Constructing Shapely Nested Graph Transformations*

Frank Drewes[1], Berthold Hoffmann[2], and Mark Minas[3]

[1] Umeå Universitet, drewes@cs.umu.se
[2] Universität Bremen, hof@informatik.uni-bremen.de
[3] Universität Erlangen-Nürnberg, minas@informatik.uni-erlangen.de

Abstract. *Shapely nested graph transformation* is the computational model for DiaPlan, a language for programming with graphs representing diagrams that is currently being developed. This model supports nested structuring of graphs, graph variables, and structural graph types (*shapes*), but is still intuitive. In this paper, we show that the *construction* of shapely nested graph transformation steps can be reduced to solving the subgraph isomorphism and variable matching problem for the components of a structured graph, and devise restrictions of the transformation rules that improve efficiency. Shapes provide useful structural information about the graphs involved in a transformation step, and may therefore further improve efficiency.

1 Introduction

DiaGen [19] is a tool for implementing the *syntax* of diagram languages. The editors generated by DiaGen represent diagrams as graphs, perform scanning and structure editing by graph transformation, and parse their syntax according to a graph grammar. DiaPlan [13, 15], a programming language that is currently being designed by the authors, shall complement DiaGen by a tool for implementing the *semantics* of diagrams.

Shapely nested graph transformation [14] has been devised as the computational model of DiaPlan. This model is rather intuitive, although it supports powerful concepts that are not found in other graph transformation languages like Progres [23] and Agg [9]:

- Edges may contain graphs in a nested fashion, for a compositional *structuring* of graphs.
- *Graph variables* allow subgraphs of arbitrary size to be duplicated, compared, or deleted in a single transformation step.
- The admissible *shape* of graphs can be specified by syntactic rules that allow for type checking.

* The second author has been partially supported by the ESPRIT Working Group *Applications of Graph Transformation* (Appligraph).

In general, graph transformation is difficult to implement. So this paper studies the *construction* of shapely nested graph transformation steps. We show that this construction can be reduced to subgraph isomorphism and variable matching for the components of a graph, and propose restrictions of the transformation rules that improve efficiency, in particular by cutting down nondeterminism.

We investigate nested graph transformation (without shapes) in Section 2. Shapes are introduced afterwards, in Section 3, because shapely nested graph transformation leads to considerably different algorithms. Related and future work is discussed in Section 4.

2 Nested Graph Transformation

Our notion of graphs follows [14]; it is more general than usual ones: edges may connect an arbitrary number of nodes, not just two, and they may contain graphs in a nested fashion. We also distinguish a sequence of interface nodes at which graphs may be glued together.

More precisely, let L be a *ranked alphabet* where every symbol $l \in L$ comes with an *arity* $\mathrm{arity}(l) \geqslant 0$. The set $\mathcal{G}$ of *graphs* over L consists of sixtuples

$$G = \langle V, E, \mathrm{lab}\colon E \to L, \mathrm{att}\colon E \to V^*, \mathrm{cts}\colon E \to \mathcal{G}, p \in V^* \rangle^1$$

with finite sets V of *nodes* and E of *edges*, where every edge $e \in E$ has a *labelling* $\mathrm{lab}(e)$, a sequence $\mathrm{att}(e)$ of $\mathrm{arity}(\mathrm{lab}(e))$ *attached nodes*, and a *contents* $\mathrm{cts}(e)$, and where p designates a sequence of *points*.[2] It is required that p does not contain repetitions, and that the same holds for every $\mathrm{att}(e)$ $(e \in E)$. This is a well-known normal form which does not restrict the expressiveness of the concepts defined below.

By $\langle\rangle$ we denote the *empty graph*; the *handle graph* $\langle l \rangle$ of a label l consists of an edge e with $\mathrm{cts}(e) = \langle\rangle$ that is labelled with l and attached to $\mathrm{arity}(l)$ points. In a graph G, an edge e is called a *frame* if $\mathrm{cts}(e) \neq \langle\rangle$, and *plain* otherwise; e is qualified as a *k-ary l-edge* if it has k attachments and label l. G is called *plain* if it contains no frames, and k-ary if it has k points.

The tree-like nesting of frames in a graph G as above defines *nested edges*

$$\Delta_G = \{\varepsilon\} \cup \{ew \mid e \in E, w \in \Delta_{\mathrm{cts}(e)}\}$$

for selecting the *subcomponent* G/w contained in a nested edge $w \in \Delta_G$, and *assigning* a graph U to that nested edge, written $G[w \leftarrow U]$. By $G(w)$ we denote the *plain graph at* w: the one which is obtained from G/w by replacing the contents of each frame with $\langle\rangle$.

Two graphs G and H are *isomorphic*, written $G \cong H$ if they are equal up to the identities of their nodes and edges.

[1] V^* denotes the set of *sequences* over some vocabulary V. The *empty sequence* is denoted by ε.

[2] A more precise definition would define graphs by induction over the nesting depth of edges.

In contrast to notions of *hierarchical graphs* that are used for system modeling [8], our nesting concept is *compositional*: it forbids edges between components so that component assignment is possible. This is important for programming.

Figure 2 below shows three *control flow graphs*. Their nodes represent execution states, and their edges represent assignments, branches, and procedure calls; calls are frames that contain the control flow graph of the called procedure.

Variables. Let X be a ranked alphabet of *variable names* disjoint with L. A graph P over $L \cup X$ is called a *pattern* if all its *nested variables* (the nested edges labelled by X) are plain. By $\underline{P}$ we denote the *skeleton* of a pattern P where all variables have been removed.

Let P be a pattern with a k-ary nested variable $we \in \Delta_P$. The *replacement* of e in P/w by a k-ary graph U is then defined by gluing the attachments of e to the corresponding points of U, removing e, and assigning the result to P/w.

A pattern C with a single nested variable we is called a *context*. The *embedding* of a graph U in C is denoted as $C[U]$ and defined by replacing the nested variable we by U.

A function $\sigma\colon X \to \mathcal{G}$ is a *substitution* if it maps variable names onto graphs with the same arity.

The *instantiation* of a pattern P according to σ is obtained by the simultaneous replacement of all nested variables we in P by the graph $\sigma(\mathrm{lab}_{G/w}(e))$; the resulting *instance graph* is denoted by $P\sigma$.

Graph Transformation. Using the notions summarized above, transformation rules and steps can be defined in a similar way as in the area of term rewriting [16]. In doing so, it seems sensible to apply the same restrictions as for the rules of a term rewrite system: Their left-hand side patterns must not be variables, as such rules apply to every graph so that the system diverges, and their right-hand side patterns must not contain uninstantiated variables, since then arbitrary substructures have to be created "out of thin air".

A *(transformation) rule* $P \to_t R$ consists of two patterns P and R such that the left-hand side P is not a variable handle, and only variable names from P occur in the right-hand side R. Then t *transforms* a graph G into another graph H, written $G \Rightarrow_t H$, if t can be instantiated with a substitution σ, and embedded into some context C so that $G \cong C[P\sigma]$ and $H \cong C[R\sigma]$.

Example 3 (Control Flow Graph Transformation). In Figure 1 we show a rule l that performs a *loop transformation*, and a rule u that *unfolds* the body of a procedure call. Figure 2 shows two transformations of a control flow graph using l and u. The context for the first step equals the graph $G[f \leftarrow \langle D \rangle]$ (where f is the frame in G, and $\langle D \rangle$ is D's handle graph), and the substitution maps D onto the handle graph $\langle x := e \rangle$. For the second step, the context equals the graph obtained by replacing the frame f by a D-variable, and the substitution maps D onto H/f.

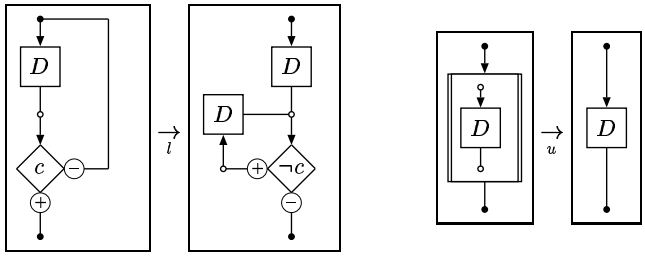

Fig. 1. Rules for loop transformation and for procedure unfold

Let l^{-1} and u^{-1} be obtained by interchanging the sides in rules l and u. Then u^{-1} is not a rule because its left-hand side is the handle graph $\langle D \rangle$. Indeed this rule could always be applied, to *fold* any control flow graph to a procedure call. However, l^{-1} is a transformation rule that could transform H back to G.

Note that a single graph transformation may affect arbitrary large subgraphs of the host graph. Every application of l *duplicates* the subgraph bound to the variable name D. Similarly, a rule *deletes* the subgraph bound to a variable name in its left-hand side if that name does not occur in its right-hand side. And, a rule may require to *compare* arbitrarily large subgraphs: the rule l^{-1} applies only to a host graph like H, where both D-variables on its left-hand side match isomorphic subgraphs. This is a rather complex applicability condition, and therefore often forbidden in applications based on term rewriting. So implementations of (shapely) nested graph transformation may also require that rules are left-linear, i.e. that every variable name occurs at most once on the left-hand side.

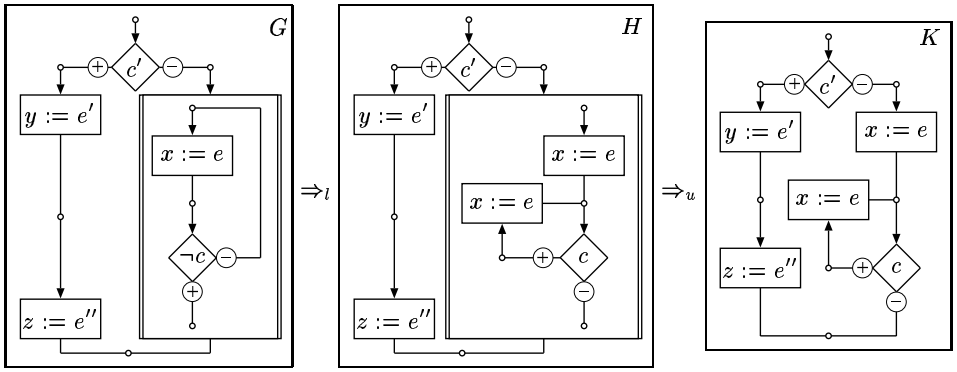

Fig. 2. Two transformation steps

Construction of Transformations. For a graph G and a rule $P \to_t R$, every valid transformation step can be constructed as follows:

1. FIND P's skeleton $\underline{P}$ in G, i.e. determine an *occurrence graph* O such that $\underline{P} \cong O \subseteq G/w$ for some nested frame w in G.
2. BIND P's variable names by a *matching substitution* σ such that $P\sigma \cong G/w$.[3]
3. REWRITE, i.e. determine H as $G[w \leftarrow R\sigma]$.

For plain graphs, FIND and BIND together constitute the *graph matching problem* studied in [20]. Obviously FIND cannot be solved without solving the *subgraph isomorphism problem*, which is NP-complete. BIND is somewhat easier: Using the techniques presented in [20] it is solvable in polynomial time. Once this has been done, REWRITE is easy.

A closer look at FIND and BIND reveals that the main difficulty is to match the plain graph $P(u)$ against $G(wv)$, for nested frames u and wv. From such local solutions the required global one can be constructed rather efficiently. Thus, nesting alone makes FIND and BIND more efficient if graphs consist of many small components, since these can be considered in isolation. (Notice the benefit of compositionality.)

However, BIND may still produce an exponential number of matching substitutions for some occurrence. If evaluation is done with backtracking (as in PROLOG), all of them may have to be tried out, eventually. It is thus important to cut down the number of matches. Fortunately, rules may be restricted so that they have at most one matching substitution for any occurrence:

Theorem 1. *Let $\underline{P} \cong O \subseteq G/w$, where no variable in $P(\varepsilon)$ is attached to a point. The* BIND *step can yield at most one matching substitution σ so that $P\sigma \cong G/w$, provided that every $u \in \Delta_P$ satisfies one of the following properties:*

1. *The variables in $P(u)$ have pairwise disjoint sets of attached nodes and $G(wu)$ is connected,*
2. *$P(u)$ contains at most one variable,*
3. *$P(u)$ is a handle graph or contains no variable at all.*

The conditions 1–3 are ordered with increasing strength. Rule u in Figure 1 of Example 3 satisfies condition 3. So do all the rules used for specifying a graphical version of Quicksort in [5]. (Actually also the right-hand sides of all rules in that paper satisfy condition 3.) This indicates that even rigidly restricted variable concepts suffice for many nontrivial programming situations.

Rule l^{-1} in Figure 1 of Example 3 violates condition 2 since the variables of its left-hand side share one of their attached nodes. Rule l in that figure, although satisfying condition 2, fails as the D-variable on its left-hand side is attached to

[3] To be precise, one must extend P by a *hole variable* h that is attached to P's points. Then $G[w \leftarrow \sigma(\mathrm{lab}_P(h))]$ determines the skeleton of the context C for the transformation. Furthermore, substitution σ has to be consistent with the occurrence of $\underline{P}$ in G which has been found in the FIND step, i.e., the occurrence morphism has to be a submorphism of the isomorphism $P\sigma \to G/w$.

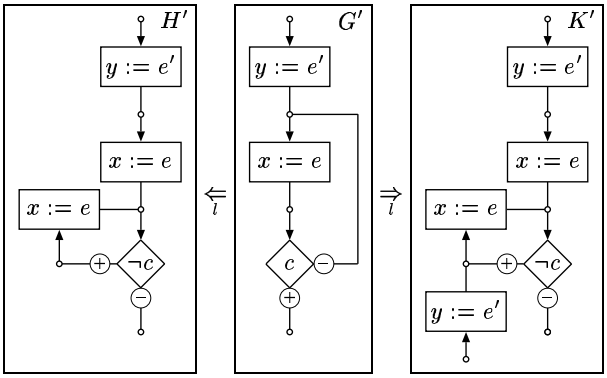

Fig. 3. Two loop transformations

a point. Figure 3 illustrates that the occurrence of l's pattern skeleton may then be transformed with different instantiations. Our intuition about control flow graphs may mislead us to believe that l transforms G' in just one way, yielding H'. In that transformation, the subgraph $\langle y := e' \rangle$ in G' is part of the context wherein the instance of l is embedded. However, this subgraph may also belong to the instantiation of l. Then, the transformation yields the graph K', where the subgraph $\langle y := e' \rangle$ is duplicated, but introduced as dead code that will never be executed.

3 Shaped Nested Graph Transformation

Figure 3 points out an inherent feature of nested graph transformation: All graphs and pattern over the label alphabets may occur as host graphs, in substitutions of variables, and in rules. The construction of transformations has to cope with the general case, even if the graphs that actually occur have particular properties. (For example, all control flow graphs are connected, and have a unique start node.) These properties could be used to construct transformations more efficiently.

Therefore we devise rules specifying the syntax of graphs in a context-free way so that graphs and patterns can be checked against these rules.

Syntax Graphs. Let N be a ranked alphabet of *nonterminals* disjoint with the vocabularies L and X. Graphs over $L \cup N$ are called *syntax graphs* if their N-edges are plain (as for patterns).

Let Σ be a finite set of *syntactic rules* of the form $n ::= R$, where n is a nonterminal, and R is a syntax graph with arity(n) points. A *direct derivation* of a syntax graph G to a syntax graph H under Σ, written $G \Rightarrow_\Sigma H$, is obtained

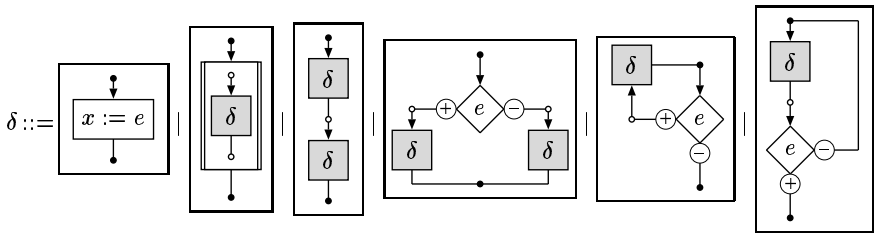

Fig. 4. A grammar for structured control flow graphs

by replacing a nested n-edge we in G by R, where $n ::= R$ is a syntactic rule in Σ. The reflexive and transitive closure of $\Rightarrow_\Sigma$ is denoted by $\Rightarrow_\Sigma^*$.

Example 4 (Grammar for Structured Control Flow Graphs). Figure 4 shows the rules defining the syntax of *structured control flow graphs* that are built (from left to right) over assignment and procedure call by sequential composition, conditional statement, pre-checked and post-checked loop. In this example, δ is the only nonterminal. We write $\delta ::= R_1 | \cdots | R_n$ to abbreviate several rules $\delta ::= R_1, \ldots, \delta ::= R_n$ for the same nonterminal.

Syntactic rules specify *hyperedge replacement*, which derives one of the best-studied classes of context-free graph languages (see [11, 4] for details). Such rules allow to define recursive "algebraic" data types of functional and logical languages, and sophisticated pointer structures like cyclic lists, or leaf-connected trees, which cannot be defined in imperative languages (see also [10]).

Membership in these languages is decidable:

Theorem 2. *The question "$\langle n \rangle \Rightarrow_\Sigma^* G$?" is decidable for all syntax graphs G.*

Shaped Graphs and Patterns. For the rest of this paper, we fix a finite set Σ of syntactic rules over the nonterminals N, and use it to specify shapes of graphs. (The term *shape analysis* is used for inferring properties of pointer structures in imperative programs [22].) We assume that every variable name $x \in X$ is *typed* with a nonterminal $\mathrm{type}(x) \in N$.

The *shape* $\lceil P \rfloor$ of a pattern P is the syntax graph obtained by relabelling every x-variable in P by its type.

A pattern P is *shaped* by some nonterminal $n \in N$ if $\langle n \rangle \Rightarrow_\Sigma^* \lceil P \rfloor$ (or just *shaped* if n is not relevant). A shaped context C is called *n-context* if its unique variable is of type $n \in N$. (Note that in general, C may be shaped by another nonterminal.) A substitution σ is *shaped* if the graph $\sigma(x)$ is shaped by $\mathrm{type}(x)$ for all $x \in X$.

Shapely Transformation. We now refine graph transformation so that it preserves shapes.

A rule $P \to_t R$ is *shapely* if P and R are shaped by the same nonterminal, say n. Then t *transforms* a graph G into another graph H, written $G \Rrightarrow_t H$, if t can be instantiated with a shaped substitution σ, and embedded into some n-context C so that $G \cong C[P\sigma]$ and $H \cong C[R\sigma]$.

Our definition of shapes is consistent, since the result of a shapely transformation is a shaped graph again (see [14] for the straightforward proof).

Theorem 3. *In the situation above, G and H are of the same shape as C.*

Altogether, shapes set up a *type discipline* that can be *statically checked*: Theorem 2 allows to confirm whether a set T of transformation rules is shapely or not. If the rules are shapely, and a graph G (the "input") has been checked to be shaped, Theorem 3 guarantees that every transformation sequence $G \Rightarrow^* H$ will yield a shaped "output" graph H. Type-checking between the steps ("at runtime") is not necessary.

Example 5 (Shapely Control Flow Graph Transformations). The patterns in Figure 1 of Example 3 are shaped according to δ, so the rules l, U, and l^{-1} are shapely. The contexts of the transformations in Figure 2 are δ-contexts, and the matching substitutions are shaped so that the transformations are shapely as well.

In Figure 3, the transformation $G' \Rightarrow_{l^{-1}} H'$ is shapely. However, the transformation $G' \Rightarrow_l K'$ is not shapely, because neither the context, nor the substitution used in it are shaped.

Construction of Shapely Transformations. As pointed out earlier, matching a pattern P against (a subcomponent of) G can be done by computing matches of the plain patterns $P(u)$ against the plain graphs $G(v)$. The obtained results can be combined using a top-down or bottom-up procedure. Let us briefly describe the bottom-up case; the top-down procedure is similar. Assume we are given a matching algorithm for plain graphs. In the first step we consider all $u \in \Delta_P$ such that P/u is plain. For every $v \in \Delta_G$ we use the given algorithm to determine whether $P(u) = P/u$ matches $G(v) = G/v$. Next, we consider all $u \in \Delta_P$ such that P/u has nesting depth 1 and repeat the procedure, applying the given algorithm to $P(u)$ and each $G(v)$ to find out whether P/u matches G/v (making use of the already computed information). This is repeated until we reach the root of P.

Obviously, the recursive part of this procedure can be implemented efficiently. It can be used to find a single matching, but also to enumerate all possibilities if an evaluation strategy involving backtracking is desired or needed.

Thus, the complexity is mainly determined by the complexity of matching $P(u)$ against $G(v)$. As mentioned above, this problem generalizes the subgraph-isomorphism problem and is thus not efficiently solvable unless P=NP. However,

in the presence of shapes one does not need to solve the problem in all its generality. Below, we briefly discuss some possibilities to gain efficiency.

Functional languages correspond to the (very restrictive) case where the $G(v)$ are taken from a finite set. More precisely, a term $f(t_1, \ldots, t_k)$ is represented by a graph consisting of an f-labelled frame containing a graph with k points and k frames $e_1, \ldots, e_k$. Frame e_i is attached to the ith point and represents, recursively, the subterm t_i. In this case, plain graph matching can be done in constant time and thus occurrences can be found efficiently.

In more general cases, one can make use of algorithms which compute the set of all derivation trees of $G(v)$ (with respect to the shape grammar; see [4] for a discussion of derivation trees). If this can be done in polynomial time, matchings can usually be found in polynomial time as well. This is because P is also shaped, and hence each $P(u)$ can be represented by its set of derivation trees. The latter can be matched against the derivation trees of $G(v)$, which essentially reduces the problem to the question of tree matching.

Unfortunately, parsing of hyperedge-replacement languages is NP-complete in general [17], so this approach is not always useful. However, restrictions under which parsing becomes polynomial have been studied by Lautemann, Vogler, and Drewes [18, 24, 3]. Let us discuss the way in which the algorithm by Lautemann can be used. The algorithm is a generalization of the well-known parsing algorithm by Cocke, Kasami, and Younger [25]. It can be reformulated in such a way that it returns a representation of all derivation trees of the (plain) input graph. Since there may be exponentially many derivation trees, sharing is used to represent them on polynomial space. In other words, the returned representation of the forest of derivation trees is a directed acyclic graph (dag).

To explain the structure of this *derivation dag* let us first discuss a possible representation of a derivation tree of an n-shaped plain graph $H = G(v)$. Let $n ::= R$ be the rule applied to $\langle n \rangle$ in the first step of the derivation, where R contains k nonterminal edges $e_1, \ldots, e_k$. Hence, H is obtained from an isomorphic copy R' of R by replacing each e_i with a graph $H_i \subseteq H$ which is derivable from $\langle \mathrm{lab}_R(e_i) \rangle$. To account for this fact, the derivation tree has a root node v that represents the pair (n, H) and a frame with contents R' which is attached to $v_1, \ldots, v_k, v$ where $v_1, \ldots, v_k$ are the root nodes of the derivation trees obtained recursively from the derivations $\langle \mathrm{lab}_R(e_i) \rangle \Rightarrow_\Sigma^* H_i$. Now, the derivation dag D representing the set of all derivation trees of H is obtained by taking the forest of all these derivation trees and identifying all nodes which represent the same pair (n', H').[4] Under the conditions of [18] the algorithm by Lautemann computes D in polynomial time. In particular, the size of D is polynomial. The mentioned conditions basically state that the graphs generated by the shape grammar fall apart into at most a constant number of components by deleting k nodes, where k is the maximum type of nonterminal edges. The shape grammar for control flow graphs discussed in Example 4 satisfies this requirement.

[4] Here, "the same" really means that the graphs are identical, not just isomorphic, since we are working with concrete subgraphs of H.

As mentioned above, the considered component $P(u)$ of the pattern graph can be turned into a corresponding derivation dag D' as well, since the allowed patterns are shaped with respect to the same shape grammar. Then it is not hard to find all possible matchings by a bottom-up or top-down procedure that runs in time $O(d \cdot d')$, where d and d' are the sizes of D and D'. In fact, it should be mentioned that this performs even the BIND step discussed in Section 2 since each matching found in this way associates every variable in $P(u)$ with a node in D. The graph represented by this node is the one to be bound to the variable.

Even though it runs in polynomial time, from the point of view of efficiency the procedure just described has the drawback that the derivation dag D' must be reconstructed after each step. This is because the application of a rule may invalidate some of the derivation trees represented in D while on the other hand creating new possible derivation trees. However, the specific derivation tree to which D' is mapped is, by the shapedness of rules, turned into a correct derivation tree of the resulting graph by performing the corresponding replacement on the level of trees. If we have unique derivation trees, we can therefore represent graphs by their derivation trees throughout and reduce transformation to the (much more efficient) replacement of subtrees of derivation trees. As such, this option is not very realistic because the restriction to grammars with unique derivation trees would be much to strong. However, unlike the class of grammars considered by Lautemann, those studied in [24, 3] have unique derivation trees modulo a certain type of associativity and commutativity rules. We do not wish to go into the details here, but it seems quite clear that the technique sketched above can be extended in order to work in this, somewhat more realistic case as well. For instance, the syntax rules in Example 4 have unique derivation trees, modulo the third rule that expresses associativity of sequencing in control flow graphs.

It is worth pointing out that implementations may select, for each $G(v)$, the matching algorithm which is most suitable for its shape. Hence, a mix of different matching algorithms may be applied to match P against G. For example, if Σ contains shapes n and m which satisfy the restrictions of matching algorithms A respectively B we may use algorithm A for n-shaped components whereas B is used for m-shaped components. In this way, efficient algorithms can be applied whenever possible without restricting the language in general.

4 Conclusions

Nested graph transformation is closely related to other ways of graph transformation. On the one hand, it lifts the substitutive transformation of flat graphs [20] to nested graphs; on the other hand, it extends hierarchical graph transformation [5] with respect to the use of variables. Hierarchical graph transformation has in turn been defined by lifting double pushout graph transformation [6] to hierarchical graphs (for injective occurrences, as studied in [12]). The paper [8] defines double pushout transformation of hierarchical graphs where edges may cross the border of components (called *packages*), yet without investigating un-

der which conditions the hierarchy stays intact. A general framework for the transformation of (many kinds of) hierarchical graphs is developed in [2]. Shape specifications (for plain graphs) have been considered in Structured Gamma [10].

This paper indicates that nested graph transformation is not only intuitive and expressive, but may also be implemented in a reasonable way. Nesting helps for the general case, and Theorem 1 gives reasonable conditions that eliminate the overhead for binding. Shapes may even improve these results since the conditions on the pattern graphs may be relaxed if the host graphs have structural properties that simplify the task of finding matchings and bindings. The investigation of such structural properties is an important and interesting question for future work.

Several other questions remain to be studied as well. Let us focus our discussion on shapes here. The syntactic rules can be extended by *embedding rules* [19], without sacrificing Theorem 2. Then one can also specify non-context-free shapes like, for instance, general control-flow diagrams (of "spaghetti programs"). The parsing algorithm explained in [1] allows more general syntax rules. However, it is unclear how efficient it will be in practice, and more important, whether it is consistent with the operations of context embedding and variable instantiation that are fundamental with our way of graph transformation.

Acknowledgment We thank the referees for useful comments.

References

1. P. Bottoni, A. Schürr, and G. Taentzer. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In M. M. Burnett et al., editors, *Proc. VL'2000*. IEEE Press, 2000. Full version appeared as Technical Report SI-2000-06 at the University of Rome.
2. G. Busatto. *An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation*. Dissertation, Universität Paderborn, 2002. Forthcoming.
3. F. Drewes. Recognising k–connected hypergraphs in cubic time. *Theoretical Computer Science*, 109:83–122, 1993.
4. F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In Rozenberg [21], chapter 2, pages 95–162.
5. F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, to appear 2002. (A short version appeared in number 1784 of Lecture Notes in Computer Science, pages 98–113, 2000).
6. H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science, pages 1–69. Springer, 1979.
7. G. Engels, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Specification and Programming*. World Scientific, Singapore, 1999.
8. G. Engels and R. Heckel. Graph transformation as a conceptual and formal framework for system modelling and evolution. In U. Montanari, J. Rolim, and E. Welz, editors, *Automata, Languages, and Programming (ICALP 2000 Proc.)*, number 1853 in Lecture Notes in Computer Science, pages 127–150. Springer, 2000.

9. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In Engels et al. [7], chapter 14, pages 551–603.
10. P. Fradet and D. Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2/3):263–289, 1998.
11. A. Habel. *Hyperedge Replacement: Grammars and Languages.* Number 643 in Lecture Notes in Computer Science. Springer, 1992.
12. A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
13. B. Hoffmann. From graph transformation to rule-based programming with diagrams. In M. Nagl, A. Schürr, and M. Münch, editors, *Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (*AGTIVE*'99), Selected Papers*, number 1779 in Lecture Notes in Computer Science, pages 165–180. Springer, 2000.
14. B. Hoffmann. Shapely hierarchical graph transformation. In *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 30–37. IEEE Computer Press, 2001.
15. B. Hoffmann and M. Minas. Towards rule-based visual programming of generic visual systems. In N. Dershowitz and C. Kirchner, editors, *Proc. Workshop on Rule-Based Languages*, Montréal, Quebeq, Canada, Sept. 2000.
16. J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.
17. K.-J. Lange and E. Welzl. String grammars with disconnecting or a basic root of the difficulty in graph grammar parsing. *Discrete Applied Mathematics*, 16:17–30, 1987.
18. C. Lautemann. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27:399–421, 1990.
19. M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 2002. To appear.
20. D. Plump and A. Habel. Graph unification and matching. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science, pages 75–89. Springer, 1996.
21. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations.* World Scientific, Singapore, 1997.
22. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.
23. A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Engels et al. [7], chapter 13, pages 487–550.
24. W. Vogler. Recognizing edge replacement graph languages in cubic time. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. Fourth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, volume 532 of *Lecture Notes in Computer Science*, pages 676–687. Springer, 1991.
25. D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:189–208, 1967.