ELSEVIER
Science of Computer Programming 44 (2002) 205–227
Science of Computer Programming
www.elsevier.com/locate/scico

Designing the automatic transformation of visual languages

Dániel Varró*, Gergely Varró, András Pataricza[1]

Department of Measurement and Information Systems, Budapest University of Technology and Economics, H-1521, Budapest, Magyar tudósok körútja 2, Hungary

Abstract

The design process of complex systems requires a precise checking of the functional and dependability attributes of the target design. The growing complexity of systems necessitates the use of formal methods, as the exhaustiveness of checks performed by the traditional simulation and testing is insufficient.

For this reason, the mathematical models of various formal verification tools are automatically derived from UML-diagrams of the model by mathematical transformations guaranteeing a complete consistency between the target design and the models of verification and validation tools.

In the current paper, a general framework for an automated model transformation system is presented. The method starts from a uniform visual description and a formal proof concept of the particular transformations by integrating the powerful computational paradigm of graph transformation, planner algorithms of artificial intelligence, and various concepts of computer engineering.

Keywords: System verification; Validation; Graph transformation; Model transformation; Visual languages; Planner algorithms; MOF; UML; Dependability

1. Introduction

For most computer controlled systems, especially dependable, real-time systems for critical applications, an effective design process requires an early conceptual and

* Corresponding author.
E-mail addresses: varro@mit.bme.hu (D. Varró), gervarro@cs.bme.hu (G. Varró), pataric@mit.bme.hu (A. Pataricza).

[1] This work was supported by the Hungarian National Scientific Foundation Grant (OTKA T030804) and the Foundation for the Hungarian Higher Education and Research (FKFP 0193).

0167-6423/02/$ - see front matter
PII: S0167-6423(02)00039-4

architectural validation prior to the implementation in order to avoid costly re-design cycles. All relevant system characteristics have to be checked during this *system verification* phase in order to have a guaranteed design quality. These parameters identify critical bottlenecks to which the system is highly sensitive.

The increasing need for effective design has necessitated the development of standardized and well-specified design methods and languages, which allow system, developers to work on a common platform of design tools. The *Unified Modelling Language* (*UML*) is a visual specification language for pure software systems, as well as for embedded real-time systems (systems reactively interacting with their environment). The UML represents a collection of best engineering practises that have proven successful in the modelling of large and complex systems. Recently, UML has been regarded as the standard object-oriented modelling language.

1.1. Formal methods in system design

Formal methods are mathematics-based techniques offering a rigorous and effective way to model, design and analyze computer systems. They have been a topic of research (in projects like IOSIP [8], SafeRail [4], SpeciMen [6] or HIDE [2]) for many years with valuable academic results. However, their industrial utilization is still limited to specialized development sites, despite their vital necessity originating in the complexity of IT products and the increasing requirements for dependability and Quality of Service (QoS).

The use of formal verification tools (like e.g. PVS [13]) in IT system design is hindered by a gap between practice-oriented CASE tools and sophisticated mathematical tools. On the one hand, system engineers usually show no proper mathematical skills required for applying formal verification techniques in the software design process. On the other hand, even if a formal analysis is carried out, the consistency of the manually created mathematical model and the original system is not assured. Moreover, the interpretation of analysis results, thus the re-projection of the mathematical analysis results to the designated system is problematical. From the engineering point of view, the notion of dependability is a composite one necessitating the analysis of multiple mathematical properties by using different verification tools.

The aim of our ongoing research is to provide a provenly correct and complete, automated transformation between UML-based system models and formal mathematical verification tools for an effective software design.

1.2. Mathematical model transformation

The step generating the description of the target design on the input language of mathematical tools from the UML model of the system is called *mathematical model transformation*. The inverse direction of model transformation (referred as *back-annotation*) is of a similar practical importance as well when some problems (e.g. a deadlock) are detected during the mathematical analysis. After an automated

back-annotation these problems can be observed in the same UML system model allowing the designer to fix conceptual bugs within his well-known UML environment.

The practical application of transformation based design and verification necessitates the analysis of the UML model from different aspects. This way a transformation environment has to support the implementation of several transformations towards different mathematical tools.

Several semi-formal transformation algorithms have already been designed and implemented for different purposes (e.g. formal verification of functional properties and quantitative analysis of dependability attributes [3]). Unfortunately, this conventional way (i.e. experiences in the experimental implementation process) of model transformation raised several problems.

- The lack of unique and formal descriptions of the transformation algorithms resulted in hand-written and rather ad hoc implementations (inconvenient for implementing complex transformations).
- Any formal proof of correctness and completeness of these transformation scripts is almost impossible, hence their uncertain quality remains a bottleneck of the entire transformation based verification approach.
- Each model had to be verified individually although the transformation algorithms have similar underlying algorithmic skeletons.

As a conclusion, a general and automated transformation method was missing, which would generate the target models from a well-formed, high-level specification.

1.3. Research objectives

Our proposal for the previous problems is a general mathematical model transformation system supporting the automated generation of transformation code of a proven quality for an arbitrary number of transformations. Such an automated model transformation system has to fulfil at least the following requirements [20]:

- *Requirement* 1: The easy-to-understand *description of source and target models* (based on metamodels) in order to support a variety of transformations;
- *Requirement* 2: A visual but mathematically precise *description of transformation rules* clearly indicating the correspondence between the elements of the UML visual programming paradigm and the target mathematical notation;
- *Requirement* 3: An efficient *back-annotation* of mathematical analysis results (aiming to visualize the results of the analysis in a UML notation);
- *Requirement* 4: An engine for *proving semantic correctness and completeness* of transformations;
- *Requirement* 5: An *automatic model generation* from the visual transformation rules.

Transformations correctly proved necessitate a precise underlying mathematical structure for both source models (like UML) and target models (such as Kripke structures, Petri Nets, computational tree logic, etc.). Additionally, model transformation and back-annotation also have to be ruled strictly and precisely (Requirements 1–3).

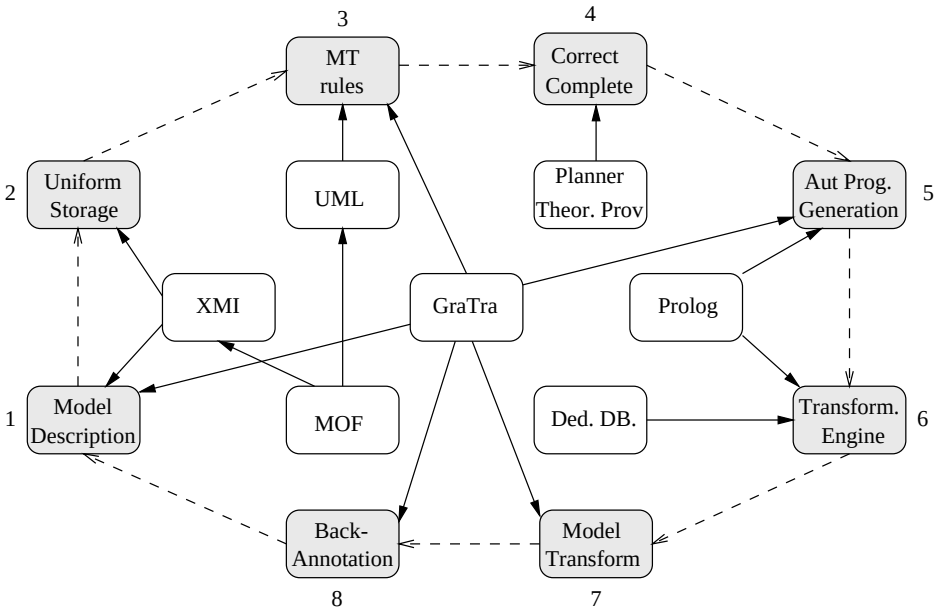


Fig. 1. An overview of VIATRA.

The quality of model transformation (Requirements 4 and 5) should be ensured by an automated proof method for correctness and completeness, which step will be followed by an automated program generation phase. The program derived takes a specific UML model as input and generates the language of a particular verification tool as the output. As a result, the quality bottlenecks originating in the former heuristic implementation (manual coding) could be eliminated.

1.4. VIATRA: a visual automated model transformation system

Our model transformation approach (which is an integration of different disciplines of artificial intelligence, and computer engineering) is based on formal mathematical background and provides a general transformation description language and methodology for a large scale of transformations.

The process of model transformation is characterized by a model analysis round-trip (illustrated by the sequence of rounded grey boxes in Fig. 1).

(1) *Model description*. A model transformation for practical applications necessitates on one hand a uniform and precise description of source and target models to improve the quality of such transformations. But, on the other hand, it should follow the main standards of the industry in order to be integrated to software design methodologies.

For this reason, the *Meta Object Facility* (*MOF*) metamodelling techniques are used in VIATRA. MOF metamodels provide graphical means to define

metaobjects for similarly behaving instances in various domains by combining the expressive power of UML Class diagrams (concerning the structure) with the Object Constraint Language (OCL) for describing semantic issues. MOF metamodels are used as a basis for describing UML models (following the standard metamodel of UML) as well as mathematical structures (by creating non-standard metamodels for them).

A typical UML model contains more details than required for a specific mathematical analysis (for instance, documentation or use case diagrams are frequently of little importance). Thus, in the sequel, a UML model will only contain the relevant pieces of information with respect to a specific analysis, and this reduced model can be obtained from the original user-created system model by some filtering mechanism.

In VIATRA, filtering is expressed by metamodels. Exactly those constructs are regarded as relevant (thus transformable) that is are included in the metamodel of the source language (hence if specific constructs are irrelevant for one purpose, they are simply omitted from the metamodel).

(2) *Uniform representation of models.* The front-end and back-end of transformations (UML as the source model and a formal verification tool as the target model) is defined by a uniform, standardized description language of system modelling, that is, *XMI* (*XML Metadata Interchange*). XMI is a special metamodel dependent collection of XML constructs providing an XML representation for arbitrary (MOF based) models.

XMI seems to be a natural choice as a large number of UML tool vendors provide a facility to export their models into XMI, moreover, several academic communities (e.g. the graph transformation community [19]) have started discussion to settle on a general XML based interchange format for their tools.

(3) *Model transformation rules.* The visual specification of model transformations is supported by *graph transformation* [1,8,15], which combines the advantages of graphs and rules into an efficient computational paradigm.

A *graph transformation rule* is a special pair of pattern graphs where the instance defined by the left-hand side is substituted with the instance defined by the right hand side when applying such a rule (similarly to the well-known grammar rules of Chomsky in computational linguistics).

Model transformation rules (in the form of graph transformation rules) are specified by using a visual notation of UML. However, for obtaining a tool-independent transformation specification, the transformation rules will also be exported in an XML based format, conforming to the evolving standard of graph transformation systems [19].

(4) *Correctness and completeness.* Automated transformations necessitate an automated proof method aiming to verify that the generated target models are semantically correct (correctness problem). Moreover, each construct allowed in the source model should be handled by a corresponding rule (completeness problem).

Instead of verifying the semantic correctness of individual target models we put the stress on the *correctness and completeness of transformation rules*, i.e. starting

from a source model that fulfils some semantic criteria, the derivation steps should always keep these properties invariant for the target design.

(5) *Automated program generation.* Even if the description of the transformation is theoretically correct and complete, additionally, the source and target models are also mathematically precise, the implementation of these transformations has a high risk on the overall quality of a transformation system. As a possible solution, *automatic transformation algorithm generation* is carried out for implementing visual transformation rules and control structures.

(6) *The transformation engine.* As being a logic programming language based on powerful unification methods, Prolog seems to be a suitable language for a prototype implementation of the transformation engine. Thus, the XMI based models and rule descriptions are translated into a Prolog graph notation serving as the input data and the program to be executed, respectively. After a successful prototyping phase, Prolog could be substituted with a more powerful but lower abstraction level language (like C++ or Java).

(7) *Model transformation.* Model transformation is performed by executing the automatically generated Prolog program supplied with the Prolog description of the source model. According to our experiments (see Section 4 for benchmark applications), the time required for the transformation is just a few percentage of the total time spent on the formal analysis exploring the entire state space.

(8) *Back-annotation of analysis results.* The results of the mathematical model transformation are planned to be automatically back-annotated to the UML based system model. Thus, the system analysts are reported from conceptual bugs in their well-known UML notation. Unfortunately, the current version of UML does not directly support the representation of analysis traces. For instance, the sequence of fired statechart transitions that leads to a deadlock (according to the verification tool) completely lacks a fine-grained UML representation.

As model transformations are frequently projections in a mathematical sense, thus, they cannot be inverted in general. Moreover, several formal analysis methods often perform another model transformation (e.g. a deadlock detection algorithm may take the description of a transition system as input and may generate a sequence of fired transitions as output). For this reason, back-annotation is not equivalent with an inverse model transformation, as it only requires the identification of related source and target objects.

1.5. Related results

In Section 1.3, we set up several requirements that has to be *simultaneously* fulfilled by a general purpose model transformation system. Currently, a brief overview of previous results (typically achieved in projects with different objectives and orientation) is provided concerning these individual fields.

- *Model description*: Model transformations necessitate a formal specification of models in arbitrary domains with a special emphasis on defining a precise semantics for UML. PROGRES [17] provides a general framework for graph models and their

transformations with typed graphs conforming to graph schemata. In FUJABA, UML models are formally specified by combining UML and graph transformation [10], while an algebraic presentation of models is used for defining the semantics of UML in [11].

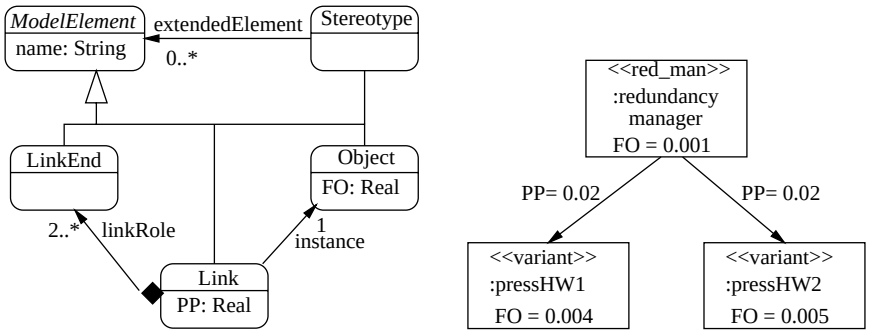
As a convenient handling of multiple metamodels (e.g. establishing correspondence between source and target languages) are not addressed by these approaches, reference graphs (and metamodels) are introduced in VIATRA.

- *General rule representation*: Model transformation rules require a visual, easy-to-understand and platform independent notation. The idea of using UML as a specification language of graph transformation rules (by collaboration and activity diagrams) first appeared in [12]. Alternately, a general description of rules can be obtained on an XML level [19]. In VIATRA, rules are specified in a (slightly different) UML notation, and intended to be exported conforming to the evolving GTXL standard.
- *Correctness and completeness*: Traditional graph transformation approaches (e.g. double pushout [5], single pushout [9]) elegantly deal with the soundness of a *single* transformation step (proving that the resulting object is a graph). However, in model transformation systems, the notion of correctness and completeness is related to an arbitrary sequence of transformation rules.
- Automatic program generation: Using graph transformation as a programming language forms the basic idea of PROGRES ([17]), which generates C code for the implementation of graph transformation rules. VIATRA generates executable Prolog code from high level UML based rule specifications as further verification steps will be based on these Prolog term representations.
- *Model transformation approaches*: MTRANS [14] provides an XSLT based framework for transforming MOF compliant models. Transformation scripts are specified in a purely textual language. Individual model transformations were designed in RIVIERA [16], where UML models are transformed to the Maude language in order to carry out formal analysis and verification. VIATRA is a general model transformation system for automating the creation of *different* models for formal verification.
- *Back-annotation*, *inverse transformation*: The triple graph grammar (TGG) approach [18] provides a general bi-directional transformation mechanism for graph languages. TGG uses correspondence graphs coupling the source and target objects. However, in order to obtain bi-directionality, there is a loss in efficiency by having costly graph pattern matching for both directions. Model transformation rules in VIATRA are uni-directional (i.e. traditional graph transformation rules), thus back-annotation can be more efficient by using simple reference relations instead of graph pattern matching.

2. Model description

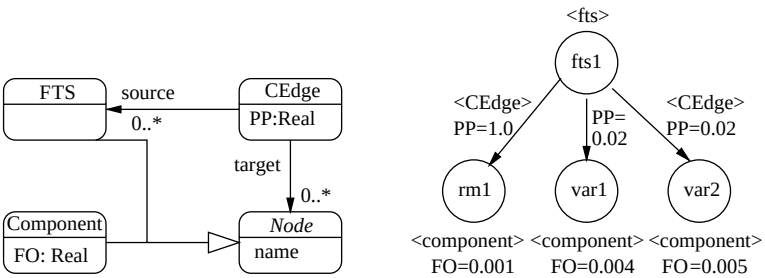
2.1. Running example

In order to simultaneously demonstrate the technicalities of model transformation and the typical questions from the engineering background, we selected a small fragment



(a) The UML metamodel (b) The source UML model

Fig. 2. The source language: UML object diagram.



(a) The IM metamodel (b) The target IM model

Fig. 3. The target language: IM hypergraph.

of a complex transformation as the basis of our running example. The complete transformation (discussed in details in [3]) generates stochastic Petri Nets from static UML models enriched with special dependability attributes (e.g. failure rate of components). Each static relation between high-level objects is regarded as a potential error propagation path. This Petri Net based analysis aims at the identification of dependability bottlenecks in an early phase of design.

The entire transformation is divided into two major steps. At first, a Intermediate Model (in the form of a simple hypergraph) is derived in order to extract important dependability attributes from UML models. Afterwards, the Petri Net model can be transformed straight from this intermediate hypergraph representation (without the use of original UML models).

In our running example, the transformation of fault tolerant structures will be performed from static UML models (depicted in Fig. 2) to this intermediate graph representation called Intermediate Model (IM in Fig. 3).

The simplified metamodel of UML (describing stereotyped objects and links between them with abstract classes printed in italics) is depicted in Fig. 2(a). The

metamodel is enriched with two dependability parameters: the fault occurrence rate FO in objects and fault propagation probability PP of links (as potential propagation paths).

The sample source UML model (in a visual UML notation in Fig. 2(b)) represents a fault-tolerant structure which consists of three objects, a redundancy manager (redundancy_manager) and two variants (pressHW1 and pressHW2) identified by the corresponding stereotypes (red_man and variant) and the links between them. The redundancy manager is responsible for switching from one variant to the other when an error is detected.

The target IM metamodel in Fig. 3(a) specifies a hypergraph consisting of (i) nodes of type FTS representing the fault tolerant structure as a whole, (ii) nodes of type Component standing for system components, (iii) and edges of type CEdge indicating the "composed of" relation.

In the sample target IM hypergraph model in Fig. 3(b), (i) a single graph node of type Component is assigned to each variant object (var1, var2), (ii) two distinct nodes (fts1 and rm1 of types FTS and Component, respectively) are assigned to each redundancy manager, (iii) the fts node is in a 'composed of' relation with the remaining three nodes as indicated by the edges of type CEdge.

2.2. Graph models

In this section, basic concepts of graph transformation systems (such as graphs, graph transformation rules, transformation units, etc.) are applied to the special needs of model transformation built upon MOF metamodels in order to provide a precise (but still practice oriented) mathematical background. For the basic definitions (such as directed, typed and attributed graphs), the reader is referred to [1].

Definition 1. *A **model graph** G is a directed, typed and attributed graph with the following structure (expressed e.g. by a corresponding schema graph).*

- *A graph node is associated with a unique identifier* Id, *and a type label* $\mathsf{T_n}$.
- *An edge has an own* Id, *a reference to a source* $\mathsf{Id_S}$ *and a target* $\mathsf{Id_T}$ *identifier, and a type label* $\mathsf{T_e}$.
- *Both nodes and edges may be related to attributes (represented e.g. as special graph nodes) with an* Id *identifier (referring to the graph element the attribute is related to), a type label* $\mathsf{T_a}$ *and a data value* V.

Model graphs may also contain *n*-ary relations between nodes (denoted as relations or hyperedges), but these relations are represented by a special class of model graph nodes connected to "original" graph nodes by special (reserved) types of edges. Model graph relations are closely related (in their use and functionality) to PROGRES path expressions with multiple source and/or target nodes.

From MOF models to model graphs: Model graphs are derived automatically from MOF based models. Each node and edge must have type labels corresponding to an

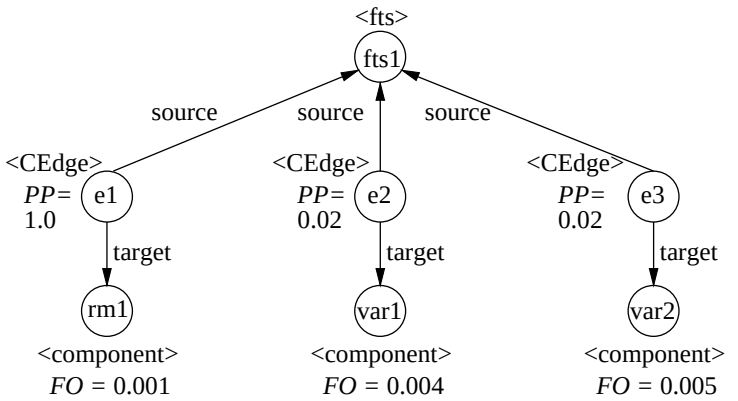


Fig. 4. Model graphs from MOF based models.

MOF construct. The most fundamental rules of this derivation are the following:

- Instances of an MOF Class (`A`) are mapped into model graph nodes with identically named types.
- Associations in MOF based models are typically non-directed links between instances. However, model graphs are directed graphs, each (navigable) AssociationEnd of a MOF Association (`E`) between two MOF Classes (from `A` to `B`) is projected into a separate model graph edge. A further type restriction states that all the graph edges of type `E` have to connect a graph node of type `A` to a node of type `B`.
- MOF attributes are directly mapped into model graph attributes.

With this respect, the model graph of Fig. 4. is an equivalent of the IM model of Fig. 3(b) (attributes are printed in italics).

Reference graphs: As the main goal of model transformation is to derive a target model from a given source model, source and target objects must be linked to each other in some way to form a single graph. For this reason, the following definition introduces the concepts of a *reference graph*. The structure of a reference graph is also constrained by a corresponding metamodel, which contains (i) references of existing source and target metamodel nodes; (ii) novel (so-called) reference nodes that provide a typed coupling of source and target objects, and (iii) reference edges connecting all these nodes.

Definition 2. *A* ***reference graph*** $\mathsf{G_{ref}} = (\mathsf{G_s}, \mathsf{G_t}, \mathsf{NODES_{ref}}, \mathsf{EDGES_{ref}})$ *contains a source and a target model graph* ($\mathsf{G_s}$ *and* $\mathsf{G_t}$, *respectively*), *and an additional set of reference nodes* $\mathsf{NODES_{ref}}$ *and edges* $\mathsf{EDGES_{ref}}$, *where*

- *a* ***reference node*** *is a model graph node* (*thus associated with a unique identifier* Id, *and a type label* $\mathsf{T_n}$) *of the reference metamodel.*
- *a* ***reference edge*** *is a model graph edge* (*of the reference metamodel*) *that may lead* from a reference node *to either a source, a target or a reference element of a specific type.*

As several models (such as statecharts, subnets in high-level Petri nets) are structured into a hierarchy, the previous definition of reference graphs can be extended by allowing reference nodes to relate (sub)graphs to (sub)graphs (in addition to single nodes). As a result, a more flexible reference structure is obtained, however, the reference graph is no longer a simple graph but a hierarchical graph.

2.3. Transformation rules

Definition 3. *A* ***graph transformation rule*** $\mathsf{r} = (\mathsf{L}, \mathsf{R}, \mathsf{App})$ *contains a left-hand side* (*LHS*) *graph* L, *a right-hand side* (*RHS*) *graph* R, *and application conditions* App.

The *application* of r to a *host graph* (graph instance) G replaces an occurrence of the LHS L in G by the RHS R. In general, this is performed by

(1) finding an occurrence of L in G (also denoted as graph pattern matching),
(2) checking the application conditions App (such as negative application conditions which prohibit the application of the rule in the presence of certain nodes and edges),
(3) removing a part of the graph G determined by the occurrence of L yielding the *context* graph D,
(4) gluing R and the context graph D and obtaining the **derived** graph H.

The adaption of graph transformation rules to model transformations prescribe special requirements for the structure of these rules. As the target model is constructed from scratch, model transformation rules are frequently non-deleting, which ensures the pleasant property of being able to handle all the LHS matches parallelly.

On the other hand, when the deletion of certain graph objects is prescribed by a rule, we must ensure that distinct parallel matches do not conflict with each other. In our model transformation approach, parallelly executable rules cannot remove any part of the graph to avoid such problems.

Following the classification of different graph transformation approaches that can be found in [15], a model transformation rule is defined as follows:

Definition 4. *A* ***model transformation rule*** r_{mt} *is a special graph transformation rule, where*

- *both graphs* L *and* R *are reference graphs*;
- *an occurrence of* L *in* $\mathsf{G}_{\mathsf{ref}}$ *is required to be an isomorphic image of* L;
- *all the dangling edges are deleted automatically*;
- *non-deleting rules are matched* (*and executed*) *parallelly as default*.

A sample model transformation rule is depicted in Fig. 5. Please note that in order to improve the clarity of the illustrations, only a graphical representation of rules is indicated and the underlying model graph structure is omitted. Negative application conditions are denoted by objects embedded in a region painted gray.

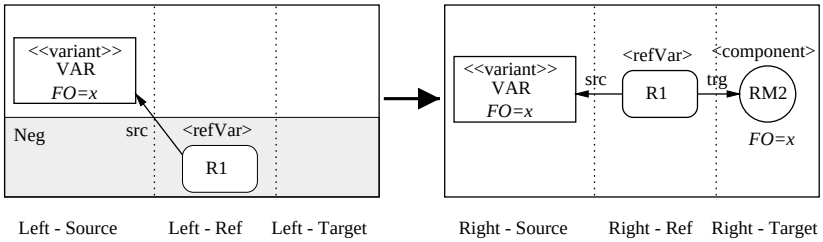


Fig. 5. A sample model transformation rule.

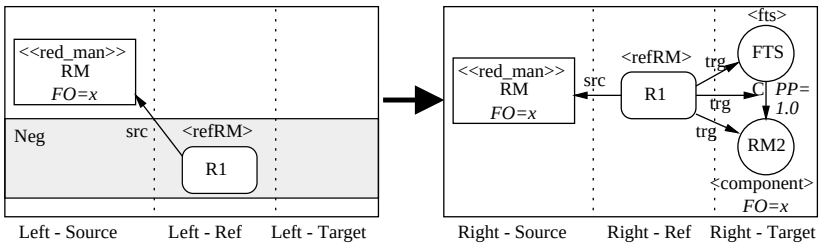


Fig. 6. Model transformation rule "ftsR".

The LHS of this rule requires a UML object with the stereotype `variant` to be present on the source side without a reference edge to a reference node of type `refVar` (indicated by the negative condition), while there are no restrictions for the target design. According to the RHS, a new IM node of type `component` and a new reference node of type `refVar` is inserted and connected to the UML object by corresponding reference edges (of type `src` and `targ`). In addition to structural modifications, the value of an object's attribute `FO` is also projected into the target design.

As possible industrial applications of model transformation surely consist of very large and complex models containing hundreds of rules, model transformation rules must be extended by a sophisticated structuring mechanisms that allow to compose them in a modular way. In the graph transformation community, the concepts of *transformation units* were introduced for this purpose (e.g. [1]), which units are adapted for structuring model transformations in VIATRA.

2.4. A sample transformation

Our sample model transformation is carried out by three transformation rules applied in the specific order: `ftsR`, `variantR` and `linkR`. The process of our sample transformation is the following (characterized by the *parallel* application of rules in correspondence with the default semantics).

(1) Transform all the redundancy managers in the UML model into two connected IM nodes (`ftsR`; Fig. 6).

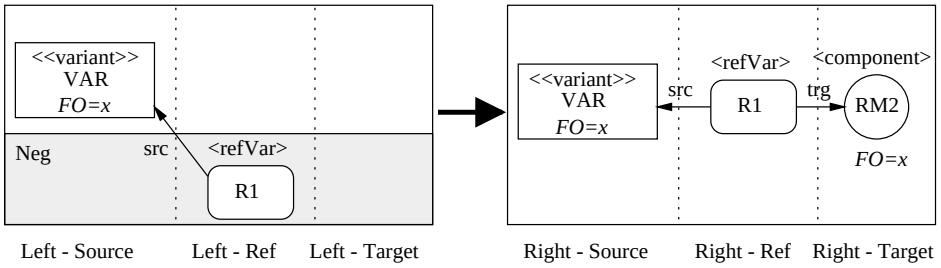


Fig. 7. Model transformation rule "variantR".

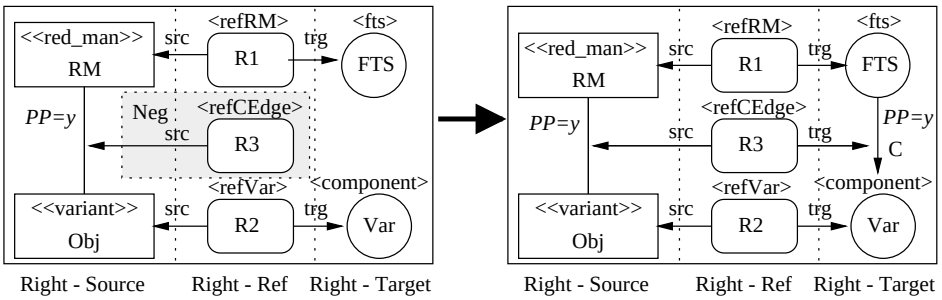


Fig. 8. Model transformation unit "linkR".

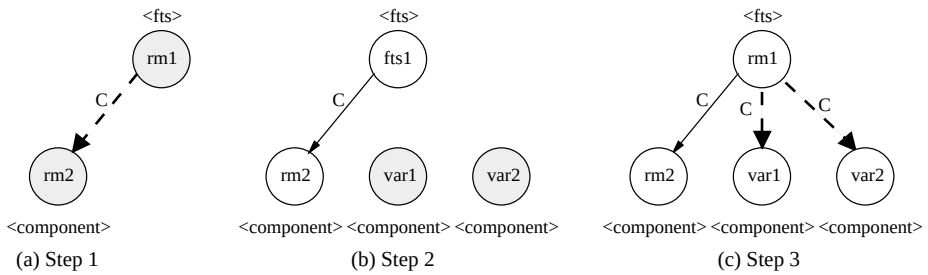


Fig. 9. Tracing the transformation step by step.

(2) Create a new IM node for each UML object with stereotype "*variant*" (`variantR`; Fig. 7).
(3) Link (by applying `linkR` in Fig. 8) each IM equivalent of variant objects with the corresponding equivalent of a node of type `fts` (i.e. fault tolerant structure). By equivalent we mean related reference nodes and edges between the elements.

The construction of the target IM model is illustrated in Fig. 9. Nodes that have been created most recently are colored grey while new edges have dashed lines.

Step 1: An `fts` node and an `component` node are created by applying the transformation rule `ftsR`.

Step 2: The `component` nodes are constructed by applying transformation rule `variantR` for the two occurrences of a source variant object.

Step 3: The `fts` nodes are connected to the `component` nodes (as a result of applying `linkR`) by adding a new IM edge. Each pair of these nodes are linked only once, which is ensured by the negative context condition on the LHS.

As a conclusion, we illustrated that model transformation can be specified by means of graph transformation rules. In the following, we concentrate on the proven quality of transformations (namely, correctness and completeness).

3. Correctness and completeness of transformations

The validity of some system requirement formulated in a temporal logics representation is typically verified by applying model checking or theorem proving techniques. The proofs constructed by theorem provers are general in the sense that they hold in any models of the specification. Model checkers, on the contrary, operate on given model instances (i.e. finite representations of a general underlying theory). In this section, the concepts of correctness and completeness for model transformation systems (called *syntactic well-formedness* in the sequel) will be defined analogously on two levels.

- *Model dependent approach*: In this first case, the well-formedness of individual transformation instances (i.e. the transformation of a specific source model) are checked.
- *Metamodel/grammar dependent approach*: In this case, we are aiming to prove that the transformation is correct for any instance of the source metamodel (i.e. any sentence of the source grammar).

Before being able to discuss the correctness of model transformations, one has to decide what a correct model and a correct transformation is.

- A first idea may be to prove the *semantic equivalence* of source and target models. Unfortunately, many model transformations are projections (i.e. some constructs of the source model are not transformed) thus semantic equivalence cannot be proven.
- In the future, we are planning to set up special *invariant criteria* that a transformation must preserve, which criteria would be checked by of model checking or theorem proving techniques.
- However, in the current paper, correctness is defined by means of visual languages and graph grammars in analogy with traditional computational linguistics (thus on a *syntactic level*).
  - *Model dependent* (*simple*) *correctness* is defined by means of parsing the visual sentences generated by a model transformation by using the graph grammar of the target language.
  - *Metamodel dependent* (*total*) *correctness* is stronger than the previous as it aims to prove that *each model* generated by a model transformation system is a sentence of the target language. As the structure of model transformation rules mainly resembles to the structure of the source model, this problem is not at all trivial.

In the sequel, we suppose that a graph grammar exists for each metamodel (e.g. a graph grammar of UML or IM) which controls the construction of well-formed visual sentences. As the process of editing such models is not considered, we may also suppose that *no rules of these graph grammars prescribe the deletion of some elements*. Naturally, the structure of model transformation rules is not restricted in this sense.

3.1. Model correctness

Thus, in the current paper, a model is considered to be correct whenever it can be derived from the start graph (regarded as an axiom) by graph transformation rules (regarded as deduction rules).

Definition 5 (Graph grammar). *Let* $\mathcal{G} = (\mathcal{S}, \mathcal{R}_{\mathcal{G}})$ *be a production system with the axiom* $\mathcal{S}$ (*start graph*) *and deduction rules* $\mathcal{R}_{\mathcal{G}}$ (*graph transformation rules*). *This system is called a* graph grammar (*considering that all graph nodes are terminal nodes*).

Definition 6 (Derivable). *Let* $\mathcal{G} = (\mathcal{S}, \mathcal{R}_{\mathcal{G}})$ *be a graph grammar. We call a graph* $\mathcal{M}$ (*called* model *or* sentence *later*) derivable *from* $\mathcal{G}$ (*denoted as* $\mathcal{G} \vdash \mathcal{M}$) *iff* $\mathcal{M}$ *can be obtained from the start graph* $\mathcal{S}$ *by a finite sequence of graph transformation steps using deduction rules* $\mathcal{R}_{\mathcal{M}}$.

Definition 7 (Visual language). *Let* $\mathcal{G}$ *be a graph grammar. The* visual language (*of the graph grammar*), *denoted as* $\mathcal{L}_{\mathcal{G}}$, *contains all the graphs that are derivable from* $\mathcal{G}$.

$$\mathcal{L}_{\mathcal{G}} = \{\mathcal{M} \mid \mathcal{G} \vdash \mathcal{M}\},$$

where $\mathcal{M}$ *is graph called* (*visual*) sentence.

3.2. Transformation correctness

After discussing the correctness of models, correctness of model transformation will be introduced, built upon well-formed source models as axioms.

Definition 8 (Model transformation). *Let* $\mathcal{T} = (\mathcal{M}_{\mathcal{A}}, \mathcal{R}_{\mathcal{T}})$ *be a graph grammar with the start graph* $\mathcal{M}_{\mathcal{A}}$, *called source model* (*a sentence of the source visual language*), *and model transformation rules* $\mathcal{R}_{\mathcal{T}}$. $\mathcal{T}$ *is denoted as* model transformation.

Definition 9 (Model transformation system). *A* model transformation system *is a tuple* $\mathcal{MTS} = (\mathcal{A}, \mathcal{R}_{\mathcal{T}}, \mathcal{B})$, *where* $\mathcal{A}$ *and* $\mathcal{B}$ *are graph grammars defining the source and target language, respectively, and* $\mathcal{R}_{\mathcal{T}}$ *is a set of model transformation rules.*

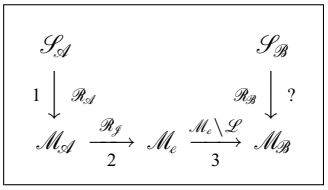


Fig. 10. Concepts of correctness.

Corollary 10. *Let* $\mathcal{MTS} = (\mathcal{A}, \mathcal{R}_{\mathcal{T}}, \mathcal{B})$ *be a model transformation system.* $\forall \mathcal{M}_{\mathcal{A}} : \mathcal{A} \vdash \mathcal{M}_{\mathcal{A}}$, $\mathcal{T} = (\mathcal{M}_{\mathcal{A}}, \mathcal{R}_{\mathcal{T}})$ *is a model transformation.*

Several models may be derived by a model transformation system from different source models. However, such a system is of little importance if these derived models are incorrect sentences of the target graph grammar. To express the difference, models that are derived by a model transformation system will be denoted as *model candidates*. This concept of correctness is illustrated in Fig. 10.

Definition 11 (Derivable as target). *Let* $\mathcal{T} = (\mathcal{M}_{\mathcal{A}}, \mathcal{R}_{\mathcal{T}})$ *be a model transformation. A graph* $\mathcal{M}_{\mathcal{B}}$ (*called* model candidate) *is* derivable as target *from* $\mathcal{T}$ (*denoted as* $\mathcal{T} \vdash_t \mathcal{M}_{\mathcal{B}}$ or $\{\mathcal{M}_{\mathcal{A}}, \mathcal{R}_{\mathcal{T}}\} \vdash_t \mathcal{M}_{\mathcal{B}}$) *iff*

$$\exists \mathcal{M}_{\mathcal{C}} : (\mathcal{T} \vdash \mathcal{M}_{\mathcal{C}}) \wedge (\mathcal{M}_{\mathcal{A}} \cup \mathcal{M}_{\mathcal{B}} = \mathcal{M}_{\mathcal{C}} \setminus \mathcal{Q}),$$

where $\mathcal{Q}$ *is the set of all reference nodes and edges in* $\mathcal{M}_{\mathcal{C}}$.

Model transformation rules build a common supergraph $\mathcal{M}_{\mathcal{C}}$ containing the original source model $\mathcal{M}_{\mathcal{A}}$ and the novel target model candidate $\mathcal{M}_{\mathcal{B}}$, which are connected by reference nodes and edges $\mathcal{Q}$. The target model candidate can be obtained from this supergraph if the original source model and the reference objects are removed.

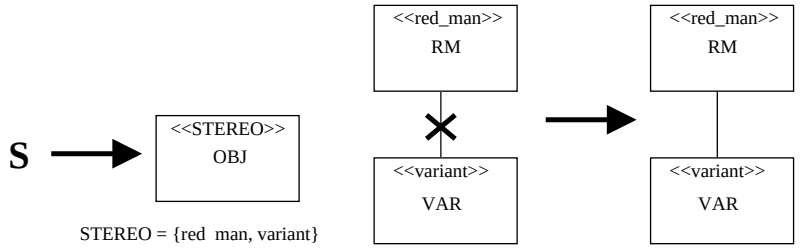
In the following, the most important notions of model transformation systems (namely, correctness and completeness) are defined.

- Informally, a *model transformation is correct*, if the derived target candidate is a sentence of the target language (model dependent).
- A *model transformation system is correct*, whenever correctness holds for each source model (metamodel dependent).

Definition 12 (Correctness: model transformation). *Let* $\mathcal{T} = (\mathcal{M}_{\mathcal{A}}, \mathcal{R}_{\mathcal{T}})$ *be a model transformation and* $\mathcal{B}$ *be a graph grammar defining the target language.* $\mathcal{T}$ *is correct* (*with respect to* $\mathcal{B}$) *iff*

$$\mathcal{T} \vdash_t \mathcal{M}_{\mathcal{B}} \Rightarrow \mathcal{B} \vdash \mathcal{M}_{\mathcal{B}}.$$

Such a correctness will also be denoted as simple correctness.



(a) Adding an object (`addObject`) (b) Adding a UML link (`addLink`)

Fig. 11. Graph grammar of the source UML models.

Definition 13 (Correctness: model transformation system). *Let* $\mathcal{MTS} = (\mathcal{A}, \mathcal{R}_{\mathcal{T}}, \mathcal{B})$ *be a model transformation system.* $\mathcal{MTS}$ *is correct iff*

$$\forall \mathcal{M}_{\mathcal{A}} : \mathcal{A} \vdash \mathcal{M}_{\mathcal{A}} \wedge \{\mathcal{M}_{\mathcal{A}}, \mathcal{R}_{\mathcal{T}}\} \vdash_t \mathcal{M}_{\mathcal{B}} \Rightarrow \mathcal{B} \vdash \mathcal{M}_{\mathcal{B}}.$$

Such a system is also called total correct.

Completeness is also defined on two levels: finding an appropriate source model for a given target sentence and for each target model.

Definition 14 (Completeness: model transformation). *Let* $\mathcal{B}$ *be a graph grammar, and* $\mathcal{M}_{\mathcal{B}}$ *be a model of this grammar. A model transformation* $\mathcal{T}$ *is complete* (*with respect to* $\mathcal{M}_{\mathcal{B}}$) *iff*

$$\mathcal{B} \vdash \mathcal{M}_{\mathcal{B}} \Rightarrow \mathcal{T} \vdash_t \mathcal{M}_{\mathcal{B}}.$$

This process (finding one or more source models for a given target sentence) is also called *back-annotation* or simple completeness.

Definition 15 (Completeness: model transformation system). *Let* $\mathcal{MTS} = (\mathcal{A}, \mathcal{R}_{\mathcal{T}}, \mathcal{B})$ *be a model transformation system.* $\mathcal{MTS}$ *is complete iff*

$$\forall \mathcal{M}_{\mathcal{B}} \exists M_A : \mathcal{B} \vdash \mathcal{M}_{\mathcal{B}} \Rightarrow \{\mathcal{M}_{\mathcal{A}}, \mathcal{R}_{\mathcal{T}}\} \vdash_t \mathcal{M}_{\mathcal{B}} \wedge \mathcal{A} \vdash \mathcal{M}_{\mathcal{A}}.$$

3.3. Examples on correctness and completeness

In the sequel, theoretical foundations are illustrated on our running example of the UML-IM model transformation.

At first, a model transformation system is created by source and target languages defined by graph grammar rules. Such a small grammar for handling the running example may be the following (shown in Figs. 11 and 12, where S is a placeholder for any empty LHS). Negative application conditions follow the traditional notation

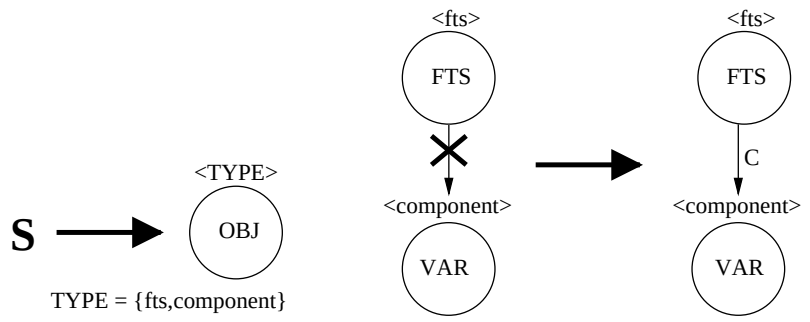


Fig. 12. Graph grammar of the target IM models.

(crossed edges) while typing constraints are interpreted as application conditions for the transformation rule.

- Let $\mathcal{MTS}$ be a model transformation system with source grammar $\mathcal{A}$ of Fig. 11, target grammar $\mathcal{B}$ of Fig. 12, and model transformation rules $\mathcal{R}_{\mathcal{T}}$ are the ones in Figs. 6–8.
- Let $\mathcal{T}$ be a model transformation with source model $\mathcal{M}_{\mathcal{A}}$ of Fig. 2(b) and model transformation rules $\mathcal{R}_{\mathcal{T}}$.
- Let $\mathcal{M}_{\mathcal{B}}$ of Fig. 3(b) be a target model candidate.

Proposition 16. *$\mathcal{T}$ is correct (with respect to $\mathcal{B}$).*

Proof. It is sufficient to show a sequence of derivations that is able to construct $\mathcal{M}_{\mathcal{B}}$ from graph grammar $\mathcal{B}$. In other words, we have to show that the sentence $\mathcal{M}_{\mathcal{B}}$ can be parsed by using the inverse rules of grammar $\mathcal{B}$.

Let us consider the following sequence (in the given order within the same group), where `addNode(A,B)` means to add the node `B` of type `A`, while `addEdge(C.D)` adds an edge between nodes `C`, and `D`:

(1) `addNode(fts,im1)`, `addNode(component,im2)`, `addEdge(im1,im2)`,
(2) `addNode(component,var1)`, `addNode(component,var2)`,
(3) `addEdge(im1,var1)`, `addEdge(im1,var2)`.

One can easily notice that the derivation process is similar to the one of Fig. 9. As a result, the same graph is constructed in each case. □

Corollary 17. *$\mathcal{T}$ is complete (with respect to $\mathcal{M}_{\mathcal{B}}$).*

Proposition 18. *$\mathcal{MTS}$ is correct.*

Proof. In this proof, each application of a model transformation rule will be related to a sequence of target grammar rules. Thus, when a model transformation rule is

Model transformation rules	Target grammar rules

`ftsR`	`addNode(fts,FTS),`
	`addNode(component,RM2),`
	`addEdge(FTS,RM2)`

`variantR`	`addNode(component,VAR)`
`linkR`	`addEdge(FTS,VAR)`

Fig. 13. Rule coupling for proving correctness.

applied, we try to apply the corresponding sequence of grammar rules. If the target model candidate and the parallelly generated target model is isomorphic after each model transformation step then the model transformation must be correct as well.

In other words, starting from the target graph on the LHS of a model transformation rule, the target graph on the RHS of the similar rule has to be created by the graph grammar rules of the target language (empty target side is related to the $\mathscr{S}$ start symbol).

Let us consider the following coupling of rules (depicted in Fig. 13).

As a result, the modifications performed by model transformation rules on the target model candidate are simulated by graph grammar rules of the target language, thus $\mathscr{MTS}$ is correct. □

Proposition 19. *$\mathscr{MTS}$ is not complete* (*unfortunately*).

Proof. For a counterexample, let us consider a target model $\mathscr{M}'_{\mathscr{B}}$ with an individual node of type `fts`. Let us suppose that there exists a source model $\mathscr{M}'_{\mathscr{A}}$ which can be transformed to $\mathscr{M}'_{\mathscr{B}}$.

Such a source model must contain a redundancy manager object as it is the only object that is projected into an `fts` node. When performing the transformation of the redundancy manager, an additional node and edge will appear in the target model candidate.

As the graph grammar of the target language does not contain any rules that would be able to remove graph nodes and edges, the original target model must be a subgraph of the resulting model candidate. □

Please note that if another set of model transformation rules were used (splitting `ftsR` into three rules: one is generating the `fts` node and the second one derives the `component` node, finally, the third one creates the link between those two), completeness could have been proven.

All the proofs presented here (especially the proofs of correctness) may serve as skeletons for further proofs in connection with model transformation systems. However, the problem of an automated verification still remains. In the following, the sketch of

such an automated proof method for syntactic correctness is presented, based upon planner algorithms of artificial intelligence.

3.4. Proving correctness by planner algorithms

Planner algorithms [21] are complex, hierarchical problem solving procedures subdividing the original problem into smaller parts before trying to solve them according to the "divide and conquer" principle. Finally, these partial solutions are merged together yielding the solution of the original problem.

Definition 20. *A* planner $\mathscr{PA}:(\mathscr{I},\mathscr{E},\mathscr{O})\mapsto\mathscr{P}$, *is a structure where* $\mathscr{I}$ *is the first-order logics formulae of the initial state,* $\mathscr{E}$ *is the first-order logics formulae of the goal state, while* $\mathscr{O}$ *is the set of permitted operations. The output is plan* $\mathscr{P}$, *which is a sequence of operations providing a trajectory from the initial to the goal state.*

Definition 21. *A* planner operation $\mathscr{O}=(\mathscr{C},\mathscr{A})$, *where* $\mathscr{C}$ *stands for the preconditions* (*first-order logics formulae*), *and* $\mathscr{A}$ *for actions. Preconditions must hold before performing the specific operation. Actions may add or remove certain basic logics formulae* (*called facts*) *to the state space.*

In the following, a planner will be constructed to prove correctness of model transformations.

- Basic facts are built up from model graphs (supposing the close world assumption, i.e. when all the true facts have to be listed explicitly).
  - From a model graph node of type `type` with an identifier `id` the predicate `type(id)` is generated.
  - From a model graph edge of type `type` with its own `id`, source `src` and target `trg` identifiers, the predicate `type(id,src,trg)` is generated.
  - From a model graph attribute attached to the node identified by `id` with a name `name`, and having value `value`, the predicate `name(id,value)` is generated.
- Graph grammar rules (of the source and target language) are encoded into planner operations according to the following mapping:
  - The LHS of a rule together with application conditions are encoded into a planner precondition.
    - — LHS objects are encoded into positive predicates in the Prolog style, i.e. with (unbound) variables for `id`s.
    - — Negative application conditions are (universally quantified) negative statements.
    - — Further general conditions concerning uniqueness (and context) are added to the precondition of each operation. As our graph grammars do not contain deleting rules, postconditions are implicitly defined by the LHS objects and the additions prescribed by the RHS. This way, general postconditions such as the dangling edge condition need not be considered.
  - the changes defined by the RHS of the rule are mapped into planner actions defining element additions (serving as new postconditions).

Definition 22. *Let* $\mathscr{A}=(\mathscr{S},\mathscr{R}_{\mathscr{A}})$ *and* $\mathscr{B}=(\mathscr{S},\mathscr{R}_{\mathscr{B}})$ *to form the model transformation system* $\mathscr{MTS}=(\mathscr{A},\mathscr{R}_{\mathscr{T}},\mathscr{B})$. *The proof planner* $\mathscr{PA}_{\mathscr{T}}$ *of correctness is sequence of* $\mathscr{PA}_i$ *sub-planners* (*one assigned for each model transformation rule* $\mathscr{R}_i \in \mathscr{R}_{\mathscr{T}}$) *which are defined as follows*:

- *the* initial state *of* $\mathscr{PA}_i$ *is defined by the* left target *side graph of the* model transformation rule,
- *a* subgoal *of* $\mathscr{PA}_i$ *is defined by the* right target *side graph of the* model transformation rule.
- *the* operations *are defined by the* graph grammar rules $\mathscr{R}_{\mathscr{B}}$ *of the* target language.

Proposition 23. *If a plan can be constructed for each* $\mathscr{R}_i$ *then the model transformation system* $\mathscr{MTS}$ *is correct*.

Proof (Sketch). Speaking in graph transformation terms, we are aiming to prove that (i) whenever a model transformation (MT) rule is applied (to one specific match), (ii) and its effects can be simulated in general by applying a specific sequence of the target graph grammar (GG) rule on the target part of MT rule graphs, (iii) this specific sequence is applicable for the specific (isomorphic) match in the host graph thus deriving the parsing steps of the host graph from parsing just the MT rule graph. (Please note the differences of rule and host graphs; graph grammar and model transformation rules). □

According to our construction, performing a planner operation is identical to applying the related GG rule (without deletions). According to the assumption, there exist a sequence of GG rules that derives the right target graph (and not the image of it) from the left target graph of the MT rule. Such a sequence must not create additional graph objects (as side effects) due to the lack of deleting rules and the closed world assumption for the subgoal.

When an MT rule is applied, an isomorphic image of the initial and goal states are required to be present in the host graph. Thus, applying the same sequence of GG rules to that specific matching image, it will derive the image of the goal state (and nothing else).

Constructing a proof planner for correctness was only a demonstration. Similar planners can be built for completeness as well by slight modifications.

4. Conclusion

In the current paper, our initial results towards a complex model transformation method was presented intended to perform mathematical model transformations in order to integrate UML-based system models and mathematical models of formal verification tools. Due to the large complexity of IT systems, model transformations are supported by an integrated environment composed of various fields of computer science (planner algorithms, graph transformation) and software engineering (UML, MOF).

For obtaining a higher quality of transformations, the syntactic correctness and completeness of each transformation are proved, additionally, an executable Prolog code is derived automatically from high-level model transformation rules. As a result of an automated transformation and back-annotation of analysis results, a variety of formal verification tools will become available for system designers, without the thorough knowledge of underlying mathematics.

The following benchmark transformations have already been designed and implemented according to the model transformation concepts of VIATRA:

- Transforming the static aspects of UML models enriched with dependability parameters into stochastic Petri Nets for dependability analysis in an early phase of system design;
- Transforming UML Statecharts into Extended Hierarchical Automaton that provide a formal operational semantics for these UML diagrams;
- Automatic Prolog program generation for visual control structures.

In our future plans, the reconsideration of the proof method and the implementation of further model transformation are aimed at first. Semantic criteria that must be invariant to a model transformation may also be proved by using theorem provers and model checkers instead of planner algorithms.

References

[1] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, G. Taentzer, Graph transformation for specification and programming, Sci. Comput. Programming 34 (1999) 1–54.

[2] A. Bondavalli, M. Dal Cin, D. Latella, A. Pataricza, High-level integrated design environment for dependability, Proc. WORDS'99, Workshop on Real-Time Dependable System, 1999.

[3] A. Bondavalli, I. Majzik, I. Mura, Automatic dependability analyses for supporting design decisions in UML, Proc. HASE'99: The 4th IEEE International Symposium on High Assurance Systems Engineering, 1999, pp. 64–71.

[4] E. Canver, Einsatz von model-checking zur analyse von MSCs über statecharts, Technical Report, University of Ulm, April 1999.

[5] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe, in: G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, vol. 1, Foundations, World Scientific, Singapore, 1997, Ch. Algebraic Approaches to Graph Transformation—Part I: Basic Concepts and Double Pushout Approach, pp. 163–245.

[6] J. Desel, G. Juhas, R. Lorenz, Process semantics of Petri Nets over partial algebra, in: ICATPN: International Conference on the Application and Theory of Petri Nets 2000, 2000, pp. 146–165.

[7] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook on Graph Grammars and Computing by Graph Transformation, vol. 2, Applications, Languages and Tools, World Scientific, Singapore, 1999.

[8] H. Ehrig, R. Geisler, M. Grosse-Rhode, M. Klar, S. Mann, On formal semantics and integration of object oriented modeling languages, EATCS, vol. 70, 2000, pp. 77–81.

[9] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini, in: G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, vol. 1, Foundations, Ch. Algebraic Approaches to Graph Transformation—Part II: Single Pushout Approach and Comparison with Double Pushout Approach, World Scientific, Singaore, 1997, pp. 247–312.

[10] G. Engels, J.H. Hausmann, R. Heckel, S. Sauer, Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML, in: A. Evans, S. Kent, B. Selic (Eds.), UML

2000—The Unified Modeling Language. Advancing the Standard, Lecture Notes in Computer Science, vol. 1939, Springer, Berlin, 2000, pp. 323–337.
[11] J.L. Fernandez Aleman, A. Toval Alvarez, Seamless formalizing the UML semantics through metamodels, in: K. Siau, T. Halpin (Eds.), Unified Modeling Language: Systems Analysis, Design and Development Issues, Ch. 14. Idea Publishing Group, Hershey, PA, 2001, pp. 224–248.
[12] T. Fischer, J. Niere, L. Torunski, A. Zündorf, Story diagrams: a new graph transformation language based on UML and Java, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Proceedings of the Theory and Application to Graph Transformations (TAGT'98), Lecture Notes in Computer Science, vol. 1764, Springer, Berlin, 1998, pp. 269–309.
[13] S. Owre, N. Shankar, J. Rushby, D. Stringer-Calvert, The PVS language reference, Version 2.3, Technical Report, SRI International, September 1999.
[14] M. Peltier, J. Bézivina, G. Guillaume, MTRANS: a general framework, based on XSLT, for model transformations, in: J. Whittle et al. (Eds.), Workshop on Transformations in UML, 2001, pp. 93–97.
[15] G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, vol. 1, Foundations, World Scientific, Singapore, 1997.
[16] J. Saez, A. Toval Alvarez, J.L. Fernandez Aleman, Tool support for transforming UML models to a formal language, in: J. Whittle et al. (Eds.), Workshop on Transformations in UML, 2001, pp. 111–115.
[17] A. Schürr, Introduction to PROGRES, an attributed graph grammar based specification language, in: M. Nagl (Ed.), Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science, vol. 411, Springer, Berlin, 1990, pp. 151–165.
[18] A. Schürr, Specification of graph translators with triple graph grammars, Technical Report, RWTH Aachen, Fachgruppe Informatik, Germany, 1994.
[19] G. Taentzer, Towards common exchange formats for graphs and graph transformation systems, in: J. Padberg (Ed.), UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques, 2001.
[20] D. Varró, G. Varró, A. Pataricza, Designing the automatic transformation of visual languages, in: H. Ehrig, G. Taentzer (Eds.), GRATRA 2000, Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems, 2000, pp. 14–21.
[21] D.S. Weld, An introduction to least commitment planning, AI Mag. 15 (4) (1994) 27–61.