_T)$, where S_T is a set of nodes called steps; F_T is a set of edges called flow labels $source_T : F_T \rightarrow S_T$ is the source function for edges; and $target_T : F_T \rightarrow S_T$ is a target function for edges.
- An action function $action_T : F_T \rightarrow \{success, failure, don't\ care\}$
- A starting step $start_T \in S_T$, such that $\nexists f \in F_T : target_T(f) = start_T$
- A set A_T of assertions that are sentences of the *Assertion Description Language*.

A control structure describes the control graph of a model transformation. The separate building elements of the transformations are called *step*s, because we want to distinguish between traditional rewriting rules and the elements of *transformation description*s. A step can be a traditional rewriting rule, and a complex sub-transformation as well. The application of a rewriting rule can be successful, or unsuccessful, and based on this fact, branches can be defined in the

control flow. We decided to keep this feature and extend it for steps as well. Therefore, to each flow edge an action value is assigned: (i) Value *success* means that the flow edge is followed if the application of the source step was successful. (ii) Value *failure* means that the flow edge is followed if the application of the source step was unsuccessful. (iii) Value *dontcare* means that the edge is followed in both cases.

Assertions are sentences of the Assertion Description Language (ADL)**.** An assertion consists of two parts: (i) a *formula* that states something about the model or the transformation, and (ii) a part that places the formula at a concrete point of the transformation control structure. For example: *there are no nodes of type T in the input model* [(i)] *when the transformation starts* [(ii)]. We can distinguish two types of formulas: (i) static formulas and (ii) dynamic formulas. While formulas of type (i) (for example the first part of the previous assertion) states something about the current state of the model under transformation or the transformation itself, formulas of type (ii) describe the dynamic behavior of a transformation step. An example for formulas of type (ii) can be: *step s deletes all nodes of type* T_1. The sentences of ADL are called assertions. The syntax of an assertion $assertion_T$ of a model transformation T is as follows:

$$
\begin{array}{rll}
assertion_T ::= & \mathbf{pre}(s) & Formula_{Static} \mid \\
& \mathbf{post}_{\mathbf{succ}}(s) & Formula_{Static} \mid \\
& \mathbf{post}_{\mathbf{fail}}(s) & Formula_{Static} \mid \\
& \mathbf{at}_{\mathbf{succ}}(s) & Formula_{Dynamic} \mid \\
& \mathbf{at}_{\mathbf{fail}}(s) & Formula_{Dynamic} \mid \\
& \mathbf{before}(s) & Formula_{Static} \mid \\
& \mathbf{after}(s) & Formula_{Static}
\end{array}
$$

Bold words are operators in ADL, $s \in S_T$ is a step of the corresponding model

transformation. The prefix of an assertion describes the point of the transformation where the current *Formula* is true. The second part of each assertion is a formula, which can be either static or dynamic. The semantics of possible prefix values is summarized in Table 1.

Table 1
Assertion Prefixes

Prefix	Semantic
$\mathbf{pre}(s)$ *Formula*	The application of step s is successful if *Formula* is true.
$\mathbf{post}_{\mathbf{succ}}(s)$ *Formula*	*Formula* will be true after step s if the application of s was successful.
$\mathbf{post}_{\mathbf{fail}}(s)$ *Formula*	*Formula* will be true after step s if the application of s was unsuccessful
$\mathbf{at}_{\mathbf{succ}}(s)$ *Formula*	*Formula* describes the modification that is performed by the step if the rule can be applied successfully.
$\mathbf{at}_{\mathbf{fail}}(s)$ *Formula*	*Formula* describes the modification that is performed by the step if the rule cannot be applied successfully.
$\mathbf{before}(s)$ *Formula*	*Formula* is true just before the processing of step s.
$\mathbf{after}(s)$ *Formula*	*Formula* is true just after the processing of step s.

Patterns in ADL

Before introducing assertion formulas, we need to present the definition of *pattern*s. Patterns will be used throughout the definition of different formulas in assertions.

Definition 2. A model *pattern* $P = (TG, AC)$ consists of:

- A typed graph *TG*=(*N, E, T, type, source, target*), which is a *G*=(*N, E, source, target*) graph extended with a set *T* of types and a type function $type: N \cup E \rightarrow T$ that assigns a type value for each node or edge in the graph.
- A set of attribute constraints *AC*.

Attribute constraints are sentences of the Attribute Constraint Description Language (CDL). Each sentence states some constraints about the attributes of the nodes and edges in the pattern. In this paper, we will use only simple attribute constraints. A simple attribute constraint defines that the value of an attribute must be a certain value, or defines that the value of an attribute must not be a certain value. The syntax of the sentences of CDL is as follows:

$$constraint_P ::= c_P : e \rightarrow a \ \{\textbf{requires}|\textbf{forbids}\} \ value$$

if *P* is the corresponding pattern,

- c_P is the name of the constraint
- *e* is a node or an edge of the corresponding pattern $P (e \in N \cup E)$
- *a* is the name of the attribute
- exactly one of operators **requires** or **forbids** has to be present
- *value* is an arbitrary value of an attribute

Patterns are not attributed graphs, since nodes and edges of the pattern does not have attributes. The attribute constraints will be evaluated when an instance of the pattern is found in a concrete model.

Definition 3. Let *I* be a model, which is a typed attributed graph. *I* is an *instance of a pattern P* (denoted by $I \triangleleft P$) if there exists an injective graph morphism *m* between the base graph of *I* (*I* without types and attributes) and the base graph of *TG* (without types) such that:

- $n \in N_I \Rightarrow type_I(n) = type_P(m(n))$ or $type_I(n)$ inherits from $type_P(m(n))$ by the metamodel of the models to be transformed
- $e \in E_I \Rightarrow type_I(e) = type_P(m(e))$, or $type_I(e) \in E_T$ inherits from $type_P(m(e))$ by the metamodel of the models to be transformed

- $l \in N_I \in E_I \Rightarrow$ l all attribute constraints in P that concerns $m(l)$ are satisfied for l.

The metamodel of the models to be transformed is also used for the analysis. Therefore, in the case of instance matching, inheritance is taken into account.

The semantic of a constraint $constraint_P ::= c_P : l \rightarrow a \{\textbf{requires} \mid \textbf{forbids}\} v$ is as follows: if I is an instance of pattern P with the morphism m, then

- the attribute a of element l' of I ($m(l') = l$) has to have the value v in the case of **requires** operator is present,
- the attribute a of element l' of I ($m(l') = l$) is forbidden to have the value v in the case of **forbids** operator.

We introduce a notation that will be used later in this paper: if I is an instance of pattern P with morphism m and $n \in N_P$, or $e \in E_P$ than $n(I) = n'$ and $e(I) = e'$ such that $m(n') = n$ and $m(e') = e$.

Formulas of ADL

Table 2
Static Formulas in ADL

Name	Syntax	Semantic
None	**None** P	No instances of pattern P exist in the model.
Exists	**Exists** P	At least one instance of pattern P exists in the model.
Any	**Any** $P_1 \rightarrow P_2$	It is true that for each instance I_1 of pattern P_1 an instance I_2 of pattern P_2 can be found, such that the following conditions hold: (i) if $n \in N_{P_1} \wedge n \in N_{P_2} \Rightarrow n(I_1) = n(I_2)$ (ii) if $e \in E_{P_1} \wedge e \in E_{P_2} \Rightarrow e(I_1) = e(I_2)$

Table 3
Dynamic Formulas in ADL

Name	syntax	semantic
Forone	**ForOne** $P_1 \rightarrow P_2$	an instance I_1 of pattern P_1 is matched from the model and will be replaced with an instance I_2 of pattern P_2 such that: (i) $n \in N_{P_1} \wedge n \notin N_{P_2} \Rightarrow n(I)$ will be deleted (ii) $e \in E_{P_1} \wedge n \notin E_{P_2} \Rightarrow e(I)$ will be deleted (iii) $n \notin N_{P_1} \wedge n \in N_{P_2} \Rightarrow n(I)$ will be a newly created node in the model (iv) $e \notin E_{P_1} \wedge e \in E_{P_2} \Rightarrow e(I)$ will be a newly created edge in the model. (v) $\forall l : (l \in N_{P_2} \cup E_{P_2}) \wedge \forall c_{P_2} \in AC_{P_2} : c_{P_2} = l \rightarrow a$ **requires** $v \Rightarrow$ the attribute a of $l(I_2)$ will be v. (v) $\forall l : (l \in N_{P_2} \cup E_{P_2}) \wedge \forall c_{P_2} \in AC_{P_2} : c_{P_2} = l \rightarrow a$ **forbids** $v \Rightarrow$ the attribute a of $l(I_2)$ will not be v.
Foreach	**ForEach** $P_1 \rightarrow P_2$	A foreach means that the **ForEach** $P_1 \rightarrow P_2$ modification is applied repeatedly until it cannot be applied any more.
Termination	**Terminates**	The current step terminates.

The last elements of ADL are formulas, in Table 2, and Table 3, we have collected the syntaxes and semantics of static and dynamic formulas respectively. A *None formula* states that a pattern does not exist in the model at the current point of the

transformation specified by the prefix of the assertion. Its opposite is the An *Exists formula* which states that at least one instance of a concrete pattern exists in the model. An any formula is used describe some structural constraints of the model, for example, if a node of type T_1 is present in the model, than it has at least one connecting node (connected with an edge) of type T_2. The base dynamic formula that is used is called *Forone formula*. It can be identified with the definition of a rule by specifying its left hand side and right hand side patterns. We have introduced a formula (*Foreach*) for the exhaustive application of a *Forone formula*, which means that the modification is applied repeatedly until it cannot be applied any more. We can explicitly define that a step terminates with the *Termination* formula.

2.3 Summary

In this section, we have concluded a formal description language for model transformations. Given an initial assertion set, using the appropriate deduction rules, we can derive additional assertions that describe additional properties of the model transformation. A key question of our approach is how to create the initial assertion set. We want to provide an automated or semi-automated system, which means that a base transformation description with a base assertion set that describes the main characteristics of the transformation should be generated automatically from the definition of a transformation. Semi-automation means that transformation experts should have the possibility to add new assertions to the original assertion set, in order to extend the knowledge of the system with the results of the manual analysis.

3 Related Work

Offline analysis of model transformations have been performed in several cases, but the approaches presented can usually be applied for only certain (type of) transformations, or only for certain (type of) properties. In [9], syntactic correctness and semantic correctness are aimed to be verified by the VIATRA transformation system. Semantic correctness covers: (i) verification of correctness requirements, (ii) termination, (iii) completeness, (iv) uniqueness. The paper presents a static method to verify model transformations. Converting system models into Kripke structures allows to verify certain properties of a single transformation starting from a single model. [10] presents how behavior preservation of a model transformation can be verified via goal-directed certification. The proposed verification realized in GReAT is a static technique, which is based on the generation of assurances for code produced by automatic synthesis tools. [11] presents an approach to verify that during the transformation of a model the semantic preserves, which is formally proved for each rule. Some

papers propose approaches for verification of model transformations in special domains, such as mechatronic systems [12], or Java code generation [13]. [14] presents an approach similar to the one presented in this paper: UML metamodels along with embedded well-formedness rules (typically OCL constraints) can be translated to the formalism Alloy. Then, the Alloy Analyzer can conduct fully automated analysis of the transformation. The difference between our approach and the one presented in that paper is that the Alloy Analyzer uses a simulation that generates a random instance model of the input metamodel, then analyzes the behavior of the transformation by transforming this instance model.

4 Contributions

4.1 Consistency of Transformation Descriptions

Until now, we have concluded the formalism to describe model transformations. The base elements of a transformation description are the control graph and the set of assertions, these entities will be used to derive additional assertions.

When formally analyzing transformation descriptions, it is important to prove that a description is consistent in the sense that there are no assertions that contradict each other. It is a fundamental issue, because we want the developer to manually extend the set of assertions. Another important question is: what can be stated about the consistency of a description that is extended with the derived assertions? It is not a trivial problem, since only a subset of assertions are used when deriving new assertions.

Definition 4. Let D=(C, *start*, *action*, A) be a transformation description. An assertion a is *derivable* from the transformation description (denoted by $D \vdash a$), if all assertions in A are true implies that a is true.

Definition 5. Let a = ($prefix_1$ $Formula_1$), and a_2=($prefix_1$ $Formula_2$) be two assertions. a_1 is the *opposite* of a_2 if a_1 is true (false) if and only if a_2 is false (true).

Definition 6. A transformation description D=(C, *start*, *action*, A) is *weakly consistent* if $\nexists a_1, a_2 \in A$ such that a_1 is the opposite of a_2.

Definition 7. A transformation description $D=(C, start, action, A)$ is *strongly consistent* (or *consistent*) if weakly consistent and $\forall a, A' : D'=(C, start, action, A')$

$\vdash a, A' \subseteq A \Rightarrow D''=(C, start, action, A \cup \{a\})$ is *consistent*.

The analysis of the property consistency is crucial in the application of our concept, but may exceed the limits of this paper. In the rest of this paper, we assume that our transformation descriptions are consistent.

4.2 Deduction Rules for the Reasoning System

In this section, we present deduction rules for the automated reasoning system. With these rules, a reasoning system can derive additional assertions from an initial assertion set that describes the base characteristics of a concrete transformation. Deduction rules are needed to (i) propagate formulas through the control flow, (ii) derive new formulas at certain points of the transformation. Propagation is, for example, when we prove that if a formula is true before a concrete rule, it will be so after the rule as well. Derivation means proving a formula that could not be proven before a concrete point of a transformation.

Proposition 1. Let D be a consistent transformation description. In D, if (**after**(s) *Formula*)$\in A$, and $\exists$ e $\in F$: *source*(e) = s, *target*(e) = s', and $\nexists$ $e' \in F$: $e' \neq$ e

target(e') = s', then $D \vdash$ (**before**(s') *Formula*).

Intuitively, Proposition 1 claims that if there is a flow edge from step s to step s', and there is no other edge to s' then each formula that will be true after the application of s will be true as well before the application of s'.

Proof (of Proposition 1). There are no assertions on flow edges (which means, flow edges do not modify the properties of the model or model transformation), therefore, in a concrete application of a transformation if a formula is true after the execution of a step (s) it will be true as well before the application of the next step

(s'). The conditions above assures that in each execution of the transformation, independently from the concrete model, the step before s' will always be s, therefore, if a formula is true after s it will be always true before s'.

Proposition 2. Let D be a consistent transformation description. In D, if (**pre**(s) **Exists** P) $\in A$, and (**at**$_{\mathbf{succ}}$(s) **ForEach** $P \rightarrow P' \in A$, and $\nexists\ a \in A : a =$ (**at**$_{\mathbf{fail}}$(s) *Formula*), and (**at**$_{\mathbf{succ}}$(s) **Terminates)** $\in A$, then $D \vdash$ (**after**(s) **None** P).

Proof (of Proposition 2) The application of the current step surely terminates because of the conditions of the proposition. There are two possible cases: (i) the application of the step is unsuccessful and (ii) the application of the step is successful.

In case of (i), **pre**(s) **Exists** P is not satisfied, therefore, **before**(s) **None** P is true.

The model will not be modified, since $\nexists\ a \in A : a =$ **at**$_{\mathbf{fail}}$(s) *Formula*, therefore,

None P can be propagated through the step, which results that **after**(s) **None** P is true.

In case of (ii), **at**$_{\mathbf{succ}}$(s) **ForEach** $P \rightarrow P'$ will be true, which means that after the step finishes, the rewriting can not be applied anymore, therefore, **after**(s) **Exists** P will not be true, which means that **after**(s) **None** P will be true.

Proposition 3. Let D be a consistent transformation description. Assume that

assertion $a =$ (**before**(s) **None** P) is true, where $s \in S$ and P is a pattern. **None** P

can be propagated through s, in other words, (**after**(s) **None** P) will be true, if $\forall\ a$

: $D \vdash a$, $a = (\mathbf{at}_{\mathbf{succ}}(s)\ Formula)$, or $a = \mathbf{at}_{\mathbf{fail}}(s)\ Formula$ implies that, if $Formula =$

ForEach $P_1 \Rightarrow \Rightarrow P_2$, or $Formula =$ **ForOne** $P_1 \rightarrow P_2 \Rightarrow \nexists\ n_A, n_B$: $n_A \in N_{P2}$, or n_A

$\in N_{P1}$, and $n_B \in N_P$ such that $type_{P2}(n_A)$ is equal with, or inherits from $type_P(n_B)$.

Also, $\nexists\ e_A, e_B$: $e_A \in E_{P2}$, or $eA \in E_{P1}$, and $e_B \in E_P$, such that $type_{P2}(e_A)$ is equal

with, or inherits from $type_P(e_B)$

Proof (of Proposition 3). The conditions in Proposition 3 claim that there are no instances of a pattern P in the model under transformation before the current step, and the step does not create, does not delete and does not modify any elements that can appear in an instance of pattern *P*. If an instance of a pattern exists after the application of the step, its elements must have been present in the model before the application of the rule with the same attribute values, therefore, an instance of *P* would have been present before the application of the step. It would be a contradiction, which results that no instances of pattern *P* is present after the application of the step.

Finally, to demonstrate that functional and non-functional properties can be analyzed together using our concept we present a proposition for proving termination of a model transformation.

Proposition 4. Let *D* be a consistent model transformation description. If there are

no loops in the control graph *C* and for each step $s \in S$, $D \vdash (\mathbf{at}_{\mathbf{succ}}(s)$

Terminates), and $D \vdash (\mathbf{at}_{\mathbf{fail}}(s)$ **Terminates**) then the execution of the

transformation terminates.

Proof of (Proposition 4.) Since there are no loops in the control graph of the transformation, it means that each step will be executed at most once. The conditions of the proposition states that the execution of each step terminates, irrespectively of the fact if the application of the step was successful. Hence, the execution consists of finite number of parts, each of these parts terminates, therefore, the whole transformation terminates.

Proposition 5. Let D be a consistent transformation description. Assume that the following assertions with static formulas can be derived from the description: $D \vdash a_1 = prefix$ **Any** $P_1 \rightarrow P_2$, and $D \vdash a_2 = prefix$ **None** P_2. In this case, $D \vdash a_3 = prefix$ **None** P.

Proof (of Proposition 5). *prefix* selects a concrete point of the transformation, where no instance of P_2 exists (a_2). If an instance of P_1 would be present at this point, an instance of P_2 would also be present, because a_1. Therefore, no instances of P_1 can be present at this point of the transformation.

4.3 Realization of the Reasoning System

VMTS is our n-level metamodeling and model transformation framework. For the reasoning system, we use SWI-Prolog [15] that is an Prolog environment licensed under the Lesser GNU Public License. Deduction rules, including the ones presented in Section 4.1, are defined in the Prolog environment. In VMTS, a transformation description with basic assertions describing the rewriting rules can be generated automatically from the definition of any model transformation: (i) Rules of the original transformation will be steps of the generated descriptions. (ii) For each rules a *ForEach*, or *ForOne* Formula is automatically generated, including the appropriate patterns. The generator parses constraints and imperative code attached to the rules and use them to generate attribute constraints for the patterns. At its current state, VMTS can handle only a very restricted set of types of code constructions, such as value assignment and simple conditions, such as if an attribute has a specified value.

Conclusions

In this paper, we have presented our concept of an automated reasoning system for verifying properties of model transformations. We have presented a set of deduction rules for the reasoning system and have proven the correctness of the derived assertions. Our proposed solution is a new approach that supports offline

analysis of model transformations. Even the number of deduction rules defined in this paper are small, our system is designed to be as extendable as possible, therefore, future development is possible. Our concept supports semi-automated verification as well, which means that the knowledge of transformation experts and the results of their manual analysis can be built in and used by the reasoning system. A realization of the reasoning system in VMTS has been outlined. We have shown that an initial assertion set of the system is generated automatically from the definition of model transformations.

Our work can be continued in multiple directions. In order to analyze more and more aspects of model transformations and make our concept able to be applied in more complex case studies, new types of assertions should be introduced and more deduction rules need to be provided. We want to investigate consistency of different assertion sets, because manually added assertions may conflict with the existing ones. We also want to analyze complex control structures of model transformations, such as loops. Hopefully, this approach may result a complex system that is able to be applied in industrial solutions as well.

References

[1] Object Management Group (OMG): OMG Unified Modeling Language (OMG UML) Superstructure Specification, V2.1.2. (November 2007)

[2] Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. Software, IEEE 20(5) (Sept.-Oct. 2003) 42-45

[3] Mens, T., Van Gorp, P., Varró, D., Karsai, G.: Applying a model transformation taxonomy to graph transformation technology. Electronic Notes in Theoretical Computer Science 152 (March 2006) 143-159

[4] Visual Modeling and Transformation System (VMTS) website. http://vmts.aut.bme.hu/

[5] Rozenberg, G.: Handbook on graph grammars and computing by graph transformation, foundations, vol.1. World Scientific, (1997)

[6] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Volume XIV of Monographs in Theoretical Computer Science. An EATCS Series. Springer (2006)

[7] Plump, D.: Termination of graph rewriting is undecidable. Fundam. Inf. 33(2) (1998) 201-209

[8] Asztalos, M., Lengyel, L., Levendovszky, T.: A formalism for describing modeling transformations for verification. In: MODEVVA 2009, Model-Driven Engineering, Verification, And Validation, Denver, Colorado, USA (2009)

[9] Varró, D.: Towards formal verification of model transformations. In: PhD Student Workshop of FMOODS 2002, Formal Methods for Open Object-Based Distributed Systems,. Enschede, Hollandia. (2002)

[10] Karsai, G., Narayanan, A.: Towards verification of model transformations via goal-directed certification. Model-Driven Development of Reliable Automotive Services: Second Automotive SoftwareWorkshop, ASWSD 2006, San Diego, CA, USA, March 15-17, 2006, Revised Selected Papers (2008) 67-83

[11] Bisztray, D., Heckel, R.: Rule-level verification of business process transformations using csp. ECEASST 6 (2007)

[12] Blech, J. O., Glesner, S., Leitner, J.: Formal verification of java code generation from uml models, Fujaba Days (september 2005)

[13] Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic invariant verification for systems with dynamic structural adaptation. In: ICSE '06: Proceedings of the 28th international conference on Software engineering, New York, NY, USA, ACM (2006) 72-81

[14] Anastasakis, K., Bordbar, B., Kster, J. M.: Analysis of model transformations via alloy. In: Proceedings 4th Workshop on Model-Driven Engineering, Verification and and Validation (MoDeVVA'07). (October 2007) 47-56

[15] SWI-Prolog wesite. http://www.swi-prolog.org/

[16] Dijkman, R. M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in bpmn. Inf. Softw. Technol. 50(12) (2008) 1281-1294