Graph Transformation Units Guided by a SAT Solver*

Hans-Jörg Kreowski, Sabine Kuske, and Robert Wille

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

Abstract. Graph transformation units are rule-based devices to model graph algorithms, graph processes, and the dynamics of systems the states of which are represented by graphs. Given a graph, various rules are applicable at various matches in general, but not any choice leads to a proper result so that one faces the problem of nondeterminism. As countermeasure, graph transformation units provide the generic concept of control conditions which allow one to cut down the nondeterminism and to choose the proper rule applications out of all possible ones. In this paper, we propose an alternative approach. For a special type of graph transformation units including the solution of many *NP*-complete and *NP*-hard problems, the successful derivations from initial to terminal graphs are described by propositional formulas. In this way, it becomes possible to use a SAT solver to find out whether there is a successful derivation for some initial graph or not and how it is built up in the positive case.

1 Introduction

Graph transformation units are rule-based devices to model algorithms and processes on graphs and the dynamic behavior of systems the states of which are represented by graphs [11]. Such a unit provides descriptions of initial and terminal graphs, a set of rules, and a control condition. The initial graphs are the inputs of the modeled computational processes, the terminal graphs are the potential outputs, the rules can be applied to graphs yielding derived graphs so that the iteration of rule applications establishes the running processes. In general, rule applications are quite nondeterministic because various rules may be applicable at various matches. The control condition is a feature to get rid of undesired nondeterminism. A typical example of control conditions are regular expressions which specify that the rules are applied in a certain order. Another kind are priorities that make sure that a rule with highest priority is applied in each step.

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

H. Ehrig et al. (Eds.): ICGT 2010, LNCS 6372, pp. 27–42, 2010.

In this paper, we propose an alternative approach to deal with nondeterminism. Graph transformation units of a special type (using only rules that keep the set of nodes invariant) are transformed into propositional formulas. If the formula corresponding to some graph transformation unit is satisfied by some assignment of truth values to the Boolean variables, then the assignment tells which rule must be applied at which match in which step such that a successful derivation is efficiently constructed. If the formula is unsatisfiable, then there is no successful derivation, and, thus, no need to start the derivation process at all. If the formula is extended properly, then the same correspondence between successful derivations and satisfying truth assignments holds for a single initial graph. As a consequence, one can employ a SAT solver to find successful derivations.

The proposed approach has a theoretical and a practical inspiration. In [4], Cook proved his seminal theorem that the satisfiability problem of the propositional calculus is *NP*-complete by describing the runs of a polynomial Turing machine in terms of propositional formulas. Despite this proven complexity, in the last decade efficient SAT solvers have been developed which can handle instances including hundreds of thousands variables and clauses in very short run-time. As a consequence, SAT solvers are successfully applied in the domain of verification, planning, and many more (see, e.g., [2]). In, e.g., [1,8] it is shown that chip designs can be verified by translating them into propositional formulas such that the unsatisfiability of the latter corresponds to the correctness of the respective chip. In contrast, if the formula turns out to be satisfiable, a counterexample showing the error can be derived. Recently, SAT solvers are applied in order to verify system descriptions given in the *Unified Modeling Language* (UML) [15]. Thus, SAT solvers have been proved to be practical for many relevant problems. Using this as motivation, the key idea of this paper is to replace Turing machines on one hand and chip designs or UML specifications on the other hand by a special type of graph transformation units. But the use of a SAT solver is only feasible if the size of the input formulas is polynomially bounded. Hence, we restrict the consideration to derivations the lengths of which are bounded by a polynomial. Therefore, the present approach applies particularly to polynomial graph transformation units including many solutions of *NP*-complete and *NP*-hard graph problems.

This paper is organized as follows. In Section 2, we recall the concepts of graph transformation and graph transformation units used in this paper. Section 3 shows how polynomial derivations can be modeled by propositional formulas in such a way that each satisfying instantiation of the formula represents a derivation. Section 4 generalizes Section 3 to graph transformation units, i.e, instead of polynomial derivations we model polynomial graph transformation units by propositional formulas. Moreover, an example of a polynomial graph transformation unit is given that can be represented as a propositional formula. This unit checks for any input graph whether it contains a Hamiltonian path. In Section 5 a very first implementation of the Hamiltonian path example is presented and tested for a series of concrete input graphs. The paper ends with the conclusion in Section 6.

2 Preliminaries

In this section, we recall basic notions and notations of graphs, graph transformation, and graph transformation units.

Directed edge-labeled simple graphs. For a set Σ of labels, a (*directed edge-labeled simple*) *graph* over Σ is a pair $G = (V, E)$ where V is a finite set of *nodes* and $E \subseteq V \times \Sigma \times V$ is a finite set of labeled *edges*. An edge $e = (v, x, v')$ is called a *loop* if $v = v'$. The components of G are also denoted by V_G and E_G. The set of all graphs over Σ is denoted by $\mathcal{G}_\Sigma$. We reserve a specific label $*$ which is omitted in drawings of graphs. In this way, graphs where all edges are labeled with $*$ may be seen as *unlabeled graphs*. The number of nodes is the *size* of G, denoted by $size(G)$. If Σ is finite with $|\Sigma|$ labels, the number of edges of a graph is bounded by $|\Sigma| \cdot size(G)^2$ because of the simplicity.

Graph morphisms. For graphs $G, H \in \mathcal{G}_\Sigma$, a *graph morphism* $g\colon G \to H$ is a mapping $g_V\colon V_G \to V_H$ that is structure- and label-preserving, i.e., for all $(v, x, v') \in E_G$, $(g_V(v), x, g_V(v')) \in E_H$. If the mapping g_V is an inclusion, then G is called a *subgraph* of H, denoted by $G \subseteq H$. For a graph morphism $g\colon G \to H$, the image $g(G) = (g_V(V_G), g_E(E_G)) \subseteq H$ of G in H with $g_E(E_G) = \{(g_V(v), x, g_V(v')) \mid (v, x, v') \in E_G\}$ is called a *match* of G in H.

Rules. A *rule* $r = (L \supseteq K \subseteq R)$ consists of three graphs $L, K, R \in \mathcal{G}_\Sigma$ such that K is a discrete subgraph of L and R, i.e., $E_K = \emptyset$. The components L, K, and R of r are called *left-hand side*, *gluing graph*, and *right-hand side*, respectively.

Rule application. Let $r = (L \supseteq K \subseteq R)$ be a rule and let $G \in \mathcal{G}_\Sigma$. Moreover, let $g\colon L \to G$ be a graph morphism satisfying the following conditions:

- *dangling condition*: If a node v in $g(L)$ is the source or the target of an edge in $E_G - g_E(E_L)$, then $v \in g_V(V_K)$.
- *identification condition*: $g_V(v) = g_V(v')$ for $v, v' \in V_L$ implies $v = v'$ or $v, v' \in V_K$.

Then the application of r to G with respect to g consists of the following three steps.

1. Remove the nodes of $g_V(V_L - V_K)$ and the edges of $g_E(E_L - E_K)$ yielding the *intermediate graph* $Z \subseteq G$.
2. Let $d\colon K \to Z$ be the restriction of g to K and Z, then the pushout of d and the inclusion of K into R yields the resulting graph H and graph morphisms $h\colon R \to H$ and $i\colon Z \to H$. Without loss of generality, one can assume that i is the inclusion of Z into H and that h is the identity on $R - K$. This provides an explicit construction of H because $Z \cup h(R) = H$ and $Z \cap h(R) = d(K) = h(K)$.

The application of a rule r to a graph G with respect to g is denoted by $G \underset{r,g}{\Rightarrow} H$, where H is the graph resulting from this application. As usual, a rule application is called a *direct derivation*. The subscript r, g may be omitted if it is clear from the context. The notion of a direct derivation coincides with a rule application in the double-pushout approach (cf., e.g., [5,7]), but we need the explicit set-theoretic construction in the next section.

The sequential composition $d = G_0 \underset{r_1,g_1}{\Longrightarrow} G_1 \underset{r_2,g_2}{\Longrightarrow} \cdots \underset{r_n,g_n}{\Longrightarrow} G_n$ $(n \in \mathbb{N})$ is called a *derivation* from G_0 to G_n. The derivation from G_0 to G_n can also be denoted by $G_0 \overset{n}{\underset{P}{\Longrightarrow}} G_n$ where $\{r_1, \ldots, r_n\} \subseteq P$, or just by $G_0 \overset{*}{\underset{P}{\Longrightarrow}} G_n$. The subscript P may be omitted if it is clear from the context. The string $r_1 \cdots r_n$ is the *application sequence* of the derivation d.

Given a finite set of rules and a graph G, the number of graph morphisms from left-hand sides to G is bounded by a polynomial in the size of G because the sizes of left-hand sides of rules are bounded by a constant. Given a match, the check, whether the dangling and the identification condition hold, and the construction of the directly derived graph is linear in the size of G. Therefore, polynomial time is needed to find a match and to construct a direct derivation, and there is a polynomial number of matches. Moreover, the size of the resulting graph differs from the size of the host graph by a constant.

Control conditions. The nondeterminism of rule application can be restricted via control conditions. A typical example is a regular expression over a set of rules (or any other string-language-defining device). Let C be a regular expression specifying the language $L(C)$. Then a derivation with application sequence s is *permitted* by C if $s \in L(C)$.

Graph class expressions. The initial and terminal graphs of derivations can be specified by graph class expressions. For every graph class expression e the set of graphs specified by e is denoted by $SEM(e)$. A typical example is a subset $\Delta \subseteq \Sigma$ with $SEM(\Delta) = \mathcal{G}_\Delta \subseteq \mathcal{G}_\Sigma$. Requested or forbidden subgraphs and graphs that are reduced with respect to some rule set are also frequently used.

Graph transformation unit. Every graph transformation unit transforms initial graphs to terminal graphs via the successive application of rules according to a control condition. A (*simple*) *transformation unit* is a system $tu = (I, P, C, T)$, where I and T are graph class expressions, P is a finite set of rules, and C is a control condition. tu is *polynomial* if:

- there is a polynomial p such that for each initial graph $G \in SEM(I)$ and each derivation $G \overset{n}{\underset{P}{\Longrightarrow}} G'$, $n \leq p(size(G))$,
- the membership problems of $SEM(initial)$ and $SEM(T)$ are polynomial, and
- it can be checked in polynomial time whether the $G \overset{n}{\underset{P}{\Rightarrow}} G'$ is permitted by C.

Each transformation unit tu specifies a binary relation $SEM(tu) \subseteq SEM(I) \times SEM(T)$ that contains a pair (G, H) of graphs if and only if there is a derivation $G \overset{*}{\underset{P}{\Longrightarrow}} H$ permitted by C.

Transformation units are presented in [10,11]. In this first approach towards translating transformation units into propositional formulas, we consider simple transformation units where the structuring component is omitted. More about control conditions for transformation units can be found in, e.g., [12,9].

Further notions and notations. For each language $L \subseteq P^*$ and each number $m \in \mathbb{N}$, the finite sublanguage $L(m) = \{w \in L \mid 0 \leq |w| \leq m\}$ denotes the set of all words in L of length at most m. For each $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$. For each propositional formula Q, the set of variables of Q is denoted by $\mathcal{V}_Q$ and the number of literals by $L^{\#}(Q)$.

3 From Polynomial Derivations to Propositional Logic

In this section, we show how derivations can be modeled by propositional formulas. More precisely, every finite rule set P together with a given initial graph G_0 and a polynomial p can be automatically transformed into a propositional formula that describes all derivations from G_0 that are not longer than $p(size(G_0))$.

In this first approach, we restrict the consideration to graphs with nodes that are numbered from 1 to n for some $n \in \mathbb{N}$, to rules which keep the set of nodes invariant, and to injective graph morphisms. Moreover, we assume that the label alphabet Σ is finite. A well-known instance of graph transformation systems with invariant node sets are graph relabelling systems (see, e.g., [13]).

It is worth noting that all constructions in this section work for arbitrary lengths bounds of derivations and not only for polynomial bounds, but then the resulting propositional formulas can become intractable.

3.1 Representing Simple Graphs as Propositional Formulas

Every propositional formula Q with $\mathcal{V}_Q = [n] \times \Sigma \times [n]$ specifies a set of graphs because each instantiation $f\colon \mathcal{V}_Q \to \text{BOOL}$ represents the graph $graph(f) = ([n], E)$ where $E = \{(v, a, v') \in [n] \times \Sigma \times [n] \mid f(v, a, v') = \text{TRUE}\}$. Hence, the set $Graphs(Q)$ of graphs specified by Q is equal to $\{graph(f) \mid f\colon \mathcal{V}_Q \to \text{BOOL}, f(Q) = \text{TRUE}\}$. In the following, for each $n \in \mathbb{N}$, a propositional formula Q with $\mathcal{V}_Q = [n] \times \Sigma \times [n]$ is called a *propositional graph formula of* n. The set of all propositional graph formulas of n is denoted by $FG(n)$. A particular form of propositional graph formulas of n is $\bigwedge_{e \in E} e \wedge \bigwedge_{e \in ([n] \times \Sigma \times [n]) - E} \neg e$ where E is a subset of $[n] \times \Sigma \times [n]$. For each such subset E, this formula represents the single graph $G = ([n], E)$ and will be denoted by $fg(G)$.

3.2 Representing Derivations as Propositional Formulas

As said before, for every rule $r = (L \supseteq K \subseteq R)$, we require that $V_L = V_K = V_R$. This implies that every morphism from L into some graph G satisfies the dangling

condition. Additionally, since we use only injective graph morphisms in rule applications the identification condition is also fulfilled. In the following, the set V_L will be also denoted by V_r. The number of nodes in V_r will be denoted by $size(r)$. For an underlying rule set P its maximal size $\max\{size(r) \mid r \in P\}$ will be denoted by max.

Given a polynomial p, a polynomial derivation with respect to p has the form $G_0 \underset{r_1,g_1}{\Longrightarrow} G_1 \underset{r_2,g_2}{\Longrightarrow} \ldots \underset{r_m,g_m}{\Longrightarrow} G_m$ with $m \leq p(size(G_0))$. Due to the assumption for rules, the set of nodes $[n]$ is invariant through the whole derivation. Therefore, the occurring graphs are fully described by fixing their edges where each edge is a triple $(i, a, j) \in [n] \times \Sigma \times [n]$. This is possible by means of Boolean variables of the form $edge(i, a, j, k)$ which should be true if and only if (i, a, j) is an edge of G_k.

The actual edges of all the derived graphs depend on the initial graph G_0. Let $G_0 = ([n], E_0)$. Then G_0 can be directly described via the formula

$$\bigwedge_{(v,a,v') \in E_0} edge(v, a, v', 0) \wedge \bigwedge_{(v,a,v') \in ([n] \times \Sigma \times [n]) - E_0} \neg edge(v, a, v', 0).$$

Consider for example, the derivation in Figure 2 where the rule of Figure 1 is applied twice. The graph G_0 of this derivation is described by the preceding formula by choosing $E_0 = \{(1, ok, 1), (1, *, 2), (2, *, 2), (2, *, 3), (3, *, 3)\}$, $n = 3$ and $\Sigma = \{ok, *\}$.

In the following, for each propositional graph formula $Q \in FG(n)$ and each $k \in \mathbb{N}$ the term $Q(k)$ denotes the formula that is obtained from Q by replacing every variable (v, a, v') by $edge(v, a, v', k)$. Hence, the above formula for G_0 is equal to $fg(G_0)(0)$. Moreover, for $k \geq 1$, let $\mathcal{V}(n, k) = \{edge(v, a, v', k) \mid v, v' \in [n], a \in \Sigma\}$. Then for every mapping $f \colon \mathcal{V}(n, k) \to \text{BOOL}$, the graph induced by f is $graph(f) = ([n], E)$ with $E = \{(v, a, v') \mid f(edge(v, a, v', k)) = \text{TRUE}\}$.

The edges of the derived graphs depend not only on the initial graph but also on the rules and matches in the derivation steps. Let $r = (L \supseteq K \subseteq R)$ be a rule and let $g \colon V_L \to [n]$ be an injective function that maps the nodes of L to the nodes of G_{k-1}. The fact that g is a graph morphism from L to G_{k-1} is expressed by the formula

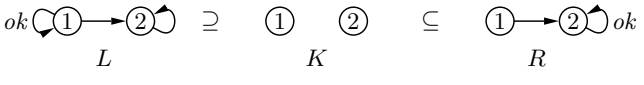

Fig. 1. A rule

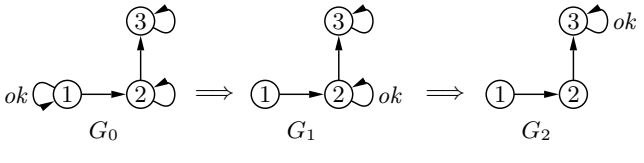

Fig. 2. Example of a derivation describable by a propositional formula

$$morph(r, g, k) = \bigwedge_{(v,a,v') \in E_L} edge(g(v), a, g(v'), k-1),$$

which means that every edge in E_L must have an image in G_{k-1} through g. Concerning the first derivation step in Figure 2, there exist six injective mappings from V_L to the set $[3] = \{1, 2, 3\}$ but the formula $morph(r, g, 1)$ only holds if $g(i) = i$ for $i = 1, 2$. In this case we get $morph(r, g, 1) = edge(1, ok, 1, 0) \wedge edge(1, *, 2, 0) \wedge edge(2, *, 2, 0)$.

The application of r to G_{k-1} removes the image of every edge of the left-hand side L from G_{k-1} provided that it is not re-inserted via the right-hand side R. This is expressed by the propositional formula

$$rem(r, g, k) = \bigwedge_{(v,a,v') \in E_L - E_R} \neg edge(g(v), a, g(v'), k).$$

With respect to our example derivation we get

$$rem(r, g, 1) = \neg edge(1, ok, 1, 1) \wedge \neg edge(2, *, 2, 1)$$

which evaluates to TRUE.

Afterwards, the edges of the right-hand side R are added, which corresponds to the formula

$$add(r, g, k) = \bigwedge_{(v,a,v') \in E_R} edge(g(v), a, g(v'), k).$$

For our example derivation we get $add(r, g, 1) = edge(1, *, 2, 1) \wedge edge(2, ok, 2, 1)$.

The number of literals in each of the formulas $morph(r, g, k)$, $rem(r, g, k)$, and $add(r, g, k)$ is bounded by $\max^2 \cdot |\Sigma|$, i.e.,

$$L^{\#}(Q) \in O(1), \text{ for } Q \in \{morph(r, g, k), rem(r, g, k), add(r, g, k)\}.$$

All edges of G_{k-1} that are neither deleted nor inserted must be kept in the derivation step k. This leads to the formula

$$keep(r, g, k) = \bigwedge_{(v,a,v') \in ([n] \times \Sigma \times [n]) - g(E_L \cup E_R)} \Big(edge(v, a, v', k) \leftrightarrow edge(v, a, v', k-1) \Big),$$

where $g(E_L \cup E_R) = \{(g(v), a, g(v')) \mid (v, a, v') \in E_L \cup E_R\}$. The number of literals in $keep(r, g, k)$ is bounded by $2n^2 \cdot |\Sigma|$, i.e., $L^{\#} \in O(n^2)$. With respect to the example we get

$$\begin{aligned} keep(r, g, 1) = {} & (edge(1, *, 1, 1) \leftrightarrow edge(1, *, 1, 0)) \wedge \\ & (edge(1, ok, 2, 1) \leftrightarrow edge(1, ok, 2, 0)) \wedge \\ & \textstyle\bigwedge_{x \in \{*, ok\}} ((edge(1, x, 3, 1) \leftrightarrow edge(1, x, 3, 0)) \wedge \\ & (edge(2, x, 1, 1) \leftrightarrow edge(2, x, 1, 0)) \wedge \\ & (edge(2, x, 3, 1) \leftrightarrow edge(2, x, 3, 0)) \wedge \\ & (edge(3, x, 1, 1) \leftrightarrow edge(3, x, 1, 0)) \wedge \\ & (edge(3, x, 2, 1) \leftrightarrow edge(3, x, 2, 0)) \wedge \\ & (edge(3, x, 3, 1) \leftrightarrow edge(3, x, 3, 0))). \end{aligned}$$

Summarizing, the fact that g is a graph morphism from L to G_{k-1} such that G_k is obtained from G_{k-1} via the application of r with respect to g is expressed by the conjunction of the formulas *morph*, *rem*, *add*, and *keep*:

$$apply(r,g,k) = morph(r,g,k) \wedge rem(r,g,k) \wedge add(r,g,k) \wedge keep(r,g,k).$$

The following lemma concerning the semantics and the number of literals of the formula *apply* can be proved in a straightforward way by using the introduced definitions.

Lemma 1. 1. Let $f\colon \mathcal{V}(n,k-1) \cup \mathcal{V}(n,k) \to \text{BOOL}$. Then

$$graph(f|\mathcal{V}(n,k-1)) \underset{r,g}{\Longrightarrow} graph(f|\mathcal{V}(n,k))$$

if and only if $f(apply(r,g,k)) = \text{TRUE}$.[1]

2. $L^{\#}(apply(r,g,k)) \in O(n^2)$.

In each step k of a derivation, the formula $apply(r,g,k)$ must hold for at least one rule r of the underlying rule set P and for at least one injective mapping $g\colon V_r \to [n]$. Let $M(r,n)$ be the set of injective mappings from V_r to $[n]$. Then a k^{th} derivation step is described by the formula

$$step(k) = \bigvee_{r\in P, g\in M(r,n)} apply(r,g,k),$$

which has at most $|P| \cdot n^{\max} \cdot (3\max^2 \cdot |\Sigma| + 2n^2 \cdot |\Sigma|)$ literals, i.e., $L^{\#}(step(k)) \in O(n^{2+\max})$.

The following formula $fder(G_0,m)$ describes all derivations of length m that start in G_0, for each graph G_0 and each natural number m.

$$fder(G_0,m) = fg(G_0)(0) \wedge \bigwedge_{k=1}^{m} step(k),$$

where for $m = 0$ the formula $\bigwedge_{k=1,\ldots,0} step(k)$ is the empty formula which evaluates to TRUE. The next theorem follows from Lemma 1 and the definitions of *step* and *fder*. It states that each instantiation of the formula $fder(G_0,m)$ represents a derivation from G_0 of length m. In particular, all graphs of this derivation can be explicitly constructed from the corresponding variable instantiation. The number of literals in $fder(G_0,m)$ are bounded by a polynomial.

Theorem 1. Let $G_0 = ([n],E)$ be a graph, let $m \in \mathbb{N}$, and let $f\colon \mathcal{V}_{fder(G_0,m)} \to \text{BOOL}$. Then

1. $G_0 \underset{P}{\overset{m}{\Longrightarrow}} G$ if and only if $f(fder(G_0,m)) = \text{TRUE}$;
2. $L^{\#}(fder(G_0,m)) \in O(n^{2+\max} \cdot m)$.

[1] For $A \subseteq B$ and a function $f\colon B \to C$, $f|A\colon A \to C$ is defined by $f|A(x) = f(x)$, for each $x \in A$.

Remarks

1. The formula $fder(G_0, m)$ can be used in the following more general formulas:
   (a) Let p be a polynomial. Then the set of all derivations of length at most $p(n)$ that start in G_0 is specified via the formula $fder(G_0)$ as follows: [2]

$$fder(G_0) = \bigvee_{m=0}^{p(n)} fder(G_0, m).$$

   For the number of literals of $fder(G_0)$ we get

$$L^{\#}(fder(G_0)) \in O(n^{\max+2} \cdot p(n)^2).$$

   The quadratic factor of the polynomial $p(n)$ in the number of literals can be reduced to a linear one. One may add the empty rule $mt = (\emptyset \supseteq \emptyset \subseteq \emptyset)$ to the set of rules which is always applicable with the empty match, but does never cause any change. Then the set of all polynomial derivations starting from G_0 is described by $fder(G_0, p(n))$ because each derivation of length $m < p(n)$ may be prolonged by empty steps to the length $p(n)$. The other way round, a successful derivation of length $p(n)$ with empty steps is also successful if the empty steps are removed.
   (b) Even more generally, the set of all polynomial derivations over the rule set P starting from an arbitrary graph with n nodes is described by the formula

$$fder(n) = \Big(\bigwedge_{e \in [n] \times \Sigma \times [n]} (e \vee \neg e) \Big)(0) \wedge \Big(\bigvee_{m=0}^{p(n)} \bigwedge_{k=1}^{m} step(k) \Big).$$

   The number of literals of $fder(n)$ is also in $O(n^{2+\max} \cdot p(n)^2)$. Analogously to point (a), $p(n)^2$ can be replaced by $p(n)$ by adding the empty rule.
2. As mentioned above, the assumption that the length bound of derivations is a polynomial implies that the number of literals of the associated formulas is also polynomial and hence tractable. But the presented constructions work for every bounding function.

4 From Polynomial Transformation Units to Propositional Formulas

In this section, polynomial transformation units are translated into propositional formulas so that every valid instantiation of such a formula represents a successful derivation of the unit. More explicitly, we do not consider stand-alone sets of rules but also initial and terminal graph class expressions as well as control

[2] For not having to introduce too many different names for very similar functions, we overload the function name *fder* by making its semantics dependent of its parameter types.

conditions. Given a set of rules, polynomial graph class expressions for the initial and terminal graphs, as well as a polynomial control condition, the resulting propositional formula describes all polynomial derivations that start with an initial graph, end with a terminal graph, and are allowed by the control condition.

4.1 Propositional Graph Class Expressions

As graph class expressions we use the propositional graph formulas introduced in Subsection 3.1. More precisely, every graph class expression is a mapping e that associates with each $n \in \mathbb{N}$ a propositional graph formula $e(n)$ of n. The graph class expression e is called polynomial if for each $n \in \mathbb{N}$, the number of literals in $e(n)$ is bounded by $p(n)$ for some given polynomial p. The semantics of each such graph class expression e is the set $SEM(e) = \bigcup_{n\in\mathbb{N}} Graphs(e(n))$.

Examples of propositional graph class expressions

1. The set of all graphs in $\mathcal{G}_\Sigma$ is specified by the propositional graph class expression *all*, where for each $n \in \mathbb{N}$,

$$all(n) = \bigwedge_{e\in[n]\times\Sigma\times[n]} \big(e \vee \neg e\big).$$

2. The class of unlabeled graphs in $\mathcal{G}_\Sigma$ can be specified by the graph class expression *unlabeled*, where for each $n \in \mathbb{N}$,

$$unlabeled(n) = \bigwedge_{v,v'\in[n]} \Big((v,*,v') \vee \neg(v,*,v')\Big) \wedge \bigwedge_{e\in[n]\times(\Sigma-\{*\})\times[n]} \neg e.$$

3. More generally, the set of all graphs in $\mathcal{G}_\Sigma$ that are labeled over a set $\Delta \subseteq \Sigma$ is specified by Δ with

$$\Delta(n) = \bigwedge_{e\in[n]\times\Delta\times[n]} \big(e \vee \neg e\big) \wedge \bigwedge_{e\in[n]\times(\Sigma-\{\Delta\})\times[n]} \neg e.$$

4. For $\Sigma = \{*\}$, the set of graphs in which every node has a loop is specified by *loop* with

$$loop(n) = \bigwedge_{v\in[n]} (v,*,v) \wedge \bigwedge_{v,v'\in[n]} \Big((v,*,v') \vee \neg(v,*,v')\Big).$$

All these examples of graph class expressions are polynomial because their numbers of literals are bounded by $2n^2 \cdot |\Sigma|$.

4.2 Control Conditions of the Language Type

As control conditions we use language type conditions as introduced in Section 2. Every such control condition C specifies a language $L(C) \subseteq P^*$ where P is the

rule set of the transformation unit in which C occurs. A particular case of a language type condition is *free* that does not restrict anything because it specifies the language P^*. A control condition of the language type is called polynomial if there is a polynomial p such that for every $m \in \mathbb{N}$, $|L(C)(m)| \leq p(m)$, i.e., the number of words specified by C that are shorter than or as long as m does not exceed $p(m)$.

Example of a class of polynomial control conditions. Consider the following subset $REG(P)$ of regular expressions over P.

- $\emptyset, \lambda \in REG(P)$ with $L(\emptyset) = \emptyset$ and $L(\lambda) = \{\lambda\}$;
- for each $r \in P$, $r, r^* \in REG(P)$ with $L(r) = \{r\}$ and $L(r^*) = \{r\}^*$;
- for each $C_1, C_2 \in REG(P)$, $C_1; C_2 \in REG(P)$ with $L(C_1; C_2) = L(C_1) \cdot L(C_2)$.

For example, the regular expression $r_1^*; r_2$ specifies the language $\{r_1^m r_2 \mid m \in \mathbb{N}\}$. It is polynomial because $|L(r_1^*; r_2)(m)| = m$ for each $m \in \mathbb{N}$.

The following proposition states that all control conditions of the presented subclass of regular expressions are polynomial. More precisely, for every regular expression C in $REG(P)$ the number of words in $L(C)$ of length at most m is in $O(m^{size(C)})$ where $size(C)$ is the number of occurrences of r and r^* in C (for all $r \in P$).

Proposition 1. Let $C \in REG(P)$. Then $|L(C)(m)| \in O(m^{size(C)})$ for each $m \in \mathbb{N}$, where $size(\emptyset) = size(\lambda) = 0$, $size(r) = size(r^*) = 1$, and $size(C_1; C_2) = size(C_1) + size(C_2)$.

Proof. Obviously, for the conditions $\emptyset$, λ, and r, the number of specified words is bounded by 1, i.e., for each $m \in \mathbb{N}$, $|L(C)(m)| \in O(m^0)$ with $C \in \{\emptyset, \lambda, r\}$. Since $O(m^0) \subseteq O(m^1)$ we get that $|L(C)(m)| \in O(m^{size(C)})$. Moreover, by induction on m we get that $|L(r^*)(m)| = m + 1$, i.e., $|L(r^*)(m)| \in O(m)$. Finally, consider the condition $C_1; C_2$. Then by definition $L(C_1; C_2) = \{w_1 w_2 \mid w_i \in L(C_i), i = 1, 2\}$ which implies that the number of words in $L(C_1; C_2)(m)$ is bounded by $|L(C_1)(m)| \cdot |L(C_2)(m)|$. By induction hypothesis $|L(C_i)(m)| \in O(m^{size(C_i)})$ which implies that $|L(C_1; C_s)(m)| \in O(m^{size(C_1)+size(C_2)})$. Since $size(C_1) + size(C_2) = size(C_1; C_2)$, the condition $C_1; C_2$ satisfies the required property.

4.3 Representing the Semantics of Polynomial Transformation Units by Propositional Formulas

Let $tu = (I, P, C, T)$ be a transformation unit such that I and T are polynomial propositional graph class expressions, and $C = free$. Then for all $m, n \in \mathbb{N}$, the set of successful derivations of length m starting from a graph of size n can be described by the mapping *ftufree* defined as follows.

$$ftufree(n, m) = I(n)(0) \wedge \bigwedge_{k=1}^{m} step(k) \wedge T(n)(m).$$

This can be shown by using Theorem 1.

If the control condition C of tu is polynomial, then all successful derivations of length m are described by the formula

$$ftu(n,m) = I(n)(0) \wedge \Big(\bigvee_{r_1 \dots r_m \in L(C)} \big(\bigwedge_{k=1}^{m} step(k, r_k) \wedge T(n)(m) \big) \Big),$$

where for each $r \in P$, $step(k,r) = \bigvee_{g \in M(r,n)} apply(r,g,k)$, i.e., $step(k,r)$ requires the application of rule r in step k..

The next theorem states that for every valid instantiation of the formula $ftu(n,m)$, there exists a successful derivation of length m in tu. Moreover, the number of literals in $ftu(n,m)$ is polynomial.

Theorem 2. Let $tu = (I,P,C,T)$ be a transformation unit with $I(n), T(n) \in O(n^j)$ and $|L(C)(m)| \in O(m^l)$ for $j,l \in \mathbb{N}$.

1. There is a derivation $G_0 \underset{r_1,g_1}{\Longrightarrow} G_1 \underset{r_2,g_2}{\Longrightarrow} \dots \underset{r_m,g_m}{\Longrightarrow} G_m$ with $(G_0,G_m) \in SEM(I) \times SEM(T)$, $r_1 \cdots r_m \in L(C)$ if and only if there is an instantiation $\mathcal{V}_{ftu(n,m)} \to$ BOOL with $f(ftu(n,m)) =$ TRUE where $n = size(G_0)$.
2. The number of literals of $ftu(n,m)$ is in $O(m^{l+1} \cdot n^i)$ where $i = \max\{2 + \max, j\}$.

Remarks. Let p be a polynom.

1. If the control condition is *free*, the set of all successful polynomial derivations of tu can be described by $ftufree(n) = \bigvee_{m=0}^{p(n)} ftufree(n,m)$.
2. If the control condition is polynomial, all polynomial successful derivations can be described by generalizing the formula $ftu(n,m)$ analogously to the generalization of *ftufree* in point 1.

4.4 Example

An example for a polynomial (nondeterministic) unit that finds Hamiltonian paths is depicted in Figure 3. The underlying alphabet Σ is equal to $\{*, ok\}$. The class of initial graphs consists of all unlabeled simple graphs in which every node has a loop. These graphs are specified via the polynomial graph class expression *unlabeled&loops* with

$$unlabeled\&loops(n) = \bigwedge_{v \in [n]} (v, *, v) \wedge unlabeled(n),$$

where $unlabeled(n)$ is the graph class expression introduced in the examples of Subsection 4.1. The first rule *start* labels an unlabeled loop with *ok*. The second rule *run* converts an unlabeled neighbor of an *ok*-node v into an *ok*-node while removing the *ok*-loop of v. (Please note that we call a node with a $*$-loop an unlabeled node and a node with an *ok*-loop an *ok*-node.) The third rule *stop* removes an *ok*-loop.

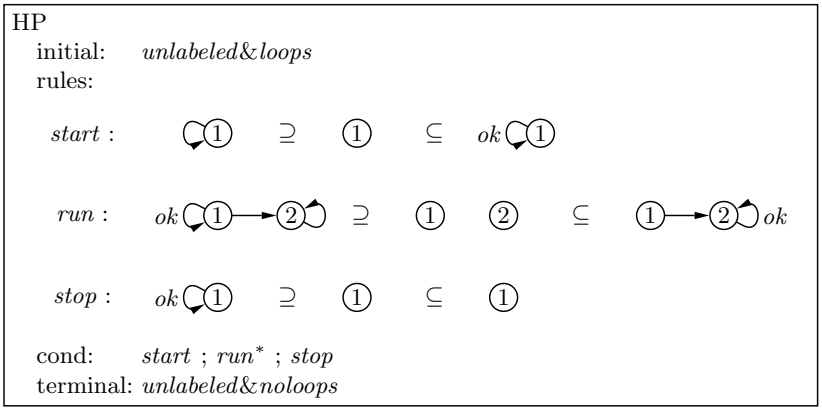

Fig. 3. A transformation unit for finding Hamiltonian paths

The control condition is the regular expression *start* ; run^* ; *stop* which guarantees that at first the rule *start* is applied exactly once; afterwards, *run* is applied arbitrarily often; and finally, *stop* is applied exactly once. Hence, the rule *start* selects an arbitrary node v of the input graph G_0, the rule *run* traverses a path in G_0 starting from v, and the rule *stop* terminates this process. The terminal graphs must not contain any loop which means that the rules must visit all nodes, i.e., the traversed path is Hamiltonian. The terminal graph class expression *unlabeled&noloops* is defined by

$$unlabeled\&noloops(n) = \bigwedge_{v \in [n]} \neg(v, *, v) \wedge unlabeled(n).$$

Successful derivations of HP can be found by feeding a SAT solver with the corresponding propositional formula.

5 Prototypical Implementation

The proposed methodology has been prototypically implemented for the Hamilton path problem. To this end, concrete unlabeled graphs (taken from [3]) have been considered. Having concrete instances available, the formulation as described in the former sections can be significantly simplified. In particular, the formulas for initial graphs as introduced in Subsection 3.2 can be removed and instead implicitly handled. Analogously, the formulas *morph*, *rem*, *add*, *keep*, and *step* needed to formulate the respective rule applications can be significantly simplified. Thus, in the following a simpler formulation derived from these constraints is used. Nevertheless, the same principles as described above are thereby applied.

More precisely, the SAT instance encoding the Hamilton path problem for a given graph instance $G_0 = ([n], E)$ only includes

- for each node $v \in [n]$ and for each $k \in \{0, \ldots, n\}$ the variables $edge(v, ok, v, k)$ stating whether the node v has an *ok*-loop in G_k and

– for each edge $(v, *, v') \in E$ of the given graph instance the variables $run_{vv'}$ stating whether the *run*-rule (see Subsection 4.4) was applied for this edge.

Then, the application of the *run*-rule between two nodes $v, v' \in V$ with $(v, *, v') \in E$ is constrained by

$$run_{vv'} \leftrightarrow \bigvee_{k=1}^{n} (edge(v, ok, v, k-1) \wedge edge(v', ok, v', k)).$$

That is, the *run*-rule can be applied between two nodes v, v' if and only if there is a node v with an *ok*-loop in G_{k-1} and a node v' with an *ok*-loop in G_k. By adding these constraints only for the edges from the given graph instance, all remaining $edge(v, a, v', k)$-variables as well as the respective *keep*-constraints are implicitly given and, thus, do not have to be added. Additionally, it is constrained that each node v is allowed to have an *ok*-loop in only one step, i.e., for each node $v \in [n]$ of the given graph instance $\sum_{k=1}^{n} edge(v, ok, v, k) = 1$ must hold which yields TRUE if and only if there is exactly one $k \in [n]$ with $edge(v, ok, v, k) = \text{TRUE}$.[3] Finally, it must be ensured that the *run*-rule is applied at most once to a node v which can be expressed by $\sum_{(v,*,v') \in E} run_{vv'} \leq 1$. This implicitly represents the *start*-rule, the *stop*-rule, as well as the initial and terminal conditions introduced in Subsection 4.4. If the SAT solver determines a satisfying assignment to all these variables, a Hamilton path can be derived from the assignments to $run_{vv'}$. Otherwise, it has been proven that no such path exists for the given graph instance. Since current state-of-the-art solvers usually work on Boolean formulas in Conjunctive Normal Form (i.e. conjunctions of clauses), the constraints introduced above have to be converted, respectively. However, this can be done in linear time and space [16].

Table 1 gives the results obtained by the proposed formulation. The first columns list applied graph instances with their numbers of nodes as well as their numbers of edges and indicate whether the considered graphs have a Hamilton

Table 1. Results obtained by the prototypical implementation

Samples (cf. [3])	Nodes	Edges	HP?	SAT-Vars	SAT-Clauses	Time (s)
anna	138	1972	no	335185	1235420	22,1
david	87	1624	no	168868	629735	5,4
games120	120	2552	no	357097	1344256	12,6
miles250	128	1548	no	251947	917624	13,4
myciel3	11	40	yes	924	2751	0,1
myciel4	23	142	yes	5280	17535	134,5
queens5_5	25	640	no	19796	73095	0,2
queens6_6	36	1160	no	49129	184816	0,8
queens10_10	100	5880	no	635641	2467440	16,3

[3] Such kind of constraints are known as *cardinality constraints* for which a couple of efficient SAT formulations exist [14].

path or not. The number of Boolean variables and the number of clauses of the resulting SAT formulation are given in the next two columns. Finally, the needed run-time (in CPU seconds) is given in the last column. MiniSAT [6] on an Intel 2.4 GHz with 1.4 GB has been utilized to obtain the results.

The results confirm that for established benchmark functions, practical relevant problems (in this case determination of Hamilton paths) can be efficiently solved using the proposed approach. In fact, for all considered graphs the considered problem can be solved in very short run-time.

6 Conclusion

In this paper, we have described derivations of a special type of graph transformation units by propositional formulas in such a way that a SAT solver can be employed to find semantically significant derivations or to prove that none of them exists. This result provides the chance to overcome the inefficiency caused by the nondeterminism of rule applications in many cases.

Clearly, the existing SAT solvers are exponential in the worst case, but they often run very fast and yield good results in the verification of chip designs, etc. Our very first tests on a sample graph transformation unit that solves the Hamiltonian-path problem look quite promising. This encourages us to undertake further steps in this direction including the following:

1. As mentioned at the end of Section 3, the description of polynomial derivations by propositional formulas can be improved by the introduction of empty steps reducing the sizes of the formulas. An interesting question is whether there are further optimizations or improvements that yield higher efficiency or more insight into the derivation process.
2. There are some first ideas how to get rid of the restriction that the used rules keep the set of nodes invariant. This would enlarge the class of polynomial graph transformation units to which SAT solvers can be applied tremendously.
3. SAT solvers are mainly used to prove the correctness of some given specification. We would like to adopt the proof principle to the situation of graph transformation units to come up with a new verification technique for the framework of graph transformation.

References

1. Biere, A., Cimatti, A., Clarke, E., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Design Automation Conf., pp. 317–320 (1999)
2. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
3. Carnegie Mellon University, Graph Coloring Instances, `http://mat.gsia.cmu.edu/COLOR/instances.html`

4. Cook, S.A.: The complexity of theorem-proving procedures. In: Proc. Third ACM Symposium on Theory of Computing, pp. 151–158 (1971)
5. Corradini, A., Ehrig, H., Heckel, R., Löwe, M., Montanari, U., Rossi, F.: Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1, pp. 163–245. World Scientific, Singapore (1997)
6. Eén, N., Sörensson, N.: An extensible SAT solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. (eds.): Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
8. Ganai, M., Gupta, A.: SAT-Based Scalable Formal Verification Solutions. Series on Integrated Circuits and Systems. Springer, Heidelberg (2007)
9. Hölscher, K., Klempien-Hinrichs, R., Knirsch, P.: Undecidable control conditions in graph transformation units. Electronic Notes in Theoretical Computer Science 195, 95–111 (2008)
10. Kreowski, H.-J., Kuske, S.: Graph transformation units with interleaving semantics. Formal Aspects of Computing 11(6), 690–723 (1999)
11. Kreowski, H.-J., Kuske, S., Rozenberg, G.: Graph transformation units – an overview. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 57–75. Springer, Heidelberg (2008)
12. Kuske, S.: More about control conditions for transformation units. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 323–337. Springer, Heidelberg (2000)
13. Litovski, I., Métivier, Y., Sopena, É.: Graph relabelling systems and distributed algorithms. In: Ehrig, H., Kreowski, H.-J., Montanari, U., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation. Concurrency, Parallelism, and Distribution, vol. 3, pp. 1–56. World Scientific, Singapore (1999)
14. Marques-Silva, J., Lynce, I.: Towards robust CNF encodings of cardinality constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 483–497. Springer, Heidelberg (2007)
15. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: Design, Automation and Test in Europe, pp. 1341–1344 (2010)
16. Tseitin, G.: On the complexity of derivation in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logic, Part 2, pp. 115–125 (1968); Reprinted in: Siekmann, J., Wrightson, G. (eds.): Automation of Reasoning, vol. 2, pp. 466–483. Springer, Berlin (1983)