

Time and Space Issues in the Generation of Graph Transition Systems

Arend Rensink

*Department of Computer Science, University of Twente
P.O.Box 217, 7500 AE, The Netherlands
rensink@cs.utwente.nl*

Abstract

GROOVE is a tool for the automatic generation of graph transition systems from graph grammars. In this type of tool, both memory and time performance are of prime importance. In this paper we discuss the implementation techniques used for optimising the tool in this regard, and we list possible future improvements.

1 Introduction

The tool reported in this paper is being developed in a project, called GROOVE (Graphs for Object-Oriented Verification), which aims to use graph-based modelling as a basis for the specification and verification of (especially) software systems. The verification approach being pursued is model checking of *graph transition systems* (GTSs) generated by graph grammars. GTSs are state/transition systems in which the states are graphs and the transitions are derivations from the graph production rules.

Model checking generally involves two main steps: generating and storing the state space, and checking temporal properties over the state space. Although there is potential benefit in doing these steps in combination, here we concentrate on the first. It should, however, be noted that in the context of graph transition systems there are quite important open issues in the second issue, in particular the fact that temporal propositional logic is not strong enough to express properties regarding the identity of nodes, such as “this particular node will never be deleted;” one has to move to a stronger logic for this, which means that fundamental questions about model checking have to be reconsidered. As we will see (Section 4), by varying the state space exploration strategy one can already check a limited set of temporal properties, namely those that can be expressed as state invariants or state-based reachability properties.

Since, to some degree, the possible implementation techniques are determined by the formal definitions of graphs and transformations, we briefly review the graph transformation approach chosen in GROOVE.

*This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs*

Graphs Graphs are flat (i.e., without hierarchical structure), untyped and without attributes. Edges are binary, directed and labelled; parallel equi-labelled edges are forbidden. Nodes have no labels. In using the tool, it is often convenient to simulate node labels or node types through (labelled) self-edges.

Transformations GROOVE basically supports single-pushout production rules with negative application conditions (see [6]). Non-injective matchings are allowed by default; of course, NACs can be used to specify injectivity where required. Currently there is no option such as in AGG ([8]) to automatically check the extra application conditions imposed by the double-pushout approach (rule overlap on deleted and preserved nodes and the dangling edge condition, see [4]); adding such an option is one of the planned (minor) tool extensions.

A non-standard feature is that, like PROGRES [20], GROOVE transformation rules allow *regular expressions* over edge labels to appear on rule edges that are shared by the left and right hand side, i.e., that are neither created nor deleted. While not increasing the expressive power of the formalism¹ it obviously enables a much more compact representation in those cases where transformations depend on non-trivial regular structures.

The remainder of this paper is structured as follows. In Section 2 we briefly go into the expected usage of the tool. In Section 3 we describe the implementation techniques used to get a reasonable time and space performance. In Section 4 we go into the exploration strategies supported by GROOVE, and their implications for model checking.

Related work. The current paper discusses some implementation issues in GROOVE (version 0.2.4). Other aspects of the tool were described before in [17], where we concentrated on the *visual interface*, and in [18], where we compared the *verification approach* pursued in this project, which is entirely graph-based, with that proposed in [21] of re-using standard model checking technology.

The functionality offered by the GROOVE state space generator has an obvious overlap with other existing tools for graph transformation, such as AGG [8], PROGRES [3] and ATOM3 [5], in that the core of all these tools is an engine to apply graph production rules. (Note that this is different from FUJABA [15], where the production rules are used for code generation instead.) In fact, these other tools are in many ways more mature and incorporate more sophisticated techniques for, e.g., graph matching and confluence detection. However, the emphasis in GROOVE on computing and storing complete state spaces is, to our knowledge, unique.

2 Tool chain

Figure 1 gives an overview of the (planned) GROOVE tool set. Rectangles represent tool components or processes and ellipses represent the data being processed.

¹ Every transformation rule with regular expressions can be transformed to a set of rules effecting the same transformation — although generally in more steps — by generating special edges that express the relation defined by the regular expressions.

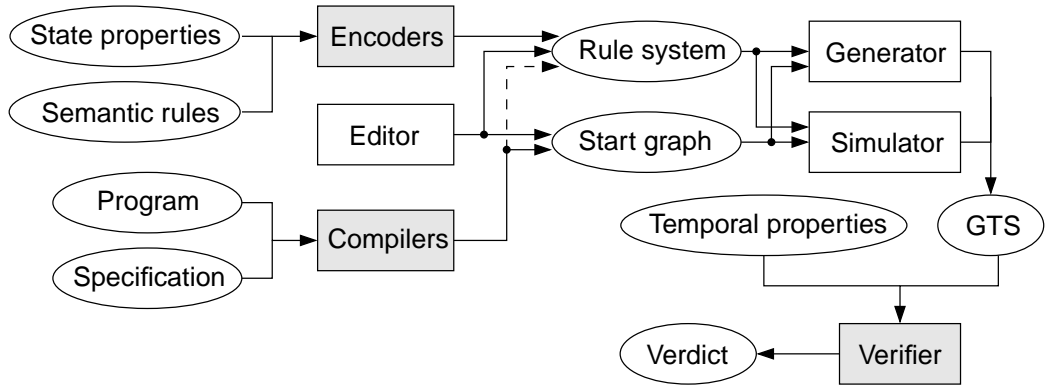


Fig. 1. Schema of the intended tool chain

The shaded components have yet to be implemented. We discuss the functionality of the various (existing and planned) components.

- The *editor* component is a graph editor in which graphs and production rules can be constructed visually. This is convenient for constructing small examples; moreover, since the editor can also be used to modify existing graphs and rules, it can also be used for experimentation purposes.
- The *generator* component is the heart of the current tool. It takes a rule system and start graph and explores the resulting graph transition system, by recursively computing all rule applications and identifying the reachable graphs up to isomorphism. The exploration may in some cases stop before the state space has been searched exhaustively; see Section 4. The output of the generator is the (explored fragment of the) graph transition system, in the form of a graph of which the nodes correspond to state graphs and the edges to rule applications.
- The *simulator* component is essentially a GUI upon the generator; it offers the additional functionality of walking through the state space step-by-step, trying out rule applications and visualising the resulting graphs. Furthermore, the simulator can save selected state graphs, besides the graph transition system.
- The planned *encoder* components serve to construct graph production rules from various sources. For instance, the *state properties* in the figure refer to invariants and bounding properties that can be formulated as graph embeddings (see Section 4). With *semantic rules* we mean the operational semantics of some specification formalism or programming languages; the idea is that such rules can be encoded in a fixed set of language-dependent but program-independent graph production rules, which then only have to be complemented with a specific start graph or possibly some additional program-dependent rules in order to analyze a particular program.
- The planned *compiler* components are intended to build graphs from individual specifications or programs. These graphs can then serve as start graphs in a rule system that reflects the semantics of the specification formalism or the programming language (see above). As a proof-of-concept, such compilers have been built for fragments of Java [13,11] and Java Bytecode [2].

	append (4:8)	phil (10)	mutex (3:2:0)
states (#)	31104	32903	262054
transitions (#)	116658	271634	620284
time (s)	212	199	162
space (MB)	13,9	24,8	88,7
nodes/graph (avg)	37.7	20.0	5.1
edges/graph (avg)	113.8	55.1	14.3

Fig. 2. Illustrative cases of generated state spaces

- The planned *verifier* component is a model checker for temporal properties. The logic we plan to support is a mixture of predicate (rather than propositional) logic and temporal operators, extending LTL as used in, e.g., SPIN [10]. See [16] for a first proposal. The *verdict* will either be “pass” if the verifier decides that a property holds, or a counter-example otherwise.

The tool set is implemented in Java, and currently consists of 11 packages, comprising some 300 classes and 50,000 lines of code. The choice of Java as a programming language, although certainly beneficial for a research prototype, has some unavoidable consequences on performance, as we will discuss in the next section.

3 Implementation techniques

In this section we describe the techniques we have employed to improve the time and space performance. We illustrate the effect of these techniques on the basis of some case studies carried out in [18]. Figure 2 contains the overall data of these cases, as reported there.² Briefly, `append` models a concurrent append operation, in this case called four times concurrently upon a list of length 8; `phil` models a system of 10 dining philosophers; and `mutex` models the mutual exclusion algorithm of [9], bounded to three processes and two resources.

3.1 Time consumption

The state space exploration as implemented in GROOVE can be divided into the following three phases: computing rule matchings, applying rules, and performing isomorphism checks. We discuss each of these phases in some detail.

Computing rule matchings. The problem of graph matching is well-known to be NP-complete in the size of the graph to be matched, which in this case is the left hand side of the graph production rules. Fortunately, left hand sides are typically small, at least compared to the graphs under transformation. The complexity is worsened by negative application conditions (which fortunately are typically also

² The memory usage reported here is somewhat larger than that in the original paper because we have taken a more accurate figure for the initial configuration.

	append		phil		mutex	
	s	%	s	%	s	%
graph matching	104	49%	55	28%	60	37%
rule application	38	18%	45	23%	53	32%
iso check	78	37%	95	48%	52	32%
total	212		199		163	

Fig. 3. Time consumption for the cases of Figure 2

small) and by regular expressions occurring on transition labels (see Section 1). Finally, obviously the complexity is linear in the number of transformation rules.

Currently we have undertaken very little effort to optimise the rule matching phase; for future improvements we hope to benefit from the experience gained in other graph transformation tools. In particular, the following seem worth pursuing:

- **Confluence issues.** If two rule matches are parallel independent, then each of them can immediately be carried over to the state reached by applying the other rule; as a result, in that next state the matching phase can be simplified. In fact, in each new state one would only have to search for matchings involving newly generated nodes or transitions. (A similar strategy is implemented in GREAT [1].) In the presence of negative conditions, however, parallel independence is itself not trivial to check. There we hope to apply a technique such as described in [22], who actually also store *failed matches* if the failure is due to the violation of a NAC; these failed matches are checked again in the next state to see if the violation is still there.
- **Control issues.** In the kind of applications that GROOVE is targeted for, viz. software verification, graph matching is arguably less complex than in general. This in itself is due to two reasons: the graphs under transformation will tend to be deterministic (i.e., have at most one outgoing edge for each edge label), since they model concrete memory structures; and the potential matches are sharply defined by the “locus of control” of the system being modelled — typically, the reference object for a process. It therefore may be worthwhile to allow classes of rules to specify (to some degree) their own matching strategy. If, for instance, a given rule applies if and only if program counter has a given value, then a match can possibly be found very efficiently.

Rule application. Constructing the derivation is a relatively simple matter of copying and modifying the start graph according to the rule by adding, removing and merging nodes and edges as required. Subsequently both the target state and the transition are coded in a more space optimised representation (see below).

Isomorphism check. An important step in the state space generation is to match each newly generated states to the existing ones up to isomorphism. Like graph matching, this is computationally expensive: the precise complexity class of graph isomorphism is unknown but thought to be strictly between P and NP — al-

	append		phil		mutex	
	#	%	#	%	#	%
certified equal	85555		239622		358871	
isomorphic	85555	100,0%	238732	99,6%	358231	99,8%
equal	12579	14,7%	84668	35,3%	128590	35,8%

Fig. 4. Isomorphism checking for the cases of Figure 2

though, as for matching, the problem becomes easier when the graphs’ outdegree is bounded. To make matters worse, in contrast to matching, the graphs we need to compare are the full states and hence tend to be large. Furthermore, this phase is quadratic in the number of states.

Fortunately there are approximations that turn out to do well in practice. Most importantly, we can “over-approximate” isomorphism by computing so-called *graph certificates*, which are necessary predictors for isomorphism; that is, two certificates are equal if, but not only if, the corresponding graphs are isomorphic. A graph certificate in GROOVE is the sum of all node certificates; a node certificate is computed recursively, as a function of the certificates of the neighbouring nodes and the edge labels used to reach them, iterated until the number of node partitions (i.e., sets of nodes with the same certificate) stabilizes. The construction of the graph certificate as a sum of node certificates has an added advantage: if two graphs have equal certificates then we can already construct a relation within which any isomorphism, if it exists, must lie: viz. by pairing off nodes with the same certificate.

It would be very useful to have, in addition, a good *sufficient* predictor for isomorphism; in other words, an “under-approximation”. For this purpose, we are currently using simple *equality*: if two graphs are equal, i.e., have equal sets of nodes and edges, then they are trivially isomorphic, under the identity.

In Figure 4 we have gathered some statistics regarding the quality of the approximation, based on the cases in Figure 2. The “certified equal” row lists the number of certificate comparisons done in the course of the state space generation that yield “true”. Only in this case do we need to go on with other checks. The second row lists the number of cases in which the graphs were indeed isomorphic, and the third row the number of cases in which they were even equal. We have expressed the accuracy as the ratio of the number of isomorphic pairs to the number of certified equal pairs, resp. the number of equal pairs to the number of isomorphic pairs.

Evaluation. In [18] we have seen that, on comparable examples, traditional model checking techniques outperforms GROOVE by a factor of 10 or more. We think that, for cases with little dynamic behaviour, there is no real hope of closing the gap entirely; the value of our approach lies in cases where the static nature of traditional model checking prevents it from modelling a given problem at all. Using the improvements described above, however, we think that close to a 50% time improvement is feasible, even while staying in Java.

		append			phil		mutex	
delta (average)		7.0			2.8		3.5	
boundary images (average)		2.9			1.8		1.6	
		B	MB	%	MB	%	MB	%
States	object	80	2,4	17%	2,5	10%	20,0	23%
	delta	4	0,8	6%	0,4	1%	3,5	4%
	gts	48	1,4	10%	1,5	6%	12,0	14%
	total		4,6	33%	4,4	18%	35,5	40%
Transitions	object	40	4,5	32%	10,4	42%	23,7	27%
	boundary	4	1,3	9%	1,9	7%	3,8	4%
	gts	32	3,6	26%	8,3	33%	18,9	21%
	total		9,3	67%	20,5	82%	46,4	52%
Open states		32	0,0	0%	0,0	0%	6,8	8%
Total			13,9		24,9		88,7	

Fig. 5. Space consumption for the cases of Figure 2

3.2 Space consumption

The memory space used to store the GTS during the generation process can be divided into three categories: memory used for states, for transitions, and for open states. An overview for the cases we are considering is given in Figure 5.

States. Since states are graphs, storing them entails storing their nodes and edges. Programming in Java, we followed the natural approach by defining classes `Node` and `Edge`, and having graphs store references to them. Since we re-use the node and edge objects among graphs, the memory space almost entirely goes to storing references, of 4 bytes each.

Rather than storing each state completely anew, we have chosen to store only *changes* or *deltas* between states, taking advantage of the fact that each individual delta, being the result of a production rule application, is of limited size; an idea which was proposed before by Mens [14] and has been thoroughly implemented in the GRAS database [12]. Thus, each state actually keeps a reference to its so-called *basis*, which is the graph with respect to which the delta is calculated, combined with arrays of *added* and *removed* elements (i.e., node and edge references). Furthermore, each state keeps a *cache* where, as long as the available space allows, the node and edge sets remain stored in a more space- but less time-consuming representation; using classes from the `java.lang.ref` package, this is implemented in such a way that the garbage collector clears the caches whenever memory grows short. Finally, in order to ensure an upper bound to the time required for reconstructing a particular graph, from time to time we store the entire graph permanently (which we call *freezing* the graph). The criterion for when to freeze a graph is decided on the basis of the *reconstruction depth* of a state, defined as the sum of the length of the chain of graphs back to the last frozen graph, plus for each element of the chain the lengths of the arrays of added and removed elements.

Obviously, there is a time penalty associated with this storage technique, linear in the number of times that states have to be reconstructed from the deltas. In the state space generation process, this only ever happens during the isomorphism check, when two graphs are found to have the same certificate. It can therefore be expected to be more frequent as the number of isomorphic states grows.

The storage space needed for each state then amounts to 80 bytes fixed plus 4 bytes per stored (delta) element, with a breakdown as reported in Figure 5. In addition, for the purpose of actually building the graph transition system, we need to collect the states that make up the graph transition system. For this purpose we use a Java `HashMap` from the certificates discussed in Section 3.1. The values in the map are either single graphs (if the certificate actually uniquely identifies a graph) or arrays of graphs with the same certificate. Each filled bucket in a `HashMap` takes 32 bytes and each unfilled bucket 4 bytes; each certificate key (encoded as an `Integer`) takes another 16 bytes. Assuming a load factor of 50% and a completely uniform distribution of states over certificates (which, as we have seen above, is not so unrealistic) we arrive at an average of 48 bytes per stored state.

Transitions. Apart from the states, we also explicitly store the transitions. It should be remarked that for some purposes this is actually superfluous; for instance, when checking an invariant or reachability property. Such properties can be checked on individual states; no transition information is required to decide whether they are valid. On the other hand, if a violation is found, it is imperative to be able to give the trace leading up to the relevant state; for this purpose we must keep track of the transitions.

The minimal amount of information for each transition is obviously its source and target state, and the rule applied. However, rule application may well be non-deterministic in that a given rule applies in a given graph more than once; and it is even possible that these applications lead to exactly the same target state. Thus, it would seem that we want to be able to recall the matching as well as the derivation morphism. This, in turn, would give rise to a high memory cost per transition.

Fortunately we can do better than storing entire mappings. First of all, for the matching, it is sufficient to store only the *images*, provided we do so in a pre-defined order. The next insight, however, is that in order to characterise the transformation of a graph it is enough to know where something actually *changes* in that graph. For instance, if a rule only adds an edge, knowing the image of that edge's source and target nodes suffices to reproduce the effect of the transformation; the rest of the rule serves as a positive application condition.³ Finally, given the matching (in this reduced form) the derivation morphism is uniquely defined up to automorphism of the target state, and so does not need to be stored at all.

The storage space needed for each transition then amounts to 40 bytes fixed (the transition object) plus 4 bytes per stored boundary image (the actual content).

³ In fact, we conjecture that, in terms of [7], this can be understood as a categorical construction: instead of the image of the left hand side we take the image of the *boundary* of the left hand side, being the source object of the initial pushout complement of the rule morphism.

Collecting the transitions of the transition system is similar to the states, although here we use a `HashSet` rather than a `HashMap`. The average overhead here comes down to 32 bytes per stored transition.

Open states. In addition to the set of all states, during the generation process we also keep track of the set of states newly generated but not yet explored fully (which are called *open* in the tool). If the exploration strategy includes a bounding condition (see Section 4), it is possible that such open states are never explored, and remain in the set of open states for the entire duration of the state space generation. This is for instance the case for the mutex example. The open states are again kept in a `HashMap`, and so take up an average of 32 bytes.

Evaluation. Java is not optimised towards memory consumption. For instance, each object, including each array, uses 8 bytes to store its actual type, and moreover, object sizes are “rounded” up to a multiple of 8 bytes. Furthermore, in the current implementation we have not gone to any great lengths to use what little control the programmer has over memory consumption. We believe that, even within Java, it should be possible to save up to 25% of memory by optimising the choice of data types; in another programming language such as C it should even be possible to save more than 50%.

4 Exploration strategies

The state space generation as implemented in GROOVE consists of two layers. On the lowest layer, the crucial step is to *close* a single state, i.e., generate all outgoing transitions. The target states of those transitions that are new, even up to isomorphism, are added to the set of *explorable* states. The exploration of the complete state space is a matter of scheduling state closings. First of all there is the well-known distinction in *depth-first* and *breadth-first* exploration strategies, which we will not go into here.

A necessary criterion for explorability is that the state is open, but there are conditions under which open states are not considered explorable. We distinguish *bounding* conditions and *halting* conditions.

Bounding conditions are such that, when they are found to be satisfied in a given state, result in that state being declared non-explorable. In other words, satisfaction of a bounding condition cuts off exploration locally at the state where it occurs. Examples are: the depth of exploration, the applicability or non-applicability of a given rule, or a combination of those. Bounding conditions are typically used either to check global reachability conditions (reachability of a condition along *all* paths) or to limit the state space to a finite fragment. For instance, in the *mutex* case reported in Figure 2 we have used a bounding condition for the latter purpose.

Halting conditions are such that, when they are found to be satisfied, result in *all* states henceforth being declared non-explorable. In other words, satisfac-

tion of a halting condition cuts off exploration globally. Examples are the (non-)applicability of a given rule. Halting conditions are typically either state invariants or local reachability conditions (reachability along *some* path).

It follows that, even without having a verifier in the sense of Figure 1, we can already check some properties, namely those that can be expressed as reachability or invariant properties. As reported in [18], in the experiments used in this paper we have in fact included invariants in this way.

5 Conclusion

We have analyzed the time and space performance of GROOVE and given an overview of the techniques used (in the current version, 0.2.4) to achieve these results. The issues we have concentrated on in the implementation are to some degree independent of the strengths of other graph transformation tools, so that there is a realistic hope that further improvements can be made by additionally re-using existing techniques. We have shown in [18] that (depending on the system modelled) GROOVE has equal or better space performance than SPIN, but time performance that is an order of magnitude worse. We conjecture that, by re-using algorithms and techniques for, e.g., confluence detection, we can speed up the tool by a factor of 50% and, by optimizing internal data structures, save 25% to 50% on memory consumption.

References

- [1] A. Agrawal, G. Karsai, and F. Shi. A UML-based graph transformation approach for implementing domain-specific model transformations. *International Journal on Software and Systems Modeling*, 2003. Submitted.
- [2] M. Arends. Graph grammars for Java bytecode simulation. Master’s thesis, University of Twente, 2003.
- [3] D. Blostein and A. Schürr. Computing with graphs and graph rewriting. *Software — Practice & Experience*, 29(3):1–21, 1999.
- [4] A. Corradini, 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 [19], chapter 3, pp. 163–246.
- [5] J. de Lara and H. Vangheluwe. AToM³: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber, eds., *Fundamental Approaches to Software Engineering (FASE)*, vol. 2306 of *LNCS*, pp. 174–188. Springer, 2002.
- [6] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation, part II: Single pushout approach and comparison with double pushout approach. In Rozenberg [19], pp. 247–312.

- [7] H. Ehrig and B. König. Deriving bisimulation congruences in the DPO approach to graph rewriting. In *Foundations of Software Science and Computation Structures (FOSSACS)*, vol. 2987 of *LNCS*, pp. 151–166. Springer, 2004.
- [8] C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, eds., *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. II: Applications, Languages and Tools. World Scientific, Singapore, 1999.
- [9] R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, ed., *Fundamental Approaches to Software Engineering (FASE)*, vol. 1382 of *LNCS*, pp. 138–153. Springer, 1998.
- [10] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.
- [11] H. Kastenbergh. Software metrics as class graph properties. Master’s thesis, Department of Computer Science, University of Twente, July 2004.
- [12] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS: A graph-oriented (software) engineering database system. *Information Systems*, 20(1):1–32, 1995.
- [13] A. Lozano. Graph grammars for Java simulation. Master’s thesis, Vrije Universiteit Brussel, 2003.
- [14] T. Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Vrije Universiteit Brussel, 1999.
- [15] U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *International Conference on Software Engineering (ICSE)*. ACM Press, 2000.
- [16] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. L. Presti, eds., *Workshop on Automated Verification of Critical Systems (AVoCS)*, Tech. Rep. DSSE–TR–2003–2, pp. 150–160. University of Southampton, 2003.
- [17] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, eds., *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, vol. 3063 of *LNCS*, pp. 479–485. Springer-Verlag, 2004.
- [18] A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In F. Parisi-Presicce, P. Bottoni, and G. Engels, eds., *International Conference on Graph Transformations (ICGT)*, LNCS, 2004. To appear.
- [19] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. I: Foundations. World Scientific, Singapore, 1997.
- [20] A. Schürr. Programmed graph replacement systems. In Rozenberg [19], pp. 479–546.
- [21] D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modelling*, 3(2):85–113, 2004.
- [22] G. Varró and D. Varró. Graph transformation with incremental updates. In *International Workshop on Graph Transformation and Visual Modeling Techniques*, ENTCS. Elsevier, 2004. To appear.