Augur 2 — A New Version of a Tool for the Analysis of Graph Transformation Systems*

Barbara König Vitali Kozioura

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

Abstract

We describe the design and the present state of the verification tool Augur 2 which is currently being developed. It is based on Augur 1, a tool which can analyze graph transformation systems by approximating them by Petri nets. The main reason for the new development was to create an open, flexible and extensible verification environment. Also, compared to the previous version, Augur 2 will include more functionality and new analysis techniques.

1 Introduction

In the last few years we have developed the verification tool Augur 1 (or simply Augur) [12] which analyzes graph transformation systems (GTSs) by approximating them with Petri nets. Using this tool we have conducted several case studies, verifying, for instance, a mobile system with a firewall [4], a mutual exclusion protocol [10] and the insertion of elements into red-black trees [2]. Other examples of systems for which our technique is suitable are dynamic pointer structures on a program heap, object graphs and reconfigurable networks with mobile processes. The tool can be obtained from http://www.fmi.uni-stuttgart.de/szs/tools/augur/.

We started with a small tool that reads GTXL files and produces GXL files (GXL respectively GTXL are XML standards for the encoding of graphs and graph transformation systems [14]). Afterwards we faced the constant necessity of adding new features and new functionality. More specifically, we added analysis algorithms for Petri nets [20] based on coverability graphs [15] and backward reachability [1]. Furthermore we established an interface to Graphviz [1] for visualization purposes. We also added the possibility to specify forbidden paths in graphs using regular expressions [16], we implemented the finite complete prefix technique for graph transformation systems [5,8] and we started to extend the tool in order to use it for

* Research supported by DFG project SANDS.

[1] http://www.graphviz.org/

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the purpose of test case generation [11]. Probably the most extensive addition was to add support for counterexample-guided abstraction refinement [13].

The architecture of AUGUR 1 was strongly oriented towards the concrete task of approximated unfolding of GTSs. This made all changes mentioned above hard to implement and led to several versions of the tool, each with a different functionality. Hence the new version of AUGUR (called AUGUR 2) will have a more general and extensible software architecture and will have more functionality concerning analysis and visualization methods.

Another new feature of AUGUR 2 will be the possibility to work with attributed graphs, i.e., graphs with (integer and string) attributes assigned to nodes and edges. As a future research topic we plan to extend existing analysis techniques accordingly. Support for input and output will also be extended, for instance we are currently working on an interface to AGG [19]. Also, we plan to have a simple pointer-manipulating programming language, which can be translated into graph rewriting, as an additional means of input.

The kernel part of the tool is already developed and has successfully passed through a number of tests. At the moment the tool is being extended with various visualization and analysis methods.

2 Graph Transformation Systems and Verification Techniques

We use hypergraph rewriting where left-hand and right-hand sides can be (almost) arbitrary hypergraphs. Compared to the double-pushout approach our GTSs have to observe some restrictions: especially, the interface graph of a rule must be discrete, no nodes can be deleted and rules must be consuming, i.e., at least one edge is deleted. While the last two restrictions are essential for the unfolding-based approach we are following, the first restriction (the interface is discrete) will be lifted in AUGUR 2.

In order to illustrate the basic ideas behind the tool, we will start with a simple example. Fig. 1 shows a GTS, which models a network consisting of connections, private servers, internal and external processes, where the network is constantly extended during runtime. Furthermore processes may cross connections. The property we want to verify is that no external process will ever visit a private server. We reduce this property to "no Error edge will be created" by adding a rule creating an edge labelled "Error" as soon as the forbidden situation has been detected.

Since GTSs are in general Turing-powerful over-approximation techniques are needed for their analysis. In our case we abstract GTSs by Petri nets, which are a conceptually simpler formalism and for which several verification techniques have already been developed. More specifically, the tool is based on an approximated unfolding technique for GTSs, presented in [3]. Compared to a standard unfolding technique we are additionally using folding steps which over-approximate, but guarantee a finite approximation. 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. Fig. 2 depicts the coarsest over-approximation for the example GTS in Fig. 1.

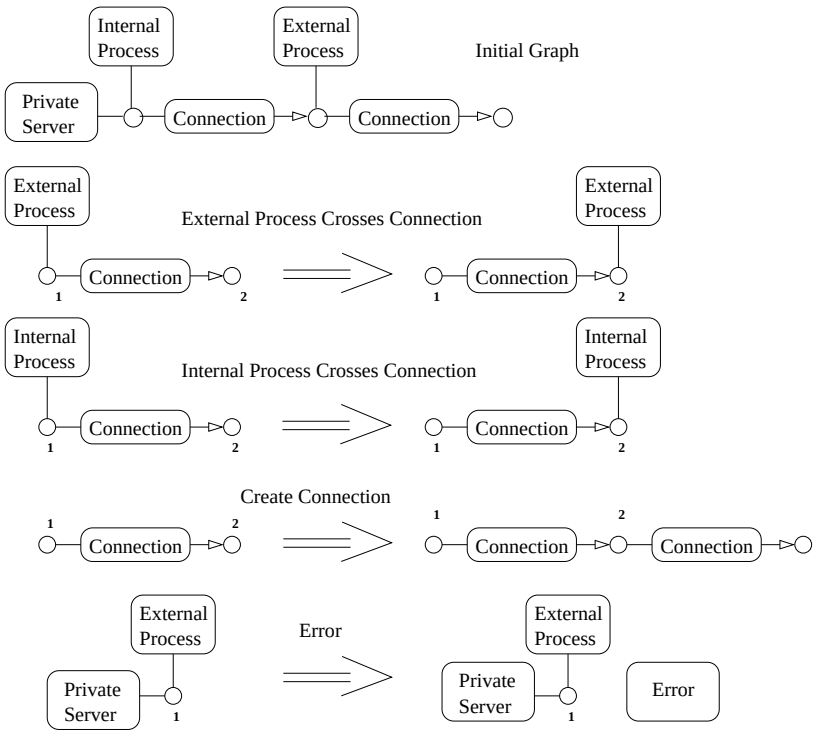

Fig. 1. Example graph transformation system.

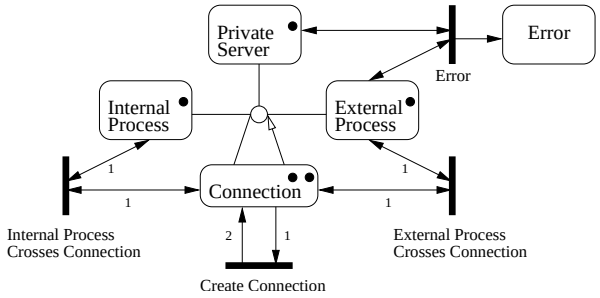

Fig. 2. Petri graph approximating the GTS (first approximation).

The 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. For instance the five edges of the initial graph correspond to the five tokens of the initial marking of the net. More generally there exists a simulation relation between the reachable graphs and the reachable markings of the net, obtained by firing enabled transitions. More details can be found in [3,6].

If the 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 [13] which starts from a concrete counterexample found by coverability checking on the Petri net. Another possibility is to use depth-based refinement [6] which constructs an over-approximation exact up to a pre-defined depth in the unfolding. Counterexample-guided abstraction refinement usually results in smaller approximations and faster verification.

The edge "Error" of the Petri graph in Fig. 2 can be covered by firing transition "Error". This means that either the property does not hold or the over-approximation is too coarse. One can show that the run is spurious, i.e., it has no counterpart in the original GTS, which indicates that we have approximated too much. Applying abstraction refinement gives us a refined Petri graph, which is depicted in Fig. 3. There exists no edge labelled "Error", i.e., such an edge can also not be covered by any reachable marking. So, from the correspondence between reachable graphs and markings it follows that the property has been successfully verified.

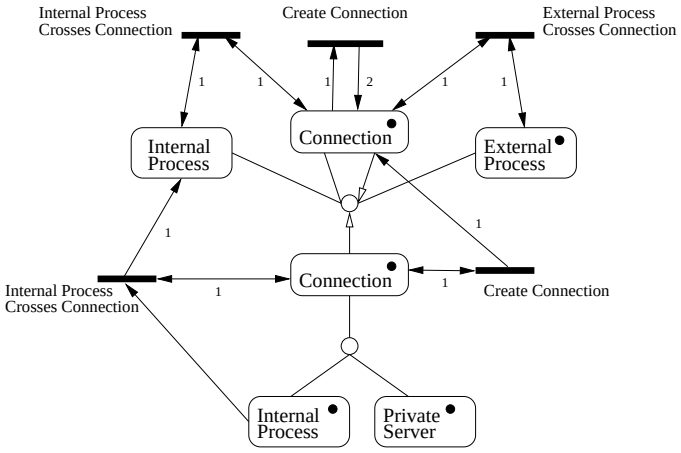

Fig. 3. Petri graph after counterexample-guided abstraction refinement.

3 Software Design

In this section we will present the main ideas behind the new implementation, which lead to an open and flexible new verification tool.

The central part of the software design is the concept of *algorithms*, which are implemented as classes. Each program module working with the common data structures should be realized as an algorithm and new algorithms can be added during the whole life time of the system. As examples of algorithms we mention here different operations on Petri graphs (firing of transitions, building the coverability graph, searching for matches of left-hand sides, performing folding/unfolding steps, etc.) and input/output operations (readers and writers from/to different data formats).

All algorithms work with the same data structures which makes it possible to use some algorithms as sub-operations inside others and to assemble new algorithms out of existing ones. A *scenario* is a special algorithm which uses as input and output only the external data sources, such as XML files (Fig. 4). Scenarios are at the top level of the system and call other algorithms. A typical example for a scenario is the approximated unfolding algorithm which reads a graph transformation system and outputs a Petri graph.

All algorithms and scenarios are managed in a central database system (see Fig. 4). The database consists of several tables, the most important being the *algorithm table*, which is shown in Table 1.

Calling Algorithm	Label	Algorithm To Call	History Path
main	a	unfold	$\emptyset$
unfold	a	findMatch2	(*,*)(a, findMatch1)(*,*)
unfold	a	findMatch1	$\emptyset$
unfold	b	findMatch3	$\emptyset$

Table 1
Example for the algorithm table in the database system.

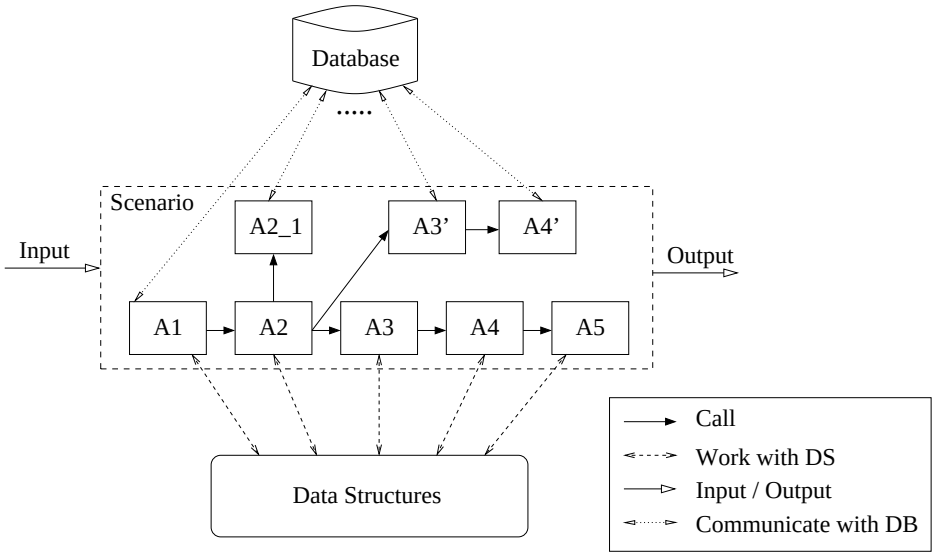

Fig. 4. Schematic depiction of a scenario.

The first column of the table represents the name of the algorithm calling another algorithm as a sub-operation. The second column is the label of the place where the sub-operation is being called. Labels are used by algorithms in order to indicate which kind of other algorithms they intend to call. Then, the information in the database determines which of the several available algorithms for this task is chosen. This information, i.e., the name of the algorithm, is given in the third column. Finally, the fourth column is a regular expression representing the history path or the call stack of the algorithm in the first column. Depending on this history different algorithms can be called from the same place in the code. We use the information of the first matching entry in the table. Hence, this table makes it easy to exchange a sub-operation by another sub-operation performing basically the same function in a different way (optimized for the concrete situation).

Table 1 represents a typical example—the control of the match finder algorithm, which searches for matches of left-hand sides in a (large) hypergraph. This operation is one of the critical parts in the calculation of the approximating unfolding and there are different ways to implement it [23,18,22]. We call three different match finder algorithms depending on the place in the unfolding algorithm where the match finder is called and on the local history of calls. At the place labelled 'b' we will always call "findMatch3". If at the place labelled 'a' the algorithm "findMatch1"

has already been called, "findMatch2" will be called next. This is very similar to what happens in our implementation since in different situations matches have to be located in slightly different ways. For instance for a folding step two matches have to be found instead of one.

Another example is coverability checking for Petri nets, for which we currently use two different algorithms: computation of coverability graphs and backward reachability. The current layout of the tool also makes it easy to replace an old inefficient version of an algorithm by a more efficient one and to use different versions of an algorithm in different situations.

Besides the algorithm table there exists some other information needed to manage the behavior of algorithms. For example, there is a table describing the reusability of algorithms, i.e., which says whether a new object should be created when a new algorithm is requested or if a previously created object can be reused. Also, there exist protocols governing the communication between algorithms. For example algorithms can notify each other about changes in the data structures (validation protocol).

4 System Architecture

After presenting the general ideas behind AUGUR 2, we will now describe the architecture behind this tool (see Fig. 5).

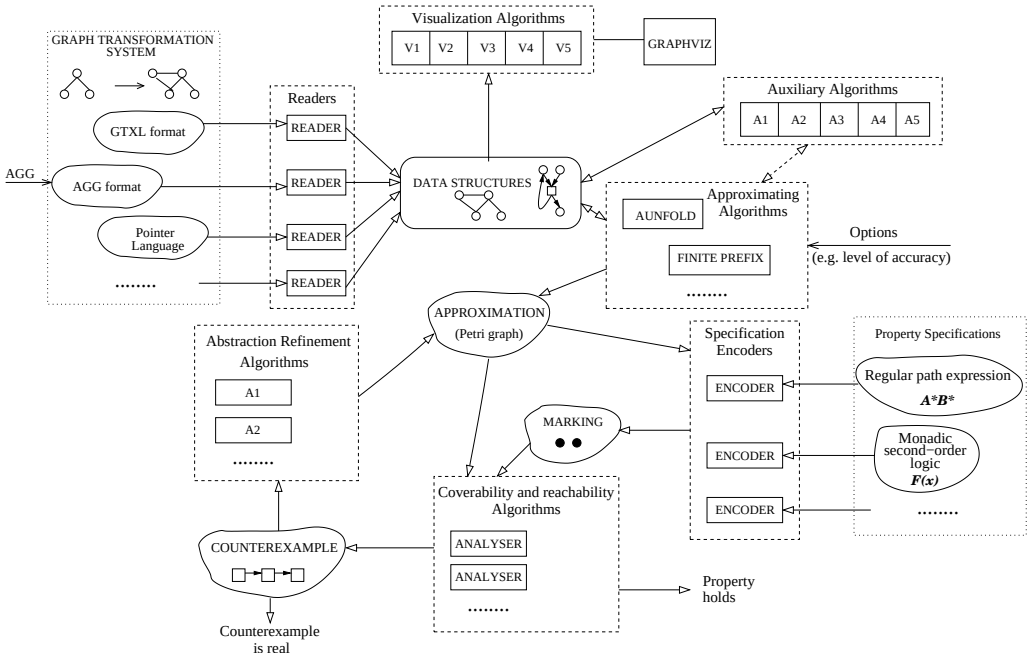

Fig. 5. Schematic representation of AUGUR 2.

We will explain Fig. 5 in roughly chronological order, starting with the input (a graph transformation system and a specification of the property to be verified) and ending with the final output, which says whether the property holds. The system starts by reading the graph transformation system from an external source. At the moment we consider the following three possibilities:

- Read the GTS from a file in GTXL-format (implemented).
- Use Agg as an input source in order to draw graph transformation systems (under development).
- Write a program in a simple pointer-manipulating language, which is then converted automatically to a graph transformation system (under development).

After reading the GTS from an input source and converting it to the internal data structures it can be visualized using Graphviz and abstracted using, for instance, the approximated unfolding algorithm aunfold. A different possibility is to calculate the finite complete prefix of the unfolding of the system (which—in the case of finite-state systems—represents all reachable graphs in a partially ordered structure).

Apart from the graph transformation system, we require the property which has to be verified as additional input. For this purpose we consider in Augur 2 the following two possibilities:

- Specify a regular expression with the set of hyperedge labels as the alphabet. This regular expression describes forbidden paths which should not occur in any reachable graph (implemented).
- Monadic second-order logic for hypergraphs (under development, for the underlying theory see [7]).

These specification languages have to be translated into properties on Petri net markings, since the analysis has to be done directly on the Petri net structure underlying the Petri graph. The coverability of these markings can then be checked using various algorithms (described above). We also plan to implement an (approximative) reachability checker for Petri nets.

If the property does not hold, a counterexample for the net is generated. In the case of spurious counterexamples one of the refinement algorithms is used to obtain a more exact approximation. This procedure can be iterated.

Whenever a non-spurious counterexample is found, we have detected an error in the GTS, i.e., the property to be verified does not hold.

5 Conclusion

Several tools are available for the analysis of graph transformation systems. While some groups [21,9] 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 [17], which led to the tool GROOVE for verifying finite-state GTS. Properties different from reachability (such as termination and confluence via critical pair analysis) can be analyzed using Agg [19].

In this paper we have summarized our plans for the development of Augur 2, a new version of an analysis and verification tool based on unfolding techniques. Some functionality is already present in the current version Augur 1, furthermore the core part of Augur 2, including the database management, has already been implemented. This tool will enable us to conduct further case studies, which will give us valuable stimulations for the future development of the verification techniques.

Among other ideas our future plans are to implement in AUGUR 2 the possibility to use and analyze attributed graph transformation systems.

Finally, we recently concluded the implementation of a graphical user interface. A screenshot, together with windows visualizing the graph and net components of a Petri graph, is shown in Fig. 6.

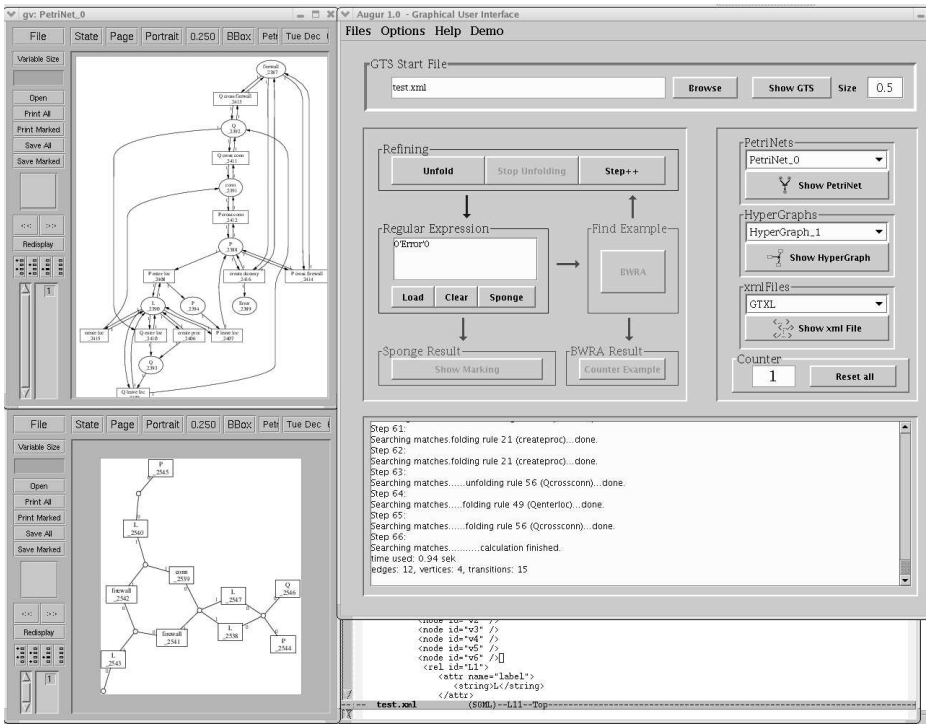

Fig. 6. Screenshot of AUGUR at work.

Acknowledgements: We want to thank all students who helped us with the implementation of AUGUR: Ingo Walther, Sinan Turan, Nicolas Relange, Julian Bart, Martin Horsch, Olga Danylevych and Ganna Monakova. Furthermore we would like to thank Paolo Baldan, Andrea Corradini and Tobias Heindel for valuable discussions on the theoretical background of the tool.

References

[1] Abdulla, P. A., B. Jonsson, M. Kindahl and D. Peled, *A general approach to partial order reductions in symbolic verification*, in: *Proc. of CAV '98* (1998), pp. 379–390, LNCS 1427.

[2] Baldan, P., A. Corradini, J. Esparza, T. Heindel, B. König and V. Kozioura, *Verifying red-black trees*, in: *Proc. of COSMICAH '05*, 2005, proceedings available as report RR-05-04 (Queen Mary, University of London).

[3] Baldan, P., A. Corradini and B. König, *A static analysis technique for graph transformation systems*, in: *Proc. of CONCUR '01* (2001), pp. 381–395, LNCS 2154.

[4] Baldan, P., A. Corradini and B. König, *Static analysis of distributed systems with mobility specified by graph grammars—a case study*, in: *Proc. of IDPT '02* (2002).

[5] Baldan, P., A. Corradini and B. König, *Verifying finite-state graph grammars: an unfolding-based approach*, in: *Proc. of CONCUR '04* (2004), pp. 83–98, LNCS 3170.

[6] Baldan, P. and B. König, *Approximating the behaviour of graph transformation systems*, in: *Proc. of ICGT '02* (2002), pp. 14–29, LNCS 2505.

[7] Baldan, P., B. König and B. König, *A logic for analyzing abstractions of graph transformation systems*, in: *Proc. of SAS '03* (2003), pp. 255–272, LNCS 2694.

[8] Bart, J., "Effiziente Entfaltungsalgorithmen für Graphersetzungssysteme," Master's thesis, Universität Stuttgart (2005), no. 2290.

[9] Dotti, F. L., L. Foss, L. Ribeiro and O. M. Santos, *Verification of distributed object-based systems*, in: *Proc. of FMOODS '03* (2003), pp. 261–275, LNCS 2884.

[10] Dotti, F. L., B. König, O. M. dos Santos and L. Ribeiro, *A case study: Verifying a mutual exclusion protocol with process creation using graph transformation systems*, Technical Report 08/2004, Universität Stuttgart (2004).

[11] Horsch, M., *Test case generation for rule-based translators* (2005), Studienarbeit (Student research project) No. 1984, Universität Stuttgart.

[12] König, B. and V. Kozioura, Augur—*a tool for the analysis of graph transformation systems*, EATCS Bulletin **87** (2005), pp. 125–137, appeared in The Formal Specification Column.

[13] König, B. and V. Kozioura, *Counterexample-guided abstraction refinement for the analysis of graph transformation systems*, in: *Proc. of TACAS '06* (2006), LNCS. to appear.

[14] Lambers, L., *A new version of GTXL: An exchange format for graph transformation systems*, in: *Proc. of GraBaTs'04*, 2004.

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

[16] Relange, N., "Verifikation dynamischer Systeme: Reguläre Ausdrücke zur Spezifikation verbotener Pfade," Master's thesis, Universität Stuttgart (2004), no. 2192.

[17] Rensink, A., *Canonical graph shapes*, in: *Proc. of ESOP '04* (2004), pp. 401–415, LNCS 2986.

[18] Rudolf, M., *Utilizing constraint satisfaction techniques for efficient graph pattern matching.*, in: *Proc. of TAGT '98* (1998), pp. 238–251, LNCS 1764.

[19] Taentzer, G., *AGG: A tool environment for algebraic graph transformation*, in: *Proc. of AGTIVE '99* (1999), pp. 481–488, LNCS 1779.

[20] Turan, S., *Effiziente Berechnung der Überdeckbarkeit bei Petri-Netzen* (2004), Studienarbeit (Student research project), No. 1935, Universität Stuttgart.

[21] Varró, D., *Towards symbolic analysis of visual modeling languages*, in: *Proc. of GT-VMT '02*, ENTCS **72** (2002).

[22] Varró, G. and D. Varró, *Graph transformation with incremental updates*, in: *Proc. of GT-VMT '04*, ENTCS **109** (2004), pp. 71–83.

[23] Zündorf, A., *Graph pattern matching in PROGRES*, in: *Proc. of TAGT '94* (1994), pp. 454–468, LNCS 1073.