Augur 2—A tool for the analysis of (attributed) graph transformation systems using approximative unfolding techniques

April 8, 2008

Contents

1 Introduction

The aim of this tool is the verification of systems described by (attributed) graph transformation systems (GTS, AGTS) using approximated unfoldings [BCK01, BCKar]. Graphs are a natural and convenient means to describe complex structures and graph transformation systems give the possibility to model dynamically changing systems. This is important for the specification of systems, where objects can be created and deleted and the system undergoes structural changes during runtime. For more information on graph transformation systems see [Roz97, EEKR99, EKMR99]. A list of publication describing the theoretical background behind AUGUR can be found at `http://www.ti.inf.uni-due.de/publications/project-augur.shtml`.

It is often convenient to extend a modelling language by adding attributes which carry values of some data types. This is for instance true for coloured Petri nets [Jen86] and attributed GTSs (AGTSs) [EEKR99, Kas05, Tae99, LKW93]. Extending GTSs with attributes allows one to combine the intuitive graphical aspects of the modelled systems with natural data structures, which makes such extended GTSs more suitable for practical applications. Usually this leads to more compact models, because many well-known operations need not be described in an artificial way using the graph structures.

Since data types are often infinite, abstraction of data types is needed in order to do automatic verification. This is usually done in one of the following ways: data abstraction [CGL94], abstract interpretation [Sif83, Cou01] or predicate abstraction [GS97, DDP99, HJMS02, JM06]. In AUGUR 2 the first two approaches are used.

In the last few years the verification tool AUGUR 1 has been developed [KK05] which analyzes graph transformation systems (GTSs)[1]. Using this tool several case studies have been already conducted, verifying, for instance, a mobile system with a firewall [BCK02], a mutual exclusion protocol [DKdSR04] and the insertion of elements into red-black trees [BCE+05a].

The development of AUGUR 1 (the previous version of the tool) started with a small tool that reads GTXL files, constructs an approximating unfolding of the given GTS and writes it in GXL files (GXL[2] respectively GTXL [Lam04] are XML standards for the encoding of graphs and graph transformation systems). In the following, in the course of the extension of the tool we faced the constant necessity of adding new features and new functionality. More specifically, the following

[1] The tool can be obtained from `http://www.ti.inf.uni-due.de/research/augur_1/`.
[2] `http://www.gupro.de/GXL/`

components were added: analysis algorithms for Petri nets [Tur04] based on coverability graphs [Rei85] and backward reachability [AJKP98], an interface to Graphviz[3] for visualization purposes, the possibility to specify forbidden paths in graphs using regular expressions [Rel04], the finite complete prefix technique for graph transformation systems [BCK04, Bar05], the extension of the tool in order to use it for the purpose of test case generation [Hor05]. Probably the most extensive addition was to add support for counterexample-guided abstraction refinement [KK06b, KK05].

The architecture of AUGUR was strongly oriented to the concrete task of approximating unfolding of GTS. 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 was needed which should have a more general and extensible software architecture and should allow an easier extension of the tool with more functionality concerning analysis and visualization methods.

Another new feature of the tool which was almost impossible to implement in the frame of AUGUR 1 was the possibility to work with attributed graphs, i.e., graphs with (integer and string) attributes assigned to nodes and edges and the corresponding extension of existing analysis techniques.

It was also rather hard to extend the input and output interface, for instance with establishing a connection to AGG [Tae99]. All this led us to the idea to create a completely new tool AUGUR 2 with a corresponding functionality and an easily extendable architecture.

In this documentation we describe technical details of installation and usage of the tool. Also, the complete verification example of (attributed) GTS and commented input and output files can be found below. The description of functionality, system architecture and software design of AUGUR 2 can be found in [KK06a, Kozar]. Additional information on the software design of the tool can be found in the design documentation (`http://www.ti.inf.uni-due.de/research/augur/design.pdf`).

Users that are mainly interested in working with the graphical user interface and who do not want to delve too deeply into the technical details are referred to the GUI documentation (`http://www.ti.inf.uni-due.de/research/augur/gui.ps`).

[3] `http://www.graphviz.org/`

2 People

The following people are or were involved either in the theoretical development or in the implementation of the tool:

- Paolo Baldan (Università Ca' Foscari di Venezia, Italy)
- Julian Bart (Universität Stuttgart, Germany)
- Andrea Corradini (Università di Pisa, Italy)
- Olga Danylevych (Universität Stuttgart, Germany)
- Salil Joshi (Indian Institute of Technology (IIT), Delhi, India)
- Tobias Heindel (Universität Duisburg-Essen, Germany)
- Lars Heinemann (Universität Stuttgart, Germany)
- Martin Horsch (Universität Stuttgart, Germany)
- Barbara König (Universität Duisburg-Essen, Germany)
- Bernhard König (Boise State University, USA)
- Vitali Kozioura (Universität Duisburg-Essen, Germany)
- Alberto Lluch Lafuente (Università di Pisa, Italy)
- Michael Mayer (Universität Duisburg-Essen, Germany)
- Anja Monakova (Universität Stuttgart, Germany)
- Nicolas Relange (Universität Stuttgart, Germany)
- Timur Tsotniashvili (Universität Stuttgart, Germany)
- Sinan Turan (Universität Stuttgart, Germany)
- Valentin Wolf (Universität Duisburg-Essen, Germany)
- Ingo Walther (Technische Universität München, Germany)
- Maxim Zaks (Universität Duisburg-Essen, Germany)

The current maintainers of the tool and the documentation are Vitali Kozioura and Barbara König. The web site of AUGUR is at `http://www.ti.inf.uni-due.de/research/tools/augur/`. Please address questions to `barbara_koenig@uni-due.de`.

3 System and Software Requirements

The code is written in C++ and uses `libxml` (`http://xmlsoft.org/`) and `lp_solve` (`http://lpsolve.sourceforge.net/`) libraries. AUGUR 2 has been tested under Linux using `libxml 2.0`. Furthermore the libxml development package (`libxml-dev/libxml-devel`) is needed. For visualization purposes the installation of the Graphviz package (which can be obtained from `http://www.research.att.com/sw/tools/graphviz/`) is required. Optionally one can use the package MetricFF (`http://members.deri.at/~joergh/metric-ff.html`) for checking coverability of non-attributed Petri nets.

4 Installation and Usage

4.1 Installation

The source code of the system is available as a tar.gz archive. After unpacking you should compile the programs by calling `make` from the root directory. In most Linux distributions the library `libxml2` is located in `/usr/include/libxml2/` and `/usr/lib/libxml2.so.2`. If the library is in another directory, please change the file `src/Makefile` (the first three lines) accordingly. You should also check the path to the lpsolve library in the file `src/Makefile`. The default path is `/usr/lib/lp_solve/liblpsolve55.a`. We have experienced some problems by using the precompiled version of the library. In this way we would recommend to compile it from the source code.

The executables can subsequently be found in the directory `/bin` and can be started as shell applications or by using a graphical user interface (GUI).

The recommended way of using AUGUR 2 is with the GUI. The GUI is written in Java and can be called in the following way:

```
java -jar aunfoldGUI.jar
```

The file `aunfoldGUI.jar` is in the distribution of AUGUR 2, while the source code of the GUI is in the directory `/gui`. For additional information on the GUI see the GUI documentation (`http://www.ti.inf.uni-due.de/research/augur/gui.ps`).

If you use AUGUR 2 as a command line application then make sure that the `/bin` directory is added to the `PATH` environment variable.

If you use MetricFF (`http://members.deri.at/~joergh/metric-ff.html`) please copy the executable file "ff" into the directory `/bin`.

In the directory `/example` several examples can be found:

1. `/example/1st_order_logic`: Examples of properties in first-order logic format.
2. `/example/agg`: Examples of GTSs in AGG format.
3. `/example/attributes`: Examples of attributed GTSs (AGTSs) in GTXL format.
4. `/example/gtxl`: Example of GTSs in GTXL format.
5. `/example/new_gtxl`: Example of GTSs in the new GTXL format.
6. `/example/spl`: Example of GTSs in single pointer language (SPL) format.

Most of the examples verified with AUGUR 2 are provided in GTXL format.

4.2 Usage

If you do not use the GUI the program can be called from a shell as follows:

```
augur -db=DATABASE -sc=SCENARIO [options] input_files output_files
```

The field DATABASE should contain the path to the XML file which contains a database of the available scenarios and algorithms. In the field SCENARIO one specifies an alias of the scenario to be executed. The following scenarios are currently available in AUGUR 2 (algorithm aliases are listed in the database description):

- `aunfold`: Construction of the unfolding. Possible algorithms are *approximating unfolding*, *finite prefix*, *finite complete prefix*. The input file is a GTS and the output file is a Petri graph.
- `property2marking`: Encoder from a property (which is a regular expression or a first order logic formula) to a marking. Possible algorithms are *simple path encoder - sponge*, *encoder to semilinear set* (the marking is then the first element in the representation of the semilinear set), *logic encoder*.
- `cover`: Checking the coverability property for the given marking. Possible algorithms are *coverability graph*, *backward coverability*, *approximating reachability*, *metric FF*.

- `refinement`: Counterexample-based abstraction refinement. Here the needed method of refinement (structural or attribute-based refinement) will be detected automatically.
- `refinement_loop`: Full mechanization of the verification procedure.
- `gts_emulator`: Emulates the application of the rules in the given GTS.
- `rules2ps`: Visualizes the given GTS and outputs it in a postscript file.
- `hg2ps`: Visualizes the hypergraph component of the Petri graph.
- `pn2ps`: Visualizes the Petri net component of the Petri graph.
- `spl`: Converts an input file in the simple pointer language (SPL) to GTXL (i.e., to a graph transformation system that is an implementation of the program).
- `test`: This scenario is used for debugging and testing.

In order to obtain the list of command line parameters needed for each scenario one can start AUGUR 2 with the chosen scenario but without parameters:

```
> bin/augur -db=db/default.xml -sc=aunfold
loading database
severe error: a SCENARIO named "aunfold" received 0 instead of 2.

Usage: augur -sc=aunfold [options] grammar petrigraph

grammar           graph grammar in GTXL format
petrigraph        output destination for the unfolding,
                     a Petri graph in GXL format
```

If the output files already exist, they are overwritten without warning.

Available options are:

`-q` quiet mode; nothing is written to stdout, only the output file is created.

`-d` debugging mode; internal information, such as the nodes of the coverability graph, is dumped to stdout during the calculation.

`-t` trace mode; before each folding or unfolding operation the intermediate Petri graph is written to the output file.

`-nc` do not check covering information when folding or unfolding; this is used for speeding up the construction in the case of complex coverability com-

putations. The result is still a correct over-approximation, but may be less precise.

`-nr` do not enforce the irredundancy condition.

`-depth=k` the k-depth over-approximation, also called k-covering, will be constructed.

`-nfold=k` no folding steps will be made and unfolding steps will only be executed up to depth less than k.

`-step=n` only n steps of the algorithm will be executed.

`-nmapping` no mapping information will be saved in the output file; mappings are needed for the next version of the system.

`-timeout=n` timeout in seconds.

`-size=s` size of the hypergraphs in the visualization of the GTS (default: 0.5).

The corresponding parameter in the command line has priority over the one set in the the database.

The program is not interactive and terminates when the calculation is finished.

5 File Formats

5.1 Database Format

Suitable database settings should be made before the tool is used. It is recommended to change the database settings using the GUI. Otherwise please orient yourself on the tags "info", where the possible algorithms are sometimes listed and on the algorithm aliases below.

The database is implemented as a single XML file and consists of three parts. The first part contains a description of the global parameters, below we give an example for the format of global variables. Some parameters are Boolean and some are of type integer.

```
<Globals>
  <debug val="false"/>   // debug mode
  <quiet val="false"/>   // quite mode
  <ncov val="true"/>     // do not use coverability by unfolding
  <coloured val="true"/> // is attributed
```

```
  <nfold val="false"/>   // do not fold
  <red val="true"/>      // use irredundancy condition
  <mapping val="true"/>  // use mappings in GTXL
  <trace val="false"/>   // give out traces
  <depth val="0"/>       // do not fold until the "val"
  <unfdepth val="-1"/>   // construct only until depth "val"
  <maxstep val="-1"/>    // make only first "val" steps
  <full_coverability_graph val="false"/>
                         // full coverability graphs
  <clear_cov_count val="4"/>
                         // clear count by incremental coverability
  <mod_base val="2"/>       // base for modulo abstraction
  <interval_from val="0"/>  // left bound of interval abstraction
  <interval_to val="1"/>    // right bound of interval abstraction
  <ref_loop_limit val="5"/> // number of refinement steps
  <attr_ref_count val="5"/> // number of attribute refinement steps
</Globals>
```

The second part describes algorithms and their interaction in the format as in the example below. Here for the algorithm *alg_1* which is not reusable (explanation see below) at the label *l_1* we have following possibilities: if in the stack of the current instance of *alg_1* the algorithms *alg_3* and *alg_4* were called at the labels *l_3* and *l_4* respectively then algorithm *alg_5* will be called. Otherwise we call the default algorithm *alg_2*. The interactions can be quite complex and the general idea is described in [KK06a].

```
<algorithm name="alg_1" reusable="false">
   <label name="l_1">
     <default algorithm="alg_2"/>
     <info>description</info>
     <expert status="true">
        <history>
          <happen algorithm="alg_3" label="l_3"/>
          <happen algorithm="alg_4" label="l_4"/>
          <call algorithm="alg_5"/>
        </history>
     </expert>
   </label>
</algorithm>
```

Below are some algorithm aliases that are currently used:

- *causality_relation*: calculation of a causality relation (needed for the merging of Petri graphs).
- *findmatch*: standard matching algorithm.
- *findmatch_by_sp*: matching algorithm using the searchplan technique.
- *first_matchhandler*: matchhandler for the unfolding.
- *find_all_matches*: computes all possible matches with the standard matching algorithm.
- *find_all_matches_by_sp*: computes all possible matches with the searchplan technique.
- *fold*: folding algorithm.
- *merge_petrigraph*: merging of a Petri graph.
- *merge_petrigraph_uf*: merging of a Petri graph using the union-find algorithm.
- *reduce_transitions*: irredundancy check.
- *second_matchhandler*: matchhandler for folding.
- *unfold*: unfolding algorithm.
- *agg_reader*: reader from AGG format.
- *coverability*: coverability calculation with a coverability graph.
- *bwra*: backward coverability algorithm.
- *reachability_analyse*: reachability (coverability) analysis using `lp_solve`.
- *abstraction_refinement*: abstraction refinement.
- *second_matchhandler_refinement*: matchhandler for abstraction refinement.
- *gtxl_reader*: reader from GTXL format.
- *new_gtxl_reader*: reader from the new GTXL format.
- *spl_reader*: reader from SPL format
- *gxl_reader*: reader from GXL format.
- *gxl_writer*: writer in GXL format.
- *approximated_unfolding*: the entire unfolding procedure.

- *coverability_test*: tests coverability graphs obtained by the incremental approach.
- *expression_engine*: standard attribute engine.
- *expression_engine_mp*: $\{-,+\}$-abstraction of attributes.
- *expression_engine_mod*: modulo-abstraction of attributes.
- *expression_engine_mp0*: $\{-,0,+\}$-abstraction of attributes.
- *expression_engine_pn*: $[0,N]$-abstraction of attributes.
- *expression_engine_mn*: $[M,N]$-abstraction of attributes.

For each algorithm we indicate in the database if the algorithm is reusable or not. Non-reusable algorithms will be created each time, when they will be called, whereas for reusable algorithms the same instance will be used during the whole session.

If you want to use incremental coverability you should set the "reusable" flag of the algorithm "coverability" to true. Also, do not forget to set the value of the "ncov" parameter to true.

Below is an example of using the algorithms in the database. Here, different matchhandlers are used depending on the previous history. The history indicates if the current algorithm runs in the abstraction refinement procedure or in the folding or (default) in the unfolding procedures.

```
<algorithm name="findmatch" reusable="false">
 <label name="call_matchhandler">
   <default algorithm="first_matchhandler"/>
   <info>string</info>
   <expert status="true">
   <history>
    <happen algorithm="abstraction_refinement"
                                          label="call_unfolding"/>
    <happen algorithm="first_matchhandler" label="call_findmatch"/>
     <call algorithm="second_matchhandler_refinement"/>
   </history>
   <history>
    <happen algorithm="first_matchhandler" label="call_findmatch"/>
     <call algorithm="second_matchhandler"/>
   </history>
   </expert>
```

```
 </label>
</algorithm>
```

The third part of the database file is a description of scenarios which are algorithms having only external files as input and output. Scenarios are high-level algorithms describing the current task of the tool. In the database we indicate which algorithm will be used in which scenario. For example in scenario *aunfold* (see below) this may either be *approximating unfolding* or *finite prefix* or *finite complete prefix*.

```
<algorithm name="main" reusable="false">
   <label name="scenario_1">
       <default algorithm="alg_1"/>
       <info>description</info>
       <expert status="false"/>
   </label>
   <label name="scenario_2">
       <default algorithm="alg_2"/>
       <info>description</info>
       <expert status="false"/>
   </label>
</algorithm>
```

As an example we give here the scenario "property2marking" The possible encoder algorithms are listed inside the "info" tag.

```
<label name="scenario_property2marking">
  <default algorithm="sponge_call"/>
    <info>sponge, sponge_call, logic_converter,
                      reg2marking, reg2semilin</info>
    <expert status="false"/>
</label>
```

5.2 Input Formats

5.2.1 Example in GTXL format

In this section we present a commented description of an AGTS from Fig 1 in GTXL format. The example consists of an initial graph, which consists one edge labelled "A"having two attributes "a1" and "a2", and one rewriting rule.

```
<?xml version="1.0"?>
<!DOCTYPE gtxl SYSTEM "gtxl.dtd">
```

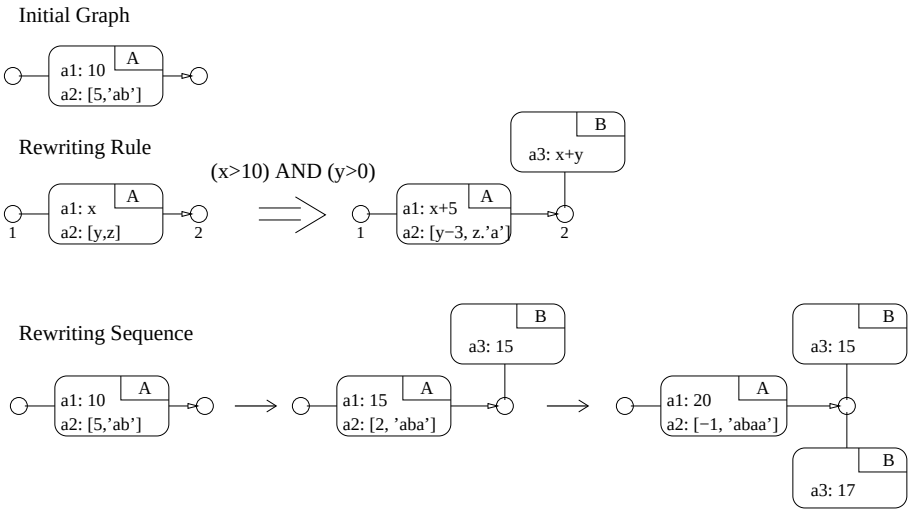

Figure 1: First example of an attributed graph transformation system

```
<GTS id="simple">
 <Initial> // initial graph
    <Graph id="Private Server and public generator" edgeids="true" hypergraph="true"
         edgemode="undirected">
      <node id="n1"/> // nodes with unique ids
      <node id="n2"/>
      <rel id="ida">  // edge with unique ids
        <attr name="label">  // label of an edge is
          <string>A</string> // written as a string
        </attr>
        <attr kind="attr-int" name="a1"> // first attribute of type integer with value 10
          <string>10</string>            // written as a string
        </attr>
        <attr kind="attr-int-str" name="a2"> // second attribute is a tuple integer-string
          <string>[5,'ab']</string>          // with a value written as a string
        </attr>
        <relend id="spriv" target="n1" startorder="0" /> // connections of an edge
        <relend id="spriv" target="n2" startorder="1" />
      </rel>
    </Graph>
 </Initial>

 <Rule id="Rule1"> // first rule
  <precondition>   // precondition = guard expression
    <condition>
      <attrCondition>
        <string>x&gt;10 AND y&gt;0</string> // expression itself as a string
      </attrCondition>
    </condition>
  </precondition>

  <LHS>                      // left-hand side of the rule
    <RuleGraph id="lr1">
      <Graph id="lgraph1" edgeids="true" hypergraph="true" edgemode="undirected">
        <node id="nl1"/>     // nodes
        <node id="nl2"/>
        <rel id="la1" >      // edge
          <attr name="label">
            <string>A</string> // label
          </attr>
```

```
        <attr kind="attr-int" name="a1"> // attributes are now from the term algebra
          <string>x</string>             // here the value is x
        </attr>
        <attr kind="attr-int-str" name="a2"> // tuple
          <string>[y,z]</string>
        </attr>
        <relend id="lspriv1" target="nl1" startorder="0" />
        <relend id="lspriv2" target="nl2" startorder="1" />
      </rel>
    </Graph>
  </RuleGraph>
</LHS>

<RHS>                      // right-hand side of the rule
  <RuleGraph id="rr1">
    <Graph id="rgraph1" edgeids="true" hypergraph="true" edgemode="undirected">
      <node id="nr1"/>     // nodes
      <node id="nr2"/>
      <rel id="ra1">       // first edge
        <attr name="label">
          <string>A</string> // label A
        </attr>
        <attr kind="attr-int" name="a1"> // attribute from the term algebra
          <string>x+5</string>           // value
        </attr>
        <attr kind="attr-int-str" name="a2">
          <string>[y-3,z.'a']</string>
        </attr>
        <relend id="rspriv1" target="nr1" startorder="0" /> // two connections to nodes
        <relend id="rspriv2" target="nr2" startorder="1" />
      </rel>

      <rel id="rb2">        // second edge
        <attr name="label">
          <string>B</string>  // label B
        </attr>
        <attr kind="attr-int" name="a3"> // attribute from the term algebra
          <string>x+y</string>           // value
        </attr>
        <relend id="rspriv3" target="nr2" startorder="0" /> // only one connection
      </rel>
    </Graph>
  </RuleGraph>
</RHS>

<Mapping id="cps_mapping">   // interface between left and right hand-sides of the rule
  <MapElem from="nl1" to="nr1" />      // as a map on nodes
  <MapElem from="nl2" to="nr2" />
</Mapping>

 </Rule> // end of the rule
</GTS>   // end of GTS
```

5.2.2 Example in the new GTXL format

In this section we present a commented description of the AGTS from Fig 1 in the new GTXL format. This format was proposed in [Lam04] and is used for instance in the AGG tool [Tae99]. In Augur 2 this format is used as an alternative to the GTXL format described in the last chapter. We use the same GTS from the

previous section in order to illustrate its usage.

The main difference of the new GTXL format compared to the previous one is the description of a type graph which is here called “Schema Graph”. Here, one describes the map from possible labels to the corresponding data types. For example if an edge has label “A” then the type graph constrains it to have the attributes “a1” and “a2” of types “integer” and “[integer,string]” correspondingly. Later, in the concrete hypergraphs, only the values of attributes (but not their types) will be given.

The next specific characteristic of the new GTXL format is that each rule is specified as consisting of three parts: “preserved”, “deleted” and “created”. In the part “preserved” one provides a rule interface, which is in our case discrete and consist only of preserved nodes. In the “deleted” (“created”) part one specifies the nodes and edges of the left-hand side (right-hand side) of the rule which are not preserved (i.e., either deleted or created).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gtxl SYSTEM "gtxl.dtd">
<gtxl xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:xalan="http://xml.apache.org/xalan"
xmlns:lxslt="http://xml.apache.org/xslt">
 <graph edgeids="true" id="Schema Graph">  // Schema Graph is a type graph of GTS
  <node id="stdNode">                      // There is only one type of nodes
   <graph edgeids="true">
    <rel id="A">                                // Declaration of an edge with a label "A"
      <attr name="a1" kind="AttrType">          // Declaration of an attribute "a1"
        <string>int</string>                    // of type integer
      </attr>
      <attr name="a2" kind="AttrType">          // Declaration of an attribute "a2"
        <string>int-str</string>                // of type integer-string
      </attr>
      <relend target="stdNode" startorder="0"/> // Connections of all edges labelled with "A"
      <relend target="stdNode" startorder="1"/> // are the same
    </rel>
    <rel id="B">                                // Declaration of an edge with a label "B"
      <attr name="a1" kind="AttrType">          // Declaration of an attribute "a3"
        <string>int</string>                    // of type integer
      </attr>
      <relend target="stdNode" startorder="0"/> // Only one connection
    </rel>
   </graph>
  </node>
 </graph>
 <gts id="I2" approach="DPO">                                  // Begin of GTS
  <type xlink:type="simple" xlink:href="#SchemaGraph"/>        // Link to the schema graph

  <initial>                                                    // Initial graph

   <graph id="ini" edgeids="true" hypergraph="true" edgemode="undirected">
    <node id="v1">                                             // First node
     <type xlink:href="stdNode"/>                              // linked to the Schema graph
    </node>
    <node id="v2">                                             // Second node
     <type xlink:href="stdNode"/>
    </node>
```

```
  <rel id="r1">                                          // Edge description
   <type xlink:href="A"/>                                // Type "A"
   <attr name="a1">                                      // Attribute "a1"
    <string>10</string>                                  // with value 10
   </attr>
   <attr name="a2">                                      // Attribute "a2"
    <string>[5,'ab']</string>
   </attr>
   <relend id="relend1" target="v1"/>                     // Connections
   <relend id="relend2" target="v2"/>
  </rel>

 </graph>
</initial>

<rule name="create proc" id="Rule1">          // Description of rule
  <precondition>                              // precondition = guard expression
    <condition>
      <attrCondition>
        <string>x&gt;10 AND y&gt;0</string>   // expression itself as a string
      </attrCondition>
    </condition>
  </precondition>

 <preserved>                                  // Preserved is usually the discrete interface
  <graph id="G1">                             // i.e., only nodes
   <node id="N1">
    <type xlink:href="stdNode"/>
   </node>
   <node id="N2">
    <type xlink:href="stdNode"/>
   </node>
  </graph>
 </preserved>

 <deleted>                                    // To delete are all edges from
                                              // the left-hand side of the rule
  <rel id="Rel_1">                            // Edge "A"
   <type xlink:href="A"/>                     // Link to the Schema Graph
    <attr name="a1">                          // Attribute "a1"
      <string>x</string>                      // with value x
    </attr>
    <attr name="a2">                          // Attribute "a2"
      <string>[y,z]</string>
    </attr>
   <relend id="Rel1_relend1" target="N1"/>     // Connections to N1
   <relend id="Rel1_relend2" target="N2"/>     // and N2
  </rel>
 </deleted>

 <created>                                    // To create are all edges from
                                              // the right-hand side of the rule
  <rel id="Rel2">                             // Edge "A"
   <type xlink:href="A"/>
    <attr name="a1">                          // Attribute "a1"
      <string>x+5</string>                    // with value x+5
    </attr>
    <attr name="a2">                          // Attribute "a2"
      <string>[y-3,z.'a']</string>
    </attr>
   <relend id="Rel2_relend1" target="N1"/>
   <relend id="Rel2_relend2" target="N2"/>
  </rel>
```

```
    <rel id="Rel3">                              // Edge "B"
     <type xlink:href="B"/>
      <attr name="a3">                           // Attribute "a3"
        <string>x+y</string>                     // with value x+y
      </attr>
     <relend id="Rel2_relend1" target="N2"/>
    </rel>

   </created>

  </rule>

 </gts>
</gtxl>
```

5.2.3 Example of an SPL program

In this section we present an example of a program in SPL format, which can be used as an input for AUGUR 2. For this purpose the corresponding reader (SPL reader) should be switched on in the database. The program will be then translated into an AGTS, which can be further approximated and analyzed. The details of the SPL language and its translation into AGTSs can be found in [Tso06].

An SPL program consists of a type declaration part, where a record consisting of one integer (`info_rec`) and a list of such records (`list_rec`) are defined. Then the variables (`list`, `help` of type "pointer to list_rec" and `i` of type "integer") are declared and initialized. After this the functionality of the program is specified: First, an infinite list of records is produced and then, during the iteration through this list, each element in the record will be increased by one.

For each pointer (`list` and `help`) the error rule will be added which should check if at some point in the execution of the program an assignment to the null-pointer is made. When the SPL program is translated to an AGTS the special edge "error" will be created. The property to verify is that no "error"-edge will be ever created.

```
TYPE info_rec;
TYPE list_rec IS RECORD (next: POINTER TO RECORD list_rec, info: POINTER TO
RECORD info_rec);
TYPE info_rec IS RECORD(value: INTEGER);

VAR list: POINTER TO RECORD list_rec;
VAR help: POINTER TO RECORD list_rec;
VAR i: INTEGER;

i:=0;
NEW(list);
NEW(list.info);
help:=list;
```

```
help.info.value:=i;

WHILE(TRUE) DO
   i:=i+1;
   NEW(help.next);
   help:=help.next;
   NEW(help.info);
   help.info.value:=i;
OD;
help:=list;
WHILE(TRUE) DO
   help.info.value:=help.info.value+1;
   help:=help.next;
OD;
SKIP;
```

5.2.4 Interface to AGG

AUGUR 2 allows one to read (attributed) GTSs visually created in the AGG [Tae99]. Details can be found in the documentation available at `s http://www.ti.inf.uni-due.de/research/augur/agg.pdf`.

5.2.5 Regular Expressions

Given an hypergraph and a regular expression, AUGUR 2 generates conditions on the markings of the hypergraph which ensure that a path in the hypergraph matches the regular expression. In other words, it computes how many parallel instances of edges are necessary in order to obtain a path corresponding to the regular expression that traverses every edge at most once. This information, given as a list of markings, can be used in order to show that no graphs containing undesirable paths can be reached.

If one wants to specify a single 0-ary edge (for example labelled with "Error") not connected to any nodes one writes `0'Error'0` if one uses the encoder `sponge` and `Error` if one uses an encoder to semilinear sets. In all other cases the regular expressions are syntactically for both encoders. For example, `'A''B'` specifies connected edges labelled 'A' and 'B'. In general, the regular expression must be given according the following grammar:

```
reg_exp ::= reg_exp + term            /* union */
term    ::= term factor               /* concatenation */
factor  ::= ( reg_exp )
            | factor *                /* kleene-closure */
            | epsilon
            | motif
```

```
epsilon ::= ~ | ""                    /* epsilon */
motif   ::= char
            | ' word '
            | digit word digit
            | ' number word number '
word    ::= char word
            | char
char    ::= [A-Za-z_]
number  ::= digit number
            | digit
digit   ::= [0-9]
```

Note: the delimiters space, tab and newline are not taken into account.

For more information on regular expressions and their usage see [HK04, Rel04].

5.2.6 First Order Logic

In AUGUR 2 there exists a possibility to specify properties in form of first order logic on hypergraphs. Please note that these properties can not be analyzed with usual coverability algorithms and hence one should use the technique based on linear equations (implemented with `lp_solve`).

For a detailed description of the format of formulas in first-order logic and also of their analysis please refer to [vM06].

5.3 Output Formats

5.3.1 Example in GXL format

In this section we present a commented description of a Petri graph obtained as an over-approximation for the example from Fig 1. The hypergraph component consists of two edges labelled "A" and "B" and two nodes. In the Petri net component there is only one transition corresponding to the rule "Rewriting Rule". Note that the guard of the transition is exactly the same as the guard of the rule.

```
<?xml version="1.0"?>
<gxl>
  <graph id="augur2.out" hypergraph="true" edgemode="undirected">
    <node id="_22">              // a node can be either a hypergraph node or a transition
      <attr name="vertex"/>      // this is a hypegraph node
      <attr name="iota_mapping"> // iota match from the initial graph
        <string>n1</string>
```

```
  </attr>
</node>
<node id="_23">               // node
  <attr name="vertex"/>       // hypegraph node
  <attr name="iota_mapping">
    <string>n2</string>
  </attr>
</node>
<rel id="_24">               // first edge
  <attr name="label">
    <string>A</string>       // labelled with "A"
  </attr>
  <attr name="initial_marking">   // initial marking of the edge
    <token>
      <int>1</int>                          // one token
      <attr kind="attr-int" name="a1">      // attribute "a1" of the token
        <string>10</string>                 // value
      </attr>
      <attr kind="attr-int-str" name="a2">  // atribute "a1" of the token
        <string>[5,'ab']</string>           // value
      </attr>
    </token>
  </attr>
  <attr name="iota_mapping">                // iota match for the edge
    <string>ida</string>                    // id in the initial graph
  </attr>
  <relend target="_22" role="vertex" startorder="0"/>    // connections
  <relend target="_23" role="vertex" startorder="1"/>
  <relend target="_26" role="postset">      // postset for the transition with id="_26"
    <attr name="weight">                    // weight on the arc is 1
      <int>1</int>
    </attr>
    <attr kind="attr-int" name="a1">        // attribute fron term algebra on the arc
      <string>x+5</string>                  // value
    </attr>
    <attr kind="attr-int-str" name="a2">    // second attribute
      <string>[y-3,z.'a']</string>
    </attr>
  </relend>
  <relend target="_26" role="preset">       // postset
    <attr name="weight">
      <int>1</int>
    </attr>
    <attr kind="attr-int" name="a1">        // first attribute
      <string>x</string>
    </attr>
    <attr kind="attr-int-str" name="a2">    // second attribute
      <string>[y,z]</string>
    </attr>
  </relend>
</rel>
<rel id="_25">                              // second edge
  <attr name="label">
    <string>B</string>                      // labelled with "B"
  </attr>
  <attr name="initial_marking"/>            // not marked initiallly
  <attr name="iota_mapping">                // iota is empty
    <string></string>
  </attr>
  <relend target="_23" role="vertex" startorder="0"/>  // connection
  <relend target="_26" role="postset"      // postset for the transition with id = "_26"
    <attr name="weight">
      <int>1</int>                          // weight on arc is 1
    </attr>
    <attr kind="attr-int" name="a3">        // attribute on arc
```

```
        <string>x+y</string>
      </attr>
    </relend>                              // not in the preset of any transition
  </rel>
  <node id="_26">                          // transition with id = "_26"
    <attr name="transition"/>
    <attr name="rule">
      <string>Rewriting Rule</string>      // name of the rule
    </attr>
    <attr kind="attr-bool">                // guard function of the type boolean
      <string>x&gt;=10 AND y&gt;0</string> // value
    </attr>
    <attr name="mu_mapping">               // mu match from a rule to the Petri graph
      <string>nl1,22;nl2,23;cps_lh_gpub,24;nr1,22;nr2,23;cps_rh_Sprv,24;idbr,25;</string>
    </attr>                                // match is based on ids
  </node>
 </graph>
</gxl>
```

6 Verification Example

In this section we represent the verification protocol for the attributed example from Fig. 2 (file `augur/example/attributed/conn2.xml` in the tool).

We use here the command line version of the tool. The property we want to verify is that no edge labelled "Error" will ever be created. We use modulo abstraction of attributes with the initial value $mod = 1$. This value should be set in the database in the following way:

```
<Globals>
  ...
  <mod_base val="1"/>
  ...
</Globals>

...

<algorithm name="storage" reusable="true">
    <label name="attribute_engine">
      <default algorithm="expression_engine_mod"/>
      <info>Here an attribute engine (abstraction) is set</info>
      <expert status="false"/>
    </label>
    ...
</algorithm>
```

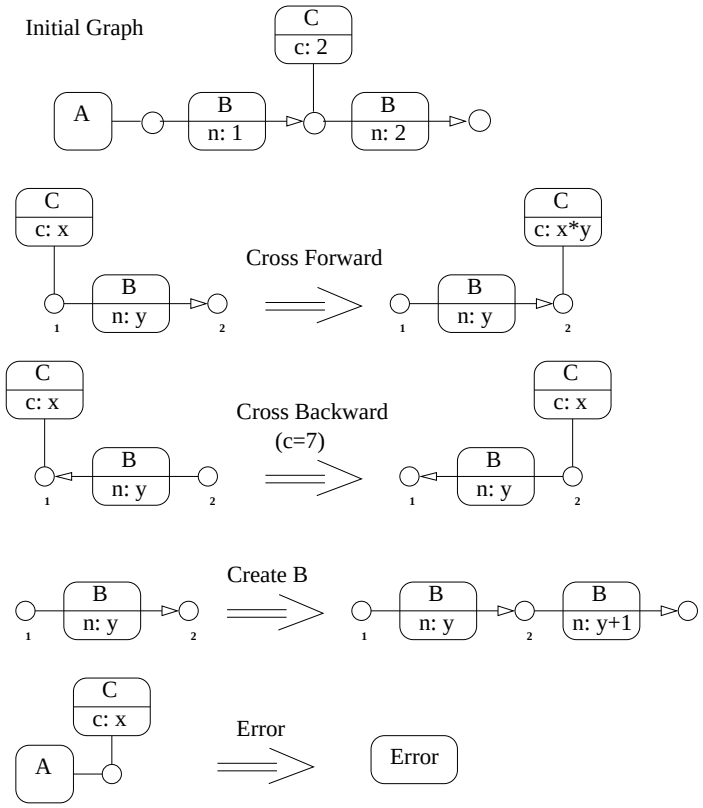

Figure 2: Second example of an attributed graph transformation system

```
  ...
```

First, construct the approximated unfolding using the scenario `aunfold`:

```
> bin/augur -db=db/default.xml -sc=aunfold
example/attributes/conn2.xml work/PetriNet_0.xml

loading database
Running scenario ...
step 1:
searching matches.unfolding rule 9 (Cross Forward) ...done.
step 2:
searching matches.folding rule 9 (Cross Forward) ...done.
step 3:
searching matches..unfolding rule 10 (Cross Backward) ...done.
step 4:
searching matches.folding rule 9 (Cross Forward) ...done.
step 5:
searching matches.folding rule 9 (Cross Forward) ...done.
step 6:
searching matches.folding rule 9 (Cross Forward) ...done.
step 7:
searching matches...unfolding rule 11 (Create C) ...done.
step 8:
searching matches.folding rule 9 (Cross Forward) ...done.
step 9:
searching matches.folding rule 9 (Cross Forward) ...done.
step 10:
searching matches....unfolding rule 12 (Error) ...done.
step 11:
searching matches....calculation finished.
time used: 0.01 sec.
edges: 4, vertices: 1, transitions: 4
```

The computed Petri graph (Petri net and graph component) can be visualized using scenarios `pn2ps` and `hg2ps`. Furthermore, the initial graph and the rules can be visualized using the scenario `rules2ps`.

For instance, the visualization of the hypergraph can be done in the following way:

```
> bin/augur -sc=hg2ps -db=db/default.xml
```

```
work/HyperGraph_0.xml work/out.hg.ps
```

The result is saved in the postscript file `out.hg.ps`.

Now we convert the regular expression `0'Error'0` describing an error edge of arity 0 and corresponding to the property to be verified. This regular expression is converted into a marking of the Petri graph via scenario `property2marking`.

```
> bin/augur -db=db/default.xml -sc=property2marking
work/PetriNet_0.xml work/regexpr work/marking

loading database
Running scenario ...
info: Hypergraph: vertices: 101   edges: 102 <S>(101 ) 103 <P>(101 )
104 <Error>() 105 <C>(101 101 )
info: result: MinCover {
Path 13926736 -> 164505152 f: 104 |-> 1
}

FORMULA ( Error_104 >= 1 )
```

The regular expression is saved in the file `work/regexpr` and the resulting marking is written to `work/marking`. Regular expression can also be more complex, consider for instance `'A'C*'B'` specifying a chain of "C"s from an edge labelled "A" to an edge labelled "B". Note however that general regular expressions are not fully supported by abstraction refinement.

We continue with our example. The file `marking` specifying markings that should *not* be coverable contains the following content:

```
MARKING Error_104: 1;
```

If this marking is not coverable, then the verified property is true. Now we are ready to check if this marking is coverable in the underlying Petri net. To this aim we call the scenario `cover` with a standard coverability algorithm, which checks whether a marking can be covered, in the following way:

```
> bin/augur -db=db/default.xml -sc=cover
work/PetriNet_0.xml work/marking work/transpath

loading database
Running scenario ...
Searching the trace to id=5
write counter-example to augur/work/transpath
```

```
The Final Marking(s) are coverable
time used: 0 sec.
```

We can see that the marking is coverable in the Petri net and the trace is saved in the file `work/transpath`:

```
Error
109
% x(int): 0
```

But this does not necessarily mean that the property is false, since we work with an over-approximation. Let us now call counterexample-based abstraction refinement with scenario `refinement` in order to obtain a more precise approximation.

```
> bin/augur -db=db/default.xml -sc=refinement
example/attributes/conn2.xml work/PetriNet_0.xml
work/transpath work/sample.xml work/PetriNet_1.xml

loading database
Running scenario ...
The example is spurious
Transition number 1, id: 109, name: Error,
could not be fired
Structural refinement is needed
equivalence class: nodes ( 2 3 4 )
optimized sample size: 1
step 1:
searching matches.unfolding rule 9 (Cross Forward) ...done.
...
step 33:
searching matches....unfolding rule 12 (Error) ...done.
step 34:
searching matches....calculation finished.
time used: 0.25 seconds
edges: 9, vertices: 3, transitions: 13
```

Now the refined over-approximation is saved in the file `work/PetriNet_1.xml`. From the output of the tool it can be deduced that the refinement step was of the structural kind. If we repeat all earlier verification steps with scenarios `property2marking` and `cover`, then we will see, that the undesirable situation is still possible in the Petri net obtained from the first refinement step (i.e., the

approximation is still too coarse).

In this case scenario `cover` gives the following answer:

```
loading database
Running scenario ...
Searching the trace to id=11
write counter-example to augur/work/transpath

The Final Marking(s) are coverable
time used: 0.01 sec.
```

This means that the marking is still coverable and the new counterexample is:

```
Cross Backward
618
% x(int): 0, y(int): 0

Error
629
% x(int): 0
```

If we restart counterexample-based abstraction refinement, then the new output is:

```
> bin/augur -db=db/default.xml -sc=refinement
example/attributes/conn2.xml work/PetriNet_1.xml
work/transpath work/sample.xml work/PetriNet_2.xml

loading database
Running scenario ...
The example is spurious
Transition number 1, id: 618, name: Cross Backward,
could not be fired
Refinement of attributes is needed
(1) Attributes have been refined
Spurious counter-example is eliminated
```

We see that in this case refinement of attributes was needed and the attribute abstraction was successfully refined. After the refinement step the modulo abstraction value becomes 2 and repeating the verification steps with scenarios `property2marking` and `cover` shows us that the marking corresponding to the edge “Error” is no more coverable:

```
loading database
Running scenario ...

The Final Marking(s) are not coverable.
time used: 0.46 sec.
```

This means we have successfully verified the example.

7 List of examples

Here we comment some of the example files contained in the directory `examples/`:

- `gtxl/dphil_finite.xml` (`gtxl/dphil_infinite.xml`): A system of dining philosophers in a finite-state and in an infinite-state version. In the latter version philosophers are allowed to reproduce and create more philosophers. This example is described in more detail in [BCK01].
- `gtxl/external-internal.xml`: A very simple toy example of external and internal processes living in a network.
- `gtxl/firewall2.xml`: A network of secure and insecure locations, divided by a firewall. This example is described in more detail in [BCK02, KK06b].
- `gtxl/mutual.xml`: A mutual exclusion protocol where a token is passed around. Only the process in possession of the token can use a shared resource. Details can be found in [DKdSR04].
- `gtxl/producer-consumer-directed.xml`, `gtxl/producer-consumer.xml`: A simple toy example describing the production and consumption of messages.
- `gtxl/pub_priv_server.xml`: A system of public and private servers interacting with mobile processes.
- `gtxl/resources.xml`: A system of several processes sharing two resources. This example is described in more detail in [BKK03].
- `gtxl/red-black.xml`, `gtxl/red-black-converted.xml`: Generates all possible red-black trees (a form of balanced search trees) and models the insertion of elements into red-black trees. Details can be found in [BCE+05b].
- `attributed/simple.xml` A very simple example of an attributed GTS.

- `attributed/conn2.xml` The example is used in this paper as a verification example (see Section 6).
- `attributed/leader.xml` A system describing a leader election protocol in a ring architecture [Lyn96] with AGTSs. Details can be found in [Kozar].

8 Limitations and Known Problems

If the abstraction refinement is used one should insert an additional rule to the system whose left-hand side L corresponds to the property to be verified. The right-hand side of this new rule contains a 0-ary edge labelled `Error` and we will check that this edge is not coverable in the approximation.

The reason for introducing such a new rule and not working with regular expressions directly is the following: Assume that we have found, in the approximating Petri net, a sequence of transitions $t_1, \ldots, t_n$, corresponding to rules $r_1, \ldots, r_n$, which generates a marking covering L. A corresponding sequence could very well exist in the graph transformation system itself, with the only difference that the graph obtained in this way does not contain L, but L', such that L can be obtained by merging nodes of L'. For instance, L' could consist of two disconnected edges. In order to recognize this as a spurious run, we will hence add the new rule, that will be fired in the approximation, but can not be applied in the original system.

A left-hand side should not contain two edges having the same label. There is no error message if this should be the case, but there is no guarantee for the correctness of the result (especially in the case of abstraction refinement).

For all tested examples, the running time is dominated by the calculation of the coverability graph. The calculation of the coverability graph can be ignored by adding the option `-nc` to the program call or by setting the `nc` flag in the database to true. This can speed up the program essentially.

In the regular expressions it is required that all edge labels consist letters only, labels containing numerical characters are disallowed. Note that if one wants to specify a single 0-ary edge (for example labelled with "Error") one writes `0'Error'0` if one uses the encoder `sponge` and `Error` if one uses an encoder to semilinear sets.

Note also that not all techniques are implemented for attributed GTSs. For example the marking check with MetricFF or with techniques based on linear equations is not possible.

Also errors in the input file are not always detected, and it is not always checked if the prerequisites for the algorithm are met.

9 License Agreement

References

[AJKP98] 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.

[Bar05] Julian Bart. Effiziente Entfaltungsalgorithmen für Graphersetzungssysteme. Master's thesis, Universität Stuttgart, June 2005. No. 2290.

[BCE+05a] 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).

[BCE+05b] 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).

[BCK01] 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.

[BCK02] Paolo Baldan, Andrea Corradini, and Barbara König. Static analysis of distributed systems with mobility specified by graph grammars—a case study. In H. Ehrig, B. Krämer, and A. Ertas, editors, *Proc. of IDPT '02 (Sixth International Conference on Integrated Design & Process Technology)*. Society for Design and Process Science, 2002.

[BCK04] 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.

[BCKar] Paolo Baldan, Andrea Corradini, and Barbara König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, to appear.

[BKK03] 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.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[Cou01] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Proc. of CAV'99*, pages 160–171, 1999.

[DKdSR04] 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.

[EEKR99] 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.

[EKMR99] 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.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proc. of CAV'97*, pages 72–83. Springer, 1997. LNCS 1254.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proc. of POPL '02*, pages 58–70. ACM, 2002.

[HK04] Markus Holzer and Barbara König. Regular languages, sizes of syntactic monoids, graph colouring, state complexity results, and how these topics are related to each other. *EATCS Bulletin*, 83:139–155, June 2004. Appeared in The Formal Language Theory Column.

[Hor05] Martin Horsch. Test case generation for rule-based translators, June 2005. Studienarbeit (Student research project), No. 1984.

[Jen86] Kurt Jensen. Coloured Petri Nets. In *Advances in Petri Nets*, pages 248–299, 1986.

[JM06] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *Proc. of TACAS'06*, pages 459–473, 2006.

[Kas05] Harmen Kastenberg. Towards attributed graphs in Groove. In *Proceedings of Workshop on Graph Transformation for Verification and Concurrency*, volume 05-34 of *CTIT Technical Report*, pages 91–98, 2005.

[KK05] 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.ti.inf.uni-due.de/research/augur_1/doc_augur_1.ps`.

[KK06a] 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 63–72, 2006. ENTCS.

[KK06b] 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 3920.

[Kozar] Vitali Kozioura. *Abstraction and Abstraction Refinement in the Verification of Graph Transformation Systems. PhD thesis.* Universität Duisburg-Essen, To appear.

[Lam04] Leen Lambers. A new version of GTXL: An exchange format for graph transformation systems. In *Proc. Workshop on Graph-Based Tools (GraBaTs'04)*, pages 51–63, 2004.

[LKW93] Michael Löwe, Martin Korff, and Annika Wagner. An algebraic framework for the transformation of attributed graphs. In *Term graph rewriting: theory and practice*, pages 185–199. John Wiley and Sons Ltd., 1993.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc, 1996.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.

[Rel04] Nicolas Relange. Verifikation dynamischer Systeme: Reguläre Ausdrücke zur Spezifikation verbotener Pfade. Master's thesis, Universität Stuttgart, September 2004. No. 2192.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, volume 1. World Scientific, 1997.

[Sif83] Joseph Sifakis. Property preserving homomorphisms of transition systems. In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, pages 458–473. Springer-Verlag, 1983.

[Tae99] 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.

[Tso06] Timur Tsotniashvili. Übersetzung von imperativen Programmen mit Zeigermanipulation in Graphtransformations-Regeln. Master's thesis, Universität Stuttgart, June 2006. No. 2431.

[Tur04] Sinan Turan. Effiziente Berechnung der Überdeckbarkeit bei Petri-Netzen, June 2004. Studienarbeit (Student research project), No. 1935.

[vM06] Arwed von Merkatz. Analyse von Graphtransformationssystemen mit Hilfe von Petrinetzen und Logiken. Master's thesis, Universität Stuttgart, July 2006. No. 2442.