Neighbourhood Abstraction in GROOVE — Tool Paper

Arend Rensink and Eduardo Zambon

6 pages
Neighbourhood Abstraction in GROOVE — Tool Paper

Arend Rensink and Eduardo Zambon

rensink@cs.utwente.nl, zambon@cs.utwente.nl

Formal Methods and Tools Group
Department of Computer Science
University of Twente, The Netherlands

Abstract: In this paper we discuss the implementation of neighbourhood graph abstraction in the GROOVE tool set. Important classes of graph grammars may have unbounded state spaces and therefore cannot be verified with traditional model checking techniques. One way to address this problem is to perform graph abstraction, which allows us to generate a finite abstract state space that over-approximates the original one. In previous work we presented the theory of neighbourhood abstraction. In this paper, we present the implementation of this theory in GROOVE and illustrate its applicability with a case study that models a single-linked list.

Keywords: Graph Abstraction, Graph Transformation, Model Checking, GROOVE

1 Introduction

Many verification methods rely on the exploration of the state space of systems. However, even for small systems the state space size tends to blow up exponentially. Moreover, one would like to be able to analyse systems independently of their instantiated size. An approach that can in principle solve both these problems is state abstraction. The idea behind this is that “similar” states are actually grouped together, and such a group of similar states is modelled in such a way the distinction between them is no longer visible. The behaviour of the abstract state is the collection of possible behaviours of the original states.

This principle has been long known and studied, e.g., in abstract interpretation [CC77] and shape analysis [SRW98, SRW02]. In the context of graph transformation we have seen several theoretical studies on suitable abstractions [Ren04, RD06, BBKR08, RN08, BKK03, KK06]. However, to the best of our knowledge only the last of these is backed up by an implementation, namely AUGUR2 [KK08].

In this paper we report an extension of GROOVE that implements the neighbourhood abstraction principle of [BBKR08], showing its feasibility at least on a small example. This gives us a basis for experimenting with different, more expressive notions of abstraction.

2 Neighbourhood Abstraction

Our notion of abstraction is based on neighbourhood similarity: two nodes are considered indistinguishable if they have the same incoming and outgoing edges, and the opposite ends of
those edges are also comparable (in a parameterisable sense). Graphs are abstracted by folding all indistinguishable nodes into one, while keeping count of their original number up to some bound of precision. The incident edges are also combined. Counting up to some bound is done using multiplicities. We use $M_k = \{0, \ldots, k, \omega\}$ with $k \in \mathbb{N}$ consisting of exact numbers up to $k$ (which is typically a low value such as 1 or 2) and the value $\omega$ standing for “many”.

Formally, a graph is a tuple $G = \langle V, E \rangle$ of nodes and edges, where (in our case) the edges are triples $(v, a, w)$ of source node, label, and target node. The abstractions are called shapes: they are 5-tuples $S = \langle G, \sim, \text{mult}_{\text{node}}, \text{mult}_{\text{in}}, \text{mult}_{\text{out}} \rangle$ in which

- $G$ is the underlying graph structure of the shape;
- $\sim \subseteq V \times V$ is a neighbourhood similarity relation;
- $\text{mult}_{\text{node}} : V \rightarrow M_v$ is a node multiplicity function, which records how many concrete nodes were folded into a given abstract node, up to bound $v$;
- $\text{mult}_{\text{in}}, \text{mult}_{\text{out}} : (V \times L \times V / \sim) \rightarrow M_\mu$ are incoming and outgoing edge multiplicity functions, which record how many edges with a certain label the concrete nodes had that were folded into an abstract node, up to a bound $\mu$ and a group of $\sim$-similar opposite nodes.

The following is pseudo-code for generating the abstract state space. $Q$ is the set of all shapes and $F$ the set of fresh, yet to be explored shapes; $P$ is the set of rules and $G$ the start graph.

```plaintext
1 let $S := \text{abstract}(G)$, $Q := \emptyset$, $F := \{S\}$
2 while $F \neq \emptyset$
3   do choose $S \in F$ (which $S$ is selected depends on the exploration strategy)
4     let $F := F \setminus \{S\}$
5     for $p \in P$, $m \in \text{prematch}(p, S)$, $S' \in \text{materialise}(m, S)$
6       do let $R := \text{normalise}(\text{apply}(p, m, S'))$
7           if $R \notin Q$
8             then let $Q := Q \cup \{R\}$, $F := F \cup \{R\}$
9           fi
10   od
11 od
```

The important phases in this algorithm are:

- abstract computes the shape of a graph. This is controlled by a parameter $i$ expressing the radius of the neighbourhood to be considered in the neighbourhood similarity relation.
- prematch computes morphisms of a rule $p$ into a shape $S$. Such a morphism is not yet a match, because the images of $p$’s left hand side may be nodes with multiplicity $> 1$; in that case they have to be materialised.
- materialise creates concrete nodes and edges for the image of $p$ in $S$. This is a non-deterministic step, as there may be choices involved in choosing multiplicities for the instantiated nodes and edges.
- apply is rule application, which can be carried out as usual because the rule now acts upon a concrete subgraph of $S'$.
- normalise merges the transformed graph back into the rest of the shape; it is thus similar to abstract except that it acts upon a (partially materialised) shape rather than a graph.
For instance, abstract converts the following graph into a shape, using $i = 1$ and $\nu = \mu = 1$. The indicated $C$-nodes in the graph are folded because they are indistinguishable by their incoming and outgoing edges.

In the right hand side figure, all $C$-labelled nodes are $\sim$-similar, and all edge multiplicities are 1. The “joined” incoming and outgoing edges indicate that the multiplicities apply to a bundle of edges, rather than a single edge. The fat $C$-node has multiplicity $\omega$.

3 Case study

As a test case for our abstraction implementation we use a graph transformation system that models a single-linked list. The list is formed by cells, representing the elements in the list, which are connected by a next pointer. Additionally, a list has a root object that indicates the first and last elements of the list, by way of pointers called head and tail, respectively. The modelling of such structure as a graph in GROOVE is trivial, as shown in the previous section. The root of the list is represented by a $L$-node and the cells by $C$-nodes. Pointers head, tail, and next are modelled by edges labelled $h$, $t$, and $n$, respectively.

We consider two list operations: one that puts a new element to the tail of the list, and another that gets the head element from the list. These operations are modelled in our graph transformation system by two rules, shown below. GROOVE combines the left and right hand side of a rule in a single graph, and colours and shapes are used to distinguish different elements. Blue (dashed thin) elements are deleted by the rule application; green (continuous fat) nodes and edges are created; and black (continuous thin) elements are preserved. For simplicity, we assume that our lists always have at least one element\(^1\).

It is clear that the concrete state space of this example is unbounded: the put rule is always enabled, and successive applications of this rule keep producing longer and longer lists. However, the abstract state space produced by our abstraction mechanism is finite. For an abstraction radius of one, the state space has 10 states and 21 transitions, as shown in Fig. 1. Each dashed box represents a state, with its numbering on the lower left corner, and the corresponding shape drawn with solid lines. The transitions between states are shown by dashed arrows, labelled with the rule applied.

There are many interesting points to note in the state space of Fig. 1. First, as long as node and edge multiplicities stay within their bounds, the abstract graph transformation corresponds to the concrete one. This is seen on states $s0$, $s1$, and $s2$, where the shapes are concrete.

\(^1\) otherwise, two more rules are necessary to insert an element to an empty list and to remove the last element.
Figure 1: The abstract state space for the parameters $i = 1$ and $\nu = \mu = 1$
Second, an abstract state may represent an unbounded number of concrete ones. State $s_3$, for example, is an abstract representative for lists with four or more elements. This is illustrated by the put transition from $s_3$ to itself.

Third, the non-determinism of the materialisation algorithm can be seen on the two get transitions from state $s_3$. Although there is only one pre-matching of the rule, when materialising this pre-match two distinct shapes are produced.

Fourth, we can see that the abstract state space has spurious configurations. For example, states $s_6$ to $s_9$ represent lists with unconnected elements, which do not occur in the concrete state space. This spurious shapes arise from the fact that the neighbourhood abstraction mechanism does not keep information regarding connectivity. This is a point where we plan to improve the current theory.

## 4 Conclusion

The results reported above are the very first steps toward the capability for GROOVE to incorporate abstraction. We look upon this as a key factor in the eventual success of the tool. Though currently we have merely implemented the theory described in [BBKR08], we know from experience that having the ability to actually experiment with smaller and larger cases provides a lot of additional motivation and can be a source of new ideas and developments.

For instance, only a working implementation makes it possible to obtain figures about actual abstract state space sizes, which is one of the most important factors in the feasibility of any abstraction-based methods. Some very first figures about the effect of increasing node multiplicity bounds $\nu$ and radii $i$ are collected in the following table. Clearly, the radius has a much greater effect on the state space size than the node multiplicity.

<table>
<thead>
<tr>
<th></th>
<th>$\nu = 1$ states</th>
<th>$\nu = 1$ trans.</th>
<th>$\nu = 2$ states</th>
<th>$\nu = 2$ trans.</th>
<th>$\nu = 3$ states</th>
<th>$\nu = 3$ trans.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 1$</td>
<td>10</td>
<td>21</td>
<td>14</td>
<td>29</td>
<td>18</td>
<td>37</td>
</tr>
<tr>
<td>$i = 2$</td>
<td>389</td>
<td>1060</td>
<td>613</td>
<td>1486</td>
<td>969</td>
<td>2318</td>
</tr>
</tbody>
</table>

Bibliography


