AUGUR—A TOOL FOR THE ANALYSIS OF GRAPH TRANSFORMATION SYSTEMS *

Barbara König Vitali Kozioura
Universität Stuttgart
Institut für Formale Methoden der Informatik
{koenigba,koziouvi}@fmi.uni-stuttgart.de

Abstract

We describe the tool AUGUR for the verification of systems with dynamically evolving structure specified by graph transformation. After giving a short introduction to graph transformation systems (GTSs), we describe the verification techniques used by the tool, namely the approximation of GTSs by Petri nets. Instead of verifying properties directly in the original system, they can be checked on the approximating Petri net. We explain the workings of the different modules of the AUGUR tool using two small case studies where we model reconfigurable networks and mobile processes.

1 Introduction

The idea behind AUGUR is to provide a tool to verify systems with dynamically evolving structure using suitable approximation techniques. Systems of this kind appear in many places: as pointer structures on a program heap or as reconfigurable networks with mobile processes. They are characterized by the creation and deletion of objects and by changes in the system topology during runtime.

AUGUR takes as input language a simple yet expressive specification language: graph transformation systems (GTS) [18, 10]. Graph transformation systems extend static graph structures with the possibility to describe dynamic changes using transformation rules. They are well-suited to describe dynamic behavior, especially of concurrent and distributed systems.

GTSs are in general Turing-powerful and hence abstraction or over-approximation techniques are needed for the analysis of such systems. In our case we approximate GTSs by Petri nets, which are a conceptually simpler formalism and for

*Research supported by DFG project SANDS.

which several verification techniques have already been developed. More specifically, the tool is based on an approximate unfolding technique for GTSs, presented in [3].

We are currently mainly interested in verifying that all reachable graphs satisfy certain structural properties. In the current implementation we support the specification of paths using regular expressions, where we check that such paths are absent in every reachable graph. These properties can be translated to coverability properties of the approximating Petri net. In order to check coverability for Petri nets we use standard algorithms, such as coverability graphs [13] and backward reachability [1].

If the obtained over-approximation is too coarse and does not allow to verify the property, techniques for refining the approximation are available. One such technique is counterexample-guided abstraction refinement which starts from a concrete counterexample found by coverability checking. Another possibility is to use depth-based refinement, which constructs an over-approximation exact up to a pre-defined depth in the unfolding.

We have conducted successful case studies such as the verification of mutual exclusion in an extending ring of processes [8] or the analysis of insertion of new elements into red-black trees [2].

2 Graph Transformation Systems

A graph transformation system consists of an initial graph and a set of rewriting rules. In order to obtain more flexibility we consider hypergraphs where an edge (a hyperedge) is connected to a sequence of nodes (instead of a pair of nodes as in a directed graph). The initial graph is a hypergraph describing the initial state of the system. Rewriting rules consist of two hypergraphs (left-hand side and right-hand side) and specify the possible dynamic transformations of the system. If an instance of the left-hand side is found in the current state of the system, then this rule can be applied and the instance of the left-hand side of the rule will be replaced by its right-hand side. Embedding rules specify how this right-hand side is connected to the rest of the graph.

One of the most common approaches to graph rewriting is the DPO (double-pushout) approach, which derives its name from the fact that a rewriting step is described by two pushouts modelling the gluing of graphs. We are currently supporting restricted versions of DPO rules, where we only allow discrete interfaces, i.e., we can not describe the preservation of edges, and merging as well as deletion of nodes is forbidden. Edges, however, can be deleted. The extension to non-discrete interfaces is not very difficult from a theoretical point of view, whereas merging and deletion lead to more serious problems. Especially deletion means

that we would have to handle negative application conditions, which can only be modelled using inhibitor arcs in Petri nets.

Other approaches to graph transformation (such as the single-pushout approach) could also be handled provided that the restrictions mentioned above are satisfied. For more information on graph transformation systems see [18, 9, 10]. Below we give an example of a very simple GTS meant to illustrate the main features of AUGUR.

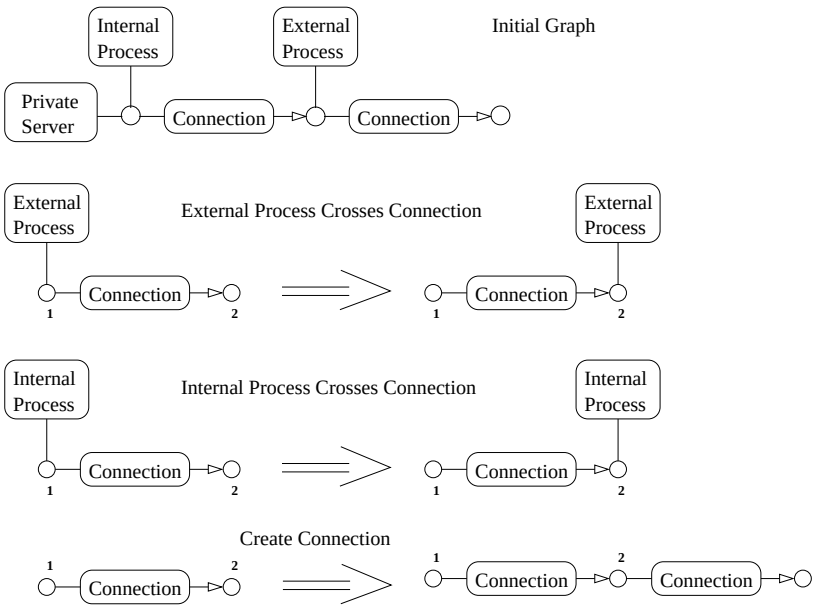

Figure 1: Example graph transformation system

In this system external and internal processes may cross connections and new connections can be created. This means we produce a tree-like structure of connections—starting with two connections—and let the mobile processes move nondeterministically along some branch of the tree. Transformations extending the network and movement of processes can be interleaved. The initial graph consists of a private server with an internal process connected to it. Separated by one connection there is an external process.

In this example we plan to verify the following property: “An external process will never reach a private server”, i.e., a hyperedge representing an external process and a hyperedge representing a private server will never share the same node.

3 Verification Techniques

We demonstrate the verification technique using the example of the previous section. To analyze this GTS the tool constructs an over-approximation, which is a so-called Petri graph (i.e., a hypergraph with a Petri net structure over it, see [3]). The hyperedges are at the same time the places of the net. For instance Fig. 2 shows the 0-depth (i.e., the coarsest) over-approximation of the GTS in Fig. 1. In Fig. 2 the small black rectangles and the arrows attached to them represent Petri net transitions, black dots represent the initial marking and the remaining structure depicts a hypergraph. Note that the places of the net coincide with the hyperedges of the graph.

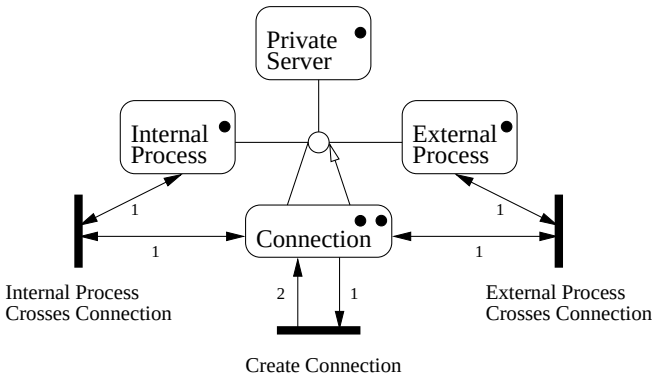

Figure 2: 0-depth approximation

This Petri graph is an over-approximation in the following sense: (i) every reachable graph can be mapped to its hypergraph component via a (usually non-injective) graph morphism and (ii) the multi-set image of its edges corresponds to a reachable marking of the net. More generally there exists a simulation relation between the reachable graphs and the reachable markings of the net. For a marking m we say that m *represents* a graph G whenever there is a mapping from G to the underlying hypergraph such that the number of edges mapped to a place agrees with the marking m.

Specifically, the initial marking represents the initial graph of the GTS (together with other graphs). Furthermore each transition is associated with a rule and over-approximates the effect of this rule when applied to a graph.

We do not describe here how the Petri graph is computed, apart from saying that the computation is based on an approximative unfolding algorithm. The algorithm is designed in such a way that nice properties of the GTS model, such as locality (state changes are only described locally) and concurrency (no unnecessary interleaving of events) are preserved in the approximating Petri net. More details can be found in [3, 5].

In this case the over-approximation is rather coarse. Observe specifically that *every* graph consisting of edges of the four types ("Private Server", "Connection", etc.) can be mapped to the underlying graph since all nodes have been merged into one. The only information we obtain via this approximation is the number of edges of a certain type. For instance the initial marking reports that in the initial graph there is one edge of type "Private Server", two edges of type "Connection", etc.

Since the approximation is too coarse, it is not possible to see whether a process has already crossed a certain connection and whether external processes may visit private servers. In fact the initial marking represents (in the sense defined above) graphs violating the property to be checked. It is also evident, that this counterexample is spurious, i.e., it has no counterpart in the original system since the "real" initial graph does not violate the property. In general there is a technique implemented in AUGUR telling the user if a counterexample is spurious, where a counterexample is a run of the Petri net producing a marking that represents graphs violating the property to be analyzed.

If some spurious counterexample is found, the over-approximation can be refined, which can be done in two different ways.

(i) One can change the level of accuracy (the depth) of the over-approximation.

(ii) One can construct a refined over-approximation by forbidding to merge certain nodes.

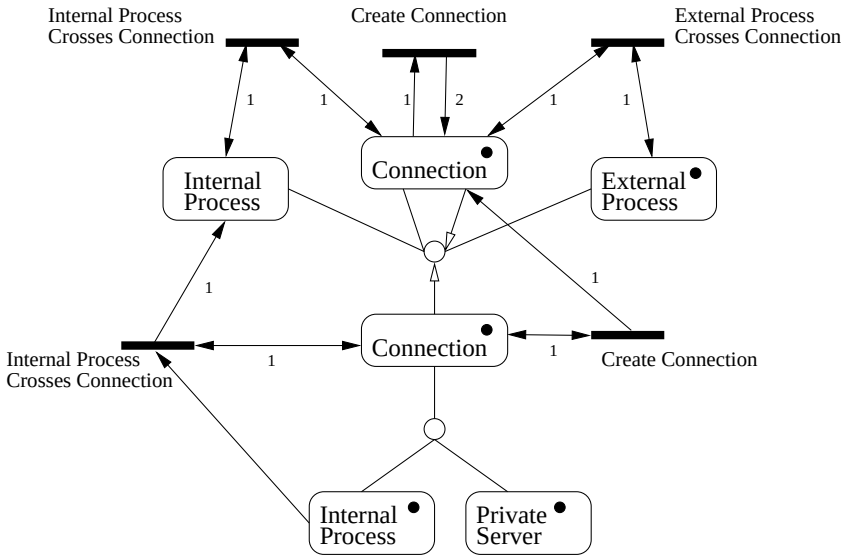

Figure 3: 1-depth approximation

In our example we choose the second possibility, which usually leads to smaller Petri graphs. We take the counterexample obtained above and construct the refined over-approximation (see Fig. 3). The edges representing the private server and the external process are now separated (since the corresponding nodes have been separated) and the spurious counterexample found above is eliminated. It is also evident that no marking is coverable which represents a "bad" graph, i.e. a graph where an external process is connected to the private server . This can be shown using AUGUR, which means that the property can be successful verified in an automatic way.

4 System Description

In this section we look at AUGUR in more detail. We briefly describe the following modules of AUGUR: AUNFOLD, SPONGE, BWRA and ABSTREF (see Fig. 4 for an overview of AUGUR).

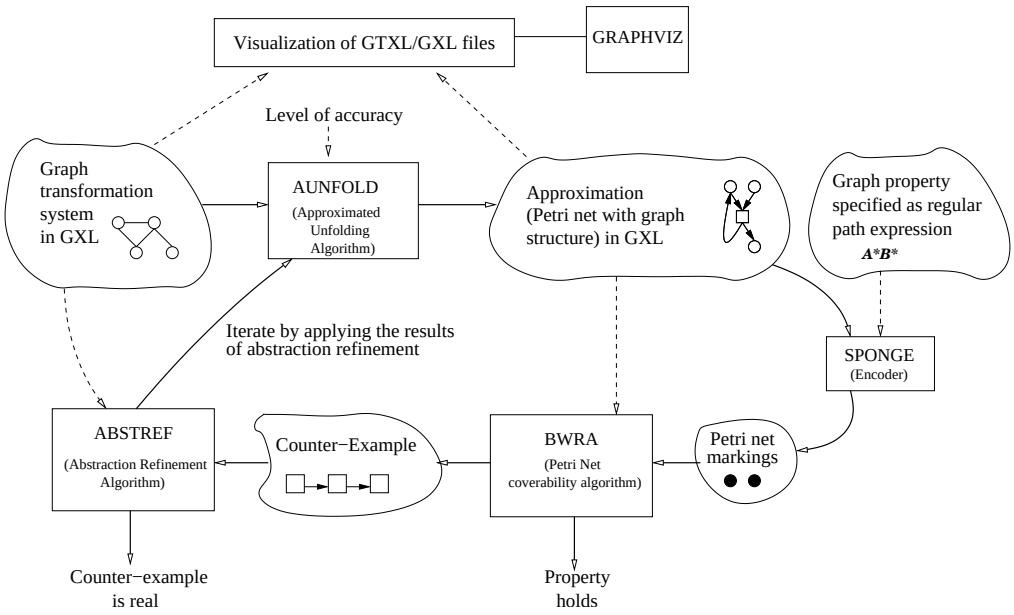

Figure 4: Overview of AUGUR

AUNFOLD is a module constructing the k-depth approximated unfolding (k-covering) for the given graph transformation system. As input and output format two XML-based standards are used (GTXL for describing graph transformation systems and the GXL for describing the over-approximating Petri graph obtained by the construction described in Section 3). For more details on GXL (Graph eXchange Language) and GTXL (Graph Transformation eXchange Language) see

also [23, 21, 12]. The approximation is constructed according to the algorithms proposed in [3, 5].

Given a hypergraph and a regular expression r (describing "forbidden" paths), the module SPONGE generates a set M of markings with the following property: a marking m represents a graph containing a path corresponding to r if and only if m covers at least one marking of M. This information can be used in order to show that no graphs containing undesirable paths can be reached.

This task is performed by BWRA, which is a module using the backward reachability algorithm from [1] in order to determine the coverability of a marking. If the markings are not coverable, then the property to be verified is true, otherwise BWRA generates a counterexample.

Finally, module ABSTREF checks first if the counterexample found by BWRA is real. If this is the case, then the property to be verified is false. Otherwise refinement conditions will be computed and the approximation procedure starts again taking into account all refinement conditions obtained earlier.

The procedure can be iterated in this way until either the property is verified, a counterexample is found or the pre-defined timeout is reached.

AUGUR also contains a visualization module, based on the GRAPHVIZ package. It can produce postscript files depicting graph transformation systems specified in GTXL and over-approximating Petri graphs represented as GXL files (see Fig. 5).

AUGUR can be obtained via `http://www.fmi.uni-stuttgart.de/szs/tools/augur/`. We would like to encourage people who download and try the tool to report their problems and experiences.

5 An Extended Example

In this section we describe the verification of a more complex example and give some more details concerning the usage of AUGUR. We consider the system "Public/Private Servers" in Fig. 6 (see also example file `pub_priv_serv.xml` in the AUGUR distribution), which is an extension of the previous example. This system consists of public and private servers linked by network connections. Generators produce an unlimited number of public servers and one private server. The servers in turn produce mobile processes (internal processes by the private and external by the public servers). New connections can be created between the servers, where however no connection is allowed from a private to a public server. Processes may cross these connections. Furthermore at some point in time the private server may decide to become a public server. The property we want to verify here is (as in the previous example): "No external process will ever reach a private server".

The corresponding graph transformation systems consists of an initial graph containing only two generators and a set of rules describing the transformations

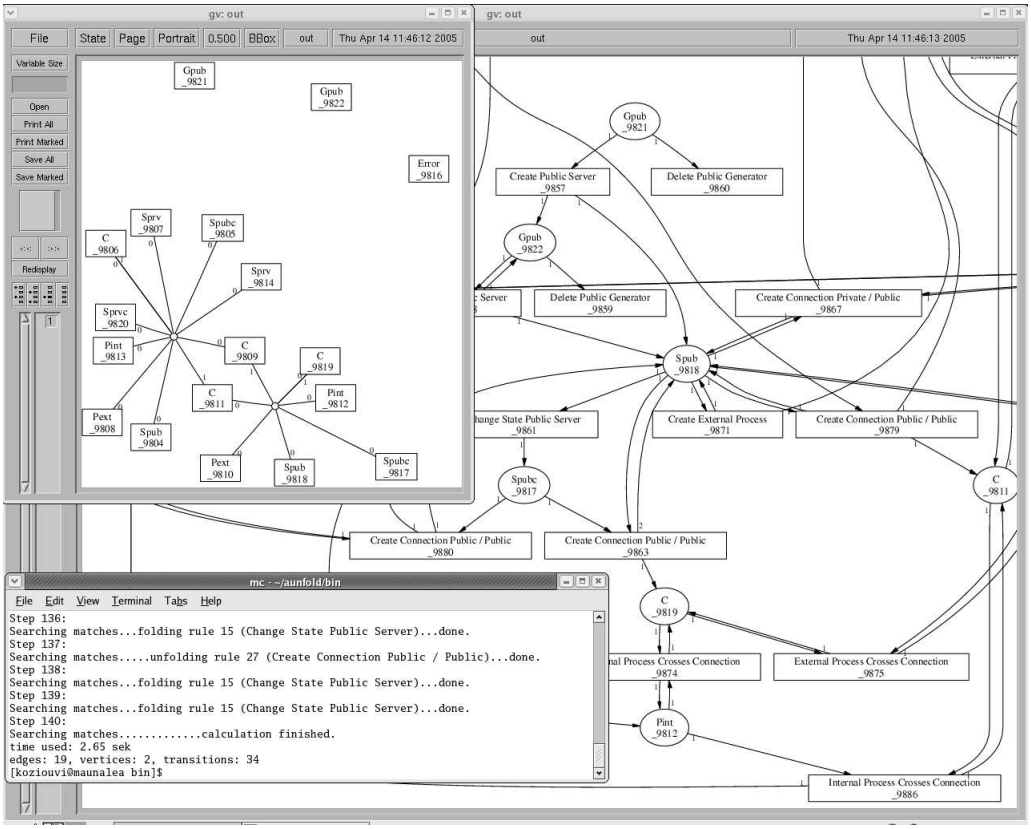

Figure 5: Screenshot of AUGUR at work

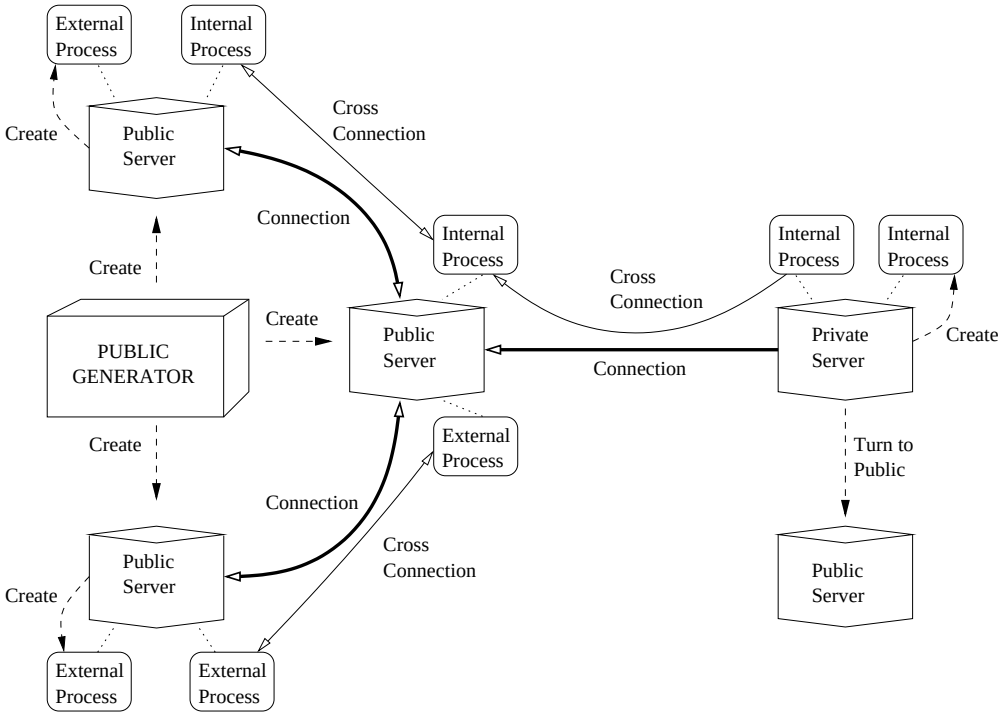

Figure 6: Example “Public/Private Servers”

schematically depicted in Fig. 6. The entire GTS is too large to be depicted in this paper.

In order to verify it, we first construct the approximated unfolding using AUNFOLD. Now we can call SPONGE with the regular expression “`Private Server`” “`External Process`” in order to obtain a set of markings M with the following meaning: all markings which cover any marking of M are exactly the markings representing “bad” graphs, i.e., graphs which violate the property to be verified.

As in the previous example the 0-depth approximation is too coarse and BWRA tells us that indeed some markings contained in M are coverable in over-approximation and gives a counterexample. After checking the obtained counterexample with ABSTREF we see that this counterexample is spurious. Using ABSTREF one can now construct the refined over-approximation as described earlier, which leads to successful verification.

Regular expression may also be more complex, for example “`Private Server`” “`Connection`”* “`External Process`” represents the property that no connection will ever be created from a public to private server. This property can also be verified after one refinement step.

Fig. 5 shows a screenshot of AUGUR during the verification of the “Public/Private Servers” system. The small window in the upper left corner shows the hypergraph

underlying the approximation, whereas the large windows shows a part of the Petri net component. A step-by-step tutorial for AUGUR that shows how to verify this example is contained in [11].

6 Conclusion

In order to conclude we would like to mention and classify related work on the verification of graph transformation systems. While some research groups [22, 7] pursue the idea of translating graph transformation systems into the input language of a model checker, others attempt to develop new specialized methods for graph rewriting. Work from our side goes in this latter direction, as well as [15, 14, 16], which led to the tool GROOVE for verifying finite-state GTS. Although it is tempting to use existing optimized model checking tools, there is good reason for developing new techniques, even for finite state spaces. Existing tools usually do not directly support the creation (and deletion) of an arbitrary number of objects while still maintaining a finite state space, making entirely non-trivial their use for checking finite-state GTSs. A nice overview comparing these two fundamentally different approaches can be found in [17].

Further related work is on shape analysis [19], i.e., techniques which address directly the problem of verifying pointer structures. We have recently started to address the problem of encoding simple pointer-manipulating programs into graph rewriting, which will enable us to directly apply our techniques to a given piece of code.

Analysis techniques for GTSs are not restricted to reachability analysis and model checking. Other properties (such as termination and confluence via critical pair analysis) can be analyzed using the AGG tool [20].

Let us also mention that structural properties of graphs can not only be specified using regular expressions, but also using a monadic second-order graph logic. While the theory for this logic has already been established [6], we have not yet implemented the encoding of graph logic into properties of Petri net markings.

Finally, while the techniques presented in this paper are in principle also applicable to finite-state systems, it will usually be better to use specialized methods, such as finite complete prefixes of unfoldings, a partial order representation for finite-state GTSs [4]. We have already started to extend our implementation in this direction.

Acknowledgements: We would like to thank Paolo Baldan, Andrea Corradini and Tobias Heindel for many interesting discussions.

References

[1] Parosh Aziz Abdulla, Bengt Jonsson, Mats Kindahl, and Doron Peled. A general approach to partial order reductions in symbolic verification. In *Proc. of CAV '98*, pages 379–390. Springer, 1998. LNCS 1427.

[2] Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel, Barbara König, and Vitali Kozioura. Verifying red-black trees. In *Proc. of COSMICAH '05*, 2005. Proceedings available as report RR-05-04 (Queen Mary, University of London).

[3] Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, pages 381–395. Springer-Verlag, 2001. LNCS 2154.

[4] Paolo Baldan, Andrea Corradini, and Barbara König. Verifying finite-state graph grammars: an unfolding-based approach. In *Proc. of CONCUR '04*, pages 83–98. Springer-Verlag, 2004. LNCS 3170.

[5] Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In *Proc. of ICGT '02 (International Conference on Graph Transformation)*, pages 14–29. Springer-Verlag, 2002. LNCS 2505.

[6] Paolo Baldan, Barbara König, and Bernhard König. A logic for analyzing abstractions of graph transformation systems. In *Proc. of SAS '03 (International Static Analysis Symposium)*, pages 255–272. Springer-Verlag, 2003. LNCS 2694.

[7] Fernando Luís Dotti, Luciana Foss, Leila Ribeiro, and Osmar Marchi Santos. Verification of distributed object-based systems. In *Proc. of FMOODS '03*, pages 261–275. Springer, 2003. LNCS 2884.

[8] Fernando Luís Dotti, Barbara König, Osmar Marchi dos Santos, and Leila Ribeiro. A case study: Verifying a mutual exclusion protocol with process creation using graph transformation systems. Technical Report 08/2004, Universität Stuttgart, 2004.

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

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

[11] Barbara König and Vitali Kozioura. AUGUR—a tool for the analysis of graph transformation systems using approximative unfolding techniques, January 2005. Available from `http://www.fmi.uni-stuttgart.de/szs/tools/augur/documentation.ps`.

[12] Leen Lambers. A new version of GTXL: An exchange format for graph transformation systems. In *Proc. Workshop on Graph-Based Tools (GraBaTs'04)*, 2004.

[13] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.

[14] Arend Rensink. Model checking graph grammars. In *Proc. of AVOCS '03 (Workshop on Automated Verification of Critical Systems)*, 2003.

[15] Arend Rensink. Canonical graph shapes. In *Proc. of ESOP '04*, pages 401–415. Springer, 2004. LNCS 2986.

[16] Arend Rensink. State space abstraction using shape graphs. In *Proc. of AVIS '04 (Third International Workshop on Automatic Verification of Infinite-State Systems)*, ENTCS, 2004. to appear.

[17] Arend Rensink and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. of ICGT '04*, pages 226–241. Springer, 2004. LNCS 3256.

[18] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, volume 1. World Scientific, 1997.

[19] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS (ACM Transactions on Programming Languages and Systems)*, 24(3):217–298, 2002.

[20] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In *Proc. of AGTIVE '99 (Applications of Graph Transformations with Industrial Relevance, International Workshop)*, pages 481–488. Springer, 1999. LNCS 1779.

[21] Gabriele Taentzer. Towards common exchange formats for graphs and graph transformation systems. In *Proc. of UniGra '01 (Uniform Approaches to Graphical Process Specification Techniques)*, volume 44.4 of *ENTCS*, 2001.

[22] Dániel Varró. Towards symbolic analysis of visual modeling languages. In *Workshop on Graph Transformation and Visual Modeling Techniques '02*, volume 72 of *ENTCS*. Elsevier, 2002.

[23] A. Winter. GXL—overview and current status. In *Proc. of GraBaTs '02 (Workshop on Graph-Based Tools)*, volume 72.2 of *ENTCS*, 2002.