Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 195 (2008) 95–111
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Undecidable Control Conditions in Graph Transformation Units

Karsten Hölscher[1,2]

Department of Mathematics and Computer Science
University of Bremen
Bremen, Germany

Renate Klempien-Hinrichs[3]

Department of Production Engineering
University of Bremen
Bremen, Germany

Peter Knirsch[4]

Department of Mathematics and Computer Science
University of Bremen
Bremen, Germany

Abstract

Graph transformation units are an approach-independent concept for programming by applying rules and imported transformation units to graphs, starting in an initial and ending in a terminal graph. This transformation process has to obey a so-called control condition, i.e. the device to select how rules or imported transformation units are to be combined in the transformation process executed by the unit. While the other parts of a unit may simply be required to be computable, this is too restrictive for control conditions. In this paper, we show that the semantics of certain control conditions is in general undecidable already when a single imported transformation unit occurs in the condition, and discuss the consequences for programming with graph transformation units.

Keywords: Graph transformation, transformation unit, control condition, decidability, computability

[1] This research was supported by the German Research Foundation (DFG) as part of the Collaborative Research Centre 637 "Autonomous Cooperating Logistic Processes" and the EC Research Training Network SegraVis (Syntactic and Semantic Integration of Visual Modelling Techniques).

[2] Email: hoelscher@informatik.uni-bremen.de

[3] Email: kh@biba.uni-bremen.de

[4] Email: knirsch@informatik.uni-bremen.de

doi:10.1016/j.entcs.2007.08.028

1 Introduction

Since diagrams and their manipulation have become increasingly popular in different domains, the field of graph transformation has become more and more important in the last years. Many applications for graph transformation can be found, with UML, MDA, software refactoring, and logistic processes being the most recent. Various graph transformation approaches, which differ in particular in the kind of graphs considered or the way in which graphs may be transformed, are proposed in the literature. For an overview, see the three-volume *Handbook of Graph Grammars and Computing by Graph Transformation* [14,5,6].

Graph transformation units were introduced in [10,9,12] as an approach-independent concept for structured computation with graph transformation and for rule-based systems in general. On this conceptual level, they are more general than other programming environments for graph transformation, such as, for instance, PROGRES [15] or AGG [16]. Consequently, one may write a graph transformation unit over any graph transformation approach, where the approach determines the class of graphs to be worked on, graph transformation rules, and the application of the rules to the graphs.

A graph transformation unit consists of a class of graphs to be used initially in a graph transformation sequence, a class of graphs with which the sequence may terminate, a set of rules and a set of graph transformation units that may be used by that unit in its transformation process, and a control condition that regulates how rules and used units may transform an initial into a terminal graph. The semantics of a graph transformation unit consists of all pairs of graphs where the first is admitted as initial and can be transformed into the second, which in turn is admitted as terminal, by the rules and the used units in a sequential way that is admitted by a control condition.

When trying to compute the semantics of a transformation unit, the termination of the derivation process is essential. In [13] it is shown that it is generally undecidable to determine whether graph rewriting terminates. In the meantime several methods of restricting graph transformation systems in order to tackle the termination problem have been investigated. Concrete termination criteria based on the number of nodes and edges have been presented in [1]. A general approach based on measurement functions can be found in [2]. This work is extended and formalized in [4]. The termination criteria proposed in the latter two have been implemented as termination checks in the AGG system.

In order to implement the approach-independent concept of graph transformation units so that they may actually be employed for graph transformation programming, one has to think about the decidability resp. computability of the various components of a unit. Obviously, it must be decidable whether a graph is initial or terminal, and whether a rule is applicable to a graph. Moreover, the result of a rule application must be computable. Requiring full decidability for control conditions would, however, be too restrictive since it excludes arbitrary iteration, which is necessary to obtain computational completeness. In this paper, we study the three

most widely used types of control conditions, namely regular expressions, *as-long-as-possible*, and priorities over rules and imported transformation units. It turns out that for a computationally complete graph transformation approach, all of them are undecidable in general. But while – as one would expect – regular expressions are still mostly semi-decidable, this is not true for *as-long-as-possible* and priorities. Roughly speaking, the reason for this is that the latter types check conditions that may be semi- or undecidable, so that the whole becomes undecidable.

The paper is organised as follows. In Section 2, a variant of the so-called double-pushout approach is defined and two classical decidability problems, namely Post's Correspondence Problem and the membership problem for Turing machines, are modelled with graph transformation units over this approach. The general definitions of a graph transformation approach, transformation units and their semantics are given in Section 3. In Section 4, the decidablity of the three types of control conditions is studied, and Section 5 contains concluding remarks.

2 Modelling Decision Problems with Graph Transformation

In this section, we introduce an example for a graph transformation approach and use graph transformation units to model two undecidable problems.

2.1 An Approach to Graph Transformation

The variant of the well-known double-pushout approach (see, e.g., [3]) that we will use in this paper is based on edge-labelled graphs and double-pushout graph transformation with injective matches.

Graphs.

A *graph* over a finite set Σ of labels is a construct $G = (V, E, s, t, l)$ consisting of a finite set V of nodes, a finite set E of edges, mappings $s, t\colon E \to V$ that assign a source and a target node, respectively, to each edge, and a mapping $l\colon E \to \Sigma$ assigning a label to each edge. We will refer to the components of G by using G as index, e.g. V_G, E_G, s_G, t_G, l_G. In pictures, a node will be drawn as a bullet, an edge as an arrow from its source to its target node, and the label of an edge will be written next to that edge. A sample graph is shown in Figure 1.

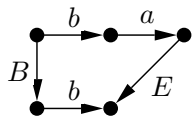

Fig. 1. A graph

A special kind of graph is a *string graph* that consists, for a string w, e.g. of nodes $0, \ldots, |w|$ and edges $1, \ldots, |w|$ with edge j having source $j-1$, target j, and the jth symbol of w as label. A sample string graph for the string $w = abbab$ is shown in Figure 2.

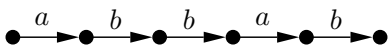

Fig. 2. A string graph

Given two graphs G, H, a *graph morphism* $f\colon G \to H$ is a pair of mappings $f = (f_V, f_E)$ with $f_V\colon V_G \to V_H$ and $f_E\colon E_G \to E_H$ such that for all edges $e \in E_G$, we have $s_H(f_E(e)) = f_V(s_G(e))$, $t_H(f_E(e)) = f_V(t_G(e))$, and $l_H(f_E(e)) = l_G(e)$. The graph morphism is *injective* if both f_V and f_E are. For notational convenience, we will also write f for f_V or f_E.

Rules.

A *rule* $r = (L, K, R)$ consists of two graphs L and R, called the left- and right-hand sides of r, respectively, and a set K of *gluing* nodes that belong to both L and R. In drawings of a rule we will also write $L ::= R$, where L and R will be shown explicitly and K is represented by small numbers close to the respective nodes in L and R. For instance, the rule shown in Figure 3 has two gluing nodes.

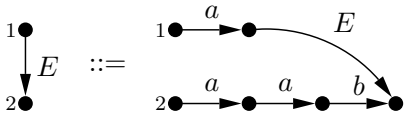

Fig. 3. A rule

Rule application.

A rule $r = (L, K, R)$ can be *applied* to a graph G if there is an injective graph morphism $g\colon L \to G$ such that for every node $g(v)$ in G that is linked by an edge to a node not in the image of g, v belongs to K. Then the application of r to G deletes all (images of) items of L with the exception of the gluing nodes, and inserts disjointly all items of R that are not gluing nodes, where the source (resp. target) of a newly inserted edge is the one of R if that is not a gluing node, and the image of the gluing node under g otherwise. The derived graph $H = (V, E, s, t, l)$ is formally defined as follows (we assume G and R to be disjoint here):

- $V = (V_G \setminus g(V_L \setminus V_K)) \cup (V_R \setminus V_K)$,
- $E = (E_G \setminus g(E_L)) \cup E_R$,
- for all $e \in E$, $s(e) = \begin{cases} s_R(e) & \text{if } e \in E_R \text{ and } s_R(e) \in V_R \setminus V_K, \\ g(s_R(e)) & \text{if } e \in E_R \text{ and } s_R(e) \in V_K, \\ s_G(e) & \text{otherwise} \end{cases}$

  and analogously for $t(e)$, and
- for all $e \in E$, $l(e) = \begin{cases} l_R(e) & \text{if } e \in E_R, \\ l_G(e) & \text{otherwise.} \end{cases}$

For instance, the rule shown in Figure 3 can be applied to the graph of Figure 1. It replaces the E-labelled edge with a larger graph, resulting in the graph shown in Figure 4 (where the images of the gluing nodes are drawn in grey for easier orientation).

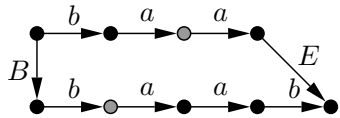

Fig. 4. The graph resulting from the application of the rule in Figure 3 to the graph in Figure 1

If r can be applied to G yielding H, we also write $G \underset{r}{\Longrightarrow} H$ and call this a *direct derivation* or a *derivation step*. The index r may be omitted if clear from the context or not needed, and one may also use R in the index if $r \in R$. The reflexive and transitive closure of $\underset{R}{\Longrightarrow}$ is denoted by $\overset{*}{\underset{R}{\Longrightarrow}}$, denoting derivations of arbitrary length.

2.2 Modelling Post's Correspondence Problem

An instance of *Post's Correspondence Problem* (PCP), see, e.g., [8], consists of two lists $U = u_1, \ldots, u_k$ and $V = v_1, \ldots, v_k$ of strings over some alphabet Δ, with the two lists being of equal length k. For each i, the pair (u_i, v_i) is a *corresponding* pair. This instance of PCP *has a solution* if there is a nonempty sequence $i_1 \cdots i_n \in \{1, \ldots, k\}$ such that $u_{i_1} \cdots u_{i_n} = v_{i_1} \cdots v_{i_n}$.

We will model the search for a solution of such a given instance of PCP by successively adding corresponding pairs to a prefix graph. Such a prefix graph consists of two disjoint string graphs. Moreover, the prefix graph contains a B-labelled edge from the first node of the first string graph to the first node to the second string graph, and an E-labelled edge connecting analogously the respective last nodes of the string graphs. If we have for instance $u_{i_1} = ba$, $u_{i_2} = a$, $v_{i_1} = b$, $v_{i_2} = aab$, then the graphs in Figures 1 and 4 are prefix graphs $G(i_1)$ and $G(i_1 i_2)$, respectively, and the rule for corresponding pair i_2 is the one of Figure 3.

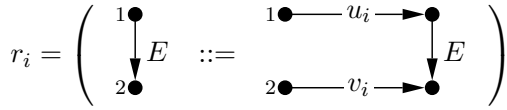

Fig. 5. The PCP rule r_i

In general, each corresponding pair (u_i, v_i) gives rise to a rule r_i as sketched in Figure 5. Here, an "edge" with a whole string w on it refers to the respective string graph. Starting with the prefix graph $G()$ for the empty prefix, which consists of two nodes and two parallel edges between them with labels B and E, respectively, the derivation

$$G() \overset{*}{\Longrightarrow} G(i_1 \cdots i_m) \underset{r_{i_{m+1}}}{\Longrightarrow} G(i_1 \cdots i_{m+1})$$

is sketched in Figure 6.

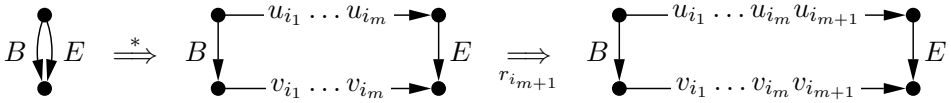

Fig. 6. Deriving prefix graphs for a PCP instance

Of course, this process can produce any index sequence, so that it must be combined with verifying the correctness of a solution. This will be done using deconstructing rules r_x, one for each $x \in \Delta$, as shown in Figure 7.

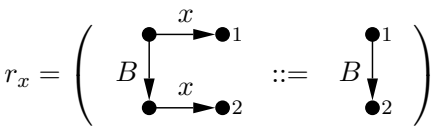

Fig. 7. The PCP rule r_x for $x \in \Delta$

Now these rules have to be combined in a useful way. This will be done in a so-called *graph transformation unit*, as shown in Figure 8. Informally, the meaning of

PCP

initial	$G()$
rules	r_i for $i = 1, \dots, k$
	r_{x_j} for $\Delta = \{x_1, \dots, x_l\}$ and $j = 1, \dots, l$
conds	$(r_1 \mid \cdots \mid r_k \mid r_{x_1} \mid \cdots \mid r_{x_l})^+$
terminal	$G()$

Fig. 8. The graph transformation unit for a PCP instance

this unit may be described as follows: Any derivation starts in an *initial* graph, here always $G()$. The given *rules* may then be applied according to the *control condition*, which is here a regular expression over the rules that admits any sequence of rules except the empty sequence, i.e. at least one rule application must be executed. Since in $G()$ there is no couple of edges that can be deleted by any of the rules r_x for $x \in \Delta$, the first rule must be one of the constructive rules r_i. The derivation may stop whenever a *terminal* graph, here again always $G()$, is reached. Thus, inbetween a double string is constructed from the corresponding pairs by applying the rules r_i, and deconstructed as long as the symbols correspond by applying the rules r_x. The semantics of the unit then contains all pairs of initial with terminal graphs such that there is a derivation from the initial to the terminal graph that obeys the control condition. In this concrete case, the semantics of **PCP** either contains only the pair of the unique initial and unique terminal graph, i.e. $(G(), G())$, and this is the case if and only if the PCP instance has a solution, or otherwise the semantics is empty.

2.3 Modelling Turing Machines

Turing machines are formal models that are frequently used to show results in computation and complexity theory. They have also been used to prove properties of graph transformation systems. For instance in [7] a Turing machine is simulated by a graph transformation program to show the computational completeness of a newly developed graph transformation-based programming language.

A Turing machine M is a device being in one of various states at any given instant. It contains a linear tape, comprising a chain of cells which are ordered from left to right. A cell can be empty or it can contain a symbol from a given finite alphabet Σ. The alphabet Σ' contains all symbols from Σ and the special symbol $\sqcup$ (denoting an empty cell). At any time the tape is finitely long, but it may be extended to the left and right with empty cells under certain circumstances. The Turing machine also includes a read-write head which is positioned on one of the cells. For this reason the tape can be described by two strings $u, v \in \Sigma'^*$. The string u contains the tape content that is left of the current head position, and v the tape content to its right including the current cell. Initially the machine is in a distinguished start state q_0, the head is positioned at the leftmost cell and no cell is empty if $v \neq \lambda$ (i.e. $u = \lambda$ and $v \in \Sigma^*$). The action of the Turing machine is determined by a configuration transition relation δ that assigns a new state q', a symbol σ', and a direction $d \in \{L, R\}$ to a current state q and a symbol σ. A concrete step of a Turing machine then changes the state, writes a symbol into the current cell and moves the head to the left or to the right, always according to the configuration transition relation. Such a step can be repeatedly executed until no further step is possible. This is obviously the case if δ does not specify a transition for the given state and the scanned symbol. This happens in particular if the current state of the machine is a final state. In case the head is positioned on the leftmost cell and δ specifies a movement of the head to the left, the tape is extended to the left by one empty cell and the head positioned on this new cell. This is done analogously for the right end of the tape. An advanced introduction to Turing machines, different existing types, and their properties can be found in, e.g., [8].

Formally, a *Turing machine* is a tuple $M = (Q, \Sigma, \Sigma', \delta, q_0, F)$. Here Q is a finite set of states, Σ is a finite alphabet, $\Sigma' = \Sigma \cup \{\sqcup\}$, δ is a configuration transition relation, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is a set of final states (we will assume w.l.o.g. that $F = \{q_F\}$). The configuration transition relation δ assigns a new state q', a symbol $\sigma' \in \Sigma'$, and a direction $d \in \{L, R\}$ to a current state q and a symbol $\sigma \in \Sigma'$. It is thus $\delta : Q \times \Sigma' \rightsquigarrow Q \times \Sigma' \times \{L, R\}$ with $(q, \sigma) \mapsto (q', \sigma', d)$.

The current state of the Turing machine, the position of the head on the tape and the contents of the tape form the current *configuration* of the Turing machine, which can be denoted by a single string. Let $uv \in \Sigma'^*$ denote the tape content as explained above, with the head currently on the first symbol of v (or an empty cell if $v = \lambda$). Then the current state q_i of the machine is inserted into this string yielding uq_iv as a string representation of the configuration. For this notation we obviously assume that the identifiers of the states are different from the symbols of

the alphabet. An initial configuration of a Turing machine M is of the form $\lambda q_0 w$ with $w \in \Sigma^*$. A final configuration is of the form $u q_{\mathrm{F}} v$ with $u, v \in \Sigma'^*$ and $q_{\mathrm{F}} \in F$. For states in F the configuration transition relation must not specify a next state. Let $q, q' \in Q$, $u, v \in \Sigma'^*$, and $a, b, c \in \Sigma'$. A next configuration then emerges as follows:

- $uqav \mapsto ubq'v$ for $(q, a, q', b, R) \in \delta$
- $uq\lambda \mapsto uq\sqcup$
- $ucqav \mapsto uq'cbv$ for $(q, a, q', b, L) \in \delta$
- $\lambda qav \mapsto q' \sqcup bv$ for $(q, a, q', b, L) \in \delta$

For this paper we want to consider a transformation unit which models an arbitrary Turing machine. As graph transformation approach we choose the double-pushout variant introduced earlier. For the graph representation of the tape a string graph is used. For technical reasons it is also necessary to be able to determine whether the head is positioned on the leftmost resp. rightmost cell. For this reason we introduce two special cells labelled $\triangleright$ (indicating the left end) and $\triangleleft$ (indicating the right end). The head can never be placed on these special cells.

The current configuration of the Turing machine can then be represented as a graph in the following way. Let q_i be the current state of the machine, $w = \alpha_1 \alpha_2 \ldots \alpha_n$ be the content of the tape, and the head be positioned on the cell containing α_1. Then the string graph representing the content of the tape is extended by adding an edge parallel to the edge labelled α_1. This new edge is then labelled with the state q_i. Figure 9 shows the graph representing the initial state of the Turing machine $q_0 w$ with $w = \alpha_1 \alpha_2 \ldots \alpha_n$.

Fig. 9. The graph representing $q_0 w$

The initial configuration $q_0\lambda$ with the empty word λ as tape content is represented as depicted in Figure 10.

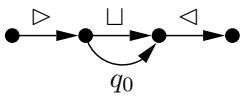

Fig. 10. Graph representation of $q_0\lambda$

For the dynamics of the Turing machine we define one graph transformation rule for every transition of the modelled Turing machine. In order to obtain a generic framework for the modeling of a Turing machine, we combine analogous rules. Let $M = (Q, \Sigma, \Sigma', \delta, q_0, F)$ be the Turing machine to be modelled. Then for every transition $(q, \sigma, q', \sigma', R) \in \delta$ two different rules are needed as depicted in Figure 11. The first rule specifies the ordinary replacement of σ with σ', the movement of the head to the right and the change of the state q to q'. Here τ is left unchanged by

the rule, since it is a variable label that matches every symbol from Σ'. The second rule is needed in case the head is positioned on the right end of the tape. In this case a new empty cell is inserted and the head placed upon it. Combine the first kind of rules to form rule set R_1, and the second kind of rules to form rule set R_2.

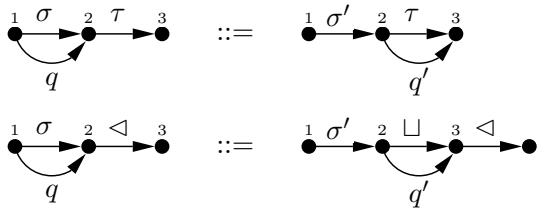

Fig. 11. Two rules for every $(q, \sigma, q', \sigma', R) \in \delta$

These rules are already sufficient to model every step of the configuration transition relation which specifies a movement of the head to the right. In order to model those steps that contain a head movement to the left, two sets R_3 and R_4 of different rules have also to be created. For all $(q, \sigma, q', \sigma', L)$ create two rules, as depicted in Figure 12. Again two rules are needed for every combination, because it may be necessary to extend the tape to its left.

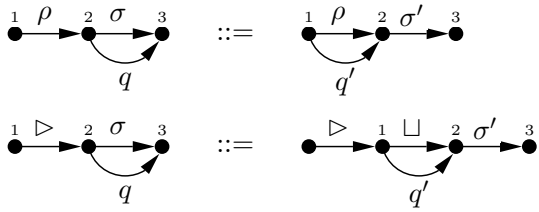

Fig. 12. Two rules for every $(q, \sigma, q', \sigma', L) \in \delta$

The final rule that is needed for this example is a rule which resets a final state of the Turing machine to the initial state, neither changing the contents of the tape nor the position of the head. This is realized by the rule r_f depicted in Figure 13.

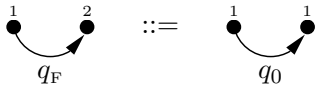

Fig. 13. Rule to reset a final state to the initial state

The transformation unit that realizes a Turing machine is depicted in Figure 14. As initial graphs we consider all string graphs for configurations with the initial state q_0, including in particular all encodings of initial configurations. The union of the rule sets R_1, R_2, R_3 and R_4 forms the set R. So the transformation unit contains all of them as rules, plus the final-state-resetting rule r_f. The control condition allows to select any one of the rules from the rule set $R = \{r_1, \ldots, r_k\}$ and repeat this selection arbitrarily often. After that repetition the rule r_f has to be applied. It is noteworthy to mention that this rule can only be applied if the Turing

machine simulation has actually reached the final state q_{F}. Thus, the semantics of this unit contains, among others, all pairs (G, H) such that G represents an initial configuration $q_0 w$ and H represents some accepting configuration that M can reach on input w, but with the initial instead of the final state.

tu(M)

initial	$\{\triangleright w_1 q_0 w_2 \triangleleft \mid w_1, w_2 \in \Sigma'^*\}$
rules	$R \cup \{r_f\}$
conds	$(r_1 \mid \ldots \mid r_k)^*; r_f$

Fig. 14. The graph transformation unit for a Turing machine M

3 Transformation Units

Transformation units are independent of a particular graph transformation approach, so first a general notion for such an approach is needed.

Graph transformation approach.

Let ID denote an arbitrary, but fixed set of identifiers that is partitioned into a set ID_R of rule identifiers and a set ID_U of transformation unit identifiers. A *graph transformation approach* then is a system $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Longrightarrow, \mathcal{X}, \mathcal{C})$ where

- $\mathcal{G}$ is a class of *graphs*,
- $\mathcal{R}$ is a class of *rules*,
- $\Longrightarrow\colon \mathcal{R} \to 2^{\mathcal{G}\times\mathcal{G}}$ is a rule application operator (yielding a binary relation $\underset{r}{\Longrightarrow} \subseteq \mathcal{G} \times \mathcal{G}$ for each $r \in \mathcal{R}$),
- $\mathcal{X}$ is a class of *graph class expressions* (such that $SEM(e) \subseteq \mathcal{G}$ specifies a subclass of $\mathcal{G}$ for every $e \in \mathcal{X}$), and
- $\mathcal{C}$ is a class of control conditions over ID (such that each $C \in \mathcal{C}$ specifies a binary relation $SEM_E(C) \subseteq \mathcal{G} \times \mathcal{G}$ for each *environment* $E\colon ID \to 2^{\mathcal{G}\times\mathcal{G}}$ that assigns a binary relation on graphs to each identifier).

If the application of a rule $r \in \mathcal{R}$ on a graph G yields the graph G' we write $G \underset{r}{\Longrightarrow} G'$ and call this a *direct derivation*. A sequence of rule applications is called a *derivation*.

The idea to have control conditions is to regulate the derivation process by enforcing some kind of order on rule applications and calls of imported transformation units and thus reducing the non-determinism inherent in graph transformation. Nevertheless, any description of a binary relation on graphs may be used as a formal control condition.

For the rest of this paper, we will assume that:

- graphs and rules are finite objects,

- it is decidable whether a given rule can be applied to a given graph,
- for every graph and every rule there is only a finite number of possibilities how this rule can be applied to the graph,
- for every such possibility, one can compute the unique result, and
- every graph class expression specifies a decidable class of graphs.

Graph transformation unit.

Let now $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Longrightarrow, \mathcal{X}, \mathcal{C})$ be a graph transformation approach. A *transformation unit over* $\mathcal{A}$ then is a system $tu = (I, U, R, C, T)$ where

- $I, T \in \mathcal{X}$ are graph class expressions,
- $U \subseteq ID$ is a finite set of identifiers,
- $R \subseteq \mathcal{R}$ is a finite set of rules, and
- $C \in \mathcal{C}$ is a control condition.

The graph class expressions I and T denote classes of initial and terminal graphs, respectively. Initial graphs define valid input for a transformation unit and terminal graphs specify what kind of graphs are expected as result of its computation. The initial graph of the PCP example is the graph $G()$, which contains only two nodes and the edges B and E. In the Turing machine example all graphs encoding a configuration with the initial state, together with the head position and the tape delimiters, are initial graphs. U defines the import of a transformation unit, thus it contains the identifiers of transformation units to import. If $U = \emptyset$, as is the case with both the PCP instance unit and the Turing machine unit, then no other transformation units are imported, meaning that we have an unstructured transformation unit of the lowest level. For this paper it suffices to consider a hierarchical import structure, i.e. each transformation unit may only import transformation units of a lower level. This facilitates the definition of the semantics given below. For transformation units that may have a cyclic import structure, a fix-point semantics is developed in [10].

Transformation units have an operational semantics that is defined as a binary relation on graphs. The intuition is that an initial graph is transformed into a terminal graph by stepwise execution of some sequence over the imported transformation units and the rules, where a step for $t \in U$ consists of using an adequate pair in the semantics of t and a step for $r \in R$ consists of applying r, such that the control condition – usually restricting the set of admissible sequences – is obeyed.

Let $tu = (I, U, R, C, T)$ be a transformation unit and E_{tu} an environment that assigns the rule application $\underset{r}{\Longrightarrow}$ to the identifier of each $r \in R$ and the semantics $SEM(t)$ to the identifier of each $t \in U$. Then $(G, H) \in SEM(tu)$ if

- $G \in SEM(I)$,
- $H \in SEM(T)$,
- there is a sequence $G_0, \ldots, G_n \in \mathcal{G}$ such that $G_0 = G, G_n = H$ and, for $i =$

$1, \ldots, n$, either $(G_{i-1}, G_i) \in SEM(t)$ for some $t \in U$ or $G_{i-1} \underset{r}{\Longrightarrow} G_i$ for some $r \in R$, and

- $(G, H) \in SEM_{E_{tu}}(C)$, which usually means that the sequence of the (G_{i-1}, G_i) must be allowed by the control condition C in the specific environment E_{tu}.

In the following section, we discuss three well-known types of control conditions, namely regular expressions over rules and imported transformation units, *as-long-as-possible* on other control conditions, and priorities over rules and imported transformation units, all of which regulate the transformation process. Note, however, that formally one may use any device as a control condition, as long as it specifies a binary relation on graphs. An example for this is a sequence of transformation units, specifying the concatenation of the respective semantics and thus a relation on graphs.

4 Decidability of Control Conditions

Control conditions regulate the control flow along rule applications and use of imported transformation units. Various kinds of control conditions and the interrelations between them are studied in [11]. Three kinds of control conditions have proved to be particularly useful in numerous examples: regular expressions, *as-long-as-possible*, and priorities. In this section, we will study these kinds of control conditions under the aspect of decidability.

Let $\mathcal{G}$ be a class of graphs, E an environment and $C \in \mathcal{C}$ a control condition with $SEM_E(C) \subseteq \mathcal{G} \times \mathcal{G}$. Then C is *decidable* if there is an algorithm that decides, for any $(G, H) \in \mathcal{G} \times \mathcal{G}$, whether $(G, H) \in SEM_E(C)$. We call C *(positive) semi-decidable* if there is a procedure that on input $(G, H) \in \mathcal{G} \times \mathcal{G}$ will eventually halt if $(G, H) \in SEM_E(C)$, and give the correct answer to the question '$(G, H) \in SEM_E(C)$?' whenever it halts.

4.1 Regular Expressions

Consider the set ID which contains only the identifiers ID_R of the given rules and the identifiers ID_U of imported transformation units. Each regular expression over ID denotes a regular language of strings over ID, and intuitively each of these strings describes a possible sequence of rule applications interleaved with calls of imported units.

The set REX of regular expressions over ID is defined as usual: $\emptyset$ and λ are regular expressions, each $id \in ID$ is a regular expression, and $(e_1; e_2)$, $(e_1|e_2)$, (e^*) are regular expressions for all regular expressions e_1, e_2, e. A regular expression is *star-free* if it does not contain a subexpression of the form (e^*).

In order to avoid parentheses, we assume that '*' has a stronger binding than ';', which in turn has a stronger binding than '|'. Moreover, ';' and '|' are associative, so that corresponding parentheses may be dropped too.

Formally, the semantics of a regular expression as control condition is a binary relation on $\mathcal{G}$ that is inductively defined as follows: For $\emptyset$, λ, and $id \in ID$, we have

- $SEM(\emptyset) = \emptyset$,
- $SEM(\lambda) = \{(G, G) \mid G \in \mathcal{G}\}$
- $SEM(id) = E(id)$, i.e. the relation assigned by the environment.

Then for $e, e_1, e_2 \in REX$ we have

- $SEM(e_1; e_2) = \{(G, H) \mid \exists G' \in \mathcal{G} : (G, G') \in SEM(e_1) \text{ and } (G', H) \in SEM(e_2)\}$,
- $SEM(e_1|e_2) = \{(G, H) \mid (G, H) \in SEM(e_1) \text{ or } (G, H) \in SEM(e_2)\}$, and
- $SEM(e^*) = \{(G, H) \mid \exists n \geq 1, G_0, \ldots, G_n \in \mathcal{G} \text{ with } (G_{i-1}, G_i) \in SEM(e) \text{ for } i = 1, \ldots, n \text{ such that } G_0 = G \text{ and } G_n = H\} \cup \{(G, G) \mid G \in \mathcal{G}\}$.

For instance, the control condition of the transformation unit **PCP** in Figure 8 is a regular expression, where the convention is used that for a given regular expression e the expression e^+ abbreviates $e; e^*$.

Decidability of regular expressions.

We can make the following observations:

(i) If C is a star-free regular expression over ID_R, then it describes a finite language over rules. Due to the assumption in Section 3 that rule applicability must be decidable, rule application computable, and all objects are finite, this implies that such a control condition is decidable.

(ii) If C is a regular expression over ID_R containing a star, then it defines an infinite (but enumerable) language over rules and is therefore (with the same argument as above) semi-decidable. However, in general it is not decidable: As a counterexample, consider the control condition $C_{\mathbf{PCP}}$ of **PCP**, which is of this kind and does not allow to decide whether $(G(); G())$ is in the semantics of $C_{\mathbf{PCP}}$ since PCP is undecidable.

(iii) Any identifier for a transformation unit with decidable finite semantics may additionally occur in a regular expression and the statements above still hold.

(iv) Any identifier for a transformation unit with decidable, but infinite semantics in an otherwise (semi-)decidable control condition will lead to a semi-decidable semantics.

(v) Any identifier for a transformation unit with undecidable semantics turns a regular expression into an undecidable control condition.

4.2 As-long-as-possible

Given any control condition C, the idea of *as-long-as-possible* is to iterate that condition until it can no longer be totally executed. For instance, if $C = r_1 r_2$ is a sequence of two rules, then the iteration $(r_1 r_2)!$ stops when this sequence cannot be applied anymore, even if r_1 alone could still be applied.

Let C be some control condition. Then $C!$ defines the set of all pairs $(G, H) \in SEM_E(C)^*$ such that there is no $H' \in \mathcal{G}$ with $(H, H') \in SEM_E(C)$.

For instance, one might replace the control condition of **tu(M)** in Figure 14 with

$(r_1|\dots|r_k)!$. Then the semantics of that unit changes to include, among others, all pairs of graphs (G, H) where G represents an initial configuration on which the Turing machine will eventually halt, and H is such a halting configuration (whether accepting or not).

Decidability of as-long-as-possible.

We can make the following observations:

(i) Iterating a star-free regular expression over rules with *as-long-as-possible* yields a semi-decidable control condition. It is in general not decidable because the halting problem for Turing machines is only semi-decidable, and that is what the variant of the Turing machine simulation given above encodes. It is still semi-decidable since star-free regular expressions over rules are decidable.

(ii) Iterating a single imported transformation unit with *as-long-as-possible* is in general undecidable and not even semi-decidable. Consider the control condition **PCP!**, where **PCP** is the unit in Figure 8, and the question whether $(G(), G()) \in SEM_E(\mathbf{PCP!})$. If the encoded PCP has a solution, then $SEM(\mathbf{PCP}) = \{(G(), G())\}$, and $(G(), G()) \notin SEM_E(\mathbf{PCP!})$. If the PCP does not have a solution, then $SEM(\mathbf{PCP}) = \emptyset$ and therefore $(G(), G()) \in SEM_E(\mathbf{PCP!})$. But this is not semi-decidable.

One may wonder whether the undecidability of *as-long-as-possible* iterating only a single transformation unit is due to the nondeterminism used in the control condition of **PCP** for the choice of the next constructive rule (the next applicable deconstructing rule is always dependent on the symbol appearing next to the B-labelled edge). However, this is not the case, which may be seen by considering **tu(M)**: Since we are interested in decidability questions rather than complexity, we may assume w.l.o.g. that M is a deterministic Turing machine. Consequently, any derivation in **tu(M)** is deterministic, too. Yet, the control condition **tu(M)!** is undecidable by analogous reasoning to the one used above for **PCP**.

4.3 Priorities

A priority control condition is a partial order on rules and used units that admits the application of a rule or a unit only if there is no rule or unit of higher priority that can be applied to the current graph.

A priority is a pair $C = (ID, <)$ where $<$ is an irreflexive partial order on ID. For $id \in ID$, let $HP_C(id) = \{id' \in ID \mid id < id'\}$ denote the set of identifiers with higher priority in C. Then $(G, H) \in SEM_E(C)$ if there are $G_0, \dots, G_n \in \mathcal{G}$ such that

- $G_0 = G$ and $G_n = H$,
- for $i = 1, \dots, n$, $(G_{i-1}, G_i) \in E(id_i)$ for some $id_i \in ID$, and for all $id \in HP_C(id_i)$ there is no $G \in \mathcal{G}$ with $(G_{i-1}, G) \in E(id)$.

For instance, one might replace the control condition of **PCP** with one that gives a higher priority to the constructing rules r_i than to the deconstructing rules r_x.

Since one can always apply a constructive rule to any graph derived from $G()$, one can derive all graphs $G(w)$ where w is a sequence over the indices $1, \ldots, k$, without any deconstruction from the side of the B-labelled edge. From such a graph, one can thus never again derive $G()$. Nevertheless, the empty derivation ensures that $(G(), G())$ is always in the semantics of the changed unit (and it is the only pair).

Decidability of priorities.

We can make the following observations:

(i) Any priority control condition that gives higher priority exclusively to rules is decidable, since the applicability of rules is required to be decidable.

(ii) Priority control conditions where a higher priority is given to some imported transformation unit are in general not decidable. Consider a rule r that replaces $G()$ with some graph H distinct from $G()$. We want to know whether $(G(), H)$ is in the semantics of the control condition $r < \mathbf{PCP}$. If $SEM(\mathbf{PCP})$ is empty, we may apply r, and the answer is yes. If $SEM(\mathbf{PCP})$ is not empty, then it contains (only) the pair $(G(), G())$, and because of the higher priority of **PCP** we may never apply r, implying that the answer is no. Since the semantics of **PCP** is undecidable, so is the semantics of the priority control condition $r < \mathbf{PCP}$.

4.4 Consequences for Implementing Graph Transformation Units

The results of this section rely on the use of a graph transformation approach that is computationally complete. Thus, it is to be expected that decidability of a control condition is the exception rather than the rule.

The semi-decidability of regular expressions over rules is due to the lack of general termination criteria. Consequently, just as with the control flow in ordinary programming languages, it is the responsibility of the programmer to ensure that their control condition in a graph transformation unit will always terminate. This task could be supported by providing tools to verify (the termination of) graph transformation units.

At the heart of the undecidability concerning *as-long-as-possible* or priorities on imported units lies the use of an undecidable condition, which is embodied in the semi- or undecidability of the semantics of the imported unit. At least for a first implementation of graph transformation units and while there is no support yet to prove the decidability for the imported unit, it seems therefore reasonable to syntactically forbid control conditions of this kind.

5 Conclusion

In this paper we have illustrated that certain control conditions used in transformation units are not decidable in general. It turns out that star-free regular expressions over rules are decidable, while regular expressions including the star are in general not decidable, as the **PCP** example shows. Iterating a star-free regular expression

over rules as well as iterating a single imported transformation unit with *as-long-as-possible* is generally undecidable and the latter is not even semi-decidable. The Turing machine example demonstrates that the undecidability of iterating a single imported transformation unit is not due to nondeterminism in control conditions. Similarly, priority control conditions that assign a higher priority to some imported unit are undecidable, whereas they are decidable if some higher priority is assigned only to rules. Therefore, a first implementation of graph transformation units should disallow *as-long-as-possible* and higher priorities on imported units.

The results of this paper are based on the use of a graph transformation approach that is computationally complete. It would be interesting to study the decidability of control conditions for less expressive approaches, in the hope of obtaining more decidability results.

Termination of graph transformation plays an important role in the presented considerations. Although thorough investigation for sequences in combination with *as-long-as-possible* has been started and results have been formulated, a general verification approach for transformation units is needed in the future. This verification method should in particular provide mechanisms for termination proofs.

Even though transformation units as a concept are independent of any particular graph transformation approach, this cannot be expected for a general verification method. In the future we will concentrate on verification of transformation units based on a given graph transformation approach. This is especially useful since we are about to implement a software system that allows to program with transformation units and eventually admits the execution of said units.

References

[1] Aßmann, U., *Graph rewrite systems for program optimization*, ACM Transactions on programming Languages and Systems (TOPLAS) **22** (2000).

[2] Bottoni, P., M. Koch, F. Parisi-Presicce and G. Taentzer, *Termination of high-level replacement units with application to model transformation.*, Electr. Notes Theor. Comput. Sci. **127** (2005), pp. 71–86.

[3] Corradini, A., U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe, *Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach*, in: Rozenberg [14], pp. 163–245.

[4] Ehrig, H., K. Ehrig, J. de Lara, G. Taentzer, D. Varró and S. Varró-Gyapay, *Termination criteria for model transformation*, in: M. Cerioli, editor, *Proc. FASE 2005*, Lecture Notes in Computer Science **3442** (2005), pp. 49–63.

[5] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools **2**, World Scientific, 1999.

[6] Ehrig, H., H.-J. Kreowski, U. Montanari and G. Rozenberg, editors, Handbook of Graph Grammars and Computing by Graph Transformation: Concurrency, Parallelism, and Distribution **3**, World Scientific, 1999.

[7] Habel, A. and D. Plump, *Computational completeness of programming languages based on graph transformation.*, in: F. Honsell and M. Miculan, editors, *Proc. FoSSaCS 2001*, Lecture Notes in Computer Science **2030** (2001), pp. 230–245.

[8] Hopcroft, J. E., R. Motwani and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison Wesley, 2001.

[9] Kreowski, H.-J. and S. Kuske, *Graph transformation units and modules*, in: Ehrig et al. [5] pp. 607–638.

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

[11] Kuske, S., *More about control conditions for transformation units*, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, *Proc. Theory and Application of Graph Transformations*, Lecture Notes in Computer Science **1764** (2000), pp. 323–337.

[12] Kuske, S., "Transformation Units—A structuring Principle for Graph Transformation Systems," Ph.D. thesis, University of Bremen (2000).

[13] Plump, D., *Termination of graph rewriting is undecidable.*, Fundam. Inform. **33** (1998), pp. 201–209.

[14] Rozenberg, G., editor, Handbook of Graph Grammars and Computing by Graph Transformation: Foundations **1**, World Scientific, 1997.

[15] Schürr, A., A. Winter and A. Zündorf, *PROGRES: Language and Environment*, in: Ehrig et al. [5] pp. 487–550.

[16] Taentzer, G., C. Ermel and M. Rudolf, *The AGG Approach: Language and Tool Environment.*, in: Ehrig et al. [5], pp. 551–603.