Electronic Communications of the EASST
Volume 30 (2010)

International Colloquium on Graph and Model Transformation On the occasion of the 65th birthday of Hartmut Ehrig (GraMoT 2010)

Stepping from Graph Transformation Units to Model Transformation Units

Hans-Jörg Kreowski, Sabine Kuske, Caroline von Totth

24 pages

Guest Editors: Claudia Ermel, Hartmut Ehrig, Fernando Orejas, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Stepping from Graph Transformation Units to Model Transformation Units

Hans-Jörg Kreowski[1], **Sabine Kuske**[2], **Caroline von Totth**[3]*

[1] kreo@informatik.uni-bremen.de
[2] kuske@informatik.uni-bremen.de
[3] caro@informatik.uni-bremen.de
Department of Computer Science
University of Bremen, Germany

Abstract: Graph transformation units are rule-based entities that allow to transform source graphs into target graphs via sets of graph transformation rules according to a control condition. The graphs and rules are taken from an underlying graph transformation approach. Graph transformation units specify model transformations whenever the transformed graphs represent models. This paper is based on the observation that in general models are not always suitably represented as single graphs, but they may be specified as the composition of a variety of different formal structures such as sets, tuples, graphs, etc., which should be transformed by compositions of different types of rules and operations instead of single graph transformation rules. Consequently, in this paper, graph transformation units are generalized to model transformation units that allow to transform such kind of composed models in a rule-based and controlled way. Moreover, two compositions of model transformation units are presented.

Keywords: graph transformation, model transformation, transformation units, model transformation units

1 Introduction

Computers are devices that can be used to solve all kinds of data-processing problems – at least in principle. The problems to be solved come from economy, production, administration, science, education, entertainment, and many other areas. There is quite a gap between the problems as one has to face them in reality and the solutions one has to provide so that they run on a computer. Therefore, computerization is concerned with bridging this gap by transforming a problem into a solution. Many efforts in computer science contribute to this need for transformation. First of all, compilers are devices that transform programs in a high-level language into programs in a low-level language where the latter are nearer and more adapted to the computer than the former. The possibility and success of compilers have fed the dream of transforming descriptions of data-processing problems automatically or at least systematically into solutions that are given by

* The authors would like to acknowledge that their research is partially supported by the Collaborative Research Centre 637 (Autonomous Cooperating Logistic Processes – A Paradigm Shift and Its Limitations) funded by the German Research Foundation (DFG).

smoothly running programs. In recent years, the term model transformation has become popular for this idea.

In this paper, graph transformation units are generalized to model transformation units as rule-based devices for modeling model transformations in a compositional framework. Our approach has three sources of inspiration:

1. Following the ideas of model-driven architecture (MDA; cf., e.g., [Fra03]), the aim of model transformation is to transform platform-independent models (PIMs), which allow to describe problems adequately, into platform-specific models (PSMs), which run properly and smoothly on a computer. As a typical description of the PIMs one may use UML diagrams, while PSMs are often just programs in some common higher-level language like Java or C++. A significant model transformation language within the framework of MDA is the QVT standard of the OMG [OMG08].

2. One encounters quite an amazing number of model transformations in theoretical computer science – in formal language theory as well as in automata theory in particular. These areas provide a wealth of transformations between various types of grammars and automata like, for example, the transformation of nondeterministic finite automata into deterministic ones, or of pushdown automata into context-free grammars (or the other way round), or of arbitrary Chomsky grammars into the Pentonen normal form (to give a less known example).

3. Graph transformation units (cf., e.g., [KKS97, KK99, KKR08]) are rule-based devices to model binary relations between initial and terminal graphs. If the initial graphs are interpreted as input models and the terminal graphs as output models, then such a unit embodies a model transformation. The transformation of UML sequence diagrams into UML collaboration diagrams in [CHK04] and the transformation of well-structured flow diagrams into *while*-programs in [KHK06] are examples of this kind. This observation supports the idea to use graph transformation units as building blocks for the modeling of model transformations.

While the models in the MDA context are often diagrammatic or textual, the examples of theoretical computer science show that models may also be tuples with components being sets of something. Accordingly, graphs as well as tuples, sequences, and sets of models are introduced as models in Section 3, while Section 2 provides the necessary mathematical preliminaries. The basic steps of model transformation are defined in Section 4 by actions that are applied componentwise to tuples of models and consist of rules in case of graph components and of data type operations in all other cases. Based on models and actions, the notion of a model transformation unit is introduced in Section 5, providing the descriptions of input, working and output models, a set of actions, and a control condition to regulate the use of actions. The semantics of such a unit is a transformation of input models into output models. In Section 6, the sequential and parallel compositions of model transformation units are studied. In this way, complex model transformations can be built up from simple ones in a modular way. While we discuss related work in Section 7, the paper ends with some concluding remarks. As a running example, the transformation of right-linear grammars into finite state automata is developed in several stages.

2 Preliminaries

In this section, we recall the notion of a graph rule base providing a class of graphs, a class of rules and a rule application operator. In the following sections graphs are used as basic visual models and rules are used for their elementary transformations. Besides graphs, we use identifiers, truth values, and non-negative integers as smallest atomic models. Moreover, cartesian products, free monoids, and powersets are recalled because these constructions will be used to build up composite models in the next section.

2.1 Graph Rule Bases

A *graph rule base* $B = (\mathscr{G}, \mathscr{R}, \Longrightarrow)$ consists of a class of graphs $\mathscr{G}$, a class of rules $\mathscr{R}$, and a rule application operator $\Longrightarrow$ with $\underset{r}{\Longrightarrow} \subseteq \mathscr{G} \times \mathscr{G}$ for every $r \in \mathscr{R}$. The rule application operator is used in infix notation, i.e, $(G,H) \in \underset{r}{\Longrightarrow}$ is denoted by $G \underset{r}{\Longrightarrow} H$. Subsections 2.2 through 2.4 present examples for the components of rule bases which are used throughout this paper.

2.2 Graph Classes

There are many different kinds of graph classes, two of which are explored here further: the class of directed edge-labeled graphs and the class of finite state graphs, the latter being a subclass of the former.

Directed edge-labeled graphs. The class of directed, edge-labeled graphs with individual, possibly multiple edges is defined as follows. Let Σ be a set of labels. A *graph* over Σ is a system $G = (V, E, s, t, l)$ where V is a set of *nodes*, E is a set of *edges*, $s,t\colon E \to V$ are mappings assigning a *source* $s(e)$ and a *target* $t(e)$ to every edge in E, and $l\colon E \to \Sigma$ is a mapping assigning a label to every edge in E. An edge e with $s(e) = t(e)$ is also called a *loop*. For a node $v \in V$ the number of edges which have v as source is denoted by $outdegree(v)$ and the number of edges that point to v is the *indegree* of v. An edge e with label x is called an x-pointer if $indegree(s(e)) = 0$ and $outdegree(s(e)) = 1$. The components V, E, s, t, and l of G are also denoted by V_G, E_G, s_G, t_G, and l_G, respectively. The set of all graphs over Σ is denoted by $\mathscr{G}_\Sigma$.

For graphs $G, H \in \mathscr{G}_\Sigma$, a *graph morphism* $g\colon G \to H$ is a pair of mappings $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that are structure-preserving, i.e., $g_V(s_G(e)) = s_H(g_E(e))$, $g_V(t_G(e)) = t_H(g_E(e))$, and $l_H(g_E(e)) = l_G(e)$ for all $e \in E_G$.

If the mappings g_V and g_E are inclusions, then G is called a *subgraph* of H, denoted by $G \subseteq H$. For a graph morphism $g\colon G \to H$, the image of G in H is called a *match* of G in H, i.e., the match of G with respect to the morphism g is the subgraph $g(G) \subseteq H$.

Finite state graphs. Two particular subclasses of $\mathscr{G}_\Sigma$ are the classes of finite state graphs and finite state graphs with word transitions respectively. More concretely, let I be some input alphabet such that $I^* \uplus \{start, final\} \subseteq \Sigma$ [1]. Then the graph in Figure 1 represents a finite state graph with word transitions over $I = \{a, b, c\}$, where the edges labeled with $w \in I^*$ represent transitions,

[1] Given sets X and Y, $X \uplus Y$ denotes the disjoint union of X and Y.

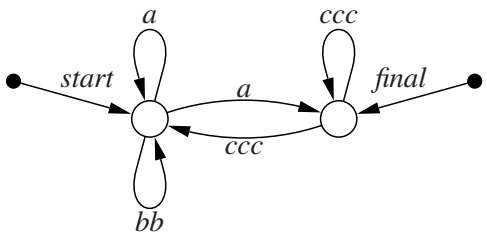


Figure 1: A finite state graph with word transitions

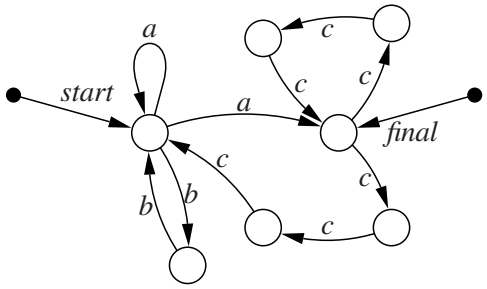


Figure 2: A finite state graph

and the sources and targets of the transitions represent states. The start state is indicated with a *start*-pointer and every final state with a *final*-pointer. States are depicted as unfilled circles whereas all other nodes are shown as small filled circles. Figure 2 shows a finite state graph where each transition is labeled with a symbol from I.

2.3 Rules

To be able to transform graphs, rules are applied to the graphs yielding graphs again. One rule class that can be used to transform graphs in $\mathscr{G}_\Sigma$ is defined as follows. A *rule* $r = (L \supseteq K \subseteq R)$ consists of three graphs $L, K, R \in \mathscr{G}_\Sigma$ such that K is a subgraph of L and R. The components L, K, and R of r are called *left-hand side*, *gluing graph*, and *right-hand side*, respectively. A rule may be depicted as $L \to R$ if K is clear from the context (the numbered nodes form the common gluing graph).

An example of a rule is given in Figure 3. The left-hand side of this rule consists of two nodes

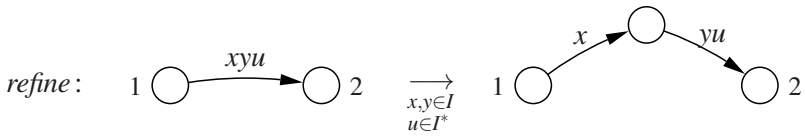


Figure 3: The graph transformation rule *refine*

1 and 2 and an edge from node 1 to node 2 that is labeled with a word xyu from some alphabet I^* where x and y are symbols of I. The gluing graph consists of the two nodes 1 and 2; the right-hand side is obtained from the gluing graph by inserting a new node v and two new edges e_1 and e_2 where e_1 points from node 1 to v and is labeled with x, and e_2 points from v to node 2 and is labeled with yu.

2.4 Rule Application

The application of a graph transformation rule to a graph G consists of replacing a match of the left-hand side in G by the right-hand side in such a way that the match of the gluing graph is kept. Hence, the application of $r = (L \supseteq K \subseteq R)$ to a graph $G = (V, E, s, t, l)$ consists of the following three steps.

1. A match $g(L)$ of L in G is chosen.
2. Now the nodes of $g_V(V_L - V_K)$ are removed, and the edges of $g_E(E_L - E_K)$ as well as the edges incident to removed nodes are removed yielding the *intermediate graph* $Z \subseteq G$.
3. Afterwards the right-hand side R is added to Z by gluing Z with R in $g(K)$ yielding the graph $H = Z \uplus (R - K)$ with $V_H = V_Z \uplus (V_R - V_K)$ and $E_H = E_Z \uplus (E_R - E_K)$. The edges of Z keep their labels, sources, and targets so that $Z \subseteq H$. The edges of R keep their labels; they also keep their sources and targets provided that those belong to $V_R - V_K$. Otherwise, $s_H(e) = g(s_R(e))$ for $e \in E_R - E_K$ with $s_R(e) \in V_K$, and $t_H(e) = g(t_R(e))$ for $e \in E_R - E_K$ with $t_R(e) \in V_K$.

The application of a rule r to a graph G is denoted by $G \overset{r}{\Longrightarrow} H$, where H is the graph resulting from the application of r to G. A rule application is called a *direct derivation*.

If the rule *refine* in Figure 3 is applied to a finite state graph, it splits a word transition labeled with a word w of length at least two into two consecutive transitions, the first of which takes the first symbol of w, while the second one gets labeled with the remainder of w. In particular, if *refine* is applied as long as possible to the finite state graph in Figure 1, one gets the finite state graph in Figure 2.

2.5 Further Basic Types

In addition to graph rule bases, we assume a set of identifiers ID, the set of truth values $\mathrm{BOOL} = \{TRUE, FALSE\}$, and the set of non-negative numbers $\mathbb{N}$. All these sets are equipped with the usual predicates and operations, i.e. the arithmetic operations like $+, -, \cdot, \leq, =$, etc. for $\mathbb{N}$, the Boolean operations like $\wedge, \vee, \neg, \rightarrow$, etc. for BOOL, and the equality predicate $=$ for ID.

All involved sets may be subject to the following three constructions that yield sets again:

1. the cartesian product $X_1 \times \cdots \times X_k$ for sets $X_1, \ldots, X_k, k \in \mathbb{N}$;
2. the free monoid X^* for a set X;
3. the powerset $set(X)$ for a set X that contains all subsets of X.

Furthermore we assume that the usual operations of these data types are available, like the projections in the case of the product, concatenation and other string-processing operations in the case of X^* and the usual operations and predicates on sets like $\cup, \cap, \in, \subseteq$, etc.

3 Models and Model Types

Many models used in computer science are of a graphical, diagrammatic, and visual nature, and they can be represented as graphs in an adequate way in most cases. Moreover, further types of elementary models such as numbers, truth values, or identifiers may be useful in addition to graphs. And models may not occur only as singular items, but also as tuples or as some other collections of models like sequences and sets. To cover this, we define models and their types in a recursive way.

Definition 1 (models and their types) Models together with their types are recursively defined as follows:

1. Let Y be a class of graphs $\mathscr{G}$, ID, BOOL, or $\mathbb{N}$. Then each $y \in Y$ is a *model of type* Y.

2. If m_i is a model of type T_i for $i = 1, \ldots, k$ for some $k \in \mathbb{N}$, then the k-tuple $(m_1, \ldots, m_k)$ is a *model of type* $T_1 \times \cdots \times T_k$.

3. If m_i is a model of type T for $i = 1, \ldots, k$ for some $k \in \mathbb{N}$, then the sequence $m_1 \cdots m_k$ is a *model of type* T^*.

4. If m is a set of models of type T, then m is a *model of type* $set(T)$.

Note that in this way every model gets a type which is a set of models, but can serve as a name on the syntactic level as well. To stress the semantic level we may write $\mathfrak{M}(T)$ for T.

Point 1 makes sure that all graphs and – in this way – all diagrams with graph representations are models. Besides graphs, truth values, numbers and identifiers become available as elementary models. Point 2 allows one to consider a $k-$tuple of models as a model and makes k models simultaneously available in this way. Point 3 and Point 4 also make many models of the same type available at the same time. While Point 3 provides them as a sequence, Point 4 collects them as a set.

The types of models as introduced above may be considered as free because they are based on the free constructions product, free monoid, and power set. But in many cases, it may not be reasonable to transform all models of a free type without any further restriction. For example, a Chomsky grammar $G = (N, T, P, S)$ is not just a quadruple of type $set(ID) \times set(ID) \times set(ID^* \times ID^*) \times ID$, but N, T and P should be finite, N and T should be disjoint, S should be a nonterminal, and a pair $(u, v) \in P$ should consist of two strings of terminals and nonterminals rather than of arbitrary identifiers. To make such restrictions possible, we introduce constrained types.

Definition 2 (constrained model types) Let T be a model type.

1. Then $\mathscr{X}(T)$ is a class of *constraints* if each $x \in \mathscr{X}(T)$ specifies a set of models of type T, i.e. $SEM(x) \subseteq \mathfrak{M}(T)$.

2. For $x \in \mathscr{X}(T)$, $\langle T \text{ with } x \rangle$ is called a *constrained model type*. The models of this type are the models of $SEM(x)$, denoted by $\mathfrak{M}(\langle T \text{ with } x \rangle)$.

The definition is used in a recursive way considering the free model types and the constrained model types both as model types. Consequently, one can build types of the form $\langle\langle T \text{ with } x \rangle \text{ with } y \rangle$ with iterated constraints.

3.1 Examples for Constraints

1. For the model type $\mathscr{G}$, constraints x with $SEM(x) \subseteq \mathscr{G}$ are called graph class expressions in the framework of graph transformation units and are extensively used there to specify initial and terminal graphs. Examples of graph class expressions are the following.

   (a) Single graphs $Z \in \mathscr{G}$ with $SEM(Z) = \{Z\}$ are useful as start graphs of graph grammars.

   (b) For $\mathscr{G} = \mathscr{G}_\Sigma$ with $\Sigma \subseteq ID$, a subset $X \subseteq \Sigma$ describes $SEM(X) = \mathscr{G}_X$ and may serve as terminal labels.

   (c) For $\mathscr{G} = \mathscr{G}_\Sigma$ and $X \subseteq \Sigma$, the expression $pointers(X)$ specifies all graphs in $\mathscr{G}_\Sigma$ in which all edges labeled with some $x \in X$ are pointers (cf. Subsection 2.2).

   (d) For $\mathscr{G} = \mathscr{G}_\Sigma$ and $X \subseteq \Sigma$, the expression $one(X)$ specifies all graphs G in which for each $x \in X$ there occurs exactly one x-labeled edge, i.e., $|\{e \in E_G \mid l_G(e) = x\}| = 1$ for each $x \in X$.

2. Logical formulas are further typical examples for constraints. They may involve model variables and the usual predicates and operations of the basic and free types:

   (a) Boolean operations in case of BOOL like $\neg, \wedge, \vee, \rightarrow$;

   (b) arithmetic operations and predicates on $\mathbb{N}$ like $+$, $\cdot$, mod, $=$, $\leq$;

   (c) string operations and predicates on X^* for some set X, like concatenation, transposition, equality;

   (d) set operations and predicates like $\cup$, $\cap$, $\uplus$, $=$, $\subseteq$, $\in$.

   Consider, for example, a model $(x, y, X, Y, m, n, u, v, G, H)$ of type $ID \times ID \times set(ID) \times set(ID) \times \mathbb{N} \times \mathbb{N} \times ID^* \times ID^* \times \mathscr{G}_\Sigma \times \mathscr{G}_\Sigma$. Then one may add the following constraints: $x = y$, $x \in X$, $y \in Y$, $X \cap Y = \emptyset$, $m \leq n$, $length(u) \geq n$, $uv \neq vu$, $u = vtranspos(v)$, $G \subseteq H$, $G \in \mathscr{G}_X$, where *length* measures the length of a word u and returns an integer, and *transpos* reverses the sequence of symbols in a word. Clearly, all the constraints may be combined by Boolean operations.

3. Another frequently used constraint for graphs and sets is the requirement of finiteness indicated by the constant model class expression *finiteness*. Instead of $\langle \mathscr{G}_\Sigma \textit{ with finiteness} \rangle$ we may write $\textit{fin}(\mathscr{G}_\Sigma)$, and $\textit{finset}(ID)$ instead of $\langle \textit{set}(ID) \textit{ with finiteness} \rangle$.

3.2 Examples for Constrained Model Types

1. Finite state automata with word transitions can be defined as a constrained model type, i.e. a *finite state automaton* $\textit{fsa} = (I,G)$ is a pair of type $\langle \textit{set}(ID) \times \mathscr{G}_\Sigma \textit{ with } (\Delta \subseteq \Sigma) \wedge (G \in (\Delta \cap \textit{pointers}(\{\textit{start},\textit{final}\}) \cap \textit{one}(\{\textit{start}\})))\rangle$ where $\Delta = I^* \uplus \{\textit{start},\textit{final}\}$. The constraint means that every state graph G is labelled over $I^* \uplus \{\textit{start},\textit{final}\}$, *final*- and *start*-edges are pointers, and there is exactly one *start*-pointer. In the following, the constrained model type of finite state automata with word transitions is denoted by FSA^*. The type of finite state automata the transitions of which are labelled only with single symbols from I, can be defined as the finite state automata in FSA^*, but where in the constraint I^* is replaced by I, i.e., $\Delta = I \uplus \{\textit{start},\textit{final}\}$. The type of all finite state automata with single-symbol transitions is denoted by FSA.

2. Chomsky grammars can be introduced in the framework above as models nearly in the same way as they are defined in the literature.

   A *Chomsky grammar* $G = (N,T,P,S)$ is a quadruple of type $\textit{set}(ID) \times \textit{set}(ID) \times \textit{set}(ID^* \times ID^*) \times ID$ *with* finite N, T and P, $N \cap T = \emptyset$, $S \in N$, and $(u,v) \in P$ implies $u,v \in (N \cup T)^*$ and $u \notin T^*$. G is *right-linear* if, in addition, $(u,v) \in P$ implies $u \in N$ and $v \in (T^+N) \cup \{\varepsilon\}$ where ε denotes the empty string.

   More formally, the constraint of an arbitrary Chomsky grammar is $\textit{with } N,T,P \in \textit{finset}(ID) \wedge N \cap T = \emptyset \wedge S \in N \wedge ((u,v) \in P \rightarrow (u,v \in (N \cup T)^* \wedge u \notin T^*))$. And in case of right-linear grammars one must add $((u,v) \in P \rightarrow (u \in N \wedge v \in T^+N \cup \{\varepsilon\}))$. The type of right-linear grammars will be denoted by *RLG*. For explicit use below we mention here also the type $RLG \times \mathscr{G}_\Sigma$ which will be used for transforming right-linear grammars into finite state automata.

4 Actions and Model Transformation Processes

In this section, the dynamic part of model transformations is introduced. The basic notion is that of an action that describes an elementary step of model transformations. Then the iteration of such steps provides more complex transformations.

It is worth noting that in this paper we do not explicitly consider infinite model transformations because the purpose of model transformation units is to convert input models into output models in finitely many steps. Infinite processes are considered in [HKK09].

Each model m can be identified with the 1-tuple (m) so that one may consider tuples of models only without loss of generality. Given such a tuple $(m_1,\ldots,m_k)$, an action is also a k-tuple $a = (a_1,\ldots,a_k)$ of component operations where, for $i = 1,\ldots,k$, a_i specifies how m_i is processed by the action. If m_i is a graph, then a_i is a rule to be applied to m_i. If m_i is an identifier or truth value, then a_i may replace it by another identifier or the negated truth value respectively. If m_i

is a number, string or set, then a_i may operate on it yielding a modified number, string or set respectively. Moreover, we employ the void action $a_i = -$ meaning that m_i remains unchanged. If the component actions are performed, then a new tuple $(m'_1, \ldots, m'_k)$ of models is obtained. This is made precise in the following definition.

Definition 3 (actions) Let $T_1 \times \cdots \times T_k$ be a model type.

1. Then an *action* $a = (a_1, \ldots, a_k)$ is a k-tuple such that one of the following holds for $i = 1, \ldots, k$:
   (a) $a_i = -$,
   (b) $a_i \in \mathscr{R}$ provided that $T_i \subseteq \mathscr{G}$,
   (c) $a_i = rename$ provided that $T_i \subseteq ID$ where $rename$ is some mapping on T_i,
   (d) a_i is a term of operations with a distinguished variable of type $\mathbb{N}$ and which evaluates to $\mathbb{N}$ provided that $T_i = \mathbb{N}$.
   (e) the same as (d) replacing $\mathbb{N}$ by BOOL, T^* and $set(T)$ for some type T,
   (f) recursively, a_i is an action provided that T_i is a product type with more than one component.
2. Let $m = (m_1, \ldots, m_k) \in \mathfrak{M}(T_1 \times \cdots \times T_k)$. Then the action $(a_1, \ldots, a_k)$ may be performed on m yielding $m' = (m'_1, \ldots, m'_k) \in \mathfrak{M}(T_1 \times \cdots \times T_k)$ denoted by $m \underset{a}{\Longrightarrow} m'$ if the following holds for $i = 1, \ldots, k$:
   (a) $m'_i = m_i$ if $a_i = -$;
   (b) $m_i \underset{a_i}{\Longrightarrow} m'_i$ if $a_i \in \mathscr{R}$;
   (c) $m'_i = a_i(m_i)$ if $a_i = rename$ or a_i is a term as described in 1.(d) or (e).
   (d) $m_i \underset{a_i}{\Longrightarrow} m'_i$ if a_i is an action.
3. Let A be a set of actions. Then a *model transformation process* is a sequence of performed actions $m = m_0 \underset{a_1}{\Longrightarrow} m_1 \underset{a_2}{\Longrightarrow} \cdots \underset{a_n}{\Longrightarrow} m_n = m'$ with the *action sequence* $a_1 \cdots a_n \in A^*$. Such a process may be denoted by $m \overset{n}{\underset{A}{\Longrightarrow}} m'$ or $m \overset{*}{\underset{A}{\Longrightarrow}} m'$ if the omitted details do not matter. The set of model transformation processes over A is denoted by $MTP(A)$.

4.1 Examples for Actions

Let (N, T, P, S, G) be an arbitrary model of type $RLG \times \mathscr{G}_\Sigma$ as defined in point 2 of Subsection 3.2.

1. An action that removes a nonterminal symbol X from the first component of the right-linear grammar (N, T, P, S) and then inserts a state labeled with X in the graph component can be defined as $(remove(X), -, -, -, node(X))$, where $remove(X)$ removes X from N (if $X \in N$) and $node(X)$ is the graph transformation rule depicted in Figure 4. If N does not contain X the action cannot be executed.

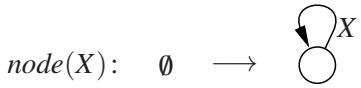


Figure 4: Graph transformation rule $node(X)$

Figure 5: Graph transformation rule $edge(X,u,Y)$

2. An action that removes a rule with a non-empty right-hand side from the right-linear grammar while inserting a corresponding transition in the graph that contains a state for every nonterminal of the rule can be defined as $(-,-,remove((X,uY)),-,edge(X,u,Y))$; the graph transformation rule $edge(X,u,Y)$ is given in Figure 5.

Model transformation processes are nondeterministic in three respects. First, the rule applications in graph model components are nondeterministic as some rules may be applicable at several matches. Second, although the operations of the basic types are functional, the evaluations of the action terms of these types may not lead to unique values as the terms can contain free variables with a variety of instantiations. Third, there may be a choice of many actions that can process a current model, and the only regulating requirement for actions is that of sequential composition, which is that one action is executed after the other. Sometimes such nondeterminism is desired, convenient, or unavoidable. But in other cases one would like to avoid nondeterminism, or cut it down at least. This can be achieved by choosing rules and actions in such a way that only one or a few of them can be applied and performed. But the rules and actions may become quite complicated. Another possibility is extra regulation which can be provided by control conditions.

Definition 4 (control conditions) Let A be a set of actions. Then $\mathscr{C}$ is a class of *control conditions* if $SEM(c) \subseteq MTP(A)$ for every $c \in \mathscr{C}$.

4.2 Examples for Control Conditions

In the area of graph transformation, control conditions are frequently expressions over rules. Many of these kinds of control conditions can be generalized by replacing rules with actions.

1. A typical kind of control conditions are regular expressions over A. Each regular expression r specifies a regular language $L(r)$. A model transformation process $m \overset{*}{\underset{A}{\Longrightarrow}} m'$ belongs to $SEM(r)$ if and only if its action sequence belongs to $L(r)$. In the following, the operators concatenation, union, and Kleene star on languages will be denoted on the level of regular expressions as a semicolon, a vertical bar and a star, respectively.

2. Another kind of control condition is a priority given by a partial reflexive and transitive relation $\leq$ on A where $a \geq a'$, but $a' \not\geq a$ means that a has higher priority than a'. A

model transformation process belongs to $SEM(\leq)$ if and only if each performed action $m_{i-1} \underset{a_i}{\Longrightarrow} m_i$ has highest priority meaning that there is no $m_{i-1} \underset{a}{\Longrightarrow} \overline{m}$ with $a \geq a_i$ but $a_i \not\geq a$.

3. For any action a, the control condition $a!$ requires to apply a as long as possible. Hence, $m \overset{*}{\underset{A}{\Longrightarrow}} m'$ is in $SEM(a!)$ if the application sequence is in $\{a\}^*$ and there is no m'' such that $m' \underset{a}{\Longrightarrow} m''$. (Due to the fact that model transformation processes are finite, this means that $SEM(a!) = \emptyset$ if a can be applied infinitely often to any model m.) This condition can be combined with regular expressions in a straightforward way. For example, the expression $a_1!;a_2!$ requires to apply first a_1 as long as possible and then a_2 as long as possible.

5 Model Transformation Units

The previous sections provide all the ingredients that are needed to introduce model transformation units as devices to specify model transformations. Such a unit consists of the type of models to be transformed, of the actions to be performed, and of the control condition that regulates the transformation process. Moreover, the types of input and output models are specified, including their relation to the type of working models. The reasons to separate input, output and working models is that input and output may have different types and that it may be convenient to use further component models for intermediate processing.

In other words, an input model m of type $\langle I_1 \times \cdots \times I_k \textit{ with } x \rangle$ (i.e. a model of type $I_1 \times \cdots \times I_k$ that satisfies the constraint x) is first of all extended to a working model $\overline{m}$ of type $T_1 \times \cdots \times T_l$ by taking the components of m as components of $\overline{m}$ according to a mapping *initial*. This mapping yields for each component of m the positions in $\overline{m}$ (i.e. the numbers out of $1,\ldots,l$) where the component should be used. Clearly, each position in $\overline{m}$ may be associated in this way with with at most one component of the input type. The components of the working model not covered by *initial* are initialized by the initial models of the respective component types. Initial models are chosen in some appropriate way, like 0 for $T_j = \mathbb{N}$, etc. Then $\overline{m}$ is transformed into $\overline{m}'$ by performing the given actions such that the control condition is satisfied. Afterwards an output model m' of type $\langle O_1 \times \cdots \times O_n \textit{ with } y \rangle$ is constructed according to a mapping *terminal*. This mapping selects for every position in m' (i.e. for every number out of $1,\ldots,n$) a component of $\overline{m}'$. Moreover, it must be assured that the obtained model m' satisfies the constraint y.

Definition 5 (model transformation unit)

1. A *model transformation unit* is a system $mtu = (ITD, OTD, WT, A, C)$ where

   - WT is a product type $T_1 \times \cdots \times T_l$ called *working type*,
   - ITD is the *input type declaration* which consists of the constrained product type $\langle I_1 \times \cdots \times I_k \textit{ with } x \rangle$ and a mapping $initial\colon [k] \to set[l]$ such that $initial(i) \cap initial(j) = \emptyset$ for $i \neq j$ and $I_i = T_j$ for $i = 1,\ldots,k$ and $j \in initial(i)$,
   - OTD is the *output type declaration* which consists of a constrained product type $\langle O_1 \times \cdots \times O_n \textit{ with } y \rangle$ and an injective mapping $terminal\colon [n] \to [l]$ with $O_i = T_{terminal(i)}$ for $i = 1,\ldots,n$,

– A is the *set of actions* with respect to the working type and
– C is the *control condition*.

The type $I = \langle I_1 \times \cdots \times I_k \; with \; x \rangle$ is called *input type* of *mtu* and the mapping *initial initialization*. The type $O = \langle O_1 \times \cdots \times O_n \; with \; y \rangle$ is called *output type* of *mtu* and the mapping *terminal terminalization*.

2. The model transformation modeled by the model transformation unit *mtu* is a mapping $SEM(mtu) \colon \mathfrak{M}(\langle I_1 \times \cdots \times I_k \; with \; x \rangle) \to set(\mathfrak{M}(\langle O_1 \times \cdots \times O_n \; with \; y \rangle))$ which is defined by $m' = (m'_1, \ldots, m'_n) \in SEM(mtu)(m_1, \ldots, m_k)$ for every $m = (m_1, \ldots, m_k) \in \mathfrak{M}(\langle I_1 \times \cdots \times I_k \; with \; x \rangle)$ if and only if the following holds:

   There are working models $\overline{m} = (\overline{m}_1, \ldots, \overline{m}_l)$, $\overline{m}' = (\overline{m}'_1, \ldots, \overline{m}'_l) \in \mathfrak{M}(T_1 \times \cdots \times T_l)$ such that

   (a) $\overline{m}_j = \begin{cases} m_i & \text{for} \quad i = 1, \ldots, k \text{ and } j \in initial(i) \\ init(T_j) & \text{for} \quad j \notin initial([k]) = \bigcup\limits_{i \in [k]} initial(i) \end{cases}$,

   (b) $\overline{m} \overset{*}{\underset{A}{\Longrightarrow}} \overline{m}' \in SEM(C)$,

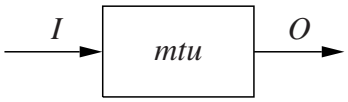
   (c) $m'_i = \overline{m}'_j$ for $i = 1, \ldots, n$ and $terminal(i) = j$,

   (d) $m' \in SEM(y)$.

   The initial model $init(T_j)$ in (a) may be chosen in some appropriate way, like 0 for $T_j = \mathbb{N}$, the empty string ε for $T_j = T^*$, the empty set $\emptyset$ for $T_j = set(T)$ or *FALSE* for $T_j = \text{BOOL}$.

In examples, *initial* will be represented in the form $i \mapsto j_1, \ldots, j_p$ if $initial(i) = \{j_1, \ldots, j_p\}$ and *terminal* in the form $i \mapsto j$ for $terminal(i) = j$.

Remark Given a model transformation unit *mtu* with input type $I = \langle I_1 \times \cdots \times I_k \; with \; x \rangle$ and output type $O = \langle O_1 \times \cdots \times O_n \; with \; y \rangle$, *mtu* can be graphically represented by



emphasizing that *mtu* specifies a transformation of input models into output models.

5.1 Examples for Model Transformation Units

A model transformation unit that transforms right-linear Chomsky grammars into finite state automata is given in Figure 6. The components of this model transformation unit *RLG2FSA** are the following:

$RLG2FSA^*$	
input:	RLG & $1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4$
add:	$5: \mathscr{G}_\Sigma$ & $init(5) = \emptyset$ for $\Sigma = (N \cup T)^* \uplus \{start, final\}$
actions:	$a_1 = (remove(X), -, -, -, node(X))$ for $X \in N$ $a_2 = (-, -, remove((X, \varepsilon)), -, final(X))$ for $X \in N$ $a_3 = (-, -, remove((X, uY)), -, edge(X, uY))$ for $X, Y \in N, u \in T^+$ $a_4 = (-, -, -, -, start(S))$ $a_5 = (-, -, -, -, remove\_loop(X))$ for $X \in N$
cond:	$a_1!; a_2!; a_3!; a_4; a_5!$
output:	FSA^* & $1 \mapsto 2, 2 \mapsto 5$

Figure 6: The model transformation unit $RLG2FSA^*$ transforms right-linear Chomsky grammars (RLG) into finite state automata with word transitions (FSA^*)

– A model of the working type is a quintuple where the first four components of the working type correspond to the four types of a right-linear grammar; the last component is equal to $\mathscr{G}_\Sigma$ and serves to build up the finite state graph. It is initialized with the empty graph $\emptyset$. The alphabet Σ must equal $(N \cup T)^* \uplus \{start, final\}$ where N are the nonterminal symbols and T the terminal symbols of the input grammar, and *start* and *final* will serve to label the start and final states of the finite state graph respectively.

– The input type declaration is composed of the constrained model type for right-linear grammars and the initialization $initial : [4] \to set([5])$ with $initial(i) = \{i\}$ for $i = 1, \ldots, 4$. This means that the four components of the input type are the first four components of the working type. Hence, the four components of every input model are used as the first four components in the model the model transformation unit starts working with.

– The output type declaration consists of the constrained model type FSA^* and the terminalization *terminal* with $terminal(1) = 2$ and $terminal(2) = 5$. Hence, every output model of the unit is the pair consisting of the second and the last component of the model the unit ends working with, provided that the type of this pair equals FSA^*.

– The set of actions of $RLG2FSA^*$ consists of five kinds of actions, each of which contains among other operations one of the graph transformation rules depicted in Figures 4, 5 and 7.

  1. The first action $a_1 = (remove(X), -, -, -, node(X))$ serves to generate a state in the graph for each nonterminal of the input grammar. More concretely, every application of this action generates a state with name X while removing the nonterminal X from the set of nonterminals.
  2. The second action $a_2 = (-, -, remove((X, \varepsilon)), -, final(X))$ inserts final pointers at all final states of the graph, while removing the corresponding rules from the grammar.

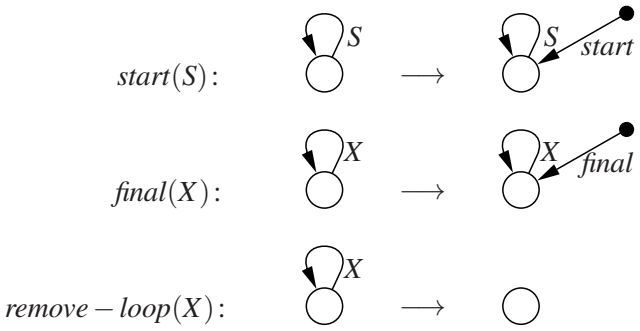


Figure 7: Graph transformation rules for the actions of model transformation unit *RLG2FSA**

3. The third action $a_3 = (-,-,remove((X,uY)),-,edge(X,u,Y))$ serves to generate transitions from those rules of the grammar that have a nonterminal in their right-hand side. Every application of a_3 removes such a rule from the rule set in the third component at the same time that a corresponding transition in the graph is generated.
4. Action $a_4 = (-,-,-,,start(S))$ inserts the start pointer at the state S if S is the start symbol of the grammar.
5. Finally, the last action $a_5 = (-,-,-,-,remove\_loop(X))$ for $X \in N$ serves to remove all state names in order to obtain a finite state graph.

– The control condition $a_1!;a_2!;a_3!;a_4;a_5!$ requires that at first all states be generated. This is achieved by applying a_1 as long as possible. The application of a_2 as long as possible inserts for every rule with the empty word as right-hand side a *final*-pointer while removing this rule. Then a_3 requires to insert a transition for every remaining rule. Then the start state is inserted by a_4 and afterwards all state names are removed by applying a_5 as long as possible.

*FSA*2FSA*	
input:	$FSA^* \,\&\, 1 \mapsto 1, 2 \mapsto 2$
actions:	$a = (-, refine)$
cond:	$a!$
output:	$FSA \,\&\, 1 \mapsto 2, 2 \mapsto 2$

Figure 8: The model transformation unit *FSA*2FSA* transforms finite state automata with word transitions (*FSA**) into finite state automata (*FSA*)

If the input model of *RLG2FSA** is the right-linear grammar $(\{S,A\},\{a,b,c\},P,S)$ with $P = \{(S,aSa),(S,aA),(S,bbS),(A,cccA),(A,\varepsilon)\}$, the output model is $(\{a,b,c\},G)$ where G is the finite state graph with word transitions in Figure 1.

Finite state graphs with word transitions can be transformed into finite state graphs with symbol transitions by the model transformation unit *FSA*2FSA* given in Figure 8.

The input type declaration consists of the constrained model type FSA^* of finite state automata with word transitions and the initialization *initial* that maps the two components of every input model to the first two components of the working type. The working type of the unit is equal to $set(ID) \times \mathscr{G}_\Sigma$; the output type declaration consists of the model type *FSA* for finite state automata and the terminalization *terminal*, which is the identity in this case. The only action a applies the rule *refine* of Figure 3 to the graph component of the current model, while the control condition requires to apply the action a as long as possible. If the input model of *FSA*2FSA* is equal to the state automaton $(\{a,b,c\},G)$ where G is the finite state graph of Figure 1, the output is equal to $(\{a,b,c\},G')$ where G' is the finite state graph in Figure 2.

6 Sequential and Parallel Composition

Model transformation units can be used as building blocks for more complex model transformation constructions obtained by sequential and parallel composition. This leads to the notion of model transformation expressions on the syntactic level. Semantically, the sequential composition of model transformations is just the usual one of relations. And the parallel composition uses the fact that all models are considered as tuples of some product types so that the product of such types yields again models of some product type.

Definition 6 (compositional expressions)

1. The set $\mathscr{CX}$ of compositional expressions is defined recursively:
   (a) model transformation units are in $\mathscr{CX}$,
   (b) $cx_1,\ldots,cx_k \in \mathscr{CX}$ implies $cx_1;\ldots;cx_k \in \mathscr{CX}$ (sequential composition),
   (c) $cx_1,\ldots,cx_k \in \mathscr{CX}$ implies $cx_1 \parallel \ldots \parallel cx_k \in \mathscr{CX}$ (parallel composition).
2. The semantic relation of a compositional expression $cx \in \mathscr{CX}$ is defined according to its syntactic structure:
   (a) If $cx = mtu$ for some model transformation unit, then $SEM(cx) = SEM(mtu)$.
   (b) If $cx_1;\ldots;cx_k$ for some model transformation units cx_i with $i = 1,\ldots,k$, then $SEM(cx_1;\ \ldots\ ;cx_k) = SEM(cx_1) \circ \ldots \circ SEM(cx_k)$where
   $$SEM(cx_i) \circ SEM(cx_{i+1})(m) = \bigcup_{m' \in SEM(cx_i)} SEM(cx_{i+1})(m')$$
   for each $i \in \{1,\ldots,k-1\}$ and each m in the domain of $SEM(cx_i)$.
   (c) $(m'_1,\ldots,m'_k) \in SEM(cx_1 \parallel \ldots \parallel cx_k)(m_1,\ldots,m_k)$ if and only if $m'_i \in SEM(cx_i)(m_i)$ for $i = 1,\ldots,k$.

6.1 Examples

The sequential composition $RLG2FSA^*;FSA^*2FSA$ of the model transformation units in Section 5 transforms right-linear grammars into finite state automata so that the language generated by the input grammar is recognized by the automaton.

The formal language theory offers many examples of sequential compositions of model transformations like the transformation of right-linear grammars into finite state automata followed by their transformation into deterministic automata followed by the transformation of the latter into minimal automata.

A typical example of a parallel composition is given by the acception processes of two finite state automata that run simultaneously. If they try to accept the same input strings, this parallel composition simulates the product automaton that accepts the intersection of the two accepted regular languages.

To make the definition of compositional expressions more transparent, one may assign an input type and an output type to each compositional expression. Then the relational semantics of an expression turns out to be a relation between input and output types.

Definition 7 (input and output types) The input type in and the output type out of a compositional expression $cx \in \mathscr{CX}$ is recursively defined.

1. If $cx = mtu$ for some model transformation unit with input type I and output type O, then $\mathsf{in}(mtu) = I$, $\mathsf{out}(mtu) = O$,

2. If $cx = cx_1;\ldots;cx_k$ for some model transformation units cx_i with $i = 1,\ldots,k$, then $\mathsf{in}(cx_1;\ldots;cx_k) = \mathsf{in}(cx_1)$ and $\mathsf{out}(cx_1;\ldots;cx_k) = \mathsf{out}(cx_k)$,

3. If $cx = cx_1 \parallel \ldots \parallel cx_k$ for some model transformation units cx_i with $i = 1,\ldots,k$, then $\mathsf{in}(cx_1 \parallel \ldots \parallel cx_k) = \mathsf{in}(cx_1) \parallel \ldots \parallel \mathsf{in}(cx_k)$ and $\mathsf{out}(cx_1 \parallel \ldots \parallel cx_k) = \mathsf{out}(cx_1) \parallel \ldots \parallel \mathsf{out}(cx_k)$, where the parallel composition of model types is defined as follows

   (a) $(T \parallel T') = (T \times T')$ provided that T and T' are free,

   (b) $T \parallel (\langle T' \mathit{\ with\ } x' \rangle) = \langle (T \parallel T') \mathit{\ with\ } x' \rangle$ provided that T is free,

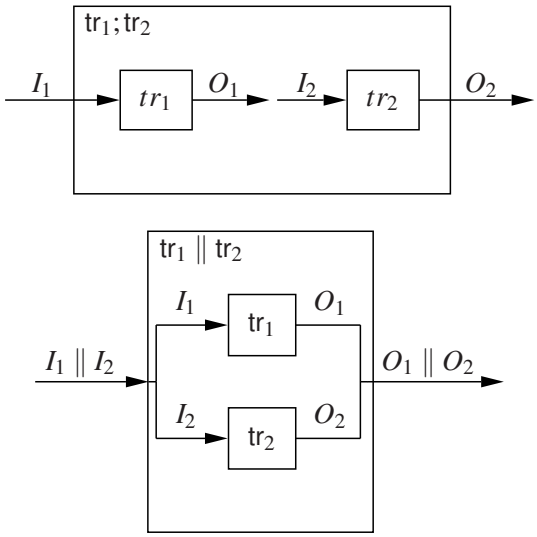
   (c) $(\langle T \mathit{\ with\ } x \rangle) \parallel T' = \langle (T \parallel T') \mathit{\ with\ } x \rangle$ provided that T' is free, and

   (d) $(\langle T \mathit{\ with\ } x \rangle) \parallel (\langle T' \mathit{\ with\ } x' \rangle) = \langle T \parallel T' \mathit{\ with\ } x \wedge x' \rangle$,

Due to these definitions, it is easy to see that compositional expressions describe transformations from input models to output models.

Observation $SEM(cx)(m) \in set(\mathfrak{M}(\mathsf{out}(cx)))$ for all $m \in \mathfrak{M}(\mathsf{in}(cx))$.

The compositions can be quite intuitively depicted:



The sequential and parallel compositions on the level of model transformation expressions have the disadvantage that their results cannot be subject to further constraints. This is particularly problematic with respect to the parallel composition because the composed units run in parallel, but without any interaction. This is quite all right provided that the components are meant to run independently of each other. But in many cases of parallel composition one intends that the components exchange information or process some data simultaneously. Such interrelations and interactions could be achieved by adding further constraints and control conditions. This requires either to extend the notion of constraints and control conditions to the level of model transformation expressions or to flatten such expressions into model transformation units. The latter is done in the following.

6.2 Sequential Composition

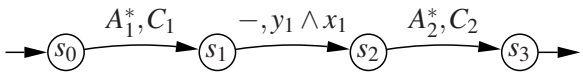
Let $mtu_i = (ITD_i, OTD_i, WT_i, A_i, C_i)$ for $i = 1,2$ be two model transformation units with input types $I_i = \langle I_{i,1} \times \cdots \times I_{i,k_i} \text{ with } x_i \rangle$ and output types $O_i = \langle O_{i,1} \times \cdots \times O_{i,n_i} \text{ with } y_i \rangle$. By definition of the semantics of the sequential composition $mtu_1;mtu_2$, the following holds: $m'' = (m''_1, \ldots, m''_{n_2}) \in SEM(mtu_1;mtu_2)(m)$ for $m = (m_1, \ldots, m_{k_1}) \in \mathfrak{M}(I_1)$ if and only if there is an m' with $m' \in SEM(mtu_1)(m)$ and $m'' \in SEM(mtu_2)(m')$. This means in particular that $m' \in \mathfrak{M}(O_1) \cap \mathfrak{M}(I_2)$ and therefore $n_1 = k_2$. To avoid too much technical trouble, we assume in addition that $WT = WT_1 = O_1 \times \cdots \times O_{n_1} = I_1 \times \cdots \times I_{k_2} = WT_2$. Then the sequential composition of mtu_1 and mtu_2 can be simulated by the model transformation unit

$$mtu(mtu_1;mtu_2) = (ITD_1, OTD_2, WT, A_1 \cup A_2, C(C_1, C_2, y_1, x_2, A_1, A_2))$$

where the control condition is chosen in such a way that a model transformation process $\overline{m} \overset{*}{\underset{A_1 \cup A_2}{\Longrightarrow}} \overline{m}''$ is accepted if and only if it decomposes into $\overline{m} \overset{*}{\underset{A_1}{\Longrightarrow}} \overline{m}' \overset{*}{\underset{A_2}{\Longrightarrow}} \overline{m}''$ with the following properties:

1. $\overline{m} \overset{*}{\underset{A_1}{\Longrightarrow}} \overline{m}'$ is accepted by C_1,
2. $\overline{m}' \in SEM(y_1) \cap SEM(x_2)$,
3. $\overline{m}' \overset{*}{\underset{A_2}{\Longrightarrow}} \overline{m}''$ is accepted by C_2.

Such a control condition may have the form of a transition system:



requiring that at the beginning the actions of A_1 are iterated regarding C_1, that the result must obey y_1 and x_2 and that finally actions of A_2 are iterated regarding C_2.

It is not difficult to show that the following correctness result holds.

Observation $SEM(mtu_1;mtu_2) = SEM(mtu(mtu_1;mtu_2))$.

6.3 Parallel Composition

Let $mtu_i = (ITD_i, OTD_i, WT_i, A_i, C_i)$ for $i = 1,2$ be two model transformation units each with input type $I_i = \langle I_{i,1} \times \cdots \times I_{i,k_i}$ *with* $x_i \rangle$ and initialization $initial_i : [k_i] \longrightarrow set[l_i]$ as well as output type $O_i = \langle O_{i,1} \times \cdots \times O_{i,n_i}$ *with* $y_i \rangle$ and terminalization $terminal : [n_i] \longrightarrow [l_i]$. Then the parallel composition of mtu_1 and mtu_2 can be simulated by the model transformation unit

$$mtu(mtu_1 \parallel mtu_2) = (ITD, OTD, WT_1 \times WT_2, A, C)$$

where

- ITD consists of the input type $I_1 \parallel I_2$ and the initialization $initial_i : [k_1 + k_2] \longrightarrow set[l_1 + l_2]$ with $initial(i) = initial_1(i)$ for $i \in [k_1]$ and $initial(i) = l_1 + initial_2(i - k_1)$ for $i = k_1 + 1, \ldots, k_1 + k_2$,
- OTD consists of the output type $O_1 \parallel O_2$ and the terminalization $terminal$: $[n_1 + n_2] \longrightarrow [l_1 + l_2]$ with $terminal(i) = terminal_1(i)$ for $i \in [n_1]$ and $terminal(i) = l_1 + terminal_2(i - n_1)$ for $i = n_1 + 1, \ldots, n_1 + n_2$,
- $A = A_1{}' \times A_2{}'$ with $A_1{}' = A_1 \cup \{-\}^{l_1}$ and $A_2{}' = A_2 \cup \{-\}^{l_2}$, and
- the control condition C is chosen in such a way that a model transformation process $(\overline{m}_1, \overline{m}_2) \overset{*}{\underset{A}{\Longrightarrow}} (\overline{m}_1{}', \overline{m}_2{}')$ is accepted if and only if it decomposes into $\overline{m}_1 \overset{*}{\underset{A_1{}'}{\Longrightarrow}} \overline{m}_1{}'$ and $\overline{m}_2 \overset{*}{\underset{A_2{}'}{\Longrightarrow}} \overline{m}_2{}'$ so that the former is accepted by C_1 and the latter by C_2 after removal of the void steps given by the performance of the void actions $(-, \ldots, -)$.

The construction relies on the cartesian product of types and actions. Because the working type components 1 to l_2 become the components l_1+1 to l_1+l_2, the initialization and terminalization must be adapted accordingly. The actions of mtu_1 and mtu_2 are extended by the void action $(-,\ldots,-)$ with l_1 and l_2 components respectively. This is necessary because the actions of mtu_1 and mtu_2 may run in parallel, but the model transformation processes are of different lengths in general so that they cannot run fully simultaneously.

It is again not difficult to show the following correctness result.

Observation $SEM(mtu_1 \parallel mtu_2) = SEM(mtu(mtu_1 \parallel mtu_2))$.

7 Related Work

In this section we briefly describe a selection of related work concerning model transformation. Since there exists quite an amount of publications we restrict ourselves to papers that are concerned with model transformations in the context of graph transformation. Moreover, we also mention some work that is concerned with the composition of model transformation definitions.

Model transformations based on graph transformation. One approach to define model transformations is by triple grammars [Sch94, KS06, SK08]. Each rule of a triple grammar can be easily transformed into a forward rule, a source rule, and a backward rule. The source rules are used to generate source models that – represented as graph triples – have the form $(S,\emptyset,\emptyset)$ where S represents the source model. The forward rules are used to produce target models from source models. These target models – represented as graph triples – have the form (S,C,T) where T is the target model. The backward rules are used to transform a target model $(\emptyset,\emptyset,T)$ to a source model (S,C,T). In [EEE+07], it is shown that any source consistent model transformation based on triple grammars is backward information preserving. This means that the target model (generated by the forward rules of the grammar) can be transformed into the source model via the backward rules of the grammar. Roughly spoken, a model transformation MT is source consistent if there is a transformation that generates the source model from $(\emptyset,\emptyset,\emptyset)$ and that completely determines the matches in the source model of the forward rules applied in MT.

In [EE08], models are graphs equipped with a semantics given as a set of simulation rules, and a model transformation is composed of generating first an integrated model by graph transformation rules and restricting it then to the target model. It is shown under which conditions semantical correctness and completeness of model transformations are achieved. In [Küs06], an approach to model transformation is presented that uses transformation units based on typed attributed graph transformation. It provides criteria for syntactic correctness as well as for termination and confluence.

Examples of model transformation tools based on graph transformation are VIATRA2 [VB07], GReAT [BNvBK06] and ATOM^3 [dLVA04]. VIATRA2 integrates graph transformation and abstract state machines. Basically, model transformation steps are captured by graph transformation rules whereas abstract state machines control the order of rule application. GReAT mainly consists of a pattern specification language, a transformation rule language and a control flow language. The graph transformation rules of GReAT include for example input and output in-

terfaces where the former can receive graph objects from previous rules and the latter can send graph objects to another rule. ATOM^3 focuses on modeling complex systems composed of various formalisms and allows to transform them into a single common formalism based on graph transformation. In [dLT04], ATOM^3 is combined with AGG for validation purposes.

In general, the mentioned publications on model transformation with graph transformation are very close to our approach – they are however restricted to transform mainly graphs, not tuples of graphs, sets or sequences as proposed in this paper.

Composition of model transformations. In the literature one can find two main types of composition techniques for model transformation definitions: external and internal composition. The first one chains model transformations sequentially whereas the second composes the rules of a set of model transformation definitions into one transformation definition. In this sense the compositions presented in Subsections 6.2 and 6.3 can be considered as internal compositions.

In [Wag08], the composition of model transformation definitions via superimposition is described, which is a feature of the ATLAS Transformation Language [JK05]. Superimposition of modules is an internal composition technique where models can be superimposed on top of each other yielding a module that contains the union of all transformation rules. In [YCWD09], the authors consider composition of model transformation definitions that transform high-level models into low-level models by defining a correspondence model that specifies the relations between the high-level meta models. The low-level correspondence model is automatically generated so that the low-level models can be composed homogeneously. In this way, new concerns can be added to existing model transformation definitions. In [CM08], two approaches for reusing model transformation definitions are proposed. The first one is called factorization and it allows to extract common parts of model transformation definitions obtaining in this way a base transformation definition which can be reused. The second concerns composition of transformation definitions which have compatible source metamodels but different target metamodels. Metamodels are related via small new metamodels and the transformations are integrated via an integration transformation definition that locates and connects the join points (without knowing the rules but some kind of trace information) by using so-called refinement rules. One approach towards composition of model transformations based on graph transformation is studied in [BHE09] where models are typed graphs that are mapped to semantic domains. The authors define spatial compositionality of semantic mappings which roughly spoken means that the semantics of a model is equal to the semantics that is obtained by embedding the semantics of a piece of the model into some context. It is assumed that the semantic mappings are graph transformation systems with a functional behavior and it is shown under which conditions they behave compositionally. In [KKS07], a first approach towards structured model transformation is proposed that allows package import, package merge and generalization according to a standardized packaging concept of the UML. In particular, the authors extend triple graph grammars by the mentioned concepts.

8 Conclusion

In this paper, we have introduced the notion of model transformation units as a generalization of graph transformation units. Models are tuples of graphs and other data structures like strings,

sets, numbers, etc. Models of this kind cover graphical models like UML diagrams as well as set-theoretic models like grammars and automata. They are transformed componentwise by rule applications in the cases of graphs and by applications of data type operations in the other cases. Besides a set of such actions, a model transformation unit provides descriptions of input, output, and working models as well as a control condition to regulate the use of actions. Semantically, a transformation of input models into output models is specified. Moreover, we have studied sequential and parallel compositions of model transformation units as means to build up complex transformations from simple ones.

Although the considerations in this paper seem to be promising, more work is needed to underpin the significance of this novel approach, including the following points.

1. As pointed out in Section 4, the introduced kind of model transformation is nondeterministic. Therefore, sufficient conditions are often of interest that guarantee termination, completeness and functionality where the first property means that there is no infinite run, the second one requires at least one output for each input, and the latter one requires at most one output for each input.

2. Concerning our running example, it is known from the literature that a right-linear grammar generates the same language as is recognized by the finite state automaton resulting from the transformation. One intention of our approach is to support such correctness proofs. Therefore, notions of correctness and an appropriate proof theory must be studied in the future.

3. An interesting question in this respect is whether and how these correctness notions are compatible with the sequential and parallel compositions so that the correctness of the components yields the correctness of the composed model transformation.

4. Further explicit and detailed examples are needed to illustrate all introduced concepts more convincingly, in particular examples for parallel and sequential composition with interaction between components.

Acknowledgments *We are grateful to the unknown referees for various helpful comments. The first author wants to thank Hartmut Ehrig for the long lasting cooperation and friendship. Hans-Jörg (being a Math student at the time) met Hartmut in 1970 when a very close relationship started. Hartmut supervised Hans-Jörg's diploma thesis in 1974 and his PhD thesis in 1978. Moreover, he guided Hans-Jörg to habilitation in 1981. Hartmut introduced Hans-Jörg to category theory. They both learned automata theory together. Hartmut convinced Hans-Jörg of the significance of graph transformation. They both together got involved in algebraic specification. Although this happened in the 1970s, it sticks: Elements of all four areas can be found in the present paper. Hans-Jörg happily acknowledges that he is one of Hartmut's grateful students.*

Bibliography

[BHE09] Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Compositionality of model transformations. *Electr. Notes Theor. Comput. Sci.*, 236:5–19, 2009.

[BNvBK06] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, and Gabor Karsai. The graph rewriting and transformation language: GReAT. *ECEASST*, 1, 2006.

[CHK04] Björn Cordes, Karsten Hölscher, and Hans-Jörg Kreowski. UML interaction diagrams: Correct translation of sequence diagrams into collaboration diagrams. In John l. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, volume 3062 of *Lecture Notes in Computer Science*, pages 275–291, 2004.

[CM08] Jesús Sánchez Cuadrado and Jesús García Molina. Approaches for model transformation reuse: Factorization and composition. In Vallecillo et al. [VGP08], pages 168–182.

[dLT04] Juan de Lara and Gabriele Taentzer. Automated model transformation and its validation using AToM^3 and AGG. In Alan F. Blackwell, Kim Marriott, and Atsushi Shimojima, editors, *Diagrammatic Representation and Inference*, volume 2980 of *Lecture Notes in Computer Science*, pages 182–198. Springer, 2004.

[dLVA04] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and System Modeling*, 3(3):194–209, 2004.

[EE08] Hartmut Ehrig and Claudia Ermel. Semantical correctness and completeness of model transformations using graph and rule transformation. In Ehrig et al. [EHRT08], pages 194–210.

[EEE+07] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. Information preserving bidirectional model transformations. In Matthew B. Dwyer and Antónia Lopes, editors, *FASE*, volume 4422 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2007.

[EHRT08] Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors. *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, volume 5214 of *Lecture Notes in Computer Science*. Springer, 2008.

[Fra03] David S. Frankel. *Model Driven Architecture. Applying MDA to Enterprise Computing*. Wiley, Indianapolis, Indiana, 2003.

[HKK09] Karsten Hölscher, Hans-Jörg Kreowski, and Sabine Kuske. Autonomous units to model interacting sequential and parallel processes. *Fundamenta Informaticae*, 92(3):233–257, 2009.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[KHK06] Hans-Jörg Kreowski, Karsten Hölscher, and Peter Knirsch. Semantics of visual models in a rule-based setting. In R. Heckel, editor, *Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004)*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 75–88. Elsevier Science, 2006.

[KK99] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.

[KKR08] Hans-Jörg Kreowski, Sabine Kuske, and Grzegorz Rozenberg. Graph transformation units – an overview. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 57–75. Springer, 2008.

[KKS97] Hans-Jörg Kreowski, Sabine Kuske, and Andy Schürr. Nested graph transformation units. *International Journal on Software Engineering and Knowledge Engineering*, 7(4):479–502, 1997.

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model transformation in the large. In Ivica Crnkovic and Antonia Bertolino, editors, *ESEC/SIGSOFT FSE*, pages 285–294. ACM, 2007.

[KS06] Alexander Königs and Andy Schürr. Tool integration with triple graph grammars - a survey. *Electr. Notes Theor. Comput. Sci.*, 148(1):113–150, 2006.

[Küs06] Jochen Malte Küster. Definition and validation of model transformations. *Software and System Modeling*, 5(3):233–259, 2006.

[OMG08] OMG. Meta object facility (MOF) 2.0 query/view/transformation (QVT). http://www.omg.org/spec/QVT/, 2008.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994.

[SK08] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In Ehrig et al. [EHRT08], pages 411–425.

[VB07] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):187–207, 2007.

[VGP08] Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors. *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, volume 5063 of *Lecture Notes in Computer Science*. Springer, 2008.

[Wag08] Dennis Wagelaar. Composition techniques for rule-based model transformation languages. In Vallecillo et al. [VGP08], pages 152–167.

[YCWD09] Andrés Yie, Rubby Casallas, Dennis Wagelaar, and Dirk Deridder. An approach for evolving transformation chains. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 551–555. Springer, 2009.