Electronic Communications of the EASST
Volume 62 (2013)

Specification, Transformation, Navigation
Special Issue dedicated to Bernd Krieg-Brückner on the Occasion of his 60th Birthday

Graph Tuple Transformation

Hans-Jörg Kreowski and Sabine Kuske

23 pages

Guest Editors: Till Mossakowski, Markus Roggenbach, Lutz Schröder
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Graph Tuple Transformation

Hans-Jörg Kreowski[1] **and Sabine Kuske**[2*]

[1] kreo@informatik.uni-bremen.de, http://www.informatik.uni-bremen.de/theorie
[2]kuske@informatik.uni-bremen.de,http://www.informatik.uni-bremen.de/~kuske
Department of Computer Science
University of Bremen, Germany

Abstract: Graph transformation units are rule-based devices to model and compute relations between initial and terminal graphs. In this paper, they are generalized to graph tuple transformation units that allow one to combine different kinds of graphs into tuples and to process the component graphs simultaneously and interrelated with each other. Moreover, one may choose some of the working components as inputs and some as outputs such that a graph tuple transformation unit computes a relation between input and output tuples of potentially different kinds of graphs rather than a binary relation on a single kind of graphs.

Keywords: Graph transformation, transformation units, graph tuples

1 Introduction

For some decades, many software engineers have dreamt about system development in such a way that data processing problems and their solution are modeled by means suitable for the application domain and then transformed into smoothly and efficiently running and trustworthy programs. This idea has been discussed under various headings like program transformation and stepwise refinement (see, e.g., Wirth [Wir71] and Basin and Krieg-Brückner [BK99]). Nowadays the term model transformation is quite popular referring to the transformation of platform-independent models into platform-dependent models (see, e.g., Frankel [Fra03]). As the former ones are often assumed to be visual models like UML diagrams or Petri nets, some researchers have proposed graph transformation as a framework for the description of visual models as well as their transformation (see, e.g., [LT04, Küs06, EEE+07, KKS07, VB07, EE08]).

In this paper, we try to enhance the usefulness of graph transformation as a base of model transformation by introducing the concept of graph tuple transformation offering the parallel processing of graph components of arbitrary tuples. The basic idea is that a visual model of a software system usually does not consist of a single diagram, but of a family of interrelated diagrams which is better covered by a graph tuple than a single graph. Our particular motivation is the observation that the area of theoretical computer science provides a wealth of model transformation examples like the transformation of automata into grammars, grammars into some normal forms, formulas into graphs, etc. where most involved models are tuples of some kind and their transformation often consists of transformations of the tuple components. We hope

* Research 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).

that it is worthwhile to exploit the experiences of model transformation in theoretical computer science.

In more explicit terms, graph transformation units are generalized to graph tuple transformation units. A graph transformation unit consists of specifications of initial and terminal graphs, a set of rules, a set of imported graph transformation units, and a control condition. The semantics is a relation between initial and terminal graphs obtained by the interleaving of rule applications and calls of imported units in such a way that the control condition holds. If the initial and terminal graphs are models of some kind, a graph transformation unit specifies a model transformation. Graph transformation units do not depend on specific rule classes, specific classes of graph class specifications or particular classes of control conditions, i.e., their components can be taken from an arbitrary graph transformation approach consisting of a graph class, a rule class, a class of graph class expressions, and a class of control conditions.

The basic components of graph transformation units are recalled in the following section (see [KK99, KKR08] for more details). The generalization to tuples of graphs is introduced in Section 3. The component graphs may be of different kinds like directed or undirected graphs. They may use different label alphabets. Or they may be of special forms representing strings, numbers or truth values for example. Corresponding to the different kinds of component graphs, there can be different kinds of rules available for the various components. The transformation of graph tuples is based on actions being tuples of rules, which can be applied in parallel to the component graphs. To get more flexibility, we allow also actions where rules may be replaced by calls of units or by the special symbol $-$. The latter means that nothing happens in the respective components. The former replaces a single rule application by an auxiliary computation of another unit. Moreover, a mechanism is provided that allows one to choose some of the components of the graph tuples processed by actions as inputs and some as outputs. Respectively, the semantics of a graph tuple transformation unit is a relation between input tuples and output tuples given by an iterated execution of actions. All new concepts are illustrated by various aspects of the recognition of strings by finite automata. A short and preliminary version of this paper appeared as [KKK04].

2 Graph Transformation

In this section we recall the main concepts of graph transformation like graphs, rules and graph transformation units and illustrate them with various small examples. In the literature one can find many more applications of graph transformation which stress the usefulness from the practical point of view. These are for example applications from the area of functional languages ([PE93, SPE93]), visual languages (e.g. [BMST99, EG07]), software engineering ([Nag96]), UML (e.g. [BKPT00, EHHS00, FNTZ00, PS00, EHK01, Kus01, KGKZ09, HZG06]), and agent systems (e.g. [Jan99, DHK02, GK07, TKKT07]).

Graph transformation comprises devices for the rule-based manipulation of graphs. Given a set of graph transformation rules and a set of graphs, one gets a graph transformation system in its simplest form. Such a system transforms a start graph by applying its graph transformation rules. The semantics can be defined as a binary relation on graphs where the first component of every pair is some start graph G and the second component is a graph derived from G by applying

a sequence of graph transformation rules. In general, the application of a graph transformation rule to a graph transforms it locally, i.e., it replaces a part of the graph with another graph part. Often one wishes to start a derivation only from certain initial graphs, and to accept as results only those derived graphs that are terminal. Moreover, in some cases the derivation process should be regulated in a certain way to cut down the nondeterminism of rule applications. For example, one may employ a parallel mode of transformation as in L systems, or one may restrict the order in which rules are applied. Graph class expressions and control conditions are suitable to restrict the derivation process where the former allow to choose initial and terminal graphs and the latter require certain derivation steps and forbid others. Altogether, graphs, rules, their application, graph class expressions, and control conditions are the basic elements of a so-called graph transformation approach.

2.1 Graphs

One of the most elementary components of a graph transformation system is a class of graphs $\mathscr{G}$. The graphs of $\mathscr{G}$ can be directed or undirected, typed or untyped, labeled or unlabeled, simple or multiple. Examples for graph classes are labeled directed graphs, hypergraphs, trees, forests, state graphs of finite automata, Petri nets, etc. The choice of graphs depends on the kind of applications one has in mind and is a matter of taste.

In this paper, we consider directed, edge-labeled graphs with individual, multiple edges. A *graph* is a construct $G = (V,E,s,t,l)$ where V is a finite set of *nodes*, E is a finite set of *edges*, $s,t\colon : E \to V$ are two mappings assigning each edge $e \in E$ a *source* $s(e)$ and a *target* $t(e)$, and $l\colon E \to C$ is a mapping labeling each edge in a given finite label alphabet C. A graph may be represented in a graphical way with circles as nodes and arrows as edges that connect source and target each, with the arrowhead pointing to the target. The labels are placed next to the arrows. In the case of a loop, i.e., an edge with the same node as source and target, we may draw a flag that is posted on its node with the label inside the box. To cover unlabeled edges as a special case, we assume a particular label $*$ that is invisible in the drawings.

For instance, the graph in Figure 1 is a state graph of a finite deterministic automaton where the edges labeled with a and b represent transitions, and the sources and targets of the transitions represent states. The start state is indicated with a *start*-flag and every final state with a *final*-flag. Moreover there is a flag labeled with *current* at the current state of the state graph of the finite deterministic automaton.

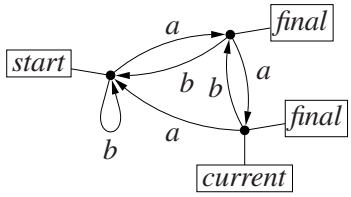


Figure 1: The state graph of a deterministic finite automaton

Two further instances of graphs are shown in Figure 2. The left one consists of five nodes and six directed edges (two of which are represented as flags). It is a string graph which represents the string *abba*. The beginning of the string is indicated with the *begin*-flag at the source of the leftmost *a*-edge. Analogously, there is an *end*-flag at the end of the string, i.e., at the target of the rightmost *a*-edge.

Figure 2: A string graph (left) and a 4-string graph (right)

If one takes this string graph and removes all occurrences of the labels a and b, one gets the string graph on the right of Figure 2 which is a graph that is a simple unlabeled path from a *begin*-flagged node to an *end*-flagged node. Such string graphs can be used to represent natural numbers. Hence, the right string graph in Figure 2 represents the number 4 because it has four unlabeled edges between its *begin*-flagged and its *end*-flagged node. Whenever a string graph represents a natural number k in this way, we say that it is the *k-string graph*.

A *graph morphism* g from a graph $L = (V,E,s,t,l)$ to a graph $G = (V',E',s',t',l')$ is a pair of mappings $(g_V(: V \to V', g_E : E \to E')$ that is structure and label preserving, i.e., $g_V(s(e)) = s'(g_E(e))$, $g_V(t(e)) = t'(g_E(e))$, and $l(e) = l'(g_E(e))$. The graph morphism g is *injective* if g_V and g_E are injective.

2.2 Rules and Rule Applications

To be able to transform the graphs in $\mathscr{G}$, rules are applied to the graphs yielding graphs. Given some class $\mathscr{R}$ of graph transformation rules, each rule $r \in \mathscr{R}$ defines a binary relation $\underset{r}{\Longrightarrow} \subseteq \mathscr{G} \times \mathscr{G}$ on graphs. If $G \underset{r}{\Longrightarrow} H$, one says that G *directly derives* H by applying r.

There are many possibilities to choose rules and their applications. Rule classes may vary from the more restrictive ones, like edge replacement [DHK97] or node replacement [ER97], to the more general ones, like double-pushout rules (Corradini, Ehrig, Löwe, Montanari, and Rossi [CEH+97]), single-pushout rules (Ehrig, Heckel, Korff, Löwe, Ribeiro, Wagner, and Corradini [EHK+97]), or PROGRES rules (Schürr [Sch97]). In general, all rule classes contain at least a left-hand side which specifies the part that should be deleted from the graph to which the rule is applied, and a right-hand side which determines the items that should be added to the graph.

In the examples of this paper, we use a simplified form of double-pushout rules, but it is worth noting that our approach works for arbitrary rule classes. Every *rule* (of this simplified rule class) is a triple $r = (L,K,R)$ where L and R are graphs (the *left-* and *right-hand side* of r, respectively) and K is a set of nodes shared by L and R. In graphical representations, L and R are drawn as usual, with numbers uniquely identifying the nodes in K. Its application means to replace an occurrence of L with R such that the common part K is kept.

A rule $r = (L,K,R)$ can be applied to some graph G directly deriving the graph H if H can

be constructed up to isomorphism (i.e., up to the renaming of nodes and edges) in the following way.

1. Find an injective graph morphism g from L to G, i.e. a subgraph $g(L)$ that coincides with L up to the naming of nodes and edges.

2. Remove all nodes and edges of $g(L-K)$, i.e., all nodes and edges of $g(L)$ except the nodes corresponding to K, provided that the remainder is a graph (which holds if the removal of a node is accompanied by the removal of all its incident edges).

3. Add R by merging each node v in K with its image $g(v)$ in G.

For example, the rule $go(a)$ in Figure 3 has as left-hand side a graph consisting of an a-edge from node 1 to node 2 and a *current*-flag at node 1. The right-hand side consists of node 1, node 2, the a-edge and a new *current*-flag at the target of the a-edge. The common part of $go(a)$ consists of nodes 1 and 2 as well as of the a-edge. The rule $go(a)$ moves a *current*-flag from the source of some a-labeled edge to the target of this edge. Its application to the state graph in Figure 1 results in the same state graph except that the current state is changed to the start state.

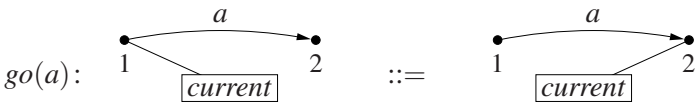


Figure 3: The rule $go(a)$

Another example of a rule is shown in Figure 4. It has as left-hand side a graph consisting of an a-edge and a *begin*-flag. The right-hand side consists of the target of a new *begin*-flag at the target of the a-edge. The common part of the rule $read(a)$ consists of the target of the a-edge. The rule $read(a)$ can be applied to the left string graph in Figure 2. Its application deletes the *begin*-flag and the leftmost a-edge together with its source. It adds a new *begin*-flag at the target of the a-edge. The resulting string graph represents the string *bba*.

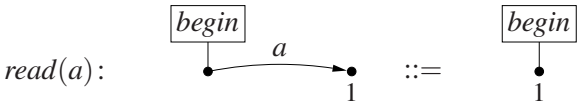


Figure 4: The rule $read(a)$

2.3 Graph Class Expressions

The aim of graph class expressions is to restrict the class of graphs to which certain rules may be applied or to filter out a subclass of all the graphs that can be derived by a set of rules. Typically, a graph class expression may be some logic formula describing a graph property like connectivity,

or acyclicity, or the occurrence or absence of certain labels. In this sense, every graph class expression e specifies a set $SEM(e)$ of graphs in $\mathscr{G}$.

For instance, *all* refers to all directed, edge-labeled graphs, whereas *empty* designates the class consisting of the empty graph *EMPTY*. Every graph $G \in \mathscr{G}$ is also a graph class expression specifying itself. In particular, we also use the graph class expression *dfsg* specifying all state graphs of deterministic finite automata as well as the expression *START* specifying all graphs in $SEM(dfsg)$ where the current state is the start state. Moreover, the expressions *string* and *nat* specify the set of all string graphs and the set of all k-string graphs, respectively. A further graph class expression used in the following is *bool* which specifies the two graphs *TRUE* and *FALSE*. Both graphs consist of a single node with a flag that is labeled *true* and *false*, respectively:

A particular kind of a graph class expression is given by a single graph $TYPE \in \mathscr{G}$ specifying the class $SEM(TYPE)$ of all graphs that can be mapped homomorphically to *TYPE*. These graphs are called typed graphs in, e.g., [EEPT06].

It is meaningful to require that graph class expressions are decidable, i.e., for any graph class expression e and any graph G there should be an algorithm that decides whether $G \in SEM(e)$. It is worth noting that this is the case in the presented examples.

2.4 Control Conditions

A control condition is an expression that determines, for example, the order in which rules may be applied. Semantically, it relates start graphs with graphs that result from an admitted transformation process. In this sense, every control condition c specifies a binary relation $SEM(c)$ on $\mathscr{G}$. As control condition we use in particular the expression *true* that allows all transformations (i.e., all pairs of graphs). Moreover, we use regular expressions as control conditions. They describe in which order and how often the rules and imported units are to be applied. In particular, for control conditions C, C_1, and C_2 the expression $C_1;C_2$ specifies the sequential composition of both semantic relations, $C_1|C_2$ specifies the union, and C^* specifies the reflexive and transitive closure, i.e., $SEM(C_1;C_2) = SEM(C_1) \circ SEM(C_2)$, $SEM(C_1|C_2) = SEM(C_1) \cup SEM(C_2)$, and $SEM(C^*) = SEM(C)^*$. Moreover, for a control condition C the expression $C!$ requires to apply C as long as possible, i.e., $SEM(C)$ consists of all pairs $(G,H) \in SEM(C)^*$ such that there is no H' with $(H,H') \in SEM(C)$. In the following the control condition $C_1|\cdots|C_n$ will also be denoted by $\{C_1,\ldots,C_n\}$.

For example, let C_1, C_2, and C_3 be control conditions that specify binary relations on graphs of a certain type. Then the expression $C_1!;C_2^*;(C_3|C_1)$ prescribes to apply first C_1 as long as possible, then C_2 arbitrarily often, and finally C_3 or C_1 exactly once. The precise meaning of a regular expression is explained where it is used. More about control conditions can be found in, e.g., [Kus00, HP01, HKK08].

2.5 Transformation Units

A class of graphs, a class of rules, a rule application operator, a class of graph class expressions, and a class of control conditions form a graph transformation approach based on which graph

transformation units as a unifying formalization of graph grammars and graph transformation systems can be defined. More precisely, a *graph transformation approach* is defined as $\mathscr{A} = (\mathscr{G}, \mathscr{R}, \Longrightarrow, \mathscr{X}, \mathscr{C})$ where $\mathscr{G}$ is a graph class, $\mathscr{R}$ is a rule class, $\Longrightarrow$ is a rule application operator which provides a rule application relation $\underset{r}{\Longrightarrow} \subseteq \mathscr{G} \times \mathscr{G}$ for each $r \in \mathscr{R}$, $\mathscr{X}$ is a class of graph class expressions with $SEM(e) \subseteq \mathscr{G}$ for all $e \in \mathscr{X}$, and $\mathscr{C}$ is a class of control conditions with $SEM(c) \subseteq \mathscr{G} \times \mathscr{G}$ for all $c \in \mathscr{C}$. In the following, the components $\mathscr{G}$, $\mathscr{R}$, $\Longrightarrow$, $\mathscr{X}$, and $\mathscr{C}$ of an approach $\mathscr{A}$ are also denoted by $\mathscr{G}_{\mathscr{A}}$, $\mathscr{R}_{\mathscr{A}}$, $\underset{\mathscr{A}}{\Longrightarrow}$, $\mathscr{X}_{\mathscr{A}}$, and $\mathscr{C}_{\mathscr{A}}$, respectively.

In general, a graph transformation system may consist of a huge set of rules that by its size alone is difficult to manage. Transformation units provide a means to structure the transformation process. The main structuring principle of transformation units relies on the import of other transformation units or – on the semantic level – of binary relations on graphs. The input and the output of a transformation unit each consist of a class of graphs that is specified by a graph class expression. The input graphs are called initial graphs and the output graphs terminal graphs. A transformation unit transforms initial graphs to terminal graphs by interleaving the applications of graph transformation rules and imported transformation units. Since rule application is nondeterministic in general, a transformation unit contains a control condition that may regulate the graph transformation process.

Let $\mathscr{A} = (\mathscr{G}, \mathscr{R}, \Longrightarrow, \mathscr{X}, \mathscr{C})$ be a graph transformation approach. A *graph transformation unit* over $\mathscr{A}$ is a system $gtu = (I, U, R, C, T)$ where I and T are graph class expressions in $\mathscr{X}$, U is a (possibly empty) set of imported graph transformation units over $\mathscr{A}$, R is a set of rules in $\mathscr{R}$, and C is a control condition in $\mathscr{C}$.

To simplify technicalities, we assume that the import structure is acyclic (for a study of cyclic imports see [KKS97]). Initially, one builds units of level 0 with empty import. Then units of level 1 are those that import only units of level 0 but at least one, and units of level $n+1$ import only units of level 0 to level n, but at least one from level n.

In graphical representations of transformation units we omit the import component if it is empty, the initial or terminal component if it is set to *all*, and the control condition if it is equal to *true*, meaning that there is no restriction.

In the following, we present some simple examples of transformation units specifying natural numbers and truth values. The latter are used in Subsection 3.5 as an auxiliary data type to model the more interesting examples concerning finite automata. The control condition of each example unit in this section is equal to *true*. The examples in Subsection 3.5 contain more sophisticated control conditions.

The first transformation unit *nat*0 depicted in Figure 5 constructs all string graphs that represent natural numbers by starting from its initial graph, which represents 0, and transforming the n-string graph into the $n+1$-string graph by applying the rule *succ*.

The second transformation unit *nat1* shown in Figure 6 is a variant of *nat0*, but now with all n-string graphs as initial graphs. Consequently, it describes arbitrary additions to arbitrary n-string graphs by sequentially increasing the represented numbers by 1.

The third transformation unit *nat2* is shown in Figure 7. It also transforms string graphs into string graphs. It has two rules *pred* and *is-zero*. The application of the rule *pred* to the n-string graph (with $n \geq 1$ since otherwise the rule cannot be applied) converts it into the $n-1$-string graph. The second rule *is-zero* can be applied only to the 0-string graph but does not transform

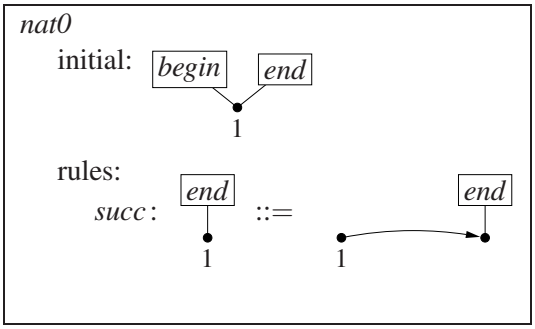


Figure 5: The transformation unit *nat*0

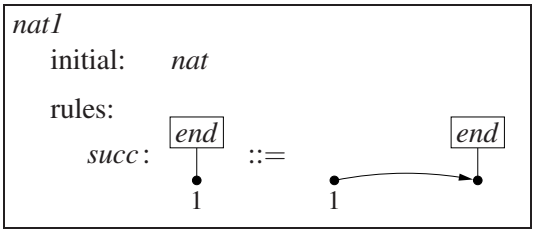


Figure 6: The transformation unit *nat*1

it, which means that this rule can be used as a test for 0.

The transformation unit *nat2* imports *nat1* so that arbitrary additions can be performed, too. The rules of *nat2* and the imported unit *nat1* can be applied in arbitrary order and arbitrarily often. Hence *nat2* converts n-string graphs into m-string graphs for natural numbers m, n. Therefore *nat2* can be considered as a data type representing natural numbers with a simple set of operations. Our model of the natural numbers is very simple providing just a graphical variant

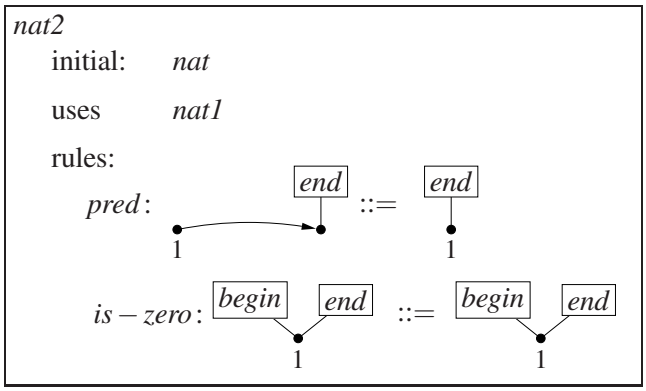


Figure 7: The transformation unit *nat*2

of the unary representation with the possibility to increase and to decrease a number by 1 as well as to test for 0. A more sophisticated model of natural numbers is not needed in this paper.

The fourth transformation unit $bool0 = (empty, \emptyset, generate\text{-}true, true, bool)$ in Figure 8 has a single initial graph, the empty graph *EMPTY*. It does not import other transformation units and it has one rule *generate-true* which turns *EMPTY* to the graph *TRUE*. The control condition allows all transformations, meaning that *TRUE* may be added arbitrarily often to *EMPTY*. However, the terminal graph class expression *bool* ensures that the rule *generate-true* is applied exactly once to the initial graph. One can consider *bool*0 as a unit that describes the type Boolean in its simplest form. At first sight, this may look a bit strange. But it is quite useful if one wants to specify predicates on graphs by nondeterministic graph transformation: If one succeeds to transform an input graph into the graph *TRUE*, the predicate holds; otherwise it fails. In other words, if the predicate does not hold for the input graph, none of its transformations yields *TRUE*.

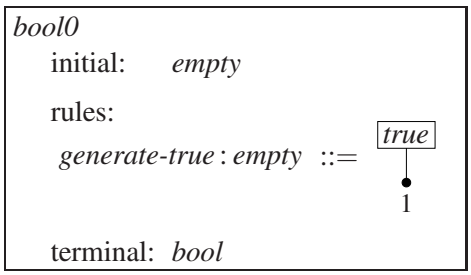


Figure 8: The transformation unit *bool*0

2.6 Interleaving Semantics of Transformation Units

Transformation units transform initial graphs to terminal graphs by applying graph transformation rules and imported transformation units so that the control condition is satisfied. Hence, the semantics of a transformation unit can be defined as a binary relation between initial and terminal graphs.

For example, the interleaving semantics of the transformation unit *nat*2 consists of all pairs (G, G') such that G is a k-string graph and G' is an l-string graph (for some $k, l \in \mathbb{N}$). In general, for a transformation unit *gtu* without import, the semantics of *gtu* consists of all pairs (G, G') of graphs such that

1. G is an initial graph and G' is a terminal graph;
2. G' is obtained from G via a sequence of rule applications, i.e., (G, G') is in the reflexive and transitive closure of the binary relation obtained from the union of all relations $\underset{r}{\Longrightarrow}$ where r is some rule of *gtu*; and
3. the pair (G, G') is allowed by the control condition.

If the transformation unit *gtu* has a non-empty import, the interleaving semantics of *gtu* consists of all pairs (G, G') of graphs which satisfy the preceding items 1 and 3, and where, in addition

to rules, imported transformation units can be applied in the transformation process of *gtu*, i.e., the second item above is extended to:

2'. G' is obtained from G via a sequence of rule applications and applications of imported units. This means that (G,G') is in the reflexive and transitive closure of the binary relation obtained from the union of all relations $\underset{r}{\Longrightarrow}$ and $SEM(u)$ where r is some rule of *gtu* and u is some imported transformation unit of *gtu*.

More formally, the interleaving semantics of *gtu* is defined as follows. Let $gtu = (I,U,R,C,T)$ be a transformation unit. Then the *interleaving semantics* $SEM(tu)$ is recursively defined as

$$SEM(tu) = SEM((I,U,R,C,T)) =$$
$$SEM(I) \times SEM(T) \cap (\bigcup_{r\in R} \underset{r}{\Longrightarrow} \cup \bigcup_{u\in U} SEM(u))^* \cap SEM(C).$$

If the transformation unit *gtu* is of level 0, the semantic relation is well-defined because the union over U is the empty set. If *gtu* is of level $n+1$, we can inductively assume that $SEM(u)$ of each imported unit u is already well-defined, so that $SEM(tu)$ is also well-defined as a union and intersection of defined relations.

3 Graph Tuple Transformation

Graph transformation in general transforms graphs into graphs by applying rules, i.e. in every transformation step a single graph is transformed with a graph transformation rule. In graph tuple transformation, this operation is extended to tuples of graphs. This means that in every transformation step a tuple of graphs is transformed with a tuple of rules. The graphs, the rules, and the ways the rules have to be applied are taken from a so-called tupling base which consists of a tuple of rule bases where a rule base $\mathscr{B} = (\mathscr{G},\mathscr{R},\Longrightarrow)$ is a graph transformation approach without control conditions and graph class expressions.

3.1 Basic Actions

The iterated application of rules transforms graphs into graphs yielding an input-output relation on graphs. But in many applications one would like to consider several inputs and maybe even several outputs, or at least an output of a type different from all inputs. In order to reach such an extra flexibility, we introduce in this section the transformation of tuples of graphs, which is the most basic operation of the graph tuple transformation units presented in Subsection 3.4.

Graph tuple transformation over a tupling base is an extension of ordinary rule application in the sense that graphs of different classes can be transformed in parallel. For example to check whether some string can be recognized by a deterministic finite automaton, one can transform three graphs in parallel: The first graph is a string graph representing the string to be recognized, the second graph is a state graph of a deterministic finite automaton, and the third graph represents the boolean value *false*. To recognize the string one applies a sequence of rule applications which consume the string graph while the corresponding transitions of the deterministic finite state graph are traversed. If after reading the whole string the current state is a final state,

the third graph is transformed into a graph representing *true*. This example will be explicitly modeled in Subsection 3.5.

In graph tuple transformation, tuples of rules are applied to tuples of graphs. A tuple of rules may also contain the symbol $-$ in some components where no change is desired. The graphs and the rules are taken from a *tupling base*, which is a tuple of rule bases $\mathscr{TB} = (\mathscr{B}_1, \ldots, \mathscr{B}_n)$. Let $(G_1, \ldots, G_n)$ and $(H_1, \ldots, H_n)$ be graph tuples over $\mathscr{TB}$, i.e. $G_i, H_i \in \mathscr{G}_{\mathscr{B}_i}$ for $i = 1, \ldots, n$. Let $a = (a_1, \ldots, a_n)$ with $a_i \in \mathscr{R}_{\mathscr{B}_i}$ or $a_i = -$ for $i = 1, \ldots, n$. Then $(G_1, \ldots, G_n) \xrightarrow[a]{} (H_1, \ldots, H_n)$ if for $i = 1, \ldots, n$, $G_i \xRightarrow[a_i]{} H_i$ if $a_i \in \mathscr{R}_{\mathscr{B}_i}$ and $G_i = H_i$ if $a_i = -$. In the following we call a a *basic action of* $\mathscr{TB}$. For a set ACT of basic actions of $\mathscr{TB}$, $\xrightarrow[ACT]{}$ denotes the union $\bigcup_{a \in ACT} \xrightarrow[a]{}$, and $\xrightarrow[ACT]{*}$ its reflexive and transitive closure.

For example, let I be some finite alphabet and let $\mathscr{B}^{basic}_{string} = (STRING, \{read(x) \mid x \in I\}, \Longrightarrow)$ and $\mathscr{B}^{basic}_{dfsg} = (DFSG, \{go(x) \mid x \in I\}, \Longrightarrow)$ be two rule bases such that $STRING$ consists of all string graphs over some alphabet I and $DFSG$ consists of all state graphs of deterministic finite automata over I. Let $(abba)^\bullet$ be the left string graph of Figure 2, let $gr(A, v_0)$ be the state graph in Figure 1 where A denotes the corresponding finite automaton and v_0 is the node with the *current*-flag. Then $((abba)^\bullet, gr(A, v_0)) \xrightarrow[a]{} ((bba)^\bullet, gr(A, v_1))$ for the basic action $a = (read(a), go(a))$ of tupling base $(\mathscr{B}^{basic}_{string}, \mathscr{B}^{basic}_{dfsg})$ where v_1 is the node with the *start*-flag.

The transformation of graph tuples via a sequence of basic actions is equivalent to the transformation of tuples of graphs where every component is transformed independently with a sequence of direct derivations of the corresponding component. This is expressed in the following proposition.

Proposition 1 *Let ACT be the set of all basic actions of $\mathscr{TB} = (\mathscr{B}_1, \ldots, \mathscr{B}_n)$. Then*

$$(G_1, \ldots, G_n) \xrightarrow[ACT]{*} (H_1, \ldots, H_n)$$

if and only if $G_i \xRightarrow[\mathscr{R}_{\mathscr{B}_i}]{*} H_i$ for $i = 1, \ldots, n$.

3.2 Graph Tuple Class Expressions

Similarly to graph class expressions, the aim of graph tuple class expressions is to restrict the class of graph tuples to which certain transformation steps may be applied, or to filter out a subclass of all the graph tuples that can be obtained from a transformation process. Typically, a graph tuple class expression may be some logic formula describing a tuple of graph properties like connectivity, or acyclicity, etc. Formally, every graph tuple class expression e over a tupling base $\mathscr{TB} = (\mathscr{B}_1, \ldots, \mathscr{B}_n)$ specifies a set $SEM(e) \subseteq \mathscr{G}_{\mathscr{B}_1} \times \cdots \times \mathscr{G}_{\mathscr{B}_n}$ of graph tuples in $\mathscr{TB}$. As graph class expressions, graph tuple class expressions should be decidable.

In many cases, such a graph tuple class expression will be a tuple $e = (e_1, \ldots, e_n)$ where the ith item e_i restricts the graph class $\mathscr{G}_{\mathscr{B}_i}$ of the rule base $\mathscr{B}_i$, i.e. $SEM_{\mathscr{B}_i}(e_i) \subseteq \mathscr{G}_{\mathscr{B}_i}$ for $i = 1, \ldots, n$. Consequently, the semantics of e is $SEM_{\mathscr{B}_1}(e_1) \times \cdots \times SEM_{\mathscr{B}_n}(e_n)$. Hence, each item e_i is a graph class expression as defined for transformation units in Section 2.

A typical example of a graph tuple class expression over the tupling base $\mathscr{TB}_3 = (\mathscr{B}, \mathscr{B}, \mathscr{B})$ for some rule base $\mathscr{B}$ is the relation of the component graphs of the triple of graphs as used in triple graph grammars [Sch94, SK08]. Let *triple* be a constant expression, then $SEM(triple)$ consists of all triples of graphs (G_1, G_2, G_3) such that G_2 is subgraph of G_1 and G_3 as well.

The graph tuple class expressions mainly used in this paper are tuples of graph class expressions. A simple example of a graph tuple class expression is $(e_1, \ldots, e_n)$ with $e_i = all$ for $i = 1, \ldots, n$ which does not restrict the graph classes of the rule bases, i.e. $SEM((e_1, \ldots, e_n)) = \mathscr{G}_{\mathscr{B}_1} \times \cdots \times \mathscr{G}_{\mathscr{B}_n}$.

3.3 Control Conditions for Graph Tuple Transformation

Similarly to control conditions in graph transformation units, a control condition for graph tuple transformation is an expression that determines, for example, the order in which transformation steps may be applied to graph tuples. Semantically, it relates tuples of start graphs with tuples of graphs that result from an admitted transformation process. In this sense, every control condition C over a tupling base $\mathscr{TB}$ specifies a binary relation $SEM(C)$ on the set of graph tuples in $\mathscr{TB}$. More precisely, for a tupling base $\mathscr{TB} = (\mathscr{B}_1, \ldots, \mathscr{B}_n)$ $SEM(C)$ is a subset of $(\mathscr{G}_{\mathscr{B}_1} \times \cdots \times \mathscr{G}_{\mathscr{B}_n})^2$.

As control conditions we use in particular actions, regular expressions over control conditions (i.e. sequential composition, union, and iteration of control conditions), as well as the expression *as-long-as-possible* (abbreviated with the symbol !). An action prescribes which rules or imported units should be applied to a graph tuple, i.e. an action is a control condition that allows one to synchronize different transformation steps. The basic actions of Subsection 3.1 are examples of actions. Roughly speaking, an action over a tupling base $\mathscr{TB} = (\mathscr{B}_1, \ldots, \mathscr{B}_n)$ is a tuple $act = (a_1, \ldots, a_n)$ that specifies an n,n-relation $SEM(act) \subseteq (\mathscr{G}_{\mathscr{B}_1} \times \cdots \times \mathscr{G}_{\mathscr{B}_n})^2$. Actions will be explained in detail in Subsection 3.4.

3.4 Graph Tuple Transformation Units

Graph tuple transformation units provide a means to structure the transformation process from a tuple of input graphs to a tuple of output graphs. More precisely, a graph tuple transformation unit transforms k-tuples of graphs into l-tuples of graphs such that the graphs in the k-tuples as well as the graphs in the l-tuples may be of different types. Hence, a graph tuple transformation unit specifies a k,l-relation on graphs. Internally, a graph tuple transformation unit transforms n-tuples of graphs into n-tuples of graphs, i.e. it specifies internally an n,n-relation on graphs. The transformation of the n-tuples is performed according to a tupling base which is specified in the declaration part of the unit. The k,l-relation is obtained from the n,n-relation by embedding k input graphs into n initial graphs and by projecting n terminal graphs onto l output graphs. The embedding and the projection are also given in the declaration part of a unit.

Graph tuple transformation units generalize the notion of triple grammars as introduced by Schürr [Sch94] which in turn are a generalization of pair grammars studied by Pratt [Pra71].

Tupling bases, graph tuple class expressions, and control conditions form the ingredients of graph tuple transformation units. Moreover, the structuring of the transformation process is achieved by an import component, i.e. every unit may import a set of other units. The transformations offered by an imported unit can be used in the transformation process of the importing

unit.

The basic operation of a graph tuple transformation unit is the application of an action, which is a transformation step from one graph tuple into another where every component of the tuple is modified either by means of a rule application, or is set to some output graph of some imported unit, or remains unchanged. Since action application is nondeterministic in general, a transformation unit contains a control condition that may regulate the graph tuple transformation process. Moreover, a unit contains an initial graph tuple class expression and a terminal graph tuple class expression. The former specifies all possible graph tuples a transformation may start with and the latter specifies all graph tuples a transformation may end with. Hence, every transformation of an n-tuple of graphs with action sequences has to take into account the control condition of the unit as well as the initial and terminal graph tuple class expressions.

A tuple of sets of rules, a set of imported units, a control condition, an initial graph tuple class expression, and a terminal graph tuple class expression form the body of a graph tuple transformation unit. All components in the body must be consistent with the tupling base of the unit.

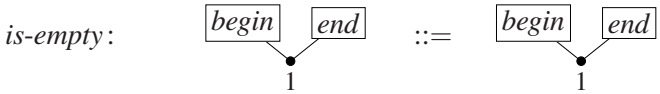
Formally, let $\mathscr{TB} = (\mathscr{B}_1, \ldots, \mathscr{B}_n)$ be a tupling base. A *graph tuple transformation unit gttu with tupling base* $\mathscr{TB}$ is a pair $(decl, body)$ where *decl* is the *declaration part* of *gttu* and *body* is the *body* of *gttu*. The declaration part is of the form $in \to out$ *on* $\mathscr{TB}$ where $in\colon [k] \to [n]$ and $out\colon [l] \to [n]$ are mappings with $k, l \in \mathbb{N}$.[1] The body of *gttu* is a system $body = (I, U, R, C, T)$ where I and T are graph tuple class expressions over $\mathscr{TB}$, U is a set of imported graph tuple transformation units, R is a tuple of rule sets $(R_1, \ldots, R_n)$ such that $R_i \subseteq \mathscr{R}_{\mathscr{B}_i}$ for $i = 1, \ldots, n$, and C is a control condition over $\mathscr{TB}$. The numbers k and l of *gttu* are also denoted by k_{gttu} and l_{gttu}. Moreover, the ith input class $\mathscr{G}_{\mathscr{B}_{in(i)}}$ of *gttu* is also denoted by $in_{gttu}(i)$ for $i = 1, \ldots, k$ and the jth output class $\mathscr{G}_{\mathscr{B}_{out(j)}}$ by $out_{gttu}(j)$ for $j = 1, \ldots, l$.

As in the case of graph transformation units, we assume that the import structure of graph tuple transformation units is acyclic.

3.5 Examples of Graph Tuple Transformation Units

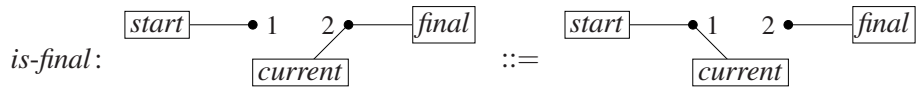
In the following we illustrate the concept of graph tuple transformation units with examples from the area of automata theory.

Example 1 The tupling base of the following example of a transformation unit is the tuple $(\mathscr{B}_{string}, \mathscr{B}_{dfsg}, \mathscr{B}_{bool})$. The rule base $\mathscr{B}_{string}$ is $(STRING, \{read(x) \mid x \in I\} \cup \{is\text{-}empty\}, \Longrightarrow)$, where the rule *is-empty* checks whether the graph to which it is applied represents the empty string. It has equal left- and right-hand sides consisting of a node with an *begin*- and an *end*-flag.



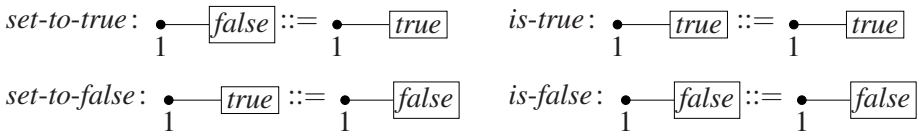
The rule base $\mathscr{B}_{dfsg}$ is $(DFSG, \{go(x) \mid x \in I\} \cup \{is\text{-}final\}, \Longrightarrow)$. The rule *is-final* checks whether the current state of a deterministic finite state graph is a final state, resetting it to the start state in that case, and can be depicted as follows.

[1] For a natural number $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$.



The rule base $\mathscr{B}_{bool}$ contains the graph class *bool* which consists of the two graphs *TRUE* and *FALSE*.

The class of rules of $\mathscr{B}_{bool}$ consists of the four rules



where *set-to-true* changes a *false*-flag into a *true*-flag, *set-to-false* does the same the other way round, *is-true* checks whether a graph of type *bool* is equal to *TRUE*, and *is-false* checks the same for *FALSE*.

Now we can define the unit *recognize* shown in Figure 9. It has as input graphs a string graph

recognize	
decl	$(string, dfsg, -) \to (-, -, bool)$ *on* $(\mathscr{B}_{string}, \mathscr{B}_{dfsg}, \mathscr{B}_{bool})$
initial	$(string, START, FALSE)$
rules	$(\mathscr{R}_{\mathscr{B}_{string}}, \mathscr{R}_{\mathscr{B}_{dfsg}}, \{set\text{-}to\text{-}true\})$
cond	$a_1!; a_2!$ where $a_1 = \{(read(x), go(x), -) \mid x \in I\}$ and $a_2 = (is\text{-}empty, is\text{-}final, set\text{-}to\text{-}true)$
terminal	$(string, dfsg, bool)$

Figure 9: A unit with empty import

and a state graph of a deterministic finite automaton and as output graph a boolean value. The mapping *in* of the declaration part of *recognize* is defined by $in\colon [2] \to [3]$ with $in(1) = 1$ and $in(2) = 2$. We use the more intuitive tuple notation $(string, dfsg, -)$ for this. The mapping *out* is denoted by $(-, -, bool)$ which means that $out\colon [1] \to [3]$ is defined by $out(1) = 3$. Hence, $in_{recognize}(1) = STRING$, $in_{recognize}(2) = DFSG$, and $out_{recognize}(1) = bool$.

The initial graph tuple class expression is $(string, START, FALSE)$, i.e. it admits all triples $(G_1, G_2, G_3) \in STRING \times DFSG \times bool$ where the *current*-edge of G_2 points to the start state and G_3 is equal to *FALSE*. The rules are restricted to the tuple

$$(\mathscr{R}_{\mathscr{B}_{string}}, \mathscr{R}_{\mathscr{B}_{dfsg}}, \{set\text{-}to\text{-}true\}).$$

Hence, just one rule from $\mathscr{B}_{bool}$ is admitted. The control condition requires to apply first the action a_1 as long as possible and then the action a_2 as long as possible, where a_1 applies $read(x)$ to the first component of the current graph tuple and $go(x)$ to the second component (for any $x \in I$). The action a_2 sets the third component to *TRUE* if the current string is empty, the current

state of the state graph is a final state, and the third component is equal to *FALSE*. This is the case where the string represented by the input string graph can be recognized by the automaton corresponding to the input state graph. Note that a_2 can be applied at most once because of *set-to-true*, and only in the case where a_1 cannot be applied anymore because of *is-empty*. In particular, if the string of the input string graph cannot be recognized by the automaton, the action a_2 cannot be applied at all. The terminal graph tuple class expression does not restrict the graph types of the tupling base, i.e. it is equal to $(string, dfsg, bool)$. The unit *recognize* does not import other units.

Example 2 The unit *recognize-intersection* shown in Figure 10 is an example of a unit with a non-empty import component. It has as input graphs a string graph and two state graphs of

recognize-intersection	
decl	$(string, dfsg, dfsg, -, -, -) \to (-, -, -, -, -, bool)$ *on* $(\mathscr{B}_{string}, \mathscr{B}_{dfsg}, \mathscr{B}_{dfsg}, \mathscr{B}_{bool}, \mathscr{B}_{bool}, \mathscr{B}_{bool})$
initial	$(string, dfsg, dfsg, bool, bool, FALSE)$
uses	*recognize*
rules	$(\emptyset, \emptyset, \emptyset, \{is\text{-}true\}, \{is\text{-}true\}, \{set\text{-}to\text{-}true\})$
cond	$a_1; a_2!$ where $a_1 = (-, -, -, recognize(1,2), recognize(1,3), -)$ and $a_2 = (-, -, -, is\text{-}true, is\text{-}true, set\text{-}to\text{-}true)$
terminal	$(string, dfsg, dfsg, bool, bool, bool)$

Figure 10: A unit with imported units combined in an action

deterministic finite state automata. The output graph represents again a boolean value. The tupling base of *recognize-intersection* is the six-tuple $(\mathscr{B}_{string}, \mathscr{B}_{dfsg}, \mathscr{B}_{dfsg}, \mathscr{B}_{bool}, \mathscr{B}_{bool}, \mathscr{B}_{bool})$. The mapping *in* of the declaration part requires to take a string graph from the first rule base of the tupling base, one state graph from the second and one from the third rule base as input graphs. The mapping *out* requires to take a graph from the last rule base as output graph.

The unit *recognize-intersection* imports the above unit *recognize* and has as local rules *is-true* and *set-to-true* where *is-true* can be applied to the fourth and the fifth component of the current graph tuples and *set-to-true* to the sixth component. The control condition requires the following.

1. Apply *recognize* to the first and the second component and write the result into the fourth component and
2. apply *recognize* to the first and the third component and write the result into the fifth component.
3. If then possible apply the rule *is-true* to the fourth and the fifth component and the rule *set-to-true* to the sixth component.

This means that in the first point *recognize* is applied to the input string graph and the first one of the input state graphs. In the second point *recognize* must be applied to the input string

graph and to the second state graph of a deterministic finite automaton. These two transformations can be performed in parallel within one and the same action denoted by the tuple $(-,-,-,recognize(1,2),recognize(1,3),-)$. (The precise semantics of this action will be given in the next subsection where actions and their semantics are introduced formally.) The rule application performed in the third point corresponds to applying the basic action a_2. Since the initial graph tuple class expression requires that the sixth graph represent *false*, this means one application due to *set-to-true*. The terminal graph tuple class expression admits all graph tuples of the tupling base.

Example 3 Let I be the alphabet consisting of the symbols a,b, let L,L_a,L_b be regular languages, and let $subst\colon I \to \mathscr{P}(I^*)$ be a substitution with $subst(a) = L_a$ and $subst(b) = L_b$. The aim of the following example is to model the recognition of the substitution language $subst(L) = \{subst(w) \mid w \in L\}$ based on a description of L,L_a,L_b by deterministic finite automata. (The model can of course be extended to arbitrarily large alphabets.)

First, consider the unit *reduce* shown in Figure 11. It takes a string graph and a state graph of

reduce	
decl	$(string,dfsg) \to (string,-)$ *on* $(\mathscr{B}_{string},\mathscr{B}_{dfsg})$
initial	$(string,START)$
rules	$(\mathscr{R}_{\mathscr{B}_{string}},\mathscr{R}_{\mathscr{B}_{dfsg}})$
cond	${a_1}^*;a_2$ where $a_1 = \{(read(x),go(x)) \mid x \in I\}$ and $a_2 = (-,is\text{-}final)$
terminal	$(string,dfsg)$

Figure 11: A unit that returns a modified input graph as output

a deterministic finite automaton as input, requiring through the initial component that the state graph be in its start state. It then reduces the string graph by arbitrarily often applying actions of the form $(read(x),go(x))$, i.e. by consuming an arbitrarily large prefix of the string and changing states accordingly in the state graph, and returns the residue of the string graph as output, but only if the consumed prefix is recognized by the state graph, i.e. only if the action $(-,is\text{-}final)$ is applied exactly once.

The unit *recognize-substitution* shown in Figure 12 makes use of *reduce* in order to decide whether an input string graph is in the substitution language given as further input by three state graphs of deterministic finite automata A,A_a,A_b that define L,L_a,L_b, in that order. Initially, the state graphs must once again be in their respective start states and the value in the output component is *false*. The idea is to guess, symbol by symbol, a string $w \in L$ such that the input string is in $subst(w)$. If the next symbol is guessed to be a, the action $(reduce(1,3),go(a),-,-,-)$ is applied that runs A_a to delete a prefix belonging to L_a from the input string $(reduce(1,3))$ and simultaneously executes the next state transition for a in A $(go(a))$. The second action $(reduce(1,4),go(b),-,-,-)$ works analogously for the symbol b. Thus, *recognize-substitution* is an example of a unit that combines an imported unit (*reduce*) and a rule $(go(x))$ in an action.

recognize-substitution	
decl	$(string, dfsg, dfsg, dfsg, -) \to (-,-,-,-, bool)$ *on* $(\mathscr{B}_{string}, \mathscr{B}_{dfsg}, \mathscr{B}_{dfsg}, \mathscr{B}_{dfsg}, \mathscr{B}_{bool})$
initial	$(string, START, START, START, FALSE)$
uses	*reduce*
rules	$(\{is\text{-}empty\}, \mathscr{R}_{\mathscr{B}_{dfsg}}, \emptyset, \emptyset, \{set\text{-}to\text{-}true\})$
cond	$(a_1 \mid a_2)^*; a_3$ where $a_1 = \{(reduce(1,3), go(a), -, -, -),$ $a_2 = \{(reduce(1,4), go(b), -, -, -),$ and $a_3 = (is\text{-}empty, is\text{-}final, -, -, set\text{-}to\text{-}true)$
terminal	$(string, dfsg, dfsg, bool, bool, bool)$

Figure 12: A unit with imported units combined in an action

Finally, a mandatory application of the action $(is\text{-}empty, is\text{-}final, -, -, set\text{-}to\text{-}true)$ produces the output value *true*, but only if the input string is completely consumed and A is in some final state.

It may be noted that even though the finite state graphs are deterministic, there are two sources of nondeterminism in this model: The symbols of the supposed string $w \in L$ must be guessed as well as a prefix of the input string for each such symbol. Consequently, the model admits only tuples with output *TRUE* in its semantics.

3.6 Semantics of Graph Tuple Transformation Units

Graph tuple transformation units transform initial graph tuples to terminal graph tuples by applying a sequence of actions so that the control condition is satisfied. Moreover, the mappings *in* and *out* of the declaration part prescribe for every such transformation the input and output graph tuples of the unit. Hence, the semantics of a graph tuple transformation unit can be defined as a k,l-relation between input and output graphs.

Let $gttu = (in \to out \text{ on } \mathscr{TB}, (I, U, R, C, T))$ be a graph tuple transformation unit with $\mathscr{TB} = (\mathscr{B}_1, \ldots, \mathscr{B}_n)$, $in\colon [k] \to [n]$, $out\colon [l] \to [n]$, and $R = (R_1, \ldots, R_n)$. If $U = \emptyset$, *gttu* transforms internally a tuple $G \in \mathscr{G}_{\mathscr{B}_1} \times \cdots \times \mathscr{G}_{\mathscr{B}_n}$ into a tuple $H \in \mathscr{G}_{\mathscr{B}_1} \times \cdots \times \mathscr{G}_{\mathscr{B}_n}$ if and only if

1. G is an initial graph tuple and H is a terminal graph tuple, i.e. $(G,H) \in SEM(I) \times SEM(T)$;
2. H is obtained from G via a sequence of basic actions over $(R_1, \ldots, R_n)$, i.e. $G \xrightarrow[ACT(gttu)]{*} H$ where $ACT(gttu)$ is the set of all basic actions $a = (a_1, \ldots, a_n)$ of $\mathscr{TB}$ such that for $i = 1, \ldots, n$, $a_i \in R_i$ if $a_i \neq -$, and
3. the pair (G,H) is allowed by the control condition, i.e. $(G,H) \in SEM(C)$.

If the graph tuple transformation unit *gttu* has a non-empty import, the imported units can also be applied in a transformation from G to H. This requires that we extend the notion of basic actions so that calls of imported units are allowed, leading to the notion of (general) actions.

Formally, an *action of gttu* is a tuple $a = (a_1, \ldots, a_n)$ such that for $i = 1, \ldots, n$ we have $a_i \in R_i$, or $a_i = -$, or a_i is of the form $(u, input, output)$ where $u \in U$, $input : [k_u] \to [n]$ with $\mathscr{G}_{\mathscr{B}_{input(j)}} \subseteq in_u(j)$ for $j = 1, \ldots, k_u$, and $output \in [l_u]$ with $out_u(output) \subseteq \mathscr{G}_{\mathscr{B}_i}$. In the latter case, we denote a_i by $u(input(1), \ldots, input(k_u))(output)$, and shorter by $u(input(1), \ldots, input(k_u))$ if u has a unique output, i.e. $l_u = 1 = output$.

The application of an action $a = (a_1, \ldots, a_n)$ to a current graph tuple of n graphs works as follows: As for rule application, if a_i is a rule of R_i, it is applied to the ith graph. If a_i is equal to $-$, the ith graph remains unchanged. The new aspect is the third case where a_i is of the form $(u, input, output)$. In this case, the mapping $input : [k_u] \to [n]$ determines which graphs of the current tuple of graphs should be chosen as input for the imported unit u. The output $output \in [l_u]$ specifies which component of the computed output graph tuple of u should be assigned to the ith component of the graph tuple obtained from applying the unit u to the input graphs selected by $input$.

For example the action $(-, -, -, recognize(1,2), recognize(1,3), -)$ of the graph tuple transformation unit *recognize-intersection* has as semantics every pair $((G_1, \ldots, G_6), (H_1, \ldots, H_6))$ such that $G_i = H_i$ for $i \in \{1, 2, 3, 6\}$, H_4 is the output of *recognize* applied to (G_1, G_2), and H_5 is the output of *recognize* applied to (G_1, G_3).

Formally, assume that every imported unit u of *gttu* defines a semantic relation

$$SEM(u) \subseteq (in_u(1) \times \cdots \times in_u(k_u)) \times (out_u(1) \times \cdots \times out_u(l_u)).$$

Then every pair $((G_1, \ldots, G_n), (H_1, \ldots, H_n))$ of graph tuples over $\mathscr{TB}$ is in the *semantics of an action* $a = (a_1, \ldots, a_n)$ of *gttu* if for $i = 1, \ldots, n$:

- $G_i \underset{a_i}{\Longrightarrow} H_i$ if $a_i \in R_i$,
- $G_i = H_i$ if $a_i = -$, and
- $H_i = H'_{output}$ if $a_i = (u, input, output)$ and $((G_{input(1)}, \ldots, G_{input(k_u)}), (H'_1, \ldots, H'_{l_u})) \in SEM(u)$.

The set of all actions of *gttu* is denoted by $ACT(gttu)$ and the semantics of an action $a \in ACT(gttu)$ by $SEM(a)$.

Now we can define the semantics of *gttu* as follows. Every pair $((G_1, \ldots, G_k), (H_1, \ldots, H_l))$ is in $SEM(gttu)$ if there is a pair $(\bar{G}, \bar{H})$ with $\bar{G} = (\bar{G}_1, \ldots, \bar{G}_n)$, $\bar{H} = (\bar{H}_1, \ldots, \bar{H}_n)$ such that the following holds.

- $(G_1, \ldots, G_k) = (\bar{G}_{in(1)}, \ldots, \bar{G}_{in(k)})$,
- $(H_1, \ldots, H_l) = (\bar{H}_{out(1)}, \ldots, \bar{H}_{out(l)})$,
- $(\bar{G}, \bar{H}) \in (SEM(I) \times SEM(T)) \cap SEM(C)$,
- $(\bar{G}, \bar{H}) \in (\bigcup_{a \in ACT(gttu)} SEM(a))^*$.

For example, the semantics of the unit *recognize* consists of all pairs of the form $((G_1, G_2), (H))$ where G_1 is a string graph, G_2 is a state graph of a deterministic finite automaton with its start

state as current state, and $H = TRUE$ if G_1 is recognized by G_2; otherwise $H = FALSE$. The semantics of the unit *recognize-intersection* consists of every pair $((G_1, G_2, G_3), (H))$ where G_1 is a string graph, G_2 and G_3 are state graphs of deterministic finite automata with their respective start state as current state, and $H = TRUE$ if G_1 is recognized by G_2 and G_3; otherwise $H = FALSE$. The semantics of the unit *reduce* contains all pairs $((G_1, G_2), (G_3))$ where G_1 and G_3 are string graphs and G_2 is a state graph of a deterministic finite automaton with its start state as current state such that G_3 represents some suffix of the string represented by G_1 and G_2 recognizes the corresponding "prefix" of G_1. The semantics of *recognize-substitution* contains all pairs $((G_1, G_2, G_3, G_4), (TRUE))$ where G_1 represents a string in the substitution language $subst(L)$, G_2 recognizes the language L, and G_3 and G_4 recognize the languages $subst(a)$ and $subst(b)$, respectively.

4 Conclusion

In this paper, we have introduced the new concept of graph tuple transformation units, which is helpful to specify transformations of combinations of various kinds of graphs simultaneously and in a structured way. To this aim a graph tuple transformation unit contains an import component which consists of a set of other graph tuple transformation units. The semantic relations offered by the imported units are used by the importing unit. The nondeterminism inherent to rule-based graph transformation can be reduced with control conditions and graph tuple class expressions. Graph tuple transformation units generalize transformation units [KK99] in the following aspects. (1) Whereas a transformation unit specifies a binary relation on a single graph type, a graph tuple transformation unit specifies a k,l-relation of graphs of different types. (2) The transformation process in transformation units is basically sequential whereas in graph tuple transformation units the component graphs are transformed in parallel.

Further investigation of graph tuple transformation units may concern the following aspects. (1) We used graph-transformational versions of the truth values, numbers, and strings, but one may like to combine graph types directly with arbitrary abstract data types, i.e., without previously modeling the abstract data types as graphs. (2) In the presented definition, we consider acyclic import structures. Their generalization to networks of graph tuple transformation units with an arbitrary import structure may be an interesting task. (3) In the presented approach the graphs of the tuples do not share any common parts and are not directly interrelated with each other in any other way while the components of the actions can share information and can be interconnected in this way. But it may also be of interest to consider graph tuple transformation where some relations (like morphisms) can be explicitly specified between the different graphs of a tuple. (4) The concepts of pair grammars [Pra71] and triple graph grammars [Sch94] are similarly motivated as graph tuple transformation units so that a close and detailed comparison may be worthwhile. As our notion of control conditions is very general, it is more or less obvious that pair and triple grammars are special cases of graph tuple transformation units. But it will need more considerations to figure out which consequences this observation has. (5) As graph tuple transformation provides a particular form of parallelism by allowing the simultaneous change of components, it may be enlightening to compare and relate it with other graph-transformational approaches to parallelism. (6) Finally, case studies of graph tuple transformation units should

also be worked out that allow to get experience with the usefulness of the concept for the modeling of (data-processing) systems and of systems from other application areas as well as of model transformations, in particular.

Acknowledgement We are grateful to the referees for their valuable remarks.

Bibliography

[BK99] D. Basin, B. Krieg-Brückner. Formalization of the development process. In Astesiano et al. (eds.), *Algebraic Foundations of Systems Specification*. IFIP State-of-the-Art Reports, pp. 521–562. Springer Verlag, 1999.

[BKPT00] P. Bottoni, M. Koch, F. Parisi-Presicce, G. Taentzer. Consistency Checking and Visualization of OCL Constraints. In Evans et al. (eds.), *Proc. UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*. Lecture Notes in Computer Science 1939, pp. 294–308. 2000.

[BMST99] R. Bardohl, M. Minas, A. Schürr, G. Taentzer. Application of Graph Transformation to Visual Languages. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. Pp. 105–180. World Scientific, Singapore, 1999.

[CEH+97] A. Corradini, H. Ehrig, R. Heckel, M. Löwe, U. Montanari, F. Rossi. Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach. Pp. 163–245 in [Roz97].

[DHK97] F. Drewes, A. Habel, H.-J. Kreowski. Hyperedge Replacement Graph Grammars. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. Chapter 2, pp. 95–162. World Scientific, 1997.

[DHK02] R. Depke, R. Heckel, J. M. Küster. Formal Agent-Oriented Modeling with UML and Graph Transformation. *Science of Computer Programming* 44:229–252, 2002.

[EE08] H. Ehrig, C. Ermel. Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation. In Ehrig et al. (eds.), *Proc. 4th International Conference on Graph Transformations (ICGT'08)*. Lecture Notes in Computer Science 5214, pp. 194–210. 2008.

[EEE+07] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer. Information Preserving Bidirectional Model Transformations. In Dwyer and Lopes (eds.), *Proc. 10th International Conference on Fundamental Approaches to Software Engineering (FASE'10)*. Lecture Notes in Computer Science 4422, pp. 72–86. 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer (eds.). *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[EG07] K. Ehrig, H. Giese (eds.). *Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*. Electronic Communications of the EASST 6. http://eceasst.cs.tu-berlin.de/index.php/eceasst/issue/archive, 2007.

[EHHS00] G. Engels, J. H. Hausmann, R. Heckel, S. Sauer. Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In Evans et al. (eds.), *Proc. UML 2000 – The Unified Modeling Language. Advancing the Standard*. Lecture Notes in Computer Science 1939, pp. 323–337. 2000.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. Pp. 247–312 in [Roz97].

[EHK01] G. Engels, R. Heckel, J. Küster. Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In Gogolla and Kobryn (eds.), *Proc. UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Lecture Notes in Computer Science 2185, pp. 272–286. 2001.

[ER97] J. Engelfriet, G. Rozenberg. Node Replacement Graph Grammars. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. Chapter 1, pp. 1–94. World Scientific, 1997.

[FNTZ00] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story Diagrams: A new Graph Transformation Language based on UML and Java. In Ehrig et al. (eds.), *Proc. Theory and Application to Graph Transformations*. Lecture Notes in Computer Science 1764, pp. 296–309. 2000.

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

[GK07] H. Giese, F. Klein. Systematic Verification of Multi-Agent Systems based on Rigorous Executable Specifications. *International Journal on Agent-Oriented Software Engineering (IJAOSE)* 1(1):28–62, 2007.

[HKK08] K. Hölscher, R. Klempien-Hinrichs, P. Knirsch. Undecidable Control Conditions in Graph Transformation Units. *Electronic Notes in Theoretical Computer Science* 195:95–111, 2008.

[HP01] A. Habel, D. Plump. Computational Completeness of Programming Languages Based on Graph Transformation. In Honsell and Miculan (eds.), *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*. Lecture Notes in Computer Science 2030, pp. 230–245. 2001.

[HZG06] K. Hölscher, P. Ziemann, M. Gogolla. On Translating UML Models into Graph Transformation Systems. *Journal of Visual Languages and Computing* 17(1):78–105, 2006.

[Jan99] D. Janssens. Actor Grammars and Local Actions. In Ehrig et al. (eds.). Pp. 57–106. World Scientific, Singapore, 1999.

[KGKZ09] S. Kuske, M. Gogolla, H.-J. Kreowski, P. Ziemann. Towards an integrated graph-based semantics for UML. *Software and Systems Modeling* 8(3):385–401, 2009.

[KK99] H.-J. Kreowski, S. Kuske. Graph Transformation Units with Interleaving Semantics. *Formal Aspects of Computing* 11(6):690–723, 1999.

[KKK04] R. Klempien-Hinrichs, H.-J. Kreowski, S. Kuske. Typing of Graph Transformation Units. In Ehrig et al. (eds.), *Proc. Second International Conference on Graph Transformations (ICGT'04)*. Lecture Notes in Computer Science 3256, pp. 112–127. 2004.

[KKR08] H.-J. Kreowski, S. Kuske, G. Rozenberg. Graph Transformation Units – An Overview. In Degano et al. (eds.), *Concurrency, Graphs and Models*. Lecture Notes in Computer Science 5065, pp. 57–75. 2008.

[KKS97] H.-J. Kreowski, S. Kuske, A. Schürr. Nested graph transformation units. *International Journal on Software Engineering and Knowledge Engineering* 7(4):479–502, 1997.

[KKS07] F. Klar, A. Königs, A. Schürr. Model transformation in the large. In Crnkovic and Bertolino (eds.), *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Pp. 285–294. ACM, 2007.

[Kus00] S. Kuske. More about control conditions for transformation units. In Ehrig et al. (eds.), *Proc. Theory and Application of Graph Transformations*. Lecture Notes in Computer Science 1764, pp. 323–337. 2000.

[Kus01] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In Gogolla and Kobryn (eds.), *Proc. UML 2001 – The Unified Modeling Language. Modeling languages, Concepts, and Tools*. Lecture Notes in Computer Science 2185, pp. 241–256. 2001.

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

[LT04] J. de Lara, G. Taentzer. Automated Model Transformation and Its Validation Using AToM 3 and AGG. In *Diagrams*. Lecture Notes in Computer Science 2980, pp. 182–198. 2004.

[Nag96] M. Nagl (ed.). *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. Lecture Notes in Computer Science 1170. Springer-Verlag, 1996.

[PE93] R. Plasmeijer, M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[Pra71] T. W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and System Sciences* 5:560–595, 1971.

[PS00] D. C. Petriu, Y. Sun. Consistent Behaviour Representation in Activity and Sequence Diagrams. In Evans et al. (eds.), *Proc. UML 2000 – The Unified Modeling Language. Advancing the Standard*. Lecture Notes in Computer Science 1939, pp. 359–368. 2000.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, Singapore, 1997.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Mayr et al. (eds.), *Proc. 20th Int. Worhshop on Graph-Theoretic Concepts in Computer Science*. Lecture Notes in Computer Science 903, pp. 151–163. 1994.

[Sch97] A. Schürr. Programmed Graph Replacement Systems. Pp. 479–546 in [Roz97].

[SK08] A. Schürr, F. Klar. 15 Years of Triple Graph Grammars. In Ehrig et al. (eds.), *Proc. 4th International Conference on Graph Transformations (ICGT'08)*. Lecture Notes in Computer Science 5214, pp. 411–425. 2008.

[SPE93] M. R. Sleep, R. Plasmeijer, M. van Eekelen (eds.). *Term Graph Rewriting. Theory and Practice*. Wiley & Sons, Chichester, 1993.

[TKKT07] I. J. Timm, P. Knirsch, H.-J. Kreowski, A. Timm-Giel. Autonomy in Software Systems. In Hülsmann and Windt (eds.), *Understanding Autonomous Cooperation & Control in Logistics The Impact on Management, Information and Communication and Material Flow*. Pp. 255–273. Springer, Berlin Heidelberg New York, USA, 2007.

[VB07] D. Varró, A. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming* 68(3):214–234, 2007.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM* 14(4):221–227, 1971.