ASTRA: A tool for abstract interpretation of graph transformation systems

Peter Backes and Jan Reineke

Universität des Saarlandes, Saarbrücken, Germany
{rtc,reineke}@cs.uni-saarland.de

Abstract. We describe ASTRA (see http://rw4.cs.uni-saarland.de/~rtc/astra/), a tool for the static analysis of infinite-state graph transformation systems. It is based on abstract interpretation and implements cluster abstraction, i.e., it computes a finite overapproximation of the set of reachable graphs by decomposing them into small, overlapping clusters of nodes. While related tools lack support for negative application conditions, accept only a limited class of graph transformation systems, or suffer from state-space explosion on models with (even moderate) concurrency, ASTRA can cope with scenarios that combine these three challenges. Applications include parameterized verification and shape analysis of heap structures.

Keywords: abstract interpretation, graph transformation systems, parameterized verification, shape analysis, tools

1 Introduction

Graph transformation is an intuitive formalism: One begins with a start graph and, by nondeterministic choice, matches and applies transformation rules to it, based on subgraph replacement. We are mainly interested in analysis of the graphs reachable by successive application of rules, to verify safety properties, for example.

One of the applications of graph transformation is modelling parameterized concurrent systems. Reasoning about such systems is hard because the state space is infinite. Hence, abstraction methods are required. In this paper, we present ASTRA, our tool for abstraction of graph transformation systems.

A number of tools are available that use abstract interpretation (each based on a different abstraction) to compute a finite over-approximation of the reachable graphs: AUGUR [7] uses a petri net based abstraction and had success with interesting examples of concurrent systems; it does not, however, support negative application conditions. `hiralysis` [5] is based on partner abstraction. It does offer negative application conditions and can analyze some concurrent systems, but requires input grammars to satisfy some rather restrictive "friendliness" properties. GROOVE [9] has an implementation of neighborhood abstraction, which has no such restriction, supports negative application conditions, but analysis of systems with concurrency leads to state space explosion.

2 Cluster abstraction

Our tool, ASTRA, implements *cluster abstraction* [3]: We consider each node in the graph (to become the *core node* of a cluster) plus its respective adjacent nodes (to become the *periphery*). We merge two or more adjacent nodes into summary nodes if both their labels and configuration (*spoke*) of edges to the core node are equal. If, by this summarization, two merged nodes disagree on to existence of some edge to a third node, we replace it by a $\frac{1}{2}$ edge. After summarization, we are left with clusters of bounded size, and we eliminate any duplicate cluster by assuming (as a further overapproximation) that there can be any number of concrete instances. An example is shown in Figure 1. The initial graph is abstracted in this way, and then rule application is lifted to the abstraction.

In this paper we describe ASTRA 2.0. An earlier version, ASTRA 1.0 [2], implemented a less precise precursor to cluster abstraction that assumed all edges in the periphery to be $\frac{1}{2}$.

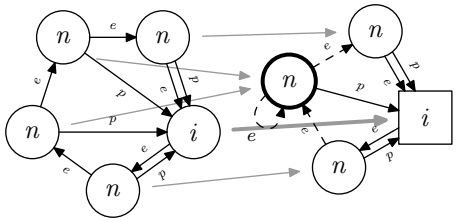

Fig. 1. An example of how a cluster is obtained by abstracting the concrete graph with respect to one specific node (here, the i-labelled one). The tool lifts the application of graph transformation rules to this abstraction.

3 Architecture and Usage

ASTRA is a command-line program that expects a start graph and graph transformation rules as input and outputs the clusters from the analysis. When running the analysis, it abstracts the start graph, then enters its main loop. The main loop searches for abstract matches; each left hand side node of each rule is matched against the core node of any cluster from the current working set, and the remaining nodes are matched to a subset of the respective peripheral nodes. In addition, one further cluster with unmatched core node, but matched peripheral nodes is materialized. Those matches are then combined into a partial concretization, with several checks done to rule out cases where no full concretization exists. Not all such cases are detected by the tool; but the result is still a valid over-approximation.

All clusters produced by rule application are added to a temporary set. After each iteration, the tool then, optionally, applies a post-pass reduction step to the temporary set, inspecting it for clusters that can be eliminated or refined. To do this, the tool searches for all partial materializations bounded to three material nodes: If a cluster cannot be used in any of them, it is eliminated, and if an edge is always present or always absent, peripheral $\frac{1}{2}$ constraints are refined. Finally, the temporary set is joined with the working set.

The tool indicates progress as it goes from rule to rule, and from iteration to iteration. After each iteration, the current working set is dumped to disk, which is useful for inspecting the current state of the analysis when running the tool on complex cases that take some time.

The main loop is executed iteratively until the working set remains unchanged, ie., a fixpoint has been reached. The tool then dumps the output to disk, prints statistics and exits.

3.1 Input file format

ASTRA uses the same ASCII-based input file format as `hiralysis` (see [5] Fig. B.1, p. 160), extended by additional application conditions. For example, the constraint `partner(x1)=neg{(out,p)}` restricts rules to apply only if the node matched by `x1` has no outgoing edge with label `p`.

Consider the following toy case as a running example. The input:

```
nodelabels n,Error,i; edgelabels e,p;
empty; // start graph
create [{x1:n,x2:n,x3:i},
   {(x1,x2):e,(x2,x3):e,(x3,x1):e,(x1,x3):p,(x2,x3):p}];// init
rule [{x1:i,x2:n},{(x1,x2):e}],                        // insert
     [{x1:i,x2:n,x3:n},{(x1,x3):e,(x3,x2):e,(x3,x1):p}];
rule [{x1:n},{},partner(x1)=neg{(out,p)}], [{x1:n,x2:Error},{}];
```

This example models singly-linked ring buffers into which an unbounded number of nodes are inserted dynamically. One special node is indicated with the label `i`. New nodes are inserted next to it with a back pointer. Here, we want to use astra to verify the safety property that each node has such a back pointer. We achieve this with the second rule. It uses a negative application condition to generate an error label if a node lacks the back pointer.

As can be seen, the input file format is mainly based on graphs, which are sets of node names, each with a label, and sets of edges (the name being a pair of node names), each with an edge label. The rules specify the subgraph to be replaced and the subgraph by which it is replaced. The node names imply a mapping from the left hand side to the right hand side.

3.2 Command-line interface

For our case study, consider the following tool run:

```
$ ./astra -Os -Op test023.gts
 0 [  2/  2] = 100% [+2, +-2]
 1 [  2/  2] = 100% [+1, +-1]
 2 [  2/  2] = 100% [+0, +-0]
done.
6 clusters, 5 matches, 1 active rules,
6 rule applications, 2 iterations
```

The `$` indicates the shell prompt; the remaining line is entered by the tool user. In this case, ASTRA is run on the input file of our running example (`test023.gts`).

In the example, we specify analysis options `-Os` and `-Op`, instructing ASTRA to apply a simple peripheral constraint satisfiability check and post-pass reduction, respectively. For our experiments, this proved to be the most practical option set, providing the best speed/precision trade-off. Removing one of the two options lead to drastic decrease in precision, while adding any other lead to merely minuscule gains. Only in specific cases where the analysis would otherwise run into state-space explosion, further analysis options were useful.

Option `-n` can be used to specify a cutoff iteration after which to prematurely terminate the analysis. This is useful to inspect the intermediate result. Run ASTRA without arguments for further details about the available options.

3.3 Status report

For each iteration, while running, the current iteration number, current rule, total number of rules and progress (current rule divided by total number of rules) is printed. After finishing the iteration, the number of clusters added and modified (i.e., with peripheral constraints weakened) is printed. Note that clusters added by the initial graph and by rules with empty left hand side are only accounted for in the final statistics printed after the fixpoint has been reached.

3.4 Output file formats

ASTRA supports DOT (as used by the graph layout tool Graphviz), GML (as used by OGDF and the GoVisual Diagram Editor, respectively), GDL (as used by VCG and its successor aiSee) and GraphML (as used by yEd and yComp, respectively). In addition, the tool supports its own native output format that is similar to the input format.

The output can be loaded or processed with any tool supporting any of those formats. The most common use will be a graph layout tool to inspect the output, but it can as well provide invariants for other analyses, like `hiralysis` [4].

For our running example, the tool outputs six clusters, visualized in Figure 2. In addition to the full analysis, we show the intermediate results obtained by using option `-n`.

These drawings were done by METAPOST, based on an experimental output module built into ASTRA that does primitive circular graph drawing. For common use, aiSee and yEd have proven most useful, especially the organic and hierarchical layout engines.

4 Experimental Evaluation

We already ran the tool on various test cases from the literature in [3], including AVL trees, red-black trees, firewalls, public/private servers, dining philosophers, resources, mutual exclusion, singly-linked lists, circular buffers, Euler walks, and the merge protocol. The merge protocol, our main example, is a distributed car platooning coordination protocol that establishes a logical communication hierarchy on top of the physical communication medium. Analysis of the protocol is hard because of its massively distributed nature, caused by the vast range of topological configurations that may evolve concurrently.

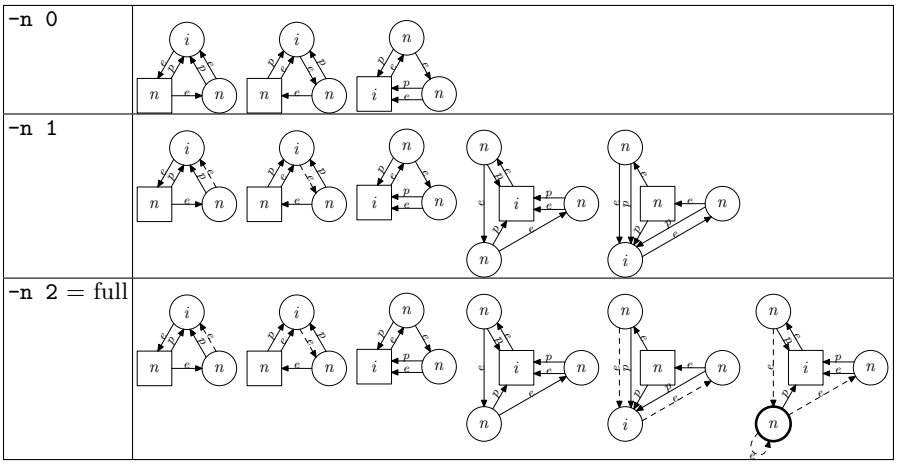

Fig. 2. Analysis results on running example.

However, all inputs from that case study were written by hand. To demonstrate the robustness of our tool, we apply it to graph transformation systems generated automatically from higher level models of the merge protocol, specified in the DCS formalism [8,6]. We used the tool `dcs2gts` [1] to translate the DCS models into graph transformation systems suitable for analysis with ASTRA. We include two new variants, follower-controlled merge.

Synchronous (leader-controlled) merge in our former case study consisted of 402 rules (plus 3 for checking safety properties), the asynchronous version 313 (plus 2). The large number is caused by the fact that many rules are generated from templates that iterate over all node labels. The automatically generated versions use 788 and 835 rules, respectively. In contrast, the number of clusters in the analysis result increased from 873 to 22509 (factor 26) and from 3069 to 142326 (factor 46). This is because the automatically generated version uses intermediate steps to model topology changes. While those steps are serialized by special labels, and thus pose no combinatorial challenge, our analysis shows that the tool does well with all those intermediate configurations absent in the manually created inputs. See Table 1 for the full results.

Table 1. Benchmark analysis statistics. cl. = clusters, m. = abstract matches, rule app. = rule applications, it. = iterations.

Benchmark	# cl.	# m.	# rule app.	# it.	time
Synchronous, leader-controlled	22509	75359	36685213	135	9m 34.200s
Synchronous, follower-controlled	24957	82569	43679468	144	22m 30.200s
Asynchronous, leader-controlled	142326	850889	1006759383	202	13136m 1.260s
Asynchronous, follower-controlled	58023	296310	83499253	157	3972m 37.560s

5 Conclusions and Future Work

We have seen how ASTRA can be used to analyze a simple graph transformation system, modelling insertion of elements into ring buffers. In contrast to related tools, it is not restricted to graph transformation systems of a special form, it supports negative application conditions and it does well when facing models involving concurrency. Our experimental evaluation showed that it is capable of handling very complex inputs generated automatically from higher-level specifications.

Future work: Our tool already has experimental support for generating an abstract labelled transition system of clusters, but the theory for actually using those with a model checker has still to be worked out. We would also like to provide more powerful application conditions, in particular non-existence of edges between two specific nodes and restrictions on the periphery of a node.

A promising way to considerably speed up analysis is parallelization. The structure of the analysis is very well suited for this and we expect a parallelized version to scale almost linearly.

Acknowledgments. We thank Dmytro Puzhay for assistance with the implementation work and Jörg Bauer-Kreiker for providing his hiralysis test cases. Conny Clausen managed copyright clearance with Saarland University to obtain permission for releasing the tool under a Free Software license. Reinhard Wilhelm provided valuable comments for a draft version of this paper.

References

1. Backes, P.: dcs2gts – An interface between XML-coded DCS protocols and the hiralysis representation of graph transformation grammars. Fopra report, Saarland University (Jan 2007)
2. Backes, P., Reineke, J.: Abstract topology analysis of the join phase of the merge proto col [using astra]. In: TTC'10. CTIT Workshop Proceedings, vol. WP10-03, pp. 127–133. University of Twente, Enschede (2010)
3. Backes, P., Reineke, J.: Analysis of infinite-state graph transformation systems by cluster abstraction. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI'15. pp. 135–152. No. 8931 in LNCS (2015)
4. Bauer, J., Schaefer, I., Toben, T., Westphal, B.: Specification and verification of dynamic communication systems. In: ACSD'06. pp. 189–200 (2006)
5. Bauer, J.: Analysis of Communication Topologies by Partner Abstraction. Ph.D. thesis, Saarland University (2006)
6. Bauer, J., Toben, T., Westphal, B.: Mind the shapes: Abstraction refinement via topology invariants. Tech. Rep. 22, SFB/TR 14 AVACS (Jun 2007)
7. König, B., Kozioura, V.: Augur 2—a new version of a tool for the analysis of graph transformation systems. In: Bruni, R., Varró, D. (eds.) GT-VMT'06. ENTCS, vol. 2011, pp. 201–210 (2008)
8. Rakow, J.: Verification of Dynamic Communication Systems. Diploma thesis, Carl-von-Ossietzky Universität Oldenburg (Apr 2006)
9. Zambon, E.: Abstract graph transformation : theory and practice. Ph.D. thesis, University of Twente (2013)

Informal Plan for an Oral Presentation of ASTRA

Peter Backes and Jan Reineke

Universität des Saarlandes, Saarbrücken, Germany
{rtc,reineke}@cs.uni-saarland.de

Our oral presentation of ASTRA would consist of two parts:

In the first part, we would give a brief introduction to *cluster abstraction*, the abstraction underlying ASTRA. This would be based on slides, and we anticipate it to take about 5-7 minutes.

The second, larger part of our presentation would be a live tool demonstration, guided by a simple running example: a singly-linked ring buffer. In this demo, we would walk the audience through all the steps necessary to use ASTRA:

- The definition of a set of graph transformation rules in ASTRA's textual input format.
- The use of the command-line interface to invoke ASTRA, touching upon various parameter options.
- The inspection of ASTRA's outputs using graph visualization tools.

The live demo should take around 15 minutes. The complexity of the running example can be adapted to the available time.