Electronic Communications of the EASST
Volume 32 (2010)

Proceedings of the
Fourth International Workshop on
Graph-Based Tools
(GraBaTs 2010)

Neighbourhood Abstraction in GROOVE

Arend Rensink and Eduardo Zambon

13 pages

Guest Editors: Juan de Lara, Daniel Varro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Neighbourhood Abstraction in GROOVE

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: Important classes of graph grammars have infinite 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 developed the theory of *neighbourhood abstraction*. In this paper, we present the implementation of this theory in GROOVE and illustrate its use with a small grammar that models operations on 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 abstraction is that "similar" states are grouped together, and these groups are modelled in a manner such that the distinction between the grouped states 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 available implementation, namely AUGUR2 [KK08].

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

The rest of this paper is organised as follows. First, we present key concepts of the theory of neighbourhood abstraction in Section 2, and we introduce our running example. In Section 3, we discuss important points of the implementation, along with some design decisions. In Section 4, we present and analyse the results obtained for the example. Conclusions and future work are given in Section 5.

[*] The work reported herein is being carried out as part of the GRAIL project, funded by NWO (Grant 612.000.632).

2 Preliminaries

This section presents the main concepts that are necessary for understanding the implementation discussion given in Section 3.

2.1 GROOVE

GROOVE is a graph transformation tool set whose main purpose is the state space exploration of graph transformation systems (also referred to as graph production systems or graph grammars). A grammar is composed by a set of graph transformation rules and an initial host graph. Rule application follows the Single-Pushout (SPO) approach, with rules composed by left-hand side (LHS) and right-hand side (RHS) graphs. In the setting of this paper we restrict rule application to injective morphisms. Exploration amounts to applying the rules to a host graph in all possible manners, starting with the initial graph and continuing with the graphs produced by the transformations. This process yields the grammar state space, which is stored as a Labelled Transition System (LTS), where states are graphs and transitions are labelled by rule applications. The tool can then model check CTL and LTL formulae on the generated LTS[1].

A problem arises when exploring graph grammars that have an infinite state space, since the corresponding LTS cannot be fully generated. The goal of the work here presented is to implement an abstraction technique in GROOVE that allows the generation of *finite* abstract LTS's for such grammars. To guarantee the soundness of the verification, the abstract LTS is required to be an over-approximation of the concrete one; the approximation should allow the verification of *safety* and *liveness* properties on the abstract LTS, i.e., if a property holds in the abstract level then we can conclude that it also holds in the concrete state space.

2.2 Running example

As an example throughout the paper 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, by way of pointers called *head* and *tail*. 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 in Figures 1(a) and 1(b). Simple abbreviations are used for conciseness. The corresponding morphisms between LHS and RHS of the rules are identified by dotted lines. Figure 1(c) shows the start graph. For simplicity, we assume that our lists always have at least one element[2].

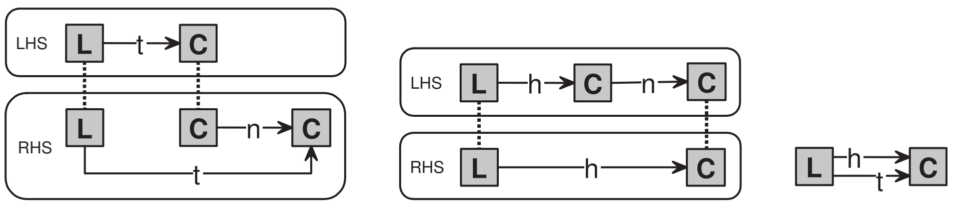
It is clear that the concrete state space of the grammar in Figure 1 is infinite: the put rule is always enabled, and successive applications of this rule keep producing longer and longer lists.

2.3 Neighbourhood abstraction

We work with directed edge-labelled graphs, where the labels are taken from a finite set Lab. Formally, a graph is a tuple $G = \langle N, E \rangle$ of nodes and edges, where the edges are triples $\langle v, a, w \rangle$

[1] More information about GROOVE can be found at the project website: http://groove.cs.utwente.nl.

[2] Otherwise, two more rules are necessary to insert an element to an empty list and to remove the last element.



(a) rule put (b) rule get (c) start graph

Figure 1: A graph transformation system modelling a single-linked list.

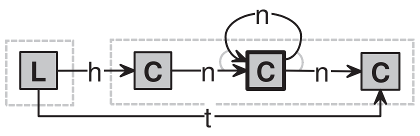


Figure 2: A shape representing lists with four or more elements.

of source node, label, and target node; such that $v,w \in N$ and $a \in \mathsf{Lab}$. Node labels are simulated with self-edges; in fact we assume that Lab is partitioned into two sub-sets: unary (node) labels, and binary (edge) labels.

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. 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 ω standing for "many".

The abstractions are called *shapes*. They are 5-tuples $S = \langle G, \sim, \mathsf{mult}^{\mathsf{n}}, \mathsf{mult}^{\mathsf{o}}, \mathsf{mult}^{\mathsf{i}} \rangle$ in which

- G is the underlying graph structure of the shape;
- $\sim \subseteq N \times N$ is a *neighbourhood similarity* relation;
- $\mathsf{mult}^{\mathsf{n}} \colon N \to M_\nu$ is a *node multiplicity* function, which records how many concrete nodes were folded into a given abstract node, up to bound ν;
- $\mathsf{mult}^{\mathsf{o}}, \mathsf{mult}^{\mathsf{i}} \colon (N \times \mathsf{Lab} \times N/{\sim}) \to M_\mu$ are outgoing and incoming *edge multiplicity* functions, which record how many concrete edges with a certain label were folded into an abstract edge, up to a bound μ and a group of $\sim$-similar opposite nodes.

Nodes and edges of a shape with multiplicity one are called *concrete*, otherwise they are called *collectors*.

Figure 2 shows an example of a shape. The graph structure of the shape is drawn as usual. The equivalence relation $\sim$ is indicated with dashed boxes. Node multiplicities are represented by line thickness: fat nodes have multiplicity ω, thin nodes have multiplicity one. All edge multiplicities are equal to one and are not shown explicitly. Gray arcs "joining" incoming and outgoing edges indicate that the multiplicities apply to a bundle of edges, rather than a single edge.

3 Implementation

In this section we discuss the most important aspects of implementing the neighbourhood abstraction theory in GROOVE.

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.

```
let $S := abstract_i(G)$, $Q := \emptyset$, $F := \{S\}$
while $F \neq \emptyset$
do choose $S \in F$        (which $S$ is selected depends on the exploration strategy)
   let $F := F \setminus \{S\}$
   for $p \in P$, $m \in prematch(p,S)$, $S' \in materialise(m,S)$
   do let $R := normalise(apply(p,m,S'))$
      if $R \notin Q$
      then let $Q := Q \cup \{R\}$, $F := F \cup \{R\}$
      fi
   od
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 non-injective morphisms of a rule p into a shape S. Such a morphism is not yet a match, because the images of p's LHS may be elements with multiplicity greater than one; in this 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 options 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'. At this step, the match of the rule is injective.
- *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.

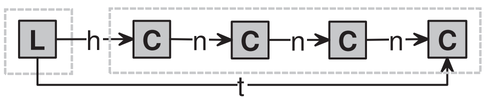
The following subsections present each of these phases in more detail.

3.1 Operation *abstract*

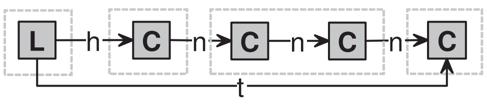
The abstraction uses the concept of neighbourhood equivalence over elements of a graph, denoted $\equiv_i$. It relates nodes with similar neighbourhoods, up to some positive "radius" i, which is a parameter of the abstraction. For a given $i > 0$ and graph G, operation *abstract* computes the relation $\equiv_i$ over N_G recursively. For any $v, w \in N_G$, we have that

- $v \equiv_0 w$, if v and w have the same node labels;
- $v \equiv_i w$, if $v \equiv_{i-1} w$, and v and w have the same number of outgoing and incoming edges for every edge label.

After the relation $\equiv_i$ is computed, we can build a shape S, where



(a) equivalence relation $\equiv_0$



(b) equivalence relation $\equiv_1$

Figure 3: Iterations on the neighbourhood equivalence relation over a graph representing a list with four elements.

- $N_S = N_G/\equiv_i$, i.e., nodes of the shape are the equivalence classes of $\equiv_i$;
- $\sim \; = \; \equiv_{i-1}$, i.e., the shape equivalence relation is taken from the previous iteration of the neighbourhood equivalence;
- $\mathsf{mult}^{\mathsf{n}}$ is the multiplicity of each equivalence class in $\equiv_i$, bounded by ν;
- $\mathsf{mult}^{\mathsf{o}}$ and $\mathsf{mult}^{\mathsf{i}}$ are the multiplicities of the set of edges between each node and equivalence class, bounded by μ.

An application of the *abstract* operation for our running example is shown in Figure 3. We assume as input a graph representing a list of four elements, and also that the abstraction radius $i = 1$. Iteration $\equiv_0$ distinguish nodes based only on their labels, as can be seen in Figure 3(a). Subsequent iterations refine the equivalence classes by looking at incoming and outgoing edges. This is shown in Figure 3(b), where the first and last cells of the list are distinguished by the head and tail edges. The resulting shape built by operation *abstract* in this example corresponds to the one depicted in Figure 2.

3.2 Operations *prematch*, *apply* and *normalise*

A pre-match m of the LHS of a rule $p = \langle L, R \rangle$ into a shape S, is a non-injective morphism $m : L \to G_S$, such that node and edge multiplicities are satisfied by the mapping. Operation *prematch* uses the normal rule matching implemented in GROOVE and then removes the invalid matches by checking the conditions on multiplicities.

A pre-match m has to be massaged into a *concrete* match m', such that: (i) m' is injective; (ii) all nodes in the image of m' are concrete and belong to a singleton equivalence class; and (iii) all edges in the image of m' are concrete. This adjustment from pre-match m to concrete match m' is done by the *materialise* operation, explained in the next section.

Given a concrete match m' into a materialised shape S', rule application is performed as usual. Operation *apply* simply uses the normal transformation code from GROOVE.

After transformation, a shape needs to be normalised, i.e., the concrete parts need to be merged back into the shape. This entails the computation of the $\equiv_i$ relation on shapes, which is similar to the one described in operation *abstract*.

3.3 Operation *materialise*

The materialisation phase is the most complex one from the abstraction algorithm. This is due to the fact that this phase resolves all non-determinism of the algorithm; the materialised shapes returned by *materialise* are ready to be transformed by conventional rule application.

Op.	Prio.	Parameters		Creates
matNode	0	$n_c \in S$	collector node, from which the new nodes will be materialised	singNode
		$N_p \subseteq N_L$	set of rule nodes mapped to n_c by the pre-match	
matEdge	1	$e_c \in S$	collector edge, from which the new edges will be materialised	pullNode
		$E_p \subseteq E_L$	set of rule edges mapped to e_c by the pre-match	
pullNode	2	$e \in S$	edge that is pulling a new node from n_c	–
		$n_c \in S$	collector node that is being pulled by e	
		$u \in M_\nu$	multiplicity for the new node that will be created	
singNode	3	$n_s \in S$	the node to be singularised	–

Table 1: Summary of the sub-operations of the materialisation phase. All operations are performed on a given shape S, guided by the pre-match of the LHS L of a rule p.

Given a pre-match m of a rule $p = \langle L, R\rangle$ into shape S, *materialise* finds all shapes S' such that $f : S' \to S$ is a shape morphism and $m' : L \to S'$ is a concrete match. The challenge in implementing this step lies in transforming the descriptive solution just given into a constructive algorithm that produces the set of all possible materialisations, based on the given shape S and pre-match m.

The materialisation algorithm iteratively changes the original shape in order to search for valid sub-shapes. The changes to be performed are divided into four *materialisation operations*. A summary of these operations is given in Table 1. The second column of Table 1 lists the operation priority, with zero being the highest priority. All materialisation operations are non-deterministic. This implies that each operation may produce zero or more new materialisation objects. If the execution of the operation yields zero results, then it is said that the operation failed, i.e., performing the operation on the materialisation object does not produce a valid shape.

Materialisation operations are put into a priority queue and traversed in a breadth-like fashion. When a materialisation is completed, it is moved to the result set of *materialise*. Not all operations can be determined when the materialisation process starts, so the execution of an operation can create other ones. The relation between creation of operations is given by the fourth column of Table 1.

The main reason for splitting the materialisation phase in sub-operations is understandability. The rationale for splitting operations is that each materialisation operation must introduce only one level of non-determinism. We proceed to explain each operation of Table 1 in detail.

3.3.1 Equation systems

An equation system is a device used for searching valid shape configurations during the materialisation phase. Operations matEdge and singNode use equation systems. We present the common points here and discuss the particularities in the sub-sections describing the operations.

The variables in these systems hold multiplicity values, and thus are called *multiplicity variables*. In the following, let x and y be such variables. Equation systems are composed of three parts

- *set constraints*, in the form $x \in U$, where U is a set of arbitrary multiplicities. Variables occurring in this part of the system are called *constrained*.
- *equations*, in the form $y = u - x$, where u is a constant multiplicity and x is a constrained variable. Variables occurring in the left-hand side of equations are called *derived* because their sets of values are obtained when the values of the constrained variables are fixed.
- *admissibility constraints*, which restrict the overall admissibility of a shape configuration. Such constraints are of the form $\sum_{t \in O} t \approx \sum_{t \in I} t$, where O and I are sets of outgoing and incoming terms, respectively. A *term* is of the form $u \cdot x$, where u is a constant multiplicity value and x is a multiplicity variable. An admissibility constraint is satisfied if the resulting multiplicities produced by both sums are overlapping, i.e., have a non-empty intersection.

It is important to note that usual arithmetic symbols (e.g., $+$ and $-$) occurring in an equation system actually stand for operations on multiplicities. In particular, an equation $y = u - x$ may admit multiple solutions for a fixed x, namely the multiplicity values that when summed with x equal u. More formally, given a value $u_x \in U$ for the constrained variable x, the possible values of the derived variable y are given by $y \in \{u_y \in M_k \mid u_x + u_y = u\}$, for a certain bound k.

Overlapping of multiplicities in the admissibility constraints is defined in terms of the intersection of multiplicity intervals. The intuition goes as follows. Two multiplicities are overlapping if: (i) their values are equal, e.g., $1 \approx 1$, $1 \not\approx 0$, and $\omega \approx \omega$; or (ii) one multiplicity is ω and the other is a concrete value greater than the multiplicity bound, e.g., $\omega \approx 2$, and $1 \not\approx \omega$ (assuming multiplicity bound $k = 1$).

Solving an equation system is done with a simple search algorithm that goes over all possible values for the constrained variables; calculates the values for the derived variables using the equations; and checks if all admissibility constraints are satisfied. Each valid solution produces a corresponding shape, obtained from the values of the variables.

3.3.2 Materialise Node

This operation materialises (creates) one or more nodes from a collector node n_c, i.e., a node with multiplicity greater than one. As shown in the third column of Table 1, the other parameter of matNode is N_p, the sub-set of nodes in the LHS of the rule that were mapped to n_c by the pre-match. The number of new copies of the collector node is determined by the cardinality of set N_p. All new materialised nodes are created with multiplicity one and the mapping of the pre-match is adjusted to the new nodes. In addition, all edges adjacent to n_c are duplicated on the new nodes. The non-determinism of this operation comes from the choice on the remaining multiplicity of n_c, once the new nodes are materialised. This operation creates a singNode operation for each of the newly materialised nodes.

Figure 4 shows an example execution of matNode. On the left side of Figure 4 we see shape S_0 with a pre-match of the LHS L of rule get. Since the image of rule node r_2 is the collector node n_2, it is necessary to materialise a concrete copy of n_2. This leads to the application of matNode with parameters $n_c = n_2$ and $N_p = \{r_2\}$. The operation creates a new concrete node n_4 and duplicates all adjacent edges of n_2, as can be seen on the right side of Figure 4. The matNode operation in this example produces two new shapes, S_1 and S_2, which differ only in the remaining multiplicity of the collector node. We consider the node multiplicity bound $\nu = 1$,

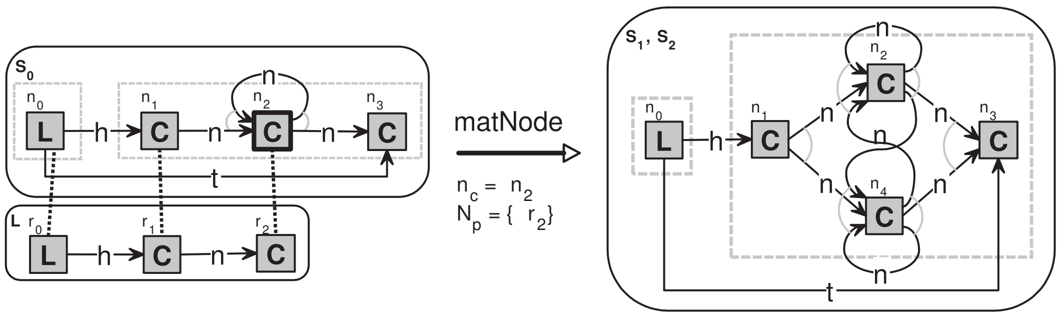


Figure 4: Example of an execution of the matNode operation. Shapes S_1 and S_2 differ only on the multiplicity of node n_2: $\mathsf{mult}^{\mathsf{n}}_{S_1}(n_2) = 1$, $\mathsf{mult}^{\mathsf{n}}_{S_2}(n_2) = \omega$.

thus, n_2 (which has multiplicity ω) originally represents two or more nodes. When materialising n_4 from n_2, the remaining multiplicity of n_2 can be either one or ω. Node n_4 will later be made singular by a singNode operation.

3.3.3 Materialise Edge

In the same vein of matNode, operation matEdge materialises (creates) one or more edges from a collector edge e_c, i.e., an edge with outgoing or incoming multiplicities greater than one, or an edge that is part of an edge multiplicity bundle (shared edge multiplicity). The additional parameter of matEdge is E_p, the sub-set of edges in the LHS of the rule that were mapped to e_c by the pre-match (see Table 1). Similarly, the number of new copies of the collector edge is determined by the cardinality of set E_p. The non-determinism of this operation comes from the choice on the remaining incoming and outgoing multiplicities of e_c, once the new edges are materialised. This operation may create new pullNode operations.

Performing this operation involves constructing and solving an equation system. We create a pair of multiplicity variables for each edge bundle that is affected by the operation. One of the variables of the pair is taken as a constrained variable and the other as a derived one. In this equation system the set constraints are always formed by singletons sets[3], with the value taken from $|E_p|$, i.e., the constrained variables stand for the number of concrete edges that will be extracted from the collector edge. The derived variables in the equations represent the remainder multiplicities of the collector edge or edge bundles. The admissibility constraints ensure that no invalid values are assigned to the derived variables, e.g., it is not possible to assign a multiplicity zero to an edge bundle that has one or more edges.

Figure 5 gives an example of application for matEdge. The input shape is S_2, one of the results of the matNode operation in Figure 4. After matNode materialises n_4 and adjusts the match, we see that the image of rule edge $\langle r_1, n, r_2 \rangle$ is an edge with shared outgoing and incoming multiplicities. This leads to the execution of the matEdge operation. In this execution, the

[3] This of course implies that the values for the constrained variables are already known. The reason for creating the equation system for matEdge in this format is just to comply to the overall format of equation systems described in Section 3.3.1. This allows for re-use of common functionalities (code).

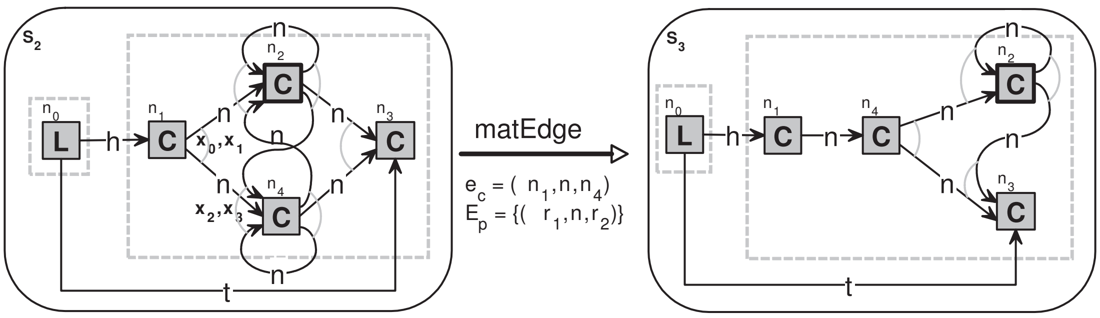


Figure 5: Example of an execution of the matEdge operation. This operation creates an equation system. The association of the equation system variables with edge multiplicities is shown in shape S_2. Variables x_0 and x_2 are constrained, and variables x_1 and x_3 are derived. This particular execution of matEdge is deterministic.

operation creates the following equation system

$$\begin{array}{rl} \text{set constraints} & x_0,\ x_2 \in \{1\} \\ \text{equations} & x_1 = 1 - x_0 \quad x_3 = 1 - x_2 \\ \text{admissibility constraints} & x_1 + \omega \approx x_3 + \omega \end{array}$$

where the association of the multiplicity variables with edge bundles is shown in shape S_2 of Figure 5. The solution of this equation system is trivial, with $x_0 = x_2 = 1$ and $x_1 = x_3 = 0$. The resulting shape S_3 is depicted on the right side of Figure 5. It is important to note that in this particular example, the operation produces only one result shape. However, in the general case, matEdge returns a set of results.

3.3.4 Pull Node

The theory of neighbourhood abstraction requires that all elements up to radius i in the image of the match have to be concrete. After performing matNode and matEdge operations, we obtain a rule match on which the elements of the image are concrete, but it is still possible to have abstract elements in the neighbourhood of the image. Operation pullNode transforms the neighbourhood, by further materialising nodes when needed.

This operation is very similar to matNode, with the exception that only one new node is created, which can have an arbitrary positive multiplicity, defined by parameter u. Note that, as in operation matNode, all adjacent edges of the collector node are duplicated. The source of non-determinism of pullNode is the same as in matNode, i.e., the choices on the remaining multiplicity of the collector node. The newly created node will not be singularised later; this operation does not create any new operations.

3.3.5 Singularise Node

This is the operation with the lowest priority, being executed after all others. When performing a singNode operation, we assume that the rule match is final and that the number of nodes in the

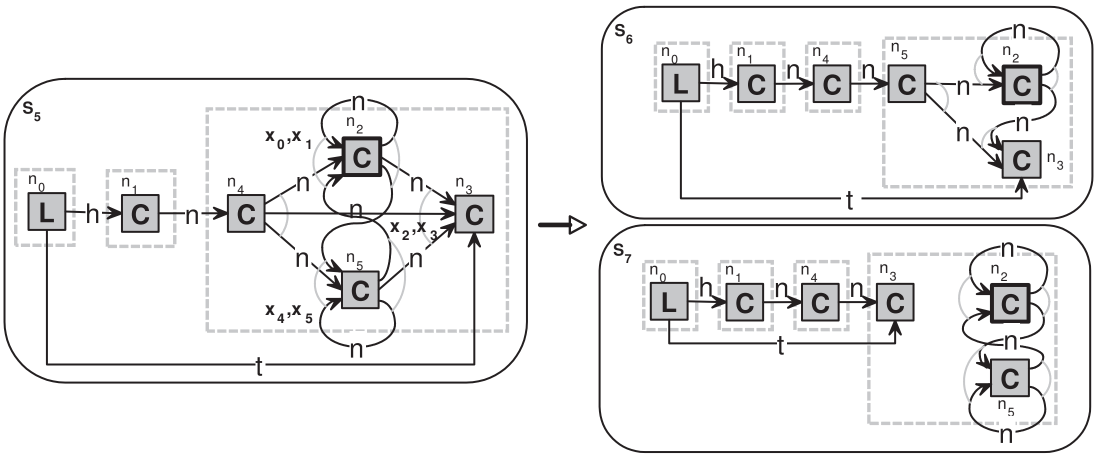


Figure 6: Example of an execution of the singNode operation with parameter $n_s = n_4$. This operation also creates an equation system. The association of the equation system variables with edge multiplicity bundles is shown in shape S_5. Variables x_0, x_2 and x_4 are constrained, and variables x_1, x_3 and x_5 are derived.

shape will no longer change. What is left to decide are the outgoing and incoming multiplicities of the edge bundles that will be affected by the operation.

In order to put a node n_s in a singleton equivalence class, singNode creates another equation system, where the variables represent the multiplicities of affected edge bundles. For each such bundles, we create a pair of variables. One variable of the pair, x, represents the multiplicity associated with the singular equivalence class that will be created; the other variable, y, stands for the multiplicity related to the remainder equivalence class after the split. Variable x is put in a set constraint, ranging on the set $\{0,1\}$, and variable y goes in an equation, i.e., x is a constrained variable and y is a derived one. The admissibility constraints care for the sanity of the shape configuration after the split, such that valid solutions of the system produce valid final shapes.

Figure 6 shows a singNode application with parameter $n_s = n_4$. The input shape S_5 is an intermediate candidate shape produced during the materialisation. This singNode execution creates the following equation system

$$\begin{array}{rlll} \text{set constraints} & x_0,\ x_2,\ x_4 \in \{0,1\} & & \\ \text{equations} & x_1 = 1 - x_0 & x_3 = 1 - x_2 & x_5 = 1 - x_4 \\ \text{admissibility constraints} & 1 \approx \omega \cdot x_0 + x_2 + x_4 & \omega \approx \omega \cdot x_1 + x_3 + x_5 & \end{array}$$

where the association of the multiplicity variables with edge bundles is shown in shape S_5 of Figure 6. This equation system has two solutions, namely: (i) $x_0 = x_2 = x_5 = 0$ and $x_1 = x_3 = x_4 = 1$; and (ii) $x_0 = x_3 = x_4 = 0$ and $x_1 = x_2 = x_5 = 1$. These solutions produce shapes S_6 and S_7, presented on the right side of Figure 6.

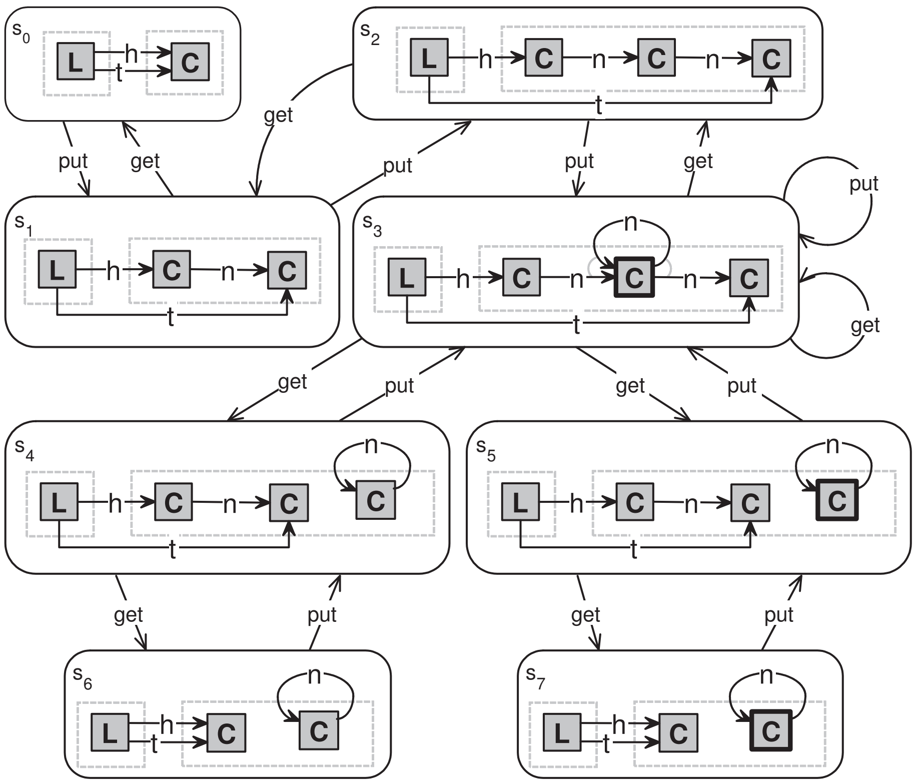


Figure 7: The abstract LTS of our running example, for the parameters $i = 1$ and $\nu = \mu = 1$.

4 Results

When applying the implemented abstraction to our running example, we obtain the abstract state space depicted in Figure 7. For an abstraction radius of one, the LTS has 8 states and 16 transitions. Each rounded box represents a state, with its numbering on the upper left corner, and the corresponding shape. The transitions between states are shown by arrows, labelled with the rule applied.

There are many interesting points to note in the state space of Figure 7. 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 s_0, s_1, and s_2, where the shapes are concrete.

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 and get transitions from s_3 to itself.

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

Fourth, we can see that the abstract state space has spurious configurations. For example, states s_4 to s_7 represent lists with unconnected elements, which do not occur in the concrete state

	$\nu = 1$		$\nu = 2$		$\nu = 3$	
	states	trans.	states	trans.	states	trans.
$i = 1$	8	16	11	22	14	28
$i = 2$	16	33	22	45	28	57

Table 2: State space sizes for a small variation of abstraction parameters ($\mu = 1$).

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.

After the abstract LTS is generated we can proceed to model check the properties of interest. For the LTS in Figure 7, we can check, for example, that the following properties hold: (i) the head cell has no predecessors; (ii) the cells are not shared; and (iii) rule get is applied infinitely often. Properties (i) and (ii) talk about safety and (iii) is a liveness property. They are informally described in English but can be easily translated into temporal logic formulae. Since these properties hold in the abstract LTS, we can then conclude that they also hold in the infinite concrete state space.

5 Conclusions and future work

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 an important factor in the feasibility of any abstraction-based methods. Some very first figures about the effect of increasing node multiplicity bounds ν and radii i are collected in Table 2; the edge multiplicity bound μ was kept equal to one. Clearly, the radius has greater effect on the state space size than the node multiplicity. All tests took just a few seconds to run. As an additional example we looked at the circular buffer grammar presented in [RD06]. In that paper the abstract state space was generated by hand, now we can mechanically reproduce it with the tool.

Current implementation efforts are aimed towards integration of the abstraction mechanism with the GUI of the Simulator in GROOVE. Experience shows that an interactive graphical interface can be of great assistance when modelling. On the theory side we are investigating different abstractions that are more adequate to handle structural properties such as connectivity and cyclicity. A key insight is that the radius used in neighbourhood abstraction is too coarse. Usually we are not interested in the whole neighbourhood of a node but instead we want to look at different radii for different types of edges. For the moment, we are aiming at a more property driven abstraction, which will allow the abstraction mechanism to be more precise, since it will retain the information that is relevant for the verification of a given property.

Bibliography

[BBKR08] J. Bauer, I. B. Boneva, M. E. Kurban, A. Rensink. A Modal-Logic Based Graph Abstraction. Pp. 321–335 in [EHRT08].

[BKK03] P. Baldan, B. König, B. König. A Logic for Analyzing Abstractions of Graph Transformation Systems. In Cousot (ed.), *Static Analysis Symposium (SAS)*. LNCS 2694, pp. 255–272. Springer, 2003.

[CC77] P. Cousot, R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. Pp. 238–252. 1977.

[EHRT08] H. Ehrig, R. Heckel, G. Rozenberg, G. Taentzer (eds.). *International Conference on Graph Transformations (ICGT)*. LNCS 5214. Springer, 2008.

[KK06] B. König, V. Kozioura. Counterexample-Guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In *TACAS*. LNCS 3920, pp. 197–211. Springer, 2006.

[KK08] B. König, V. Kozioura. AUGUR2— A New Version of a Tool for the Analysis of Graph Transformation Systems. *ENTCS* 211:201–210, 2008.

[RD06] A. Rensink, D. Distefano. Abstract Graph Transformation. In Mukhopadhyay et al. (eds.), *Software Verification and Validation*. ENTCS 157, pp. 39–59. May 2006.

[Ren04] A. Rensink. Canonical Graph Shapes. In Schmidt (ed.), *Programming Languages and Systems (ESOP)*. LNCS 2986, pp. 401–415. Springer, 2004.

[RN08] S. Rieger, T. Noll. Abstracting Complex Data Structures by Hyperedge Replacement. Pp. 69–83 in [EHRT08].

[SRW98] S. Sagiv, T. W. Reps, R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. *ACM ToPLaS* 20(1):1–50, 1998.

[SRW02] S. Sagiv, T. W. Reps, R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM ToPLaS* 24(3):217–298, 2002.