Defining and Validating Transformations of UML Models

Jochen M. Küster and Reiko Heckel and Gregor Engels
Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
Germany
{jkuester,reiko,engels}@upb.de

Abstract

With the success of the UML, the ability of transforming models into programs or formal specifications becomes a key to automated code generation or verification in the software development process. In this paper, we describe a concept for specifying model transformations by means of graph transformation rules on the UML meta model.

In order to validate the termination and uniqueness of such transformations we derive a number of sufficient criteria from basic results of the theory of graph transformation. This ensures that the rules can be executed automatically while, at the same time, providing a high-level visual model of the transformation.

1. Introduction

Models expressed in the Unified Modeling Language (UML) [13] are widely used in practice. In addition to documentation and communication means, UML models also serve as a basis for automated code generation and for studying properties of the system prior to its actual implementation.

In this context, there often arises the need to transform a model into a formal specification, program, or another model. Such model transformations are required, for example, to perform consistency checks on the translated model [12, 4], and for defining code generation from visual models.

Thus, model transformations are one of the new key technologies in software engineering. Existing approaches for specifying such transformations often follow a declarative, set-theoretic or logic approach, see for example [6]. This can be seen as a mathematical requirement specification of a transformation. However, when we have to implement model transformations and to reason about their properties, such declarative specifications provide little help.

In this paper, we will study executable model transformations and provide a sound formal model for defining and validating them. First, we discuss in more detail the requirements which arise from current applications of model transformations. Then, as a running example, we will consider a transformation of statecharts into CSP [10] and introduce rule-based model transformations with control expressions to specify this transformation. After formalizing this concept in terms of attributed graph transformation, we exploit the theory of graph transformation to derive sufficient criteria for the functionality (termination and uniqueness) of transformations.

2. Requirements for Model Transformations

A model transformation specifies a mapping from a source to a target (modeling) language, not necessarily distinct.

For semantic consistency checking, a UML model is translated into a semantic domain which provides a formal language to define and tool support to check consistency conditions. As an example, we have studied transformations from the source language UML to the formal language CSP, denoted also as the target language. Using the transformation from UML to CSP syntax, consistency conditions on UML models can be formulated in the language of CSP and validated by existing model checkers.

Besides consistency checking, other applications of model transformations include the visualization of the results of a model checker and the code generation from models. In all cases, a model transformation must be functional, i.e., it should terminate and for a given source model a unique target model should be computed. A second requirement is that model transformations are executable, i.e., that they allow an automated transformation. Finally, both consistency conditions and the policy for code generation are highly dependent on the methodology, application domain, and target platform employed. Therefore, model transformations will have to be flexible enough to be specified on
a project-by-project bases and intuitive enough to be understood by domain experts.

To meet these requirements, we will introduce rule-based model transformation which allows the specification of transformations in an operational way, thereby meeting the requirement of executability. Concerning the requirement of functionality, we will develop sufficient criteria that allow the software engineer to ensure such functionality. The graphical notation of transformation rules supports an intuitive understanding while the rule-based nature itself allows the flexibility to exchange and modify rules when the requirements for the mappings change.

In contrast to existing work on model transformations by Varro et al. [17], Whittle [18] and Akehurst et al. [1], we concentrate on achieving the functionality requirement of model transformations.

An alternative approach to model transformation could be based on techniques from compiler construction, like parsing and attribute grammars. These techniques are often more efficient than graph transformation approaches. However, their concepts and notation are more low-level and harder to use, especially if the transformation is still under development. We believe that our approach is especially suitable to try out different variants of mappings and derive prototype implementations from their conceptual descriptions.

3. Rule-based Model Transformations with Control Conditions

To provide a concept for specifying model transformations, we will now introduce rule-based model transformations with control conditions by means of an example of translating of statecharts to CSP.

3.1. Concept of Rule-based Model Transformations

A model transformation for translating a model from a source language to a target language can be defined by a synchronized transformation on the source and the target language. Each transformation can be specified by a set of model transformation rules. Typically, source transformation rules will be identical transformations leaving the source model unchanged.

In Figure 1, two model transformation rules for UML statecharts are shown (based on [9] and [16]). Each transformation rule \( r : (r_s, r_t) \) consists of two parts, a UML part and a CSP part and will therefore be referred to as a compound rule. Concerning \( p_1 \) in the figure, \( r_s \) is the UML part on the left, \( r_t \) is the CSP part on the right. Both the \( r_s \) and \( r_t \) can be viewed as graph transformation rules [2]. Note that \( r_s \) is an identical transformation with \( L_s = R_s \) and visualized by the left side only.

In general, the source transformation rule \( r_s : L_s ::= R_s \) describes the transformation of the source model, the target transformation rule \( r_t : L_t ::= R_t \) specifies the transformation of the target model. Regarding the rule \( p_1 \) in the figure, \( L_s \) is the UML model shown, \( L_t \) is the epsilon (denoting emptiness) in the CSP part and \( R_t \) is the CSP expression shown on the right.

Source and target rules are coupled by the ability of using shared variables. Such variables are denoted by \( \text{(variable)} \). For example, \( \text{SMName} \) is such a variable in the compound rule \( p_1 \). Furthermore, we allow the use of multi-objects on the left side of the source transformation rule and target transformation rule to denote sets of meta model instances. Each rule where a multi-object occurs on the left side of the source transformation rule can be viewed as a collection of rules, where each rule in the collection fixes the number of objects in the multi-object.

For translation of a source to a target model, we describe how a compound rule is applied, assuming that \( L_s = R_s \) and that \( X = \{x_1, \ldots, x_n\} \) is the set of variables of \( L_s \):

1. an occurrence of the left side \( L_s \) of the source transformation rule is searched within the source model, such an occurrence is called source match.
2. having found a source match, the variables are given concrete values, leading to a variable instantiation denoted \( X' \).
3. the left side \( L_T \) of the target transformation rule is instantiated with the values of the variables, denoted also by \( L_T(X') \).
4. an occurrence of the instantiated left side of the target transformation rule is searched within the target model, such an occurrence is called target match.
5. the right side \( R_T \) of the target transformation rule is instantiated with the values of the variables.

6. the occurrence of the instantiated left side is replaced with the instantiated right side \( R_T \) of the target transformation rule.

In order to specify model transformations with control, compound rules must be assembled to a so-called transformation unit [11] consisting of a set of compound transformation rules with control. Aiming at a computationally complete approach [7], we will assume that rules are organized in rule sets. These rule sets are then organized in a sequence of rule sets where each rule set can be considered as a layer. Within a rule set, rules may be applied nondeterministically. A transformation unit consists of a set of compound rules together with a control expression specifying the organization of rules into rule sets, layers and determining whether a rule should be applied once or as long as possible.

Syntactically, we express layers of rule sets by specifying a sequence of rule sets. For example, assuming three rules \( p_1, p_2, p_3 \), then \( \{\{p_1, p_2\}, p_3\} \) specifies two layers, the first one containing \( p_1, p_2 \) and the second one containing \( p_3 \). This means that first all rules within the first layer are applied and then the ones in the second layer. For each rule set, it is indicated whether it is applied once or as long as possible by a simple marker. For example, \( p \downarrow \) denotes that the rule \( p \) is iterated until it cannot be applied anymore.

### 3.2. Formal Definition of Transformation Units

We will now formalize our concept of rule-based model transformations as introduced in the previous section. Formally, a transformation rule on the source or target language can be seen as a typed attributed graph transformation rule where models in both the source and the target language are specified formally as typed attributed graphs.

In the following, we will briefly summarize the concepts of typed attributed graph transformation, for a detailed formal introduction to the topic the reader is referred to [8].

An attributed graph \( AG \) where only graph vertices can be attributed is a pair consisting of a directed unlabeled graph \( G = (G_V, G_E, \text{src}, \text{targ}) \), and a \( \Sigma \)-Algebra \( A \). The disjoint union of all carrier sets of \( A \) is denoted by \( |A| \). The elements of \( |A| \) represent potential attribute values which are regarded as special data vertices of the graph (besides the object vertices modeling structural entities). An object vertex \( v \in G_V \) has an attribute value \( a \in |A| \) if there is an attribute edge from \( v \) to \( a \) in \( AG \). An attributed type graph is an attributed graph where \( A \) is the \( \Sigma \)-Algebra modeling the types of object vertices. An attributed instance graph is an attributed graph with an additional typing morphism which specifies the type of all object vertices of the graph.

As an example, we will show how a UML model can be viewed as an attributed graph. In Figure 2, an attributed type graph (an extract from the UML statechart meta model) and an attributed instance graph (modeling a simple statechart) are shown. Attribute values are modeled as round vertices, graph vertices as squared vertices. By providing a meta model for CSP, CSP models can also be considered as typed attributed graphs.

Typed attributed graphs are manipulated by graph transformation rules. A typed attributed graph transformation rule \( r : L ::= R \) is composed of a rule name \( r \) and a pair of typed attributed graphs \( L \) and \( R \). It fixes a set of variables \( X \) and is attributed over the term algebra \( T_{\Sigma}(X) \), meaning that the graphs in the rule may have as attributes values obtained from terms expressed over variables in \( X \).

A graph transformation system \( GTS \) consists of a set of rules \( \mathcal{R} \). These rules induce a relation \( \implies \) on the set of graphs. One writes \( G \xrightarrow{r(o)} H \) for denoting that the graph \( H \) is derived from graph \( G \) applying the rule \( r \in \mathcal{R} \) at the occurrence \( o \). A transformation sequence \( G_0 \xrightarrow{r_1(o_1)} \cdots \xrightarrow{r_n(o_n)} G_n \) in \( GTS \) is a sequence of consecutive transformation steps such that all rules \( r_i \) are from \( \mathcal{R} \).

A compound rule as introduced informally above can now be considered as a pair of typed attributed graph transformation rules specified over the same set of variables:

**Definition 1 (Compound rule)** A compound rule \( r : (r_s, r_t, X) \) consists of a rule name \( r \), a common set of variables \( X \) and a pair \( r_s \) and \( r_t \) of typed attributed graph transformation rules specified over the same \( \Sigma \)-Algebra \( T_{\Sigma}(X) \).

Compound rule could be interpreted as rule triples as defined by Schuerr [15] by providing a suitable correspondence rule \( r_c : L_C ::= R_C \). Similar to rule triples, for each compound rule, a joined attributed graph transformation rule can be constructed. Given \( r_s : L_S ::= R_S \) and
Two attributed graphs $A$ and $B$ are joined at the data vertices. Two attributed graphs $G$ and $H$ specified over the same $\Sigma$-Algebra $A$ are joined at the data vertices by constructing a union such that $G$ and $H$ overlap in $|A|$ and are otherwise disjoint, i.e. $G \cap H = |A|$. In Figure 3, rule $p_1$ is shown as a joined typed attributed graph transformation rule in formal syntax.

The rule application strategy can be expressed by a control expression over the names of the rules. Any valid transformation sequence of a graph transformation system must comply to the control expression, i.e. the transformation sequence must be allowed by the control expression. We will now define a transformation unit [11] that can be thought of as a graph transformation system together with a control expression and a class of graphs on which rules will operate. Here, we use a simplified form of a transformation unit that can also be considered as a programmed graph transformation system. We define its syntax as follows.

Definition 2 (Syntax of a transformation unit) Let a graph transformation system $GTS = (R)$ be given and let $N$ be the set of rule names of $R$. A transformation unit $TU$ is a tuple $(G, GTS, S)$ consisting of a class of graphs, a graph transformation system and a control expression $S$ defined over the rule names, assuming that name $\in N: Exp ::= Seq \mid Set \mid Exp \mid |$ name

Seq ::= (Exp, ..., Exp)

Set ::= {Exp, ..., Exp}

The control expression $S$ restricts the possible transformation sequences that are allowed within a transformation unit. The idea of the semantics of a transformation unit is to define a new relation $\mathcal{T}U$ based on the relation of the graph transformation system that is restricted to those transformation sequences conforming to the control expression and generating graphs which are in the allowed class of graphs. We define the semantics of a transformation unit as follows:

Definition 3 (Semantics of a transformation unit) Let a transformation unit $TU = (G, GTS, S)$ be given. We define the relation $\mathcal{T}U$ based on the underlying relation in $GTS$ recursively as follows:

- $\mathcal{T}U = \{G \Rightarrow H \mid G, H \in \mathcal{G} and G \Rightarrow H is a transformation step in GTS\}$
- $\{(E_1, E_2) \Rightarrow \mathcal{T}U = \{G \Rightarrow H \mid s = s_1; \ldots; s_n and s_1: G \Rightarrow H_1 \in E_{TU} and H_1 = G_{i+1}$ for $i = 1 \ldots n-1\}$
- $\{E_1, E_2\} \Rightarrow \mathcal{T}U = \bigcup_{i \in \{1; \ldots, n\}} E_i \Rightarrow \mathcal{T}U$
- $E_1 \Rightarrow \mathcal{T}U = \{G \Rightarrow H \mid s = s_1; \ldots; s_n and s_1: G \Rightarrow H_1 \in E_{TU} and H_1 = G_{i+1}$ for $i = 1 \ldots n-1 and \not\exists s_{n+1}: H_n \Rightarrow K \in E_{TU}\}$

We say that a graph tuple $(G_0, G_n)$ is derived within $TU$ if there exists a transformation sequence $G_0 \Rightarrow \mathcal{T}U G_n$.

For the following, we will consider a transformation unit for statecharts to CSP, specified by the rules $p_1, p_2, p_5, p_{14}, p_{17}$ in Figure 1 and Figure 4. We assume a control expression of $(p_1, p_2 \downarrow, p_5 \downarrow, p_{14} \downarrow, p_{17})$. This means that first the rule $p_1$ must be applied once for generating the framework of a CSP process for a statechart, then $p_2$ is applied as long as possible for deriving behavior of each
simple state. Afterwards, transitions are encoded by applying rule \( p_5 \), a so-called static reaction is encoded by rule \( p_{14} \) and the behavior of a final state is generated by \( p_{17} \).

4. Termination and Confluence of Transformation Units

In this section, we deal with the requirement of functional behavior (functionality). Functional behavior of a graph transformation system means that it is terminating and confluent.

Given a transformation unit \( TU = (G, GTS, S) \) and fixing a start graph \( G_0 \), functionality ensures that there always exists a unique graph \( G_n \), derived after a finite number of steps in \( TU \). Note that functionality also includes the case that given a start graph \( G \) no derivation according to the control expression \( S \) exists. In the formal definition of the semantics, the resulting rewrite relation would be undefined for \( G \). In practical applications, this corresponds to the possibility that after \( n \) derivations, no rule conforming to the control expression is applicable.

Concerning the usage of mapping rules from UML to a semantic domain, functional behavior guarantees that there is exactly one model of the semantic domain. Otherwise, there could be either two or more models (if the mapping is not confluent) or no model at all (if the mapping is not terminating).

In the following, we will examine termination and confluence for transformation units with control and provide a set of sufficient criteria that allows the software engineer to ensure functional behavior.

4.1. Termination

In general, termination is undecidable for graph transformations. As a consequence, given a graph transformation system, there are no mechanisms that allow the automated decision whether or not the graph transformation system is terminating [14]. Nevertheless, termination can be achieved by respecting a set of criteria, well-known from the general theory of abstract reduction systems. Many termination proofs follow the idea of an embedding into the abstract reduction system \( (\mathbb{N}, >) \), by constructing a so-called monotone measure function. In practice, this means that it must be assured that every transformation step decreases the value of some expression that is bounded from below.

Concerning termination of a transformation unit with control, the following observations can be made: each rule (set) that is applied only once does not pose a problem because it cannot lead to infinite rule application sequences. However, if a set of rules can be applied as long as possible, it must be guaranteed that no infinite rule application occurs.

Given a control expression \( p \), this can be achieved by assuring that the left sides of a compound rule can only be matched a finite number of times and that a compound rule cannot be applied twice at the same match and that no new possibilities for matching are generated.

We can assure this either by the property of the start graph or by the effect of the rule application. As an example, the reader is referred to the transformation unit for translating statecharts to CSP introduced above. Here, \( p_2 \) is to be applied as long as possible. However, assuming a finite start graph, there exists only a finite number of \( s:\text{SimpleState} \) within the UML model, giving rise also to a finite number of \( \text{State}(\langle SName \rangle, \text{static}) \) vertices in the CSP part (note that we talk of vertices in the CSP part resulting from the formalization as attributed graphs). As the UML model is not changed and each rule application removes one \( \text{State}(\langle SName \rangle, \text{static}) \), the rule can only be applied finite number of times. Formally, this can be shown by assuming a measure function mapping each model to the number of \( \text{State}(\langle SName \rangle, \text{static}) \) vertices it contains.

Concerning further forms of control expressions we propose the following criteria for termination.

Criteria 1 For a transformation unit with a control expression \( S \) we make the following observations concerning its termination:

- \( S = p \) is terminating
- \( S = \{ p_1, ..., p_n \} \downarrow \) is terminating if the \( GTS \) with the rules \( p_1, ..., p_n \) is terminating.
- \( S = \{ E_1, ..., E_n \} \) is terminating if each \( E_i \) is terminating
- \( S = \{ E_1, ..., E_n \} \) is terminating if each \( E_i \) is terminating

The second condition is a generalization of the example above. Note that this corresponds to the classical termination problem which is often difficult to show for rule sets involving more than one rule. The second and third condition give criteria that allow to derive termination of transformation units with compound control expressions, given that each subexpression induces termination.

4.2. Confluence

Whereas termination of a transformation unit can be achieved by analyzing sets of rules being applied as long as possible for their termination, confluence of a transformation unit is more difficult to achieve. For attributed graph transformation systems, a sufficient criteria for confluence is obtained by a so-called critical pair lemma [8]. Informally, a critical pair represents a pair of non-parallel independent transformation step in a minimal context. A critical pair lemma then typically shows local confluence of a
transformation system in the case that all critical pairs are strongly joinable. In the following, we will define parallel independence for our compound rules and then develop a set of sufficient criteria for functionality of transformation units.

**Independence of Compound Rules** Given a graph transformation system $GTS$, two transformation steps $H_1 p_{\pi_1}^{o_1} G \xrightarrow{p_{\pi_2}^{o_2}} H_2$ are parallel independent, if they can be applied in any order yielding the same result. This is the case if occurrences do only overlap in elements (objects, links, attributes) that are not deleted by $p_1$ and $p_2$.

We can lift the concept of parallel independence of transformation steps to rules as follows: $p_1$ and $p_2$ are parallel independent if and only if for all $H_1 \xrightarrow{p_1} G \xrightarrow{p_2} H_2$ the two steps are parallel independent.

As an example, consider $p_2$ and $p_5$ of our transformation unit: $p_2$ deletes $State(\{SName\})$ and $p_5$ deletes $Transition(\{Source\}, \{Target\})$ in the corresponding attributed graphs. This means that a match of $p_2$ will include a $State(\{SName\})$ and a match of $p_5$ a $Transition(\{Source\}, \{Target\})$. But these two items will be preserved, i.e. $p_2$ does not delete a $Transition(\{Source\}, \{Target\})$ and vice versa. As a consequence, any two transformation steps of these two rules are parallel independent, which gives rise to parallel independence of the rules themselves.

We will often be interested in parallel independence of a rule with itself. The following proposition captures a sufficient condition for parallel independence of a compound rule:

**Proposition 1 (Parallel independence of compound rule)** Let $TU = (G, (R), S)$ be given and let $p : (r, r, X) \in R$ be a compound rule such that $r_i$ does not delete elements. Let further $Y$ be the set of variables used in data vertices on the left side of $r_i$.

If there exists at most one possible match for $L_S$ and $L_T$ for any instantiation $Y^I$ of $Y$ and if for any overlapping variable instantiations $Y^I_1, Y^I_2$ there does not exist $G \in G$ containing $L_T(Y^I_1)$ and $L_T(Y^I_2)$, then $(p, p)$ is parallel independent.

**Proof Sketch 1** Assume that there exists at most one possible match for $L_S$ and $L_T$ for any instantiation of $Y$. Consider a possible transformation step $H_3 p_{\pi_3}^{o_3} TU G p_{\pi_2}^{o_2} TU H_2$. Either the matches $o_1$ and $o_2$ involve the same or a different variable instantiation on $Y$. In the first case, the matches $o_3$ and $o_2$ must be identical, by prerequisite there exist at most one match for $L_S$ and $L_T$ for a given variable instantiation. In the second case, the two transformation steps can be applied in any order, because the matches do not overlap in the variable instantiations and $r_i$ does not delete elements.

Note first that the prerequisite that there does not exist a graph containing $L_T s$ with overlapping variable instantiations is only relevant if there is more than one variable in $Y$. As an example, with regards to $p_5$, it must be ensured that there do not exist graphs containing a $Transition$ object with more than a source and a target data vertex, leading to a possibly overlapping variable instantiation.

Applying the proposition to $p_2$ yields that $(p_2, p_2)$ is parallel independent if there exist only SimpleStates with different names (which is usually the case). Note that otherwise $(p_2, p_2)$ would not be independent: a rule with two matches, both deleting the same State(\{SName\}, static) can usually not be applied in any order. Similarly, $(p_1, p_1)$ is parallel independent if there exists only one state machine in the UML model.

Note further that if in a UML rule there exist free variables that are not part of the left side of the CSP rule, then the compound rule is usually not parallel independent, if the UML model does not fulfill certain, additional assumptions: for parallel independence of $p_5$ the UML model must fulfill two assumptions (which it does): There exists only one StateMachine vertex and, for each transition between two states, there exists a unique trigger event and action.

**Criteria** In a graph transformation system we encounter two sources of non-determinism: the selection of the rule and the selection of the match. For a transformation unit we must respect the control expression restricting the possible derivations.

The selection of the match may lead to non-functionality of the transformation unit. Assuming a transformation unit $TU$ with control expression $S = \langle p_1, ..., p_n \rangle$, derivations are of the form $G \xrightarrow{p_1} H_1 \xrightarrow{p_2} ... \xrightarrow{p_n} H_s$. Usually such a transformation unit is not functional if there exist several matches for a $p_i$ leading to different $H_s$.

On the other hand, if there is always at most one match for each $p_i$, then the transformation unit will be functional. With regards to the example of statecharts, that means that $p_1$ is functional if graphs in the class of graphs have at most one StateMachine vertex with a unique name (which is usually the case for UML models). This observation leads us to the first sufficient criteria:

**Criteria 2** Let a transformation unit $TU = (G, GTS, S)$ be given and let $S = \langle p_1, ..., p_n \rangle$. Then $TU$ is functional if for all $p_i$ and for all $G \in G$ there exists at most one match of $p_i$ in $G$.

**Proof Sketch 2** By induction on $n$:

- $n = 1$: for $S = \langle p_1 \rangle$ $TU$ is terminating because, as there exists at most one match for $p_1$, it can be applied at most at one match, leading to a unique derivation.
Concerning non-determinism induced by rule selection, we consider a control expression of the form \( S = \langle p_1, \ldots, p_n \rangle \) and additionally assume: \( p_1 \) and \( p_4 \) always have only one possible match and \( (p_2, p_3) \) is parallel independent. Then the transformation unit is usually not confluent, as the counterexample in Figure 5 shows. Here, we assume a graph with vertices that are assigned unique numbers. On the other hand, if we assume a class of graphs that either contains always even or odd numbers, the above example would be confluent because always either \( p_3 \) or \( p_4 \) would be applicable but never both, leaving no space for a choice. This allows us to formulate the following sufficient criteria:

**Criteria 3** Let a transformation unit \( TU = (G, GTS, S) \) be given and let \( S = \langle p_1, \ldots, p_n \rangle \) and assume that \( TU \) is terminating. Then \( TU \) is functional if \( (p_1, p_4) \) are pairwise parallel independent or if all critical pairs \( (p_1, p_4) \) are strongly joinable by a derivation using the rules \( \{p_1, \ldots, p_n\} \) in GTS.

**Proof Sketch 4**

- **Case 1:** \( (p_1, p_4) \) are pairwise independent. Then any order of rule applications will lead to the same result which implies confluence and, by prerequisite, also functionality.
- **Case 2:** this case can be seen as a direct consequence of the critical pair lemma shown in [8].

As an example for case 1, consider \( p_5 \downarrow \) in the statechart translation example. As \( p_5 \) is parallel independent, and assuming termination, the functionality follows.

For the computation of critical pairs involving multi-object rules, we propose to restrict oneself to the basis rule. This basis rule can be obtained by viewing each multi-object as a simple object. This is a consequence of the idea of critical pairs of representing a non-parallel independent transformation step in a minimal context.

Using each individual criteria, the functionality of a transformation unit with control can be established by using the following proposition.

**Proposition 2** Let a terminating transformation unit with control be given. Let \( S = \langle Exp_1, \ldots, Exp_n \rangle \) where \( Exp_i \) consists of plain subexpressions, i.e. \( Exp_i = \langle p_1, \ldots, p_n \rangle \) or \( Exp_i = \langle p_1, \ldots, p_n \rangle \). Then the transformation unit is functional if each subexpression conforms to its criteria, i.e. to criteria 3 or criteria 4 respectively.

**Proof Sketch 5** By induction on \( n \):

- **n = 1** follows by the criteria themselves.
- **n \implies n + 1**: if the transformation unit with a control expression of \( S = \langle Exp_1, \ldots, Exp_n \rangle \) is functional and it is also functional with respect to \( \langle Exp_{n+1} \rangle \), then the composition is also functional.

First note that the proposition is a generalization of our first criteria (Criteria 2). We have restricted ourselves to plain subexpressions, assuming that control expressions such as \( \langle p_1 \rangle, \langle (p_1, p_2) \rangle \) can be flattened to \( \langle p_1, \{p_1, p_2\} \rangle \).

Furthermore, in the proposition, we have restricted the syntax by not making any statement about control expressions of the form \( \langle (E_1), \ldots, (E_n) \rangle \) and \( \langle E \rangle \downarrow \) for implementation reasons. The first one, nondeterministic choice of sequences, would involve backtracking in an implementation. The second one (which specifies loops) has not been considered in the individual criteria and is subject to future considerations.
5. Conclusion

Model transformations of UML models are important within software engineering for a number of applications, amongst them code generation, and consistency checking. In this paper, we have studied the concept of model transformations and first established several important requirements. For meeting these requirements, we have then introduced the concept of rule-based model transformations with control which is based on a set of compound rules and a control expression specifying allowed rule application sequences.

Rule-based model transformations with control have been formalized using established theory of graph transformation, introducing a transformation unit with control expression. This formalization enabled us to introduce a set of sufficient criteria which allow the software engineer to achieve functional behavior. These criteria have been stated formally and explained along a running example of a transformation unit with control.

Future work includes the development of adequate tool support for offering automated support to the software engineer to ensure the validation criteria. Currently, we examine how the Consistency Workbench [5] can be extended for such purposes, by including mechanisms for dynamically detecting the possibility of applying a rule at more than one match.

References