Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 175 (2007) 63–72

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Verification of Random Graph Transformation Systems [⋆]

Vitali Kozioura

Universität Duisburg-Essen, Germany
vitali.kozioura@uni-due.de

Abstract

In this paper we describe some statistical results obtained by the verification of random graph transformation systems (GTSs). As a verification technique we use over-approximation of GTSs by Petri nets. Properties we want to verify are given by markings of Petri nets. We also use counterexample-guided abstraction refinement approach to refine the obtained approximation. A software tool (AUGUR) supports the verification procedure. The idea of the paper is to see how many of the generated systems can be successfully verified using this technique.

Keywords: Random graph transformation, Petri net, AUGUR

1 Introduction

In the last few years a technique for analysing of graph transformation systems (GTSs) [13] based on approximations has been developed [2]. GTSs are approximated by Petri graphs which are Petri nets with additional hypergraph structure. Petri nets can then be analyzed with standard verification techniques. The verification follows so called CEGAR approach (Counterexample-Guided Abstraction Refinement). The idea behind this approach is to start with a coarse initial abstraction or over-approximation of a system and to check whether a certain property can be verified using this abstraction. If it can not be verified, one obtains a run in the approximation that violates this property, also called counterexample. Now either this counterexample is real or it is spurious, i.e., it has been introduced by the approximation. In the latter case the approximation is refined in such a way that the counterexample disappears. This process is repeated, however in the case of infinite-state systems there is in general no guarantee that it will terminate, since the properties to be verified are usually undecidable [9].

⋆ Research supported by DFG project SANDS.

doi:10.1016/j.entcs.2007.04.017

A software tool (Augur) supporting this verification approach has been developed [7] and a number of successful case studies have been conducted (see for example the case study on red-black trees [1]). Still, verification remains undecidable in general (because of the Turing-completeness of GTSs). The interesting question is how many GTSs can be verified in practice using the over-approximation of GTSs by Petri nets and standard techniques for analysing Petri nets. This work is a first attempt to give an answer to this question.

We generate some random GTSs and verify them with the help of Augur in order to obtain statistical results. We consider some classes of GTSs identified by a number of parameters.

2 Graph Transformation Systems

Graph transformation systems (GTSs) are an expressive and useful specification formalism, allowing to describe dynamic properties of concurrent and distributed systems [13,6]. GTSs have many interesting applications in different areas of computer science, they can be used to specify evolving pointer structures on the heap as well as mobile processes in a network.

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 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 [13,5,6].

Below we give an example of a very simple GTS meant to illustrate the main features of Augur. 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 non-deterministically 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 of Graph Transformation Systems

Fig. 2 depicts the verification technique which is used in this paper. We have a GTS and a (reachability) property[1] we want to verify as an input of the system. First of all we construct a Petri graph which is an over-approximation of a GTS having both hypergraph and Petri net structures [2]. In the analysis block we first calculate the Petri net marking corresponding to the property to verify (which is usually obtained from a regular expression on the hypergraph structure of the Petri graph [11]). The marking is then analysed with the help of a coverability algorithm [10]. If the marking is not coverable, then we terminate with "VERIFIED". This means that the corresponding subgraph (described by the regular expression and a corresponding marking) cannot be reached during the reduction of the GTS. Otherwise we have

[1] Checking reachability property for GTSs is already a rather complex problem, and we here restrict our experiments to it, instead of checking general temporal properties of GTSs.

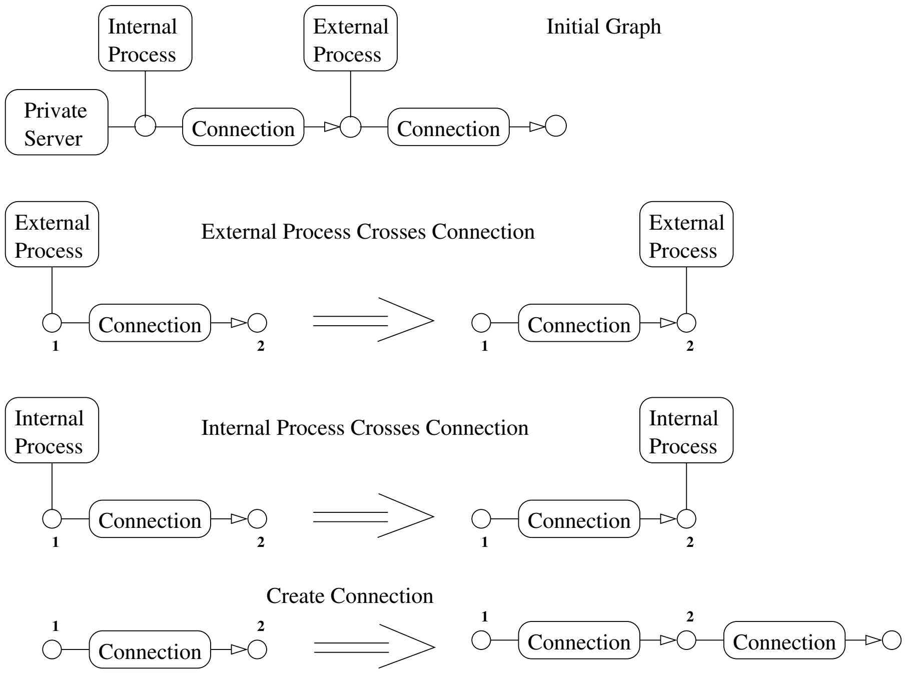

Fig. 1. Example graph transformation system

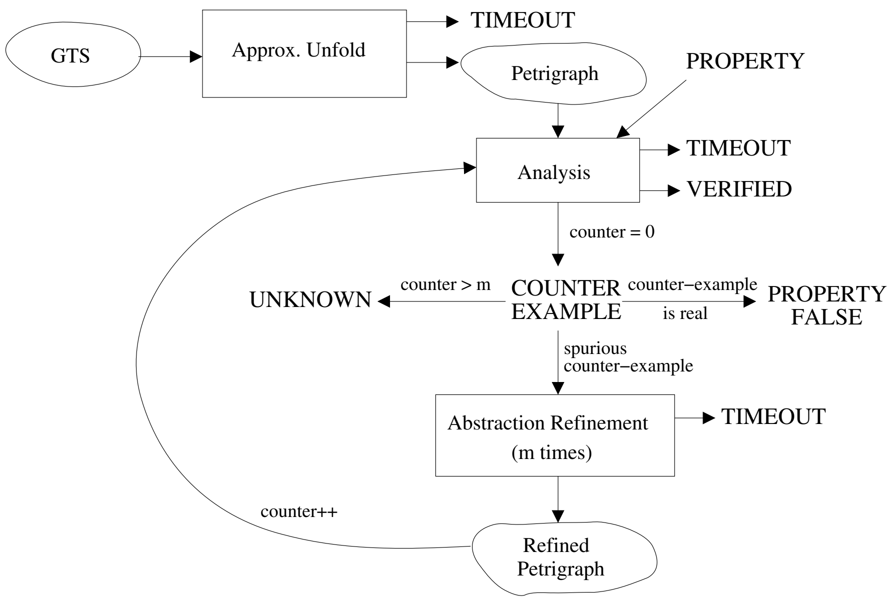

Fig. 2. Verification technique

two possibilities. The obtained trace to the coverable marking (counter-example) can be real or spurious (i.e., reproducible only in the over-approximation and not in the original GTS). In the case of a real counterexample we terminate with "PROPERTY FALSE". Otherwise we start a counterexample-guided abstraction refinement procedure [4,9] and obtain a refined Petri graph. The refinement procedure can be iterated a predefined number of times. If we still do not have a verification result, then we terminate with "UNKNOWN". For each operation a timeout is set such that when it is reached, the verification process stops with "TIMEOUT". We say that the verification problem for GTS is solved if the property is verified or we have found a (non-spurious) counter-example.

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 [2]). The hyperedges are at the same time the places of the net. For instance Fig. 3 shows the 0-depth (i.e., the coarsest) over-approximation of the GTS in Fig. 1. In Fig. 3 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.

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

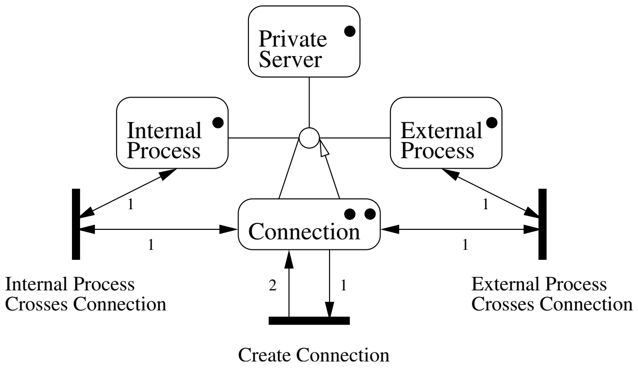

Fig. 3. 0-depth approximation

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 [2,3].

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.

In our example we choose the second possibility, which usually leads to smaller

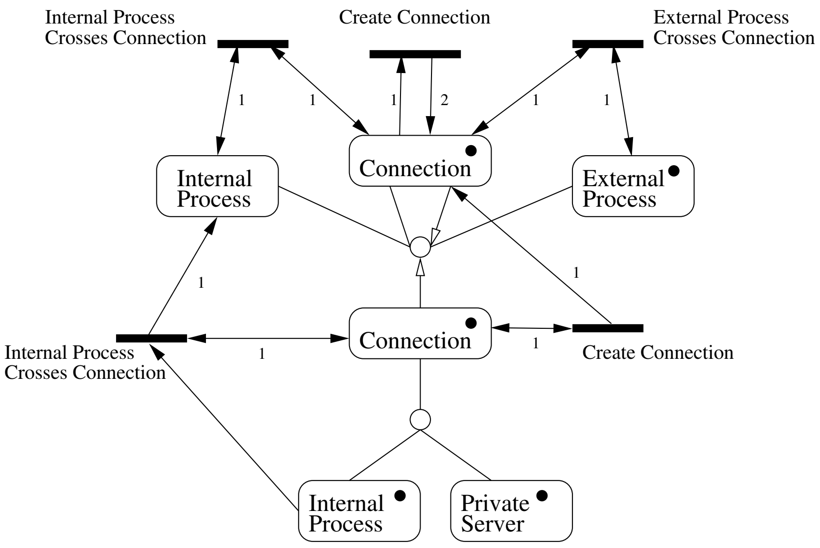

Fig. 4. 1-depth approximation

Petri graphs. We take the counterexample obtained above and construct the refined over-approximation (see Fig. 4). 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 Random Graph Transformation Systems

In this paper we generate GTSs with hyperedges having arity (number of connected nodes) one or two. Edges can be labeled (we consider two labels for each arity). We do not allow two edges having the same labels in the left-hand side of a rule. We also do not delete any nodes. Therefore we describe below only the nodes being added to the right-hand side of the rule.

The following parameters describe the class of generated GTS:

(i) Minimal/Maximal number of nodes in the left-hand side of a rule.

(ii) Minimal/Maximal number of additional nodes in the right-hand side of a rule (see the explanation above).

(iii) Minimal/Maximal number of edges in the left-hand side of a rule.

(iv) Minimal/Maximal number of edges in the right-hand side of a rule.

(v) Minimal/Maximal number of nodes in the initial graph.

(vi) Minimal/Maximal number of edges in the initial graph.

(vii) Minimal/Maximal number of rules.

In this paper we consider the following classes of random systems defined by their parameters.

(i) (1, 2; 0, 1; 1, 2; 1, 2; 2, 5; 2, 5; 3, 5)

(ii) (1, 2; 0, 2; 1, 3; 1, 3; 2, 5; 3, 7; 3, 7)

(iii) (2, 3; 1, 5; 3, 7; 3, 7; 3, 10; 3, 10; 5, 10)

Each class of GTSs is strictly included in the next one. In each class we generate 100 GTSs. The numbers are relative small because we tried to keep the sizes of generated GTSs manageable in order to obtain enough statistical material.

In each GTS we insert additionally the special rule "Error", where the left-hand side is random and the right-hand side consists only of an edge labelled "Error".

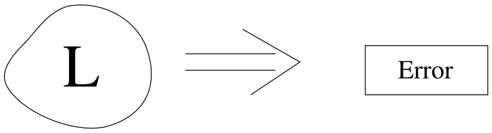

Fig. 5. Error rule

The property we want to verify is "the Error rule cannot be applied in the generated GTS", which guarantees that no reachable graph contains L as subgraph. If the rule "Error" can be applied, then the verification algorithm (Fig 2) should give the answer "FALSE" and generate a counterexample. If the rule "Error" cannot be applied, then we should obtain the answer "VERIFIED". Fig. 6 represents an example of a generated GTS from the first class.

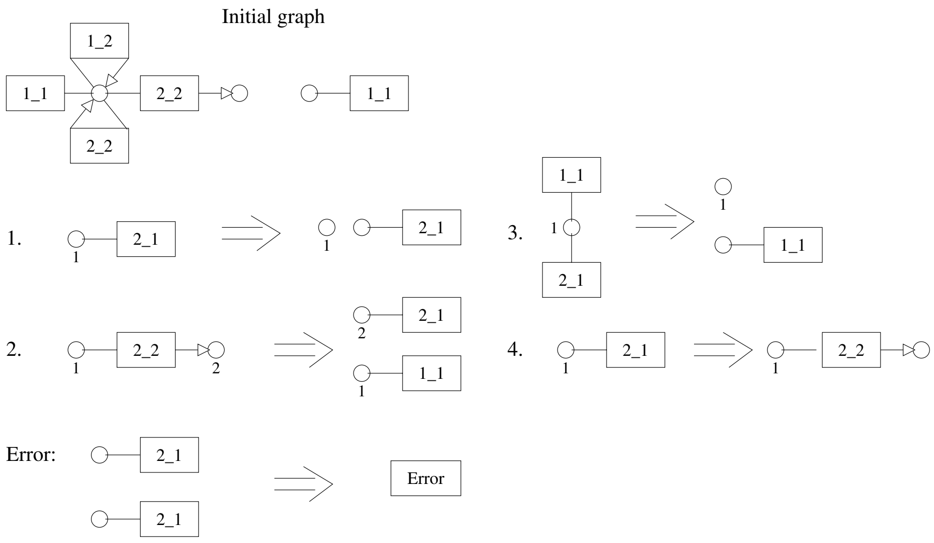

Fig. 6. Example of a generated GTS (first class of systems)

5 Statistical Results

The experiments have been done on 2*Xeon 2.4 GHz, 2GB RAM. We fix 3 iterations for the abstraction refinement procedure and 30 minutes as timeout value. In Table 1 average values obtained during the verification of generated systems are represented, namely the number of nodes, edges and transitions in the constructed over-approximations and the verification times (including the timeouts). The verification time is measured in seconds and represents the time of the whole verification procedure.

system class	nodes	edges	transitions	verification time
1	4.21	7.67	4.07	0.01
2	7.47	14.5	10.55	59.87
3	10.01	22.28	25.78	351.53

Table 1
Average values of the verified systems.

Diagrams in Fig. 7 ((a),(b) and (c), ignore (d) for the moment) describe the distribution of the verification results for the three classes of random systems described above.

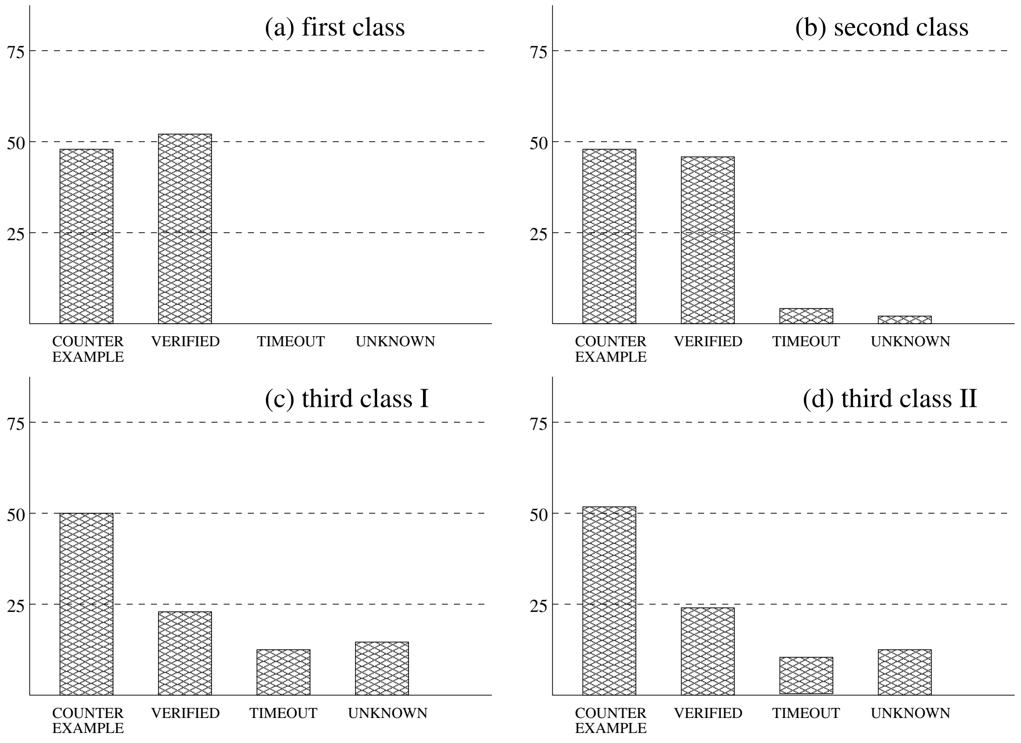

Fig. 7. Statistic of verification results

An interesting value is also the total number of refinement steps during the verification of one class of GTSs. This value grows rather quickly: 0 steps for the first class of systems, 18 steps for the second class and 83 steps for the third class. But note that the number of refinement steps for each GTS is restricted by 3.

As we can see in Fig. 7 we have successfully solved the verification problem for all 100 GTSs in the first class of systems whereas in the third class the number of problems we could not solve is about one third of the number of solved systems. These diagrams give us an idea of possibilities and constraints of the verification approach based on the over-approximation of GTSs with Petri nets. To achieve better verification results we can increase the number of refinement steps and/or the timeout interval. If we start the verification procedure for the same systems belonging to the third class with maximally five refinement steps and with two hours timeout, then we can additionally solve the verification problem for *five* more GTSs, Fig. 7(d). The average verification results in this case are represented in Table 2. The total number of abstraction refinement steps is 109.

system class	nodes	edges	transitions	verification time
3	11.87	26.57	33.06	1273.74

Table 2
Average values for the third class with five refinement steps and two hours timeout

6 Conclusion

In this paper we considered statistical results of the verification of random GTSs by approximating them with Petri nets. The verification technique is implemented in AUGUR 1 and this tool has been used as a basis for our experiments. The purpose of the paper is to show how many of the random GTSs can be verified with this technique. Obviously the systems appearing in real case studies differ from random systems by having a more regular structure, but this papers gives us some (approximative) notion about the possibilities and difficulties of this approach.

The statistical results can be seen as rather positive and hence the verification approach of approximating GTSs by Petri nets can be seen as a promising approach for the verification of GTSs. Of course it will also be necessary to compare these results with related results stemming from other methods. However we are currently not aware of any such results for random systems which have been published.

Some experimental results on the verification of GTSs have been reported in [12]. Note that we are here working in a different setting since we consider potentially infinite state GTSs, whereas [12] considers finite state GTSs.

As future work we mention here experiments on random GTSs with higher degrees of hyperedges, checking the effect of individual parameters on the results, experiments with generic systems (random GTSs generated according to some regular template), experiments with attributed GTSs and experiments with a new version of the tool AUGUR 2 [8], which is currently under development.

References

[1] 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).

[2] 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.

[3] 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.

[4] E. Clarke, S. Grumberg, S. Jha, and H. Lu, Y. und Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, pages 154–169. Springer, 2000. LNCS 1855.

[5] 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.

[6] 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.

[7] Barbara König and Vitali Kozioura. Augur—a tool for the analysis of graph transformation systems. *EATCS Bulletin*, 87:125–137, November 2005. Appeared in The Formal Specification Column.

[8] Barbara König and Vitali Kozioura. Augur 2—a new version of a tool for the analysis of graph transformation systems. In *Proc. of GT-VMT '06 (Workshop on Graph Transformation and Visual Modeling Techniques)*, pages 195–204, 2006. ENTCS.

[9] Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *Proc. of TACAS '06*, pages 197–211. Springer, 2006. LNCS.

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

[11] Nicolas Relange. Verifikation dynamischer Systeme: Reguläre Ausdrücke zur Spezifikation verbotener Pfade. Master's thesis, Universität Stuttgart, September 2004. No. 2192.

[12] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. ICGT 2004: Second International Conference on Graph Transformation*, volume 3256 of *LNCS*, pages 226–241, Rome, Italy, 2004. Springer.

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