Typing of Graph Transformation Units*

Renate Klempien-Hinrichs, Hans-Jörg Kreowski, and Sabine Kuske

University of Bremen, Department of Computer Science
P.O.Box 33 04 40, 28334 Bremen, Germany
{rena,kreo,kuske}@informatik.uni-bremen.de

Abstract. The concept of graph transformation units in its original sense is a structuring principle for graph transformation systems which allows the interleaving of rule applications with calls of imported units in a controlled way. The semantics of a graph transformation unit is a binary relation on an underlying type of graphs. In order to get a flexible typing mechanism for transformation units and a high degree of parallelism this paper introduces typed graph transformation units that transform k-tuples of typed input graphs into l-tuples of typed output graphs in a controlled and structured way. The transformation of the typed graph tuples is performed with actions that apply graph transformation rules and imported typed units simultaneously to the graphs of a tuple. The transformation process is controlled with control conditions and with graph tuple class expressions. The new concept of typed graph transformation units is illustrated with examples from the area of string parsing with finite automata.

1 Introduction

The area of graph transformation brings together the concepts of rules and graphs with various methods from the theory of formal languages and from the theory of concurrency, and with a spectrum of applications, see the three volumes of the Handbook of Graph Grammars and Computing by Graph Transformation as an overview [15, 5, 7]. The key of rule-based graph transformation is the derivation of graphs from graphs by applications of rules. In this way, a set of rules specifies a binary relation of graphs with the first component as input and the second one as output. If *graph* names the class of graphs $\mathcal{G}$, the type of such a specified relation is $graph \rightarrow graph$ where each graph is a potential input and output. To get a more flexible typing, one can employ graph schemata or graph class expressions X that specify subclasses $\mathcal{G}(X)$ of the given class of graphs. This allows one typings of the form $I \rightarrow T$ restricting the derivations to those that start in initial graphs from $\mathcal{G}(I)$ and end in terminal graphs from $\mathcal{G}(T)$. Alternatively, one may require that all graphs involved in derivations stem from

* Research partially supported by the EC Research Training Network SegraVis (Syntactic and Semantic Integration of Visual Modeling Techniques) and the Collaborative Research Centre 637 (Autonomous Cooperating Logistic Processes: A Paradigm Shift and Its Limitations) funded by the German Research Foundation (DFG).

H. Ehrig et al. (Eds.): ICGT 2004, LNCS 3256, pp. 112–127, 2004.

$\mathcal{G}(X)$ for some expression X (cf. the use of graph schemata in PROGRES [17]). Another form of typing in the area of graph transformation can be found in the notion of pair grammars and triple grammars where a pair resp. a triple of graphs is derived in parallel by applying rules in all components simultaneously (see, e.g., [14, 16]).

In this paper, we propose a new, more general typing concept for graph transformation that offers the parallel processing of arbitrary tuples of graphs. Moreover, some components can be selected as input components and others as output components such that relations of the type $I_1 \times \cdots \times I_k \to T_1 \times \cdots \times T_l$ can be specified. The new typing concept is integrated into the structuring concept of graph transformation units (see, e.g., [1, 11, 12]).

The concept of graph transformation units in its original sense is a structuring principle for graph transformation systems which allows the interleaving of rule applications with calls of imported units in a controlled way. The semantics of a graph transformation unit is a binary relation on an underlying type of graphs that transforms initial graphs into terminal ones. In order to get a flexible typing mechanism for transformation units and a high degree of parallelism this paper introduces typed graph transformation units that transform k-tuples of typed input graphs into l-tuples of typed output graphs in a controlled and structured way. The transformation of the typed graph tuples is performed with actions that apply graph transformation rules and imported typed units simultaneously to the graphs of a tuple. The transformation process is controlled with control conditions and with graph tuple class expressions. The new concept of typed graph transformation units is illustrated with examples from the area of string parsing with finite automata.

2 Typed Graph 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 typed graph 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 base type which consists of a tuple of rule bases. A rule base is composed of graphs, rules, and a rule application operator.

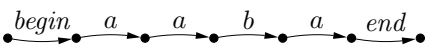
2.1 Rule Bases

A rule base $B = (\mathcal{G}, \mathcal{R}, \Longrightarrow)$ consists of a type of graphs $\mathcal{G}$, a type of rules $\mathcal{R}$, and a rule application operator $\Longrightarrow$. In the following the components $\mathcal{G}$, $\mathcal{R}$, and $\Longrightarrow$ of a rule base B are also denoted by $\mathcal{G}_B$, $\mathcal{R}_B$, and $\Longrightarrow_B$, respectively.

Examples for graph types are labelled directed graphs, graphs with a structured labelling (e.g. *typed graphs* in the sense of [3]), hypergraphs, trees, forests, 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.

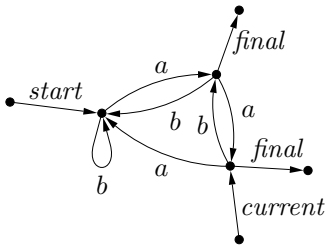
In this paper, we explicitly consider directed, edge-labelled graphs with individual, possibly multiple edges. A *graph* is a construct $G = (V, E, s, t, l)$ where V is a set of *vertices*, E is a set of *edges*, $s, t\colon E \to V$ are two mappings assigning to each edge $e \in E$ a *source* $s(e)$ and a target $t(e)$, and $l\colon E \to \Sigma$ is a mapping labelling each edge in a given label alphabet Σ.

For instance the graph



consists of seven nodes and six directed edges. It is a string graph which represents the string *aaba*. The beginning of the string is indicated with the *begin*-edge pointing to the source of the leftmost *a*-edge. Analogously, there is an *end*-edge originating from the end of the string, i.e. from the target of the rightmost *a*-edge.

Another instance of a graph is the following deterministic finite state graph where the edges labelled with *a* and *b* represent transitions, and the sources and targets of the transitions represent states. The start state is indicated with a *start*-edge and every final state with a *final*-edge. Moreover there is an edge labelled with *current* pointing to the current state of the deterministic finite state graph.



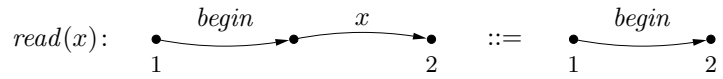
To be able to transform the graphs in $\mathcal{G}$, rules are applied to the graphs yielding graphs again. Hence, each rule $r \in \mathcal{R}$ defines a binary relation $\underset{r}{\Longrightarrow} \subseteq \mathcal{G} \times \mathcal{G}$ on graphs. If $G \underset{r}{\Longrightarrow} H$, one says that G *directly derives* H by applying r. There are many possibilities to choose rules and their applications. Types of rules may vary from the more restrictive ones, like edge replacement [4] or node replacement [8], to the more general ones, like double-pushout rules [2], single-pushout rules [6], or PROGRES rules [17].

In this paper, we concentrate on a simplified notion of double-pushout rules, i.e. every rule 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 a graphical representation of r, 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. In particular, we will use rules that add or delete a node together with an edge and/or that redirect an edge.

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.

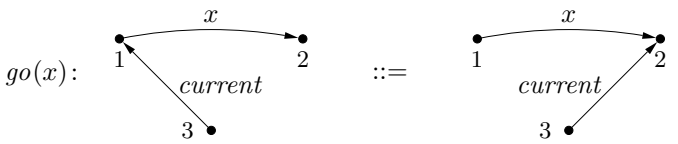
1. Find an isomorphic copy of L in G, i.e. a subgraph that coincides with L up to the naming of nodes and edges.
2. Remove all nodes and edges of this copy except the nodes corresponding to K, provided that the remainder is a graph (which holds if the removal of a node is accompanied by the removal of all its incident edges).
3. Add R by merging K with its corresponding copy.

For abbreviating sets of rules, we use also variables instead of concrete labels. For every instantiation of a variable with a label we get a rule as described above. For example, the following rule $read(x)$ has as left-hand side a graph consisting of an x-edge and a *begin*-edge. The right-hand side consists of the target of a new *begin*-edge pointing from the source of the old *begin*-edge to the target of the x-edge. The common part of the rule $read(x)$ consists of the source of the *begin*-edge and the target of the x-edge.



If the variable x is instantiated with a, the resulting rule $read(a)$ can be applied to the above string graph. Its application deletes the *begin*-edge and the leftmost a-edge together with its source. It adds a new *begin*-edge pointing from the source of the old *begin*-edge to the target of the a-edge. The resulting string graph represents the string aba.

The following rule $go(x)$ redirects a *current*-labelled edge from the source of some x-labelled edge to the target of this edge.



If x is instantiated with a, its application to the above deterministic finite state graph results in the same deterministic finite state graph except that the current state is changed to the start state.

2.2 Graph Tuple Transformation

As the iterated application of rules transforms graphs into graphs yielding an input-output relation, the natural type declaration of a graph transformation in a rule base $B = (\mathcal{G}, \mathcal{R}, \Longrightarrow)$ is $B\colon \mathcal{G} \to \mathcal{G}$. But in many applications one would like to have a typing that allows one to consider several inputs and maybe even several outputs, or at least an output of a type different from all inputs. Moreover, one may want to be able to transform subtypes of the types of input and output graphs. In order to reach such an extra flexibility in the typing of graph transformations we introduce in this section the transformation of tuples of typed graphs, which is the most basic operation of the typed graph transformation units presented in Section 4.

Graph tuple transformation over a base type is an extension of ordinary rule application in the sense that graphs of different types 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 deterministic finite state graph the current state of which is the start state, and the third graph represents the boolean value *false*. To recognize the string one applies a sequence of typed 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 modelled in Section 4.

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 *base type*, which is a tuple of rule bases $BT = (B_1, \ldots, B_n)$. Let $(G_1, \ldots, G_n)$ and $(H_1, \ldots, H_n)$ be graph tuples over BT, i.e. $G_i, H_i \in \mathcal{G}_{B_i}$ for $i = 1, \ldots, n$. Let $a = (a_1, \ldots, a_n)$ with $a_i \in \mathcal{R}_{B_i}$ or $a_i = -$ for $i = 1, \ldots, n$. Then $(G_1, \ldots, G_n) \underset{a}{\longrightarrow} (H_1, \ldots, H_n)$ if for $i = 1, \ldots, n$, $G_i \underset{a_i}{\Longrightarrow} H_i$ if $a_i \in \mathcal{R}_{B_i}$ and $G_i = H_i$ if $a_i = -$. In the following we call a a *basic action of* BT.[1] For a set ACT of basic actions of BT, $\underset{ACT}{\longrightarrow}$ denotes the union $\bigcup_{a \in ACT} \underset{a}{\longrightarrow}$, and $\overset{*}{\underset{ACT}{\longrightarrow}}$ its reflexive and transitive closure.

For example, let I be some finite alphabet and let $B^{basic}_{string} = (string, \{read(x) \mid x \in I\}, \Longrightarrow)$ and $B^{basic}_{dfsg} = (dfsg, \{go(x) \mid x \in I\}, \Longrightarrow)$ be two rule bases such that *string* consists of all string graphs over I and *dfsg* consists of all deterministic finite state graphs over I. Let G_1 be the string graph representing *aaba* and let G_2 be the above deterministic finite state graph. Then $(G_1, G_2) \underset{a}{\longrightarrow} (H_1, H_2)$ for the basic action $a = (read(a), go(a))$ of base type $(B^{basic}_{string}, B^{basic}_{dfsg})$ if H_1 represents *aba* and H_2 is obtained from G_2 by redirecting the *current*-edge to the start state.

Let ACT be the set of all basic actions of $BT = (B_1, \ldots, B_n)$. Then obviously the following holds: $(G_1, \ldots, G_n) \overset{*}{\underset{ACT}{\longrightarrow}} (H_1, \ldots, H_n)$ if and only if $G_i \overset{*}{\underset{\mathcal{R}_{B_i}}{\Longrightarrow}} H_i$ for $i = 1, \ldots, n$. This means that the transformation of graph tuples via a sequence of basic actions is equivalent to the transformation of tuples of typed graphs where every component is transformed independently with a sequence of direct derivations of the corresponding type.

In [9] the transformation of tuples of typed graphs is generalized in the sense that the transformations are performed by a product of transformation units instead of tuples of rules. In every transformation step of a product of transformation units $tu_1, \ldots, tu_n$, one can apply rules as well as imported transformation units of tu_i to the ith graph in the current graph tuple $(i = 1, \ldots, n)$. Hence, in

[1] More sophisticated actions with more expressive power will be introduced further on.

every transformation step the graphs of the current graph tuple are transformed in parallel. Such a transformation step in a product unit is called an action. The nondeterminism of actions is restricted by the control conditions and the graph class expressions of the units $tu_1, \ldots, tu_n$. Moreover one can specify control conditions on the level of actions. In order to get a flexible kind of typing, i.e. to declare a sequence of input components and a sequence of output components independently, the embedding and projection of graph products is introduced. For the same reasons, similar operations will be introduced for typed transformation units in this paper. The most striking difference of the product of transformation units in [9] and the typed graph transformation units presented here is the import component. Typed units can import other typed units whereas a product of transformation units is composed of transformation units in the original sense, i.e. it does not use other typed units.

3 Restricting the Nondeterminism

Application of rule tuples is highly nondeterministic in general. For many applications of graph transformation it is meaningful to restrict the number of possible ways to proceed with a transformation process. Hence, in order to employ typed transformation units meaningfully, they are equipped with graph tuple class expressions and control conditions to restrict the number of possible sequences of transformation steps.

3.1 Graph Tuple 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, or the occurrence or absence of certain labels. In this sense, every graph tuple class expression e over a base type $BT = (B_1, \ldots, B_n)$ specifies a set $SEM(e) \subseteq \mathcal{G}_{B_1} \times \cdots \times \mathcal{G}_{B_n}$ of graph tuples in BT.

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 $\mathcal{G}_{B_i}$ of the rule base B_i, i.e. $SEM_{B_i}(e_i) \subseteq \mathcal{G}_{B_i}$ for $i = 1, \ldots, n$. Consequently, the semantics of e is $SEM_{B_1}(e_1) \times \cdots \times SEM_{B_n}(e_n)$. Hence, each item e_i is a graph class expression as defined for transformation units without explicit typing.

The graph tuple class expressions used in this paper are also tuples of graph class expressions. A simple example of a graph class expression is *all* which specifies for any rule base B the graph type of B, i.e. $SEM_B(all) = \mathcal{G}_B$. Consequently, the graph tuple class expression $(e_1, \ldots, e_n)$ with $e_i = all$ for $i = 1, \ldots, n$ does not restrict the graph types of the rule bases, i.e. $SEM((e_1, \ldots, e_n)) = \mathcal{G}_{B_1} \times \cdots \times \mathcal{G}_{B_n}$. Another example of a graph class expression over a rule base B is a set of graphs in $\mathcal{G}_B$. The semantics of a set $e \subseteq \mathcal{G}_B$ is e itself. In particular we will use the set *initialized* consisting of all deterministic finite state graphs the current state of which is the start state.

3.2 Control Conditions

A control condition 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 base type BT specifies a binary relation $SEM(C)$ on the set of graph tuples in BT. More precisely, for a base type $BT = (B_1, \ldots, B_n)$ $SEM(C)$ is a subset of $(\mathcal{G}_{B_1} \times \cdots \times \mathcal{G}_{B_n})^2$.

As control condition we use in particular actions, 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 typed 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 the previous section are examples of actions. Roughly speaking, an action over a base type $BT = (B_1, \ldots, B_n)$ is a tuple $act = (a_1, \ldots, a_n)$ that specifies an n, n-relation $SEM(act) \subseteq (\mathcal{G}_{B_1} \times \cdots \times \mathcal{G}_{B_n})^2$. Actions will be explained in detail in Section 4.

In particular, an action act is a control condition that specifies the relation $SEM(act)$. For control conditions C, C_1, and C_2 the expression $C_1; C_2$ specifies the sequential composition of both sematic 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 n, n-relations on graphs of different types. 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.

4 Typed Graph Transformation Units

Typed transformation units provide a means to structure the transformation process from a sequence of typed input graphs to a sequence of typed output graphs. More precisely, a typed graph 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 typed transformation unit specifies a k, l-relation on typed graphs. Internally a typed transformation unit transforms n-tuples of typed graphs into n-tuples of typed graphs, i.e. it specifies internally an n, n-relation on typed graphs. The transformation of the n-tuples is performed according to a base type 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 typed unit.

4.1 Syntax of Typed Graph Transformation Units

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

The basic operation of a typed 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 typed 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 typed 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 typed graphs with action sequences has to take into account the control condition of the typed unit as well as the initial and terminal graph tuple class expressions.

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

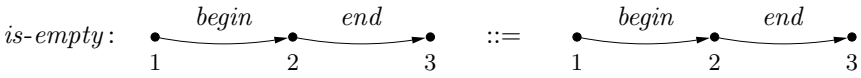
Formally, let $BT = (B_1, \ldots, B_n)$ be a base type. A *typed graph transformation unit tgtu with base type BT* is a pair $(decl, body)$ where *decl* is the *declaration part* of *tgtu* and *body* is the *body* of *tgtu*. The declaration part is of the form $in \to out$ *on* BT where $in\colon [k] \to [n]$ and $out\colon [l] \to [n]$ are mappings with $k, l \in \mathbb{N}$.[2] The body of *tgtu* is a system $body = (I, U, R, C, T)$ where I and T are graph tuple class expressions over BT, U is a set of imported typed graph transformation units, R is a tuple of rule sets $(R_1, \ldots, R_n)$ such that $R_i \subseteq \mathcal{R}_{B_i}$ for $i = 1, \ldots, n$, and C is a control condition over BT. The numbers k and l of *tgtu* are also denoted by k_{tgtu} and l_{tgtu}. Moreover, the ith input type $\mathcal{G}_{B_{in(i)}}$ of *tgtu* is also denoted by $intype_{tgtu}(i)$ for $i = 1, \ldots, k$ and the jth output type $\mathcal{G}_{B_{out(j)}}$ by $outtype_{tgtu}(j)$ for $j = 1, \ldots, l$.

To simplify technicalities, we assume in this first approach that the import structure is acyclic (for a study of cyclic imports of transformation units with a single input and output type see [13]). Initially, one builds typed units of level 0 with empty import. Then typed units of level 1 are those that import only typed units of level 0, and typed units of level $n+1$ import only typed units of level 0 to level n, but at least one from level n.

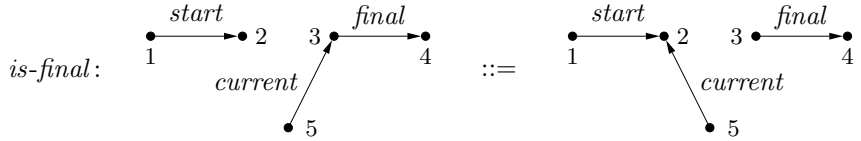
[2] For a natural number $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$.

4.2 Examples for Typed Graph Transformation Units

Example 1. The base type of the following example of a typed transformation unit is the tuple $(B_{string}, B_{dfsg}, B_{bool})$. The rule base 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 to which a *begin*- and from which an *end*-edge are pointing.

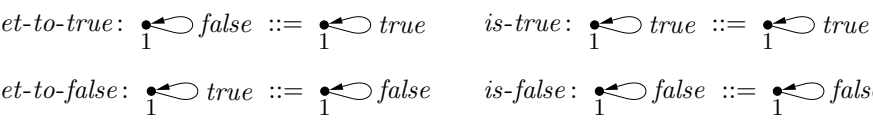


The rule base B_{dfsg} is $(dfsg, \{go(x) \mid x \in I\} \cup \{is\text{-}final\}, \Longrightarrow)$. The rule *is-final* checks whether the current state of a deterministic finite state graph is a final state, resetting it to the start state in that case, and can be depicted as follows.



The rule base B_{bool} contains the graph type *bool* which consists of the two graphs *TRUE* and *FALSE*, where *TRUE* represents the value *true* and *FALSE* the value *false*. Both graphs consist of a single node with a loop that is labelled *true* and *false*, respectively:

The rule type of B_{bool} consists of the four rules



where *set-to-true* changes a *false*-loop into a *true*-loop, *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 typed unit *recognize* shown in Figure 1. It has as input graphs a string graph and a deterministic finite state graph 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, $intype_{recognize}(1) = string$, $intype_{recognize}(2) = dfsg$, and $outtype_{recognize}(1) = bool$.

The initial graph tuple class expression is $(string, initialized, FALSE)$, i.e. it admits all tuples $(G_1, G_2, G_3) \in string \times dfsg \times bool$ where the *current*-edge of G_2

recognize	
decl:	$(string, dfsg, -) \rightarrow (-, -, bool)$ *on* $(B_{string}, B_{dfsg}, B_{bool})$
initial:	$(string, initialized, FALSE)$
rules:	$(\mathcal{R}_{B_{string}}, \mathcal{R}_{B_{dfsg}}, \{set\text{-}to\text{-}true\})$
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:	$(string, dfsg, bool)$

Fig. 1. A typed unit with empty import.

points to the start state and G_3 is equal to *FALSE*. The rules are restricted to the tuple $(\mathcal{R}_{B_{string}}, \mathcal{R}_{B_{dfsg}}, \{set\text{-}to\text{-}true\})$, i.e. just one rule from 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*. 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*. The terminal graph tuple class expression does not restrict the graph types of the base type, i.e. it is equal to $(string, dfsg, bool)$. The unit *recognize* does not import other typed units.

Example 2. The unit *recognize-intersection* shown in Figure 2 is an example of a typed unit with a non-empty import component. It has as input graphs a string graph and two deterministic finite state graphs. The output graph represents again a boolean value. The base type of *recognize-intersection* is the six-tuple $(B_{string}, B_{dfsg}, B_{dfsg}, B_{bool}, B_{bool}, B_{bool})$. The mapping *in* of the declaration part requires to take a string graph from the first rule base of the base type, one deterministic finite 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.

recognize-intersection	
decl:	$(string, dfsg, dfsg, -, -, -) \rightarrow (-, -, -, -, -, bool)$ *on* $(B_{string}, B_{dfsg}, B_{dfsg}, B_{bool}, B_{bool}, B_{bool})$
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:	$(string, dfsg, dfsg, bool, bool, bool)$

Fig. 2. A typed unit with imported units combined in an action.

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 deterministic finite state graphs. In the second point *recognize* must be applied to the input string graph and to the second deterministic finite state graph. These two transformations can be performed in parallel within one and the same action denoted by the tuple $(-,-,-,\mathit{recognize}(1,2),\mathit{recognize}(1,3),-)$. (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 $(-,-,-,\mathit{is\text{-}true},\mathit{is\text{-}true},\mathit{set\text{-}to\text{-}true})$ as long as possible. Since the initial graph tuple class expression requires that the sixth graph represent *false*, this means at most one application due to *set-to-true*. The terminal graph tuple class expression admits all graph tuples of the base type.

Example 3. Let I be the alphabet consisting of the symbols a, b, let L, L_a, L_b be regular languages, and let $\mathit{subst}\colon I \to \mathcal{P}(I^*)$ be a substitution with $\mathit{subst}(a) = L_a$ and $\mathit{subst}(b) = L_b$. The aim of the following example is to model the recognition of the substitution language $\mathit{subst}(L) = \{\mathit{subst}(w) \mid w \in L\}$ 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 typed unit *reduce* shown in Figure 3. It takes a string graph and a deterministic finite state graph 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 $(\mathit{read}(x), \mathit{go}(x))$, i.e. by consuming an arbitrarily large prefix of the string and changing states

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

Fig. 3. A typed unit that returns a modified input graph as output.

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 $(-, \textit{is-final})$ is applied exactly once.

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

Fig. 4. A typed unit with imported units combined in an action.

The typed unit *recognize-substitution* shown in Figure 4 makes use of *reduce* in order to decide whether an input string graph is in the substitution language given as further input by three deterministic finite state graphs 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 $\textit{subst}(w)$. If the next symbol is guessed to be a, the action $(\textit{reduce}(1,3), \textit{go}(a), -, -, -)$ is applied that runs A_a to delete a prefix belonging to L_a from the input string $(\textit{reduce}(1,3))$ and simultaneously executes the next state transition for a in A $(\textit{go}(a))$. The action $(\textit{reduce}(1,4), \textit{go}(b), -, -, -)$ works analogously for the symbol b. Thus, *recognize-substitution* is an example of a typed unit that combines an imported unit (*reduce*) and a rule $(\textit{go}(x))$ in an action. Finally, a mandatory application of the action $(\textit{is-empty}, \textit{is-final}, -, -, \textit{set-to-true})$ 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.

4.3 Semantics of Typed Graph Transformation Units

Typed 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 typed transformation unit can be defined as a k, l-relation between input and output graphs.

Let $tgtu = (in \to out \text{ on } BT, (I, U, R, C, T))$ be a typed transformation unit with $BT = (B_1, \ldots, B_n)$, $in\colon [k] \to [n]$, $out\colon [l] \to [n]$, and $R = (R_1, \ldots, R_n)$. If $U = \emptyset$, $tgtu$ transforms internally a tuple $G \in \mathcal{G}_{B_1} \times \cdots \times \mathcal{G}_{B_n}$ into a tuple $H \in \mathcal{G}_{B_1} \times \cdots \times \mathcal{G}_{B_n}$ 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. $G \xrightarrow[ACT(tgtu)]{*} H$ where $ACT(tgtu)$ is the set of all basic actions $a = (a_1, \ldots, a_n)$ of BT such that for $j = 1, \ldots, n$, $a_i \in R_i$ 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 transformation unit $tgtu$ 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 typed units are allowed, leading to the notion of (general) actions.

Formally, an *action of* $tgtu$ 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 $u \in U$, $input\colon [k_u] \to [n]$ with $\mathcal{G}_{B_{input(j)}} \subseteq intype_u(j)$ for $j = 1, \ldots, k_u$, and $output \in [l_u]$ with $outtype_u(output) \subseteq \mathcal{G}_{B_i}$. 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 typed graphs works as follows: As for typed 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 $input\colon [k_u] \to [n]$ determines which graphs of the current tuple of typed 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 typed unit u to the input graphs selected by $input$.

For example the action $(-, -, -, recognize(1, 2), recognize(1, 3), -)$ of the typed 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 typed unit u of $tgtu$ defines a semantic relation

$$SEM(u) \subseteq (intype_u(1) \times \cdots \times intype_u(k_u)) \times (outtype_u(1) \times \cdots \times outtype_u(l_u)).$$

Then every pair $((G_1, \ldots, G_n), (H_1, \ldots, H_n))$ of graph tuples over BT is in the *semantics of an action* $a = (a_1, \ldots, a_n)$ of $tgtu$ 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 *tgtu* is denoted by $ACT(tgtu)$ and the semantics of an action $a \in ACT(tgtu)$ by $SEM(a)$.

Now we can define the semantics of *tgtu* as follows. Every pair $((G_1, \ldots, G_k), (H_1, \ldots, H_l))$ is in $SEM(tgtu)$ 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.

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

For example, the semantics of the typed 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 deterministic finite state graph 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 typed 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 deterministic finite state graphs 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 typed 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 deterministic finite state graph 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.

5 Conclusion

In this paper, we have introduced the new concept of typed graph transformation units, which is helpful to specify structured and parallel graph transformations with a flexible typing. To this aim a typed transformation unit contains an import component which consists of a set of other typed transformation units. The semantic relations offered by the imported typed 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.

Typed transformation units are a generalization of transformation units [10] in the following aspects. (1) Whereas a transformation unit specifies a binary relation on a single graph type, a typed transformation unit specifies a k, l-relation of graphs of different types. (2) The transformation process in transformation units is basically sequential whereas in typed transformation units typed graphs are transformed simultaneously. Moreover, as described in Section 2.2 typed transformation units generalize the concept of product units [9] that also specify k, l-relations of typed graphs. With product units, however, the possibilities

of structuring (and modelling) are more restrictive in the sense that only rules and transformation units can be applied to graph tuples but no imported typed transformation unit.

Further investigation of typed transformation units may concern the following aspects. (1) We used graph-transformational versions of the truth values, but one may like to combine graph types directly with arbitrary abstract data types, i.e. without previously modelling the abstract data types as graphs. (2) In the presented definition, we consider acyclic import structures. Their generalization to networks of typed transformation units with an arbitrary import structure is an interesting task. (3) In the presented approach the graphs of the tuples do not share common parts. Hence, one could consider graph tuple transformation where some relations (like morphisms) can be explicitly specified between the different graphs of a tuple. (4) Apart from generalizing the concept of typed transformation units, a comparison with similar concepts such as pair grammars [14] and triple grammars [16] is needed. (5) Finally, case studies of typed units should also be worked out that allow to get experience with the usefulness of the concept for the modelling of (data-processing) systems and systems from other application areas.

Acknowledgement

We are grateful to the referees for their valuable remarks.

References

1. Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, 1999.
2. Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Michael Löwe, Ugo Montanari, and Francesca Rossi. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In Rozenberg [15], pages 163–245.
3. Andrea Corradini, Ugo Montanari, and Francesca Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–265, 1996.
4. Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In Rozenberg [15], pages 95–162.
5. Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools.* World Scientific, Singapore, 1999.
6. Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In Rozenberg [15], pages 247–312.
7. Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution.* World Scientific, Singapore, 1999.

8. Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In Rozenberg [15], pages 1–94.
9. Renate Klempien-Hinrichs, Hans-Jörg Kreowski, and Sabine Kuske. Rule-based transformation of graphs and the product type. In Patrick van Bommel, editor, *Handbook on Transformation of Knowledge, Information, and Data*. To appear.
10. Hans-Jörg Kreowski and Sabine Kuske. On the interleaving semantics of transformation units — a step into GRACE. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 89–108, 1996.
11. Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units and modules. In Ehrig, Engels, Kreowski, and Rozenberg [5], pages 607–638.
12. Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.
13. 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.
14. Terrence W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.
15. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, Singapore, 1997.
16. Andy Schürr. Specification of graph translators with triple graph grammars. In G. Tinnhofer, editor, *Proc. WG'94 20th Int. Worhshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163, 1994.
17. Andy Schürr. Programmed graph replacement systems. In Rozenberg [15], pages 479–546.