AUGUR—A tool for the analysis of graph transformation systems using approximative unfolding techniques

August 16, 2005

Contents

1 Introduction **2**

2 People **2**

3 Publications **3**

4 System and Software Requirements **3**

5 Installation **3**

6 aunfold—approximation tool **3**

6.1 Usage 4
6.2 Input File Format 5
6.3 Output File Format 8
6.4 Data structures 10
6.5 Algorithms 11

7 Converter tools **12**

7.1 pg2neato 12
7.2 pg2lola 13
7.3 augur-convert 13
7.4 rules2LaTex 14
7.5 pg2pnml 14

8 Verification tools **15**

8.1 Calculation of Coverability 15
8.2 sponge 15

9 Verification Example **18**

10 List of examples **20**

11 Limitations and Known Problems **20**

12 License Agreement **21**

Bibliography **21**

1 Introduction

The aim of this tool is the verification of systems described by graph grammars using approximated unfoldings [BCK01]. 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].

The current tool is only a prototype having several limitations. The following documentation describes this tool, for more information on graph transformation and analysis of graph transformation systems the relevant literature should be consulted.

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 (Universitt Stuttgart, Germany)
- Andrea Corradini (Università di Pisa, Italy)
- Olga Danylevych (Universitt Stuttgart, Germany)
- Tobias Heindel (Universitt Stuttgart, Germany)
- Lars Heinemann (Universitt Stuttgart, Germany)
- Martin Horsch (Universitt Stuttgart, Germany)
- Barbara Knig (Universitt Stuttgart, Germany)
- Bernhard Knig (Boise State University, USA)
- Vitali Kozioura (Universitt Stuttgart, Germany)
- Alberto Lluch Lafuente (Università di Pisa, Italy)
- Anja Monakova (Universitt Stuttgart, Germany)
- Nicolas Relange (Universitt Stuttgart, Germany)

- Sinan Turan (Universitt Stuttgart, Germany)
- Ingo Walther (Technische Universitt Mnchen, Germany)

The current maintainers of the tool and the documentation are Vitali Kozioura and Barbara Knig. The web site for the AUGUR tool can be found at `http://www.fmi.uni-stuttgart.de/szs/tools/augur/`. Please address questions to `augur@honolulu.informatik.uni-stuttgart.de`.

3 Publications

The theory on which this tool is based is described in the following papers: [BCK03, BKK03, CBKK03, BCK02, BK02, BCK01].

4 System and Software Requirements

The programs are written in C++ and use the libxml library. AUGUR 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.

5 Installation

The source code of the system is available as 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 some other directory, please change the file `src/Makefile` (the first three lines) accordingly. The executables can subsequently be found in the directory `/bin` and can be started as shell applications. Make sure that the `/bin` directory is added to the `PATH` environment variable. In the directory `/example` some examples of graph transformation systems in GTXL format can be found.

6 aunfold—approximation tool

The tool can construct the k-depth approximated unfolding (k-covering) for the given graph transformation system. For input and output two XML-based standards are used. These are the GTXL standard for graph transformation systems and the GXL standard for the Petri graph obtained by the construction (for GXL and GTXL see [Win02, Tae01]). The construction of the approximated unfolding is done according to the algorithms proposed in [BCK01, BK02]. In the following we use the notation of these two papers and refer to the definitions and algorithms presented there.

6.1 Usage

The program is called from a shell in the following way:

```
aunfold [-q] [-d] [-t] [-nc] [-nr] [-depth=k | -nfold=k] [-step=n]
        [-nmapping] infile outfile
```

The input file must be in GTXL format as described below. The output is written to the output file in GXL format. If the output file already exists, it is 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 computations. 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.

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

6.2 Input File Format

The input file must follow the XML-based Graph Transformation Exchange Language (GTXL) format. The format definition is only available as a draft version at the time of writing. It can be found at `http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html`.

Here is the brief description of the GTXL format. In the `/doc` directory the commented example of a GTXL file called `producer-consumer.in.comments` can be found.

In GTXL format the name of the root element has to be `GTS`. The children of the root element can be `Initial` or `Rule`. Inside the `Initial` tag is the initial graph description marked with the `Graph` tag. The left-hand side and right-hand side of the rules are marked with `LHS` and `RHS`. The rule graphs are additionally marked with the tag `RuleGraph`.

```
<GTS id="MyGTSId">
  <Initial>
     <Graph id="InitialGraphId">
     . . .
     </Graph>
  </Initial>
  <Rule id="Rule1">
    <LHS>
      <RuleGraph id="LHSRuleGraphId">
        <Graph id="LHSGraphId">
        . . .
        </Graph>
      </RuleGraph>
    </LHS>
    <RHS>
      <RuleGraph id="RHSRuleGraphId">
        <Graph id="RHSGraphId">
       . . .
```

```
        </Graph>
      <RuleGraph>
    </RHS>
    . . .
  </Rule>
</GTS>
```

Elements `GTS`, `Graph`, `Rule` and `RuleGraph` should have unique ids.

A graph description consists of two parts: a list of nodes and a list of edges. Nodes are marked with the tag `node`.

```
<Graph id="GraphId">
  <node id="NodeId">
 . . .
</Graph>
```

In hypergraphs edges are marked with `rel`, where connected nodes are marked with `relend`. Note that we have to indicate whether a graph is a hypergraph or not. In the case of hypergraphs we only use edge mode `undirected`.

```
<Graph id="HypergraphId" edgeids="true" hypergraph="true"
                         edgemode="undirected">
   . . .
   <rel id="HyperedgeId">
     <attr name="label">
       <string> . . . </string>
     </attr>
   </rel>
   <relend target="FirstNode" startorder="0">
   . . .
</Graph>
```

Attribute `label` describes the label of the given edge as string, while the tag `target` in `relend` gives the id of the connected node with the start order.

A directed graph with binary edges can either be described as a hypergraph, or alternatively it can be described in the following way:

```
<Graph id="DirectedGraphId" edgeids="true" hypergraph="false"
                            edgemode="directed">
   . . .
   <edge id="EdgeId" from="StartNodeId" to="TargetNodeId">
     <attr name="label">
```

```
      <string> . . . </string>
    </attr>
  </edge>
  . . .
</Graph>
```

The source and target nodes of the edge are indicated in this case using the attributes `from` and `to`.

There is also some special element in the rule called `Mapping`, which describes a mapping from nodes of the left-hand side to nodes of the right-hand side of the rule. It has the following syntax:

```
<Graph id="GraphId">
. . .
 <Mapping id="MappingId">
    <MapElem from="NodeLHSId" to="NodeRHSId"/>
     . . .
 </Mapping>
</Graph>
```

It maps nodes using their ids as attributes, thus describing how nodes of the left-hand side relate to nodes of the right-hand side. In our setting this mapping replaces the interface graph with morphisms.

Inside each `Graph` part all ids have to be unique.

There are further restrictions on the input file:

- There must be exactly one `Initial` instance in the file, since the algorithm can not be applied to transformation systems without or with multiple initial graphs.
- If a graph is a hypergraph (i.e., makes use of `rel` edges), it must not be declared as directed. The program does not check if a graph not declared as hypergraph includes hyperedges; such an input leads to undefined behavior.
- If a graph is not a hypergraph and is declared as undirected, all edges will internally be substituted by one directed edge in each direction.
- The `relend` children of a `rel` node must appear in the file ordered by their `startorder` attribute.
- All information except the following are ignored: `GTS`, `Initial`, `Graph`, `Rule`, `RuleGraph`, `node`, `edge`, `rel`, `relend`, `Mapping`, `MapElem`, `LHS`, `RHS`, `attr`.

- `attr` nodes are only interpreted for edges. Each edge should have a string type attribute named `label` to specify the edge label.

6.3 Output File Format

The output file is in the Graph Exchange Language (GXL) format. It contains the approximated unfolding of the input. The output is actually a Petri graph, a structure which is not directly supported by GXL. An equivalent hypergraph is used to represent the Petri graph, as follows:

Nodes of the hypergraph are represented by the tag `node`, with an attached valueless attribute named `vertex`.

Transitions are also represented by `node`s, with an attached valueless attribute named `transition`, and another string-valued attribute named `rule`. The latter contains the name identifier of the rule associated with the transition.

Edges are represented by hyperedges, with an attribute containing the edge label. The edge is connected to all its vertices by a `relend` node with role `vertex`, and to transitions by a `relend` with role `preset` if the edge is in the transition's preset, or one with role `postset` if it is in the transition's postset. Relends to transitions have an integer-valued `weight` attribute, and those to vertices have the XML attribute `startorder`.

Here is the brief description of the output GXL format. A commented example of an GXL file called `producer-consumer.out.comments` can be found in the directory `/doc`.

The root element has to be `gxl`. The child of the root element has to be `graph` element. This element may have children elements named `node` or `rel`. Each of them should have a unique id.

```
<graph id="HypergraphId">
   <node id="NodeId">
    . . .
   </node>
   . . .
   <rel id="HyperedgeId">
    . . .
   </rel>
   . . .
</graph>
```

The `node` elements have either `vertex` or `transition` attributes. A `node` element of type `transition` contains a string attribute `rule`. This indicates the rule name and is used for labeling the transitions.

```
<node id="NodeId">
 <attr name="vertex">
 . . .
</node>
<node id="TransitionId">
 <attr name="transition" />
 <attr name="rule">
   <string> . . . <string/>
 </attr>
  . . .
</node>
```

The `rel` element has a string attribute `label` and an integer attribute `initial_marking`, indicating whether the represented hyperedge is initally marked in the Petri graph.

```
<rel id="HyperedgeId">
 <attr name="label">
  <string> . . . </string>
 </attr>
 <attr name="initial_marking">
  <int> . . . </int>
 </attr>
 . . .
</rel>
```

The `rel` element may contain several `relend` entries which have to contain attributes `target` and `role`. The value of a role can be one of `vertex`, `preset` and `postset`. For `relend` entries with the value `vertex`, a value for the attribute `startorder` is additionally given. The entries with the value `preset` or `postset` contain an attribute `weight`.

```
<relend target="NodeId" role="vertex" startorder=""/>
<relend target="TransitionId" role="postset">
 <attr name="weight">
  <int> . . . </int>
 </attr>
</relend>
```

If the option `-nmapping` is not activated in `aunfold`, then each transition is supplied with information concerning the mapping of the left-hand side and right-hand side of the corresponding rule in to the Petri graph. This mapping is called μ in the relevant papers, the corresponding GXL attribute is `mu-mapping`. It has the form "$a1, b1; a2, b2; \ldots$", corresponding to $a1 \mapsto b1$, $a2 \mapsto b2$, ...

Furthermore the image of the initial graph is given by the morphism ι (`iota-mapping` in GXL). It is a string constant that gives the pre-image of a node or a hyperedge in the start graph.

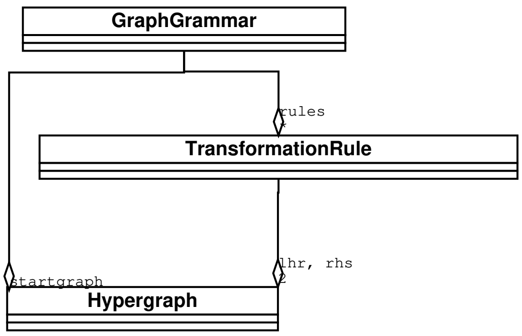
Note that both these mappings are not used in the current version of the tool, but they will be needed in future releases.

6.4 Data structures

Any edge, vertex, transition or rule has a unique integer identifier.

The C++ standard template library (STL) is used to store mappings and sets of identifiers, so the representation of most data structures is similar to the definition in the article.

Hypergraphs consist of a set of vertex ids, a set of edge ids, a connection function and a labeling function. The connection function connectedV maps each edge id to an ordered list of vertex ids; for efficiency reasons the reverse mapping is stored in a separate data structure. STL maps, lists and multisets are used for this purpose.



A **graph grammar** is an initial graph (hypergraph) and a set of transformation rules. A transformation rule consists of two hypergraphs, a left-hand side and a right-hand side. IDs of an edge e_l of the lefthand side and an edge e_r of the righthand side are the same iff $\alpha(e_l) = e_r$, where α maps nodes of the left-hand side to nodes of the right-hand side.

A **Petri net** (`PTNet`) has a set of places (represented by an integer ID), a set of transitions (consisting of a preset, postset and weights assigned to each member of the pre- or postset), and an initial marking. Markings (`PTNetMarking`) are represented as maps assigning a token number to each place.

A **Petri graph** consists of a hypergraph and a Petri net. The mappings μ and ι introduced in [BCK01] are also stored here. Edges in the hypergraph and places in the Petri net have the same IDs if they are identical.

A **Relation** on a Petri graph (class `Relation`) is a pointer to the Petri graph, and maps from a transition, vertex or edge to all transitions, vertices or edges it is related to.

6.5 Algorithms

Subgraph matching is implemented as a method of the `Hypergraph` class. A `MatchHandler` object can be passed to the method. The match handler decides whether a found subgraph matching is acceptable or if the search should continue. Two subclasses of `MatchHandler` are used in the program, to handle the two calls to subgraph matching while calculating the approximated unfolding.

Backtracking is used to find the matchings. The private method `findNextMatching` tries to map one not yet mapped edge to an edge in the larger graph, then calls itself to map the remaining unmapped edges. Every time a complete matching is found, the match handler specified by the caller is called. The backtracking method is fairly inefficient for finding matches and slows down the unfolding procedure. We plan to implement more efficient ways to find matches in future releases.

The class `PTNet` offers a method to calculate the **coverability graph** of a Petri net. Each node of the coverability graph is a marking of the net. The edges of the graph are never used, so they are not returned. The algorithm from [Rei85] is used. The coverability graph is stored as a list of pairs of markings (=directed edges) , together with a transition ID. A depth-first search algorithm is used to determine if there exists a path between two nodes of the graph, and to determine if two elements of the Petri net are in a causality relation. Note that the computation of a coverability graph can take a very long time. The unfolding procedure can be greatly accelerated by omitting the computation of the coverability graph. In this case one obtains still an over-approximation of the original graph transformation system, which, however, might be less precise. We plan to implement incremental updates of coverability graphs in the future.

The class `Relation` class has a method to extend the relation to the least closed equivalence relation. It is assumed that the relation is already consistent. The following algorithm is used:

(1) extend the relation to a reflexive relation

(2) apply Definition 7, steps (3)–(5) from [BCK01] by searching for a lefthand side of an implication and asserting the righthand side to the relation

(3) extend the relation to an equivalence relation

repeat steps (2)–(3) until the relation does not change anymore (then it is a closed equivalence relation).

The result is converted to a more convenient format. Each equivalence class gets a new ID and a mapping from the old IDs on the equivalence class IDs is calculated.

Folding and unfolding methods are offered by the Petri graph class. Folding takes a rule and two matches of the graph (where the morphisms φ, φ' from [BCK01] are called `phi`, `phi2` in the code) and uses the merging operation. Unfolding takes only one match; before merging the Petri graph, one transition is added as required by Definition 10 from [BCK01].

The algorithm to calculate the **approximated unfolding** is implemented in the `aunfold()` function (file `aunfold.cpp`). To improve efficiency, the order of tests in Definition 11 of [BCK01] has been changed:

```
for each transition
  calculate coverability graph
  search for a transition t and matches phi, phi2 as in (6)
  if found
    fold
  else
    if there is no r-labeled transition as in Def. 11,
        unfolding, line 3
      unfold
```

Note that most tests are implemented in the useThisMatch() method of the match handlers.

7 Converter tools

7.1 pg2neato

Program `pg2neato` extracts the Petri net and hypergraph out of the Petri graph and transforms it to the `neato`/`dot` input format. The output is written to stdout.

Usage:

```
pg2neato (-pn | -hg) filename
```

Available options are:

`-pn`: Petri net output

`-hg`: hypergraph output

`filename`: the file containing the Petri graph in GXL format

It is recommended to redirect output to some file and to use it as follows:

```
neato -Tps -Goverlap=false neatoInput.hypergraph > out.hg.ps
dot -Tps -Goverlap=false neatoInput.petriNet > out.pn.ps
```

See the Graphviz documentation at `http://www.research.att.com/sw/tools/graphviz/` for additional details on `dot` and `neato`.

7.2 pg2lola

The program `pg2lola` extracts the Petri net out of the Petri graph and transforms it to the LoLA [Sch00] format. The output is written to stdout.

Usage:

```
pg2lola filename
```

`filename`: the file containing the Petri graph in GXL format

The Petri net can be used as an input for the LoLA verification tool [Sch00]. The program `bwra` (backward reachability algorithm), described in Section 8 also usesthe LoLA format as input.

7.3 augur-convert

For the convenience of the user there is shell script called `augur-convert` in the directory `/bin`.

Usage:

`augur-convert filename`

`filename`: the file containing the Petri graph in GXL format

The script calls `pg2neato`, `neato`, `dot` and `pg2lola` (the paths to all of which are assumed to be in the environment variable `PATH`) and generates the following files:

- `filename.hg.ps`: A PostScript file depicting the hypergraph underlying the Petri graph.
- `filename.pn.ps`: A PostScript file depicting the Petri net component of the Petri graph.
- `filename.lola`: A file in LoLA format containing the Petri net component of the Petri graph.

7.4 rules2LaTex

The program takes the graph transformation system in GTXL format and produces a LaTeX file with a graphical representation of the initial graph and the rules.

Usage:

`rules2LaTex [-size=n] inputfile outputfile`

`-size`: the magnifying factor for left-hand and right-hand sides. Default value is 0.4

`inputfile`: the file containing graph transformation system in the GTXL format

`outputfile`: LaTeXsource file containing the graphical representation of the graph grammar.

A new directory `/<filename>_files` is created where the input files for `neato`/`dot` files and output PostScript files are saved.

Important: The program uses `neato` for converting the graphs into PostScript and it is expected that the path to `neato/dot` is contained in the `PATH` environment variable.

7.5 pg2pnml

The program extracts the Petri net out of the Petri graph and transforms it to the format of the Petri net markup language PNML ([WK03]).

```
pg2pnml infile outfile
```

`infile`: the file containing Petri graph in the GXL format

`outfile`: output in PNML format

8 Verification tools

8.1 Calculation of Coverability

These tools solve the coverability problem. This means checking if the given marking is covered by some reachable marking in the given Petri net [AJKP98].

There are four variants of the calculation implemented.

Program	Description
`bwra`	The program uses the backward reachability algorithm from [AJKP98] in order to calculate the coverability of the given marking.
`cgbatch`	The program calculates the coverability graph and then checks the coverability of the marking [Rei85].

All these programs have the same usage.

Usage:

```
program [-v | -q] file1 file2
```

Option `-v`: verbose messages

Option `-q`: quiet mode

`file1`: initial marking and Petri net in LoLA format

`file2`: test marking in LoLA format

The output then says whether one of the markings of the list can be covered in the Petri net with initial marking.

8.2 sponge

Given an hypergraph and a regular expression, `sponge` finds 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 (see also Section 8).

Usage:

```
sponge [ -q | -d ] [ -m ] [ -f ] hypergraph_file.xml
sponge --help
sponge --version
```

Option `-q` quiet mode, disable verbose output

Option `-d` debug mode, show debugging information

Option `-m` use mode 1 for hypergraphs which means that we assume that there is a binary edge from the i-th node to the j-th node of every hyperedge where $j > i$

The default is mode 0, where there is a binary edge from the 0-th node to the j-the node where $j > 0$.

This technique is used to convert hypergraphs into directed graphs. The binary edges have the same labels than the original hyperedges. (See also the explanation below.)

Option `-f` switch to formula output (default: markings output)

Option `--help` Output help information and exit.

Option `--version` Output version information and exit.

The regular expression is read from stdin. 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.

The different modes (see the `-m` option) allow to specify how to read the labels on hyperedges. If numbers are specified around the label, the first number stands for the incoming vertex, the second number for the outgoing vertex. For example, `0a2` means that one only takes into consideration the paths that go from vertex 0 to vertex 2 in the hyperedges that are labelled `a`. If no numbers are specified, the meaning of the hyperedge depends on the number of vertices the edge is connected to:

0 vertices: the hyperedge behaves like an isolated binary edge, that means that `a` and `0a0` return the same result.

1 vertex: the hyperedge is like a binary edge where source and target are the same vertex. Thus `a` is the same as `0a0`.

2 vertices: this is a binary edge and `a` means `0a1`.

$>$ 2 vertices: the behaviour of the hyperedge depends on the chosen mode.

In `mode 0`, the default mode, the hyperedge connects the first vertex 0 to all other vertices, whose number i is strictly positive ($i > 0$).

In `mode 1`, which is switched on with the option `-m`, the hyperedge connects the vertex i with all vertices $j > i$.

The input file `hypergraph_file.xml` must be in GXL (Graph eXchange Language). The output of the `aunfold` program is already in the correct format (a mix of hypergraph and Petri net). The result of the program, a condition on markings, is then written to `stdout`. Two different output formats are possible:

Default output: A list of markings with the property that a graph generated by a marking m contains a path corresponding to the regular expression if and only if m covers a marking on this list. This is the output that is appropriate for the tool `bwra`.

Formula output: This output can be switched on with the option `-f`. The output is then in the formula format as described in the documentation of LoLA.

To run `sponge` type:

```
sponge < regular_expression_file > output_file hypergraph_file.xml
```

Alternatively it is also possible to enter the regular expression in an interactive way. The end of the input is indicated by typing `<Enter> + Ctrl-D`.

```
sponge -q hypergraph_file.xml
```

The output is either a list of markings as described above or a formula describing these markings.

9 Verification Example

As a verification example let us consider the a system consisting of public and private servers. The system can produce an unlimited number of public servers and one private server connected to each other. The servers can produce mobile processes (internal processes by the private and external by the public servers). New connections can be created between the servers where no connection is allowed from private to public servers and processes can move over these connections. At some point in time the private server may change into a public server. The description in GTXL format can be found in the `/examples` directory in the file `pub_priv_server.xml`. You an visualize the rules by using `rules2LaTeX` as described above.

The property we want to verify here is: External process can not reach a private server.

First construct the approximated unfolding:

```
> aunfold pub_priv_server.xml out.xml
```

and convert it to LoLA format:

```
> pg2lola out.xml > out_lola
```

The computed Petri graph (Petri net and graph component) can be visualized using pg2neato as described above. Now we can call sponge in the following way in order to obtain the marking, corresponding to the property to be verified:

```
> sponge out.xml > marking
info: Hypergraph: vertices: 886   edges: 887 <Sprv>(886 ) 888 <Spub>(886 )
889 <Pext>(886 )890 <C>(886 886 ) 891 <Sprvc>(886 ) 892 <Dummy>()
893 <Pint>(886 ) 894 <Spubc>(886 ) 895 <Gpub>()
> . 'Sprv''Pext'
> . Ctrl-D
```

At the end `Ctrl-D` must be pressed in order to exit from the complete the input. Alternatively the regular expression can be read from stdin. Regular expression can also be more complex, for example `'Spub'C*'Sprv'` representing the property that no connection will be created from a public to private server.

We continue with our example In this case the file `marking` describing markings that should *not* be coverable looks as follows:

```
MARKING Sprv_735: 1, Pext_737: 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 `bwra`, which checks whether a marking can be covered, in the following way:

```
> bwra out_lola marking
Visited Constraints:
Constraint with ID 1    : (Sprv_735,1) (Pext_737,1)
Constraint with ID 2    : (Sprv_735,1) (Spub_736,1)
Constraint with ID 3    : (Sprv_735,1) (Gpub_742,1)
The Final Marking(s) are coverable.
Time elapsed -0.00 sec
```

We can see that the marking is coverable in the Petri net, but this does not mean that the verified property is false, since we work with an over-approximation. Let us change the depth in the unfolding construction in order to obtain a more precise approximation.

```
> aunfold -nc -depth=1 pub_priv_server.xml out.xml
```

If we repeat all earlier verification steps, then we will see, that the undesirable situation is not possible in the Petri net obtained from the 1-depth over-approximation.

Call `pg2lola`:

```
> pg2lola out.xml > out_lola
```

`sponge`:

```
> sponge out.xml > marking
```

In this case we have two markings:

```
MARKING Sprv_8849: 1, Pext_8850: 1;
MARKING Pext_8850: 1, Sprv_8855: 1;
```

and `bwra` gives the answer "The Final Marking(s) are not coverable." This means the property is verified.

10 List of examples

The directory `examples/` contains the following GTXL files:

- `dphil_finite.xml/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].
- `firewall.xml`: A network of secure and insecure locations, divided by a firewall. This example is described in more detail in [BCK02].
- `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.
- `producer-consumer-directed.xml/producer-consumer.xml`: A simple toy example describing the production and consumption of messages.
- `pub_priv_server.xml`: The example used and described in Section 9.
- `resources.xml`: A system of several processes sharing two resources. This example is described in more detail in [BKK03].

11 Limitations and Known Problems

`aunfold`:

While reading the input file, the program relies on the order of the `relend` nodes in the file, which does not conform with the XML specification. Errors in the input file are not always detected, and it is not checked if the prerequisites for the algorithm are met.

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.

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. This can speed up the program essentially.

Finally, keep in mind that this is only a prototype implementation, where some functionality is implemented in a rather inefficient way. This concerns especially the search for matches of left-hand sides and the computation of coverability graphs.

`sponge:`

`sponge` requires that all edge labels consist of letter only, labels containing numerical characters are disallowed.

12 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.

[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.

[BCK03] Paolo Baldan, Andrea Corradini, and Barbara König. Unfolding-based verification for graph transformation systems. In *Proc. of UniGra '03: Uniform Approaches to Graphical Specification Techniques (Warsaw)*, 2003.

[BK02] 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.

[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.

[CBKK03] Andrea Corradini, Paolo Baldan, Barbara König, and Bernhard König. Verifying a behavioural logic for graph transformation systems. In *Proc. of COMETA '03*, ENTCS, 2003. to appear.

[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 Transfor-*

mation, Vol.3: Concurrency, Parallellism, and Distribution. World Scientific, 1999.

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

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

[Sch00] Karsten Schmidt. LoLA: A low level analyser. In *Proc. of ATPN (Application and Theory of Petri Nets)*, pages 465–474. Springer, 2000. LNCS 1825.

[Tae01] Gabriele Taentzer. Towards common exchange formats for graphs and graph transformation systems. In *Proc. of UniGra '01 (Uniform Approaches to Graphical Process Specification Techniques)*, volume 44 of *ENTCS*, 2001.

[Win02] A. Winter. GXL—overview and current status. In *Proc. of GraBaTs '02 (Workshop on Graph-Based Tools)*, volume 72 of *ENTCS*, 2002.

[WK03] M. Weber and E. Kindler. The Petri net markup language. In *Petri Net Technology for Communication Based Systems*, pages 124–144. Springer, 2003. LNCS 2472.