Abstref — An extending module for Augur

May 20, 2005

1 Introduction

This module implements counterexample-based abstraction refinement to the over-approximation constructed with Augur. The refined over-approximation obtained in this way can then be analyzed using the tools contained in Augur (for instance with `sponge` and `bwra`). Although Augur has its own techniques for refining the over-approximation (k-depth over-approximation, also called k-covering), Abstref is in many cases much faster and constructs smaller approximations. Furthermore, there exist examples where counterexample-guided abstraction refinement is the only possibility to verify a given graph transformation system.

Abstref can also work iteratively whenever new counterexamples are found in the refined over-approximation.

The current tool is a prototype version having several limitations.

2 System and Software Requirements

The programs are written in C++ and use the XML library libxml. Abstref has been tested under Linux using libxml 2.0. Furthermore the libxml development package (libxml-dev/libxml-devel) is needed.

3 Installation

Before installing Abstref you must first install Augur which can be downloaded from `http://www.fmi.uni-stuttgart.de/szs/tools/augur`. The source code of Abstref is available as a `tar.gz` archive from the same web page. After unpacking you should copy directory `/abstref` into directory `/augur` of Augur. Compile the programs by calling `make` in directory `/abstref`.

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 `abstref/Makefile` (the first three lines) accordingly.

The executable `abstref` can subsequently be found in directory `/bin` and can be started as a shell application.

3.1 Usage

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

```
abstref [-q] [-d] [-t] [-nc] [-nr] [-step=n] [-timeout=n]
   approxFile grammarFile transitionsFile outputFile sampleFile
```

The input and output files are:

`approxFile` - file in GXL format with the approximation to be refined.

`grammarFile` - file in GTXL format with the graph grammar.

`transitionsFile` - file with the counterexample generated by `bwra`. `bwra` usually generates a file with the name `transpath`.

`outputFile` - output file in GXL format with the refined approximation.

`sampleFile` - file in XML format for storing the samples for the checked counterexamples. Samples are internal data structures. Each sample is used to avoid an appearing of the checked counterexample in the future.

If the `outputFile` already exists, it is overwritten without warning. The `sampleFile` should be deleted manually at the beginning of a new session (verification of a new GTS). Otherwise all samples stored in the `sampleFile` will be used further.

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.

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

`-timeout=n` timeout in seconds.

The program is not interactive and terminates when the calculation is finished or by the timeout condition.

4 Counterexample-guided Abstraction Refinement

ABSTREF uses the well-known technique of counterexample-guided abstraction refinement and proceeds in the following way: we compute an initial approximation and then check whether the property can be verified. If not, we obtain a run in the approximating Petri net that violates this property. Either this run corresponds to a real run—in this case we have found an error—or it is spurious. In the second case we will refine the approximation as much as is needed, such that the spurious counterexample disappears. This is done by doing another round of approximated unfolding, now avoiding to merge nodes that cause the spurious counterexample.

5 Verification Example

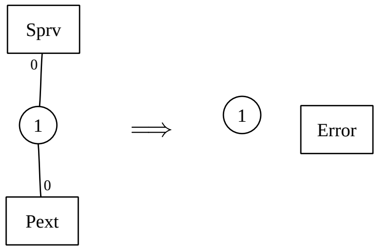
We consider the same verification example as in the documentation of AUGUR. The example is called "Public Private Servers" and its description in GTXL format can be found in directory `/examples` in the file `pub_priv_server.xml`.

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

But first we modify the example. If we plan to check that no reachable graph contains edges labelled `Sprv` and `Pext` (private server and external process) that are connected, we insert an additional rule to the system whose left-hand side L corresponds to this structure. 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 $t_1, \dots, t_n$, corresponding to rules $r_1, \dots, r_n$, 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 labelled `Sprv` and `Pext`. 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.

Below is the graphical representation and GTXL code of this new rule.




```
<Rule id="Error">
  <LHS>
    <RuleGraph id="er_lrg">
      <Graph id="er_lg" edgeids="true" hypergraph="true"
                                  edgemode="undirected">
        <node id="n1" />
        <rel id="psps_lh_Sprv" >
          <attr name="label">
            <string>Sprv</string>
          </attr>
          <relend id="psps_lh_sprv" target="n1" startorder="0" />
        </rel>
        <rel id="psps_lh_Pext" >
          <attr name="label">
            <string>Pext</string>
          </attr>
          <relend id="psps_lh_pext" target="n1" startorder="0" />
        </rel>
      </Graph>
    </RuleGraph>
  </LHS>
  <RHS>
    <RuleGraph id="er_rrg">
      <Graph id="er_rg" edgeids="true" hypergraph="true"
                                  edgemode="undirected">
        <rel id="er" >
          <attr name="label">
            <string>Error</string>
          </attr>
        </rel>
      </Graph>
    </RuleGraph>
  </RHS>
  <Mapping id="psps_mapping">
  </Mapping>
</Rule>
```

The corresponding file `pub_priv_server_2.xml` can be found in directory `abstref_examples` of ABSTREF.

To verify this example we should first compute and verify the initial approximation with AUGUR (see the AUGUR documentation available at `http://www.fmi.uni-stuttgart.de/szs/tools/augur` for more details). We create a file `error` containing the edge "Error" as a regular expression: `0'Error'0`.

Here is the log of the verification session:

```
> aunfold -nc -q pub_priv_server_2.xml out.xml

time used: 0.14 sek
edges: 9, vertices: 1, transitions: 13

> pg2lola out.xml > lola

> sponge -q out.xml < error > marking

> bwra lola marking

Visited Constraints:
Constraint with ID 1    : (Error_1020,1)
Constraint with ID 2    : (Sprv_1018,1) (Pext_1019,1)
Constraint with ID 3    : (Sprv_1018,1) (Spub_1021,1)
Constraint with ID 4    : (Sprv_1018,1) (Gpub_1026,1)
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. Now let us construct a more precise approximation using ABSTREF. The obtained counterexample is constructed by `bwra` (which is contained in AUGUR) as a side effect. The file is called `transpath` and can be found in directory `bin`.

```
file transpath
----------------
CreatePublicServer
1040
CreateExternalProcess
1048
Error
1052
----------------
```

Now we are ready to construct the refined over-approximation and to verify it. First we should delete the sample file, which could be left from the previous session

```
> rm sample.xml

> abstref -nc -q out.xml pub_priv_server_2.xml transpath
  out_2.xml sample.xml

The example is spurious
Transition number 3, id: 1128, name: Error,
could not be fired
time used: 2.14 sek
edges: 16, vertices: 2, transitions: 25

> pg2lola out_2.xml > lola

> sponge -q out_2.xml < error > marking

> bwra lola marking

Visited Constraints:
Constraint with ID 1    : (Error_12659,1)
Constraint with ID 2    : (Sprv_12657,1) (Pext_12658,1)
Constraint with ID 3    : (Sprv_12657,1) (Spub_12666,1)
Constraint with ID 4    : (Sprv_12657,1) (Pext_12661,1) (C_12662,1)
Constraint with ID 5    : (Sprv_12657,2)
Constraint with ID 6    : (Sprv_12657,1) (Spub_12668,1) (Spubc_12669,1)
Constraint with ID 7    : (Sprv_12657,1) (C_12662,1) (Spub_12668,1)
Constraint with ID 8    : (Sprv_12657,1) (Sprvc_12671,1)
Constraint with ID 9    : (Sprv_12657,1) (Spubc_12669,1) (Gpub_12672,1)
Constraint with ID 10   : (Sprv_12657,1) (C_12662,1) (Gpub_12672,1)
The Final Marking(s) are not coverable.
Time elapsed 0.02 sec.
```

The fact that the edge “Error” is not coverable means that we have successfully verified the example.

6 Limitations and Known Problems

`aunfold:`

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.

7 License Agreement