Graph Subsumption in Abstract State Space Exploration

Eduardo Zambon* Arend Rensink
Formal Methods and Tools Group
Computer Science Department
University of Twente, The Netherlands
{zambon, rensink}@cs.utwente.nl

In this paper we present the extension of an existing method for abstract graph-based state space exploration, called neighbourhood abstraction, with a reduction technique based on subsumption. Basically, one abstract state subsumes another when it covers more concrete states; in such a case, the subsumed state need not be included in the state space, thus giving a reduction. We explain the theory and especially also report on a number of experiments, which show that subsumption indeed drastically reduces both the state space and the resources (time and memory) needed to compute it.

1 Introduction

Traversal of the state space of systems is the cornerstone of many verification/analysis methods, *e.g.*, model checking [1], and is therefore a subject under intense investigation. In particular, techniques for pruning the search space (*e.g.*, partial-order reduction [10]) and duplicate state detection (*e.g.*, collapsing under isomorphism [16]) were shown to be essential ingredients in the fight to tame the ever looming problem of state space explosion. However, important classes of systems have infinite state spaces and therefore cannot be (fully) explored using traditional traversal techniques.

One way to address this problem is to perform *state abstraction*, where "similar" concrete states are collapsed under an abstract representative, with the behaviour of the abstract state encompassing all possible behaviour of the collapsed concrete states. This notion of abstraction is the basis of well-known techniques such as abstract interpretation [5] and shape analysis [20].

State "similarity" is the point where many abstractions differ; in order to define what "similar" means one has to look at the underlying framework one uses to represent systems. In our case, we use *graph transformation* as the framework for modelling system behaviour and therefore our abstraction works over graphs. In this context of graph transformation, many theoretical studies on suitable abstractions have been proposed [15, 17, 3, 19, 2, 4, 21]. However, only the last three of these were backed-up by tool support.

In previous work [18], we presented a prototype implementation of the *neighbourhood abstraction* theory developed in [3], as an extension of the GROOVE tool set [14, 9]. The main functionality of GROOVE is the ability to explore the state space of graph transformation systems (more details in Section 2), but a concrete exploration can only traverse part of the state space of an infinite state system. The abstraction extension allows GROOVE to generate a *finite* abstract state space that over-approximates the original concrete one. Over-approximation guarantees soundness of the verification, *i.e.*, properties that hold in the abstract domain also hold in the concrete counterpart.

The main goal of the prototype implementation was to serve as a practical proof-of-concept of the theoretical ideas. Unsurprisingly, performance was not optimal and only a few small systems could be properly analysed. Since then, we completely re-implemented the abstraction code and incorporated

*The work of this author is supported by the GRAIL project, funded by NWO (Grant 612.000.632).

many performance improvements. This paper describes one of such improvements, based on the concept of *state subsumption*. We illustrate the performance gain provided by the subsumption with experiments using test cases from different areas of computer science.

The rest of this paper is organised as follows. First, we present the key concepts of graph transformation and abstraction in Section 2. In Section 3, we introduce the subsumption relation for abstract states and we show how it can be used during exploration. In Section 4, we present the experiments performed and analyse the results. Related work is discussed in Section 5. Finally, Section 6 concludes the paper.

2 Graph Production Systems and Abstraction

Graph transformation [7] is a rewriting technique that operates over graphs. In its simplest form, a *transformation rule* consists of a left-hand side (LHS) and a right-hand side (RHS) graph and specifies the changes that should be performed to a *host graph*. Applying a rule r to a host graph G basically amounts to finding a match m of the LHS of r in G and replacing this matched part of G by the RHS of r, thus producing a new graph H. We write $G \xrightarrow{r,m} H$ to denote a rule application and we write $G \xrightarrow{r} H$ if there exists a match m such that $G \xrightarrow{r,m} H$.

A graph production system (or graph grammar) is formed by a set of graph transformation rules R and a start host graph. State space exploration of the grammar consists of performing all possible applications of the rules from R into the start graph, and repeating this process to all newly generated graphs. The state space obtained in this way can be represented by a Graph Transition System (GTS), which is a labelled transition system where states are host graphs and transitions are rule applications, *i.e.*, a pair of rule r and associated match m. Once generated, a GTS can be analysed as usual, for example by model checking properties written as temporal logic formulae (*e.g.*, using Computation Tree Logic – CTL). Clearly, if the rewrite system modelled by a graph grammar is non-terminating, the state space is infinite and thus a GTS cannot be fully constructed by a normal exploration method. To handle infinite state systems, some form of abstraction is required. One of such abstractions, called neighbourhood abstraction, is presented in Section 2.2. First, we formalise the graph representation that we use.

We assume the existence of a finite set of labels Lab, partitioned into unary and binary label sets, denoted Lab^U and Lab^B, respectively. We work with simple directed graphs, with labels taken from Lab.

Definition 1 (Graph) *A graph is a tuple $G = \langle N, E \rangle$, where N is a finite set of nodes and $E \subseteq N \times \mathsf{Lab} \times N$ is a finite set of directed labelled edges, such that $\langle v, l, w \rangle \in E$ with $l \in \mathsf{Lab}^\mathsf{U}$ implies $v = w$.* ◀

Given $\langle v, l, w \rangle \in E$, v and w are called source and target nodes, respectively; and l is the edge label. We simulate node labels with self-edges labelled with unary labels. Given $v \in N$, the *set* of labels of node v, denoted $\mathsf{lab}(v)$, is defined as $\mathsf{lab}(v) = \{l \in \mathsf{Lab}^\mathsf{U} \mid \langle v, l, v \rangle \in E\}$. For convenience, we write E^B to denote the set of edges with binary labels, *i.e.*, $E^\mathsf{B} = \{\langle v, l, w \rangle \in E \mid l \in \mathsf{Lab}^\mathsf{B}\}$.

2.1 Example of a Graph Grammar

As an example we use a graph grammar modelling the behaviour of a firewall in a network, taken from [12]. A *firewall* has *inner* and *outer interfaces*, to which *locations* can be connected. Locations are marked with the kind of interface they are connected to. Data are represented as *packets*, which can be transferred between locations or through the firewall. Packets can either be *safe* or *unsafe*. Safe packets can be *at* any location but unsafe packets cannot exist at inner locations. Figure 1 shows an example configuration of a network with simple abbreviations used for conciseness. The network has one outer location and two inner ones, and there are five packets being transmitted.

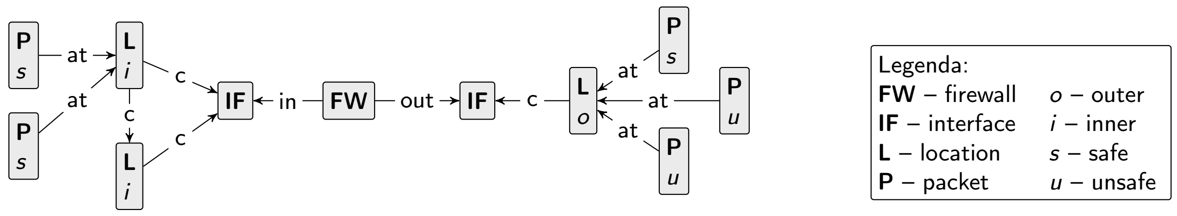
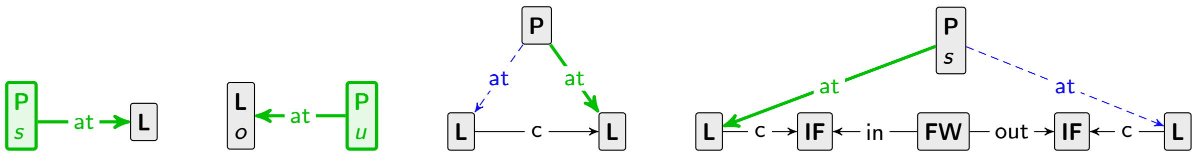


Figure 1: Concrete graph representing a possible state of a network with a firewall.



(a) Safe packet creation
(b) Unsafe packet creation
(c) Packet transfer between locations
(d) Firewall filtering

Figure 2: Examples of transformation rules for the firewall grammar.

Figure 2 shows four transformation rules of the grammar. The rules are in GROOVE single-graph notation, which uses colours and line formats to distinguish rule elements. Black (continuous thin) elements are matched and kept by rule application, blue elements (dashed thin) are matched and deleted, and green (continuous bold) elements are created[1]. Figures 2(a) and 2(b) show the rules for packet creation. A safe packet can be created at any location, whereas an unsafe packet can only be created at outer locations. Infinite behaviour stems from these two rules; since they are always enabled, an infinite number of packets can be created. Figure 2(c) depicts a rule for packet transfer between locations. Since all locations on each side of the firewall are of the same type, there is no need to distinguish between safe and unsafe packets. A dual rule (not shown here) transfers packets on the reverse direction of the connection edge, thus making the connection bi-directional. The rule in Figure 2(d) shows the firewall filter, that only allows safe packets to reach inner locations.

2.2 Neighbourhood Abstraction

Our notion of abstraction is based on neighbourhood similarity: nodes are considered equivalent if they have the same labels and the same number of incoming and outgoing edges. Graphs are abstracted by folding all equivalent nodes into one, while keeping count of their original number up to some bound of precision. Counting up to some bound is done using *multiplicities*.

2.2.1 Multiplicities

We use ω to denote an upper bound on the set of natural numbers, *i.e.*, $\omega \notin \mathbb{N}$ and for all $k \in \mathbb{N}, k < \omega$. We write $\mathbb{N}^\omega$ as a short-hand notation for $\mathbb{N} \cup \{\omega\}$. We can then define simple arithmetic operations over $\mathbb{N}^\omega$, such as *addition* and *subtraction*. For example, given $i, j \in \mathbb{N}^\omega$,

$$i + j = \begin{cases} i + j & \text{if } i, j \in \mathbb{N}, \\ \omega & \text{otherwise.} \end{cases}$$

[1]In the standard two-graph notation, the LHS is formed by black and blue elements and RHS by black and green elements.

The symbol $+$ is overloaded: the one on the left represents addition over $\mathbb{N}^\omega$ while the one on the right is the usual addition over $\mathbb{N}$. Note that the first condition of the definition implies that $i + j < \omega$.

Definition 2 (Multiplicity) *A* multiplicity *is an element of the set* $\mathbf{M} = \{\langle i, j\rangle \in (\mathbb{N} \times \mathbb{N}^\omega) \mid i \leq j\}$. ◀

Multiplicities are used to represent an interval of consecutive values taken from $\mathbb{N}^\omega$, *i.e.*, we write $\langle i, j\rangle$ as a compact representation for the set $\{k \in \mathbb{N}^\omega \mid i \leq k \leq j\}$. Given a multiplicity $\langle i, j\rangle \in \mathbf{M}$:

- if $i = j$, we call the multiplicity *singleton* and we use a short-hand notation by writing only the lower-bound i in **bold**, *i.e.*, $\mathbf{i}$. The singleton multiplicity $\mathbf{1}$ is called *concrete*; and
- if $j = \omega$, we use a short-hand notation by writing the lower-bound i in **bold**, super-scripted with $+$, *i.e.*, $\mathbf{i}^+$.

Set $\mathbf{M}$ is infinite, since i and j are taken from infinite sets. To ensure finiteness, we need to define a bound of precision, which limits the possible values of i and j.

Definition 3 (Bounded multiplicity) *A* bounded multiplicity *is an element of set* $\mathbf{M}^\mathbf{b} \subset \mathbf{M}$*, defined, for a given bound* $\mathbf{b} \in \mathbb{N}$*, as* $\mathbf{M}^\mathbf{b} = \{\langle i, j\rangle \in \mathbf{M} \mid i \leq \mathbf{b} + 1,\ j \in \{0, \ldots, \mathbf{b}, \omega\}\}$. ◀

The theory of neighbourhood abstraction presented in [3] is parameterised with two multiplicity bounds, for node and edge counting. In practice, these bounds are usually set to a low value, such as 1 or 2, since they can greatly affect the size of the abstract state space. For the remainder of this paper we consider both bounds to be 1 and we only work with bounded multiplicities. This effectively limits the possible multiplicity values to six: $\mathbf{0}$, $\langle 0, 1\rangle$, $\mathbf{0}^+$, $\mathbf{1}$, $\mathbf{1}^+$, and $\mathbf{2}^+$. Any natural number can be projected to a bounded multiplicity by means of a simple approximation function. For a given set A, we write $|A|$ to denote the bounded multiplicity approximated from the cardinality of set A.

It is simple to define arithmetic operations over multiplicities based on the operations over $\mathbb{N}^\omega$. In order to later define the state subsumption relation (Section 3) we need the concept of *multiplicity subsumption*, which amounts to interval inclusion. Given two bounded multiplicities $\mu, \nu \in \mathbf{M}^\mathbf{b}$, let $\mu = \langle i, j\rangle$ and $\nu = \langle i', j'\rangle$. We say that μ is *subsumed* by ν or that ν *subsumes* μ, denoted $\mu \sqsubseteq \nu$, if $i \geq i'$ and $j \leq j'$.

2.2.2 Neighbourhood Equivalence

We begin this section introducing some additional notation. Given a graph $G = \langle N, E\rangle$, a node $v \in N$, a binary label $l \in \mathsf{Lab}^\mathsf{B}$, and a set of nodes $C \subseteq N$, we consider the following sets of edges:

- $\mathsf{out}(v, l, C) = \{\langle v, l, w\rangle \in E^\mathsf{B} \mid w \in C\}$, *i.e.*, the set of *outgoing* l-edges from v into nodes of C; and
- $\mathsf{in}(v, l, C) = \{\langle w, l, v\rangle \in E^\mathsf{B} \mid w \in C\}$, *i.e.*, the set of *incoming* l-edges into v from nodes of C.

Formally, to define neighbourhood similarity we create a *neighbourhood equivalence relation* $\equiv$ over graph nodes.

Definition 4 (Neighbourhood equivalence relation) *Given a graph* $G = \langle N, E\rangle$*, the* neighbourhood equivalence relation $\equiv$ *over nodes of* G *is defined for two radii (with* $v, w \in N$*):*

- $v \equiv_0 w$ *if* $\mathsf{lab}(v) = \mathsf{lab}(w)$*; and*
- $v \equiv_1 w$ *if* $v \equiv_0 w$*,* $|\mathsf{out}(v, l, C)| = |\mathsf{out}(w, l, C)|$*, and* $|\mathsf{in}(v, l, C)| = |\mathsf{in}(w, l, C)|$*, for all binary labels* $l \in \mathsf{Lab}^\mathsf{B}$ *and all sets of nodes* $C \in N/{\equiv_0}$. ◀

From the definition, we see that two nodes are equivalent at radius 0 if they have the same labels. Equivalence classes are then refined at radius 1, where we look for the number of edges incoming from and outgoing to nodes of other equivalence classes.

As with multiplicity bounds, the theory is also parameterised with an abstraction radius. However, experiments with the prototype implementation showed that increasing the radius above one is not feasible in practice. The current abstraction implementation fixes the maximum radius to one.

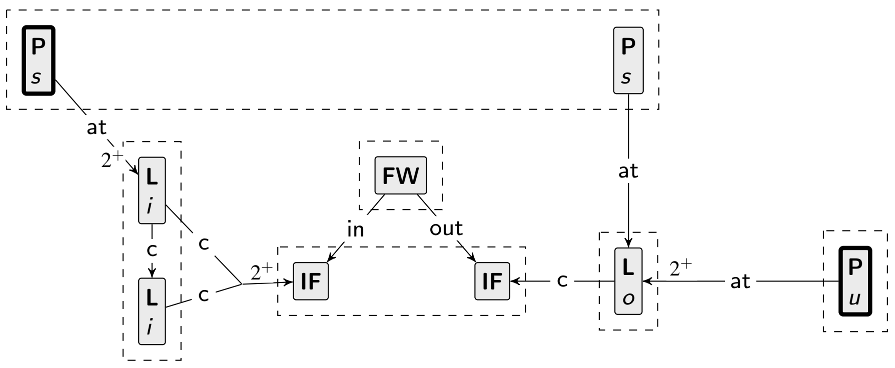


Figure 3: Shape obtained when abstracting the graph of Figure 1. Node multiplicities are indicated by thickness: bold nodes have multiplicity $\mathbf{2}^+$; thin nodes have multiplicity $\mathbf{1}$. Edge groups with multiplicity $\mathbf{2}^+$ are explicitly shown; groups without values in the figure have multiplicity $\mathbf{1}$. Edge multiplicities close to the edge arrow indicate incoming multiplicities from the opposite equivalence class.

2.2.3 Shapes

Our graph abstractions are called *shapes*, following the term defined in shape analysis [20]. A shape is a graph with some additional structure.

Definition 5 (Shape) *A* shape *is a tuple $S = \langle G_S, \simeq_S, \mathsf{mult}^{\mathsf{n}}_S, \mathsf{mult}^{\mathsf{o}}_S, \mathsf{mult}^{\mathsf{i}}_S\rangle$, where:*

- *$G_S = \langle N_S, E_S\rangle$ is the underlying graph structure of the shape;*
- *$\simeq_S \subseteq N_S \times N_S$ is a* similarity relation *over nodes of S;*
- *$\mathsf{mult}^{\mathsf{n}}_S : N_S \to \mathbf{M}$ is a* node multiplicity *function, which records how many concrete nodes were folded into an abstract node; and*
- *$\mathsf{mult}^{\mathsf{o}}_S, \mathsf{mult}^{\mathsf{i}}_S : (N_S \times \mathsf{Lab}^{\mathsf{B}} \times N_S/{\simeq_S}) \to \mathbf{M}$ are outgoing and incoming* edge multiplicity *functions, which record how many concrete edges with a certain label were folded into an abstract edge.* ◀

If a shape node v has an associated concrete multiplicity, *i.e.*, $\mathsf{mult}^{\mathsf{n}}_S(v) = \mathbf{1}$, then v is called *concrete*. Nodes that are not concrete are called *collectors*.

A shape is constructed by abstracting a concrete host graph. Given a graph G, we first compute the neighbourhood equivalence relation $\equiv$ over G. The graph component G_S of the shape is constructed by folding the nodes of each equivalence class of $\equiv_1$, while recording the multiplicities of these classes in the multiplicity maps of S. The similarity relation $\simeq_S$ is taken as $\equiv_0$.[2] A detailed explanation on shape construction is given in [3].

Figure 3 shows an example of a shape. The graph structure is drawn as usual. The similarity relation $\simeq_S$ is indicated with dashed boxes. Node multiplicities are represented by line thickness: bold nodes have multiplicity $\mathbf{2}^+$ and thin nodes have multiplicity $\mathbf{1}$. Groups of edges with incoming multiplicity $\mathbf{2}^+$ are explicitly identified; the remainder edge multiplicities are all equal to $\mathbf{1}$ and are not shown. The shape in Figure 3 is an abstraction of the graph from Figure 1, where the safe packets at the top inner location and the unsafe packets at the outer location of the graph were collapsed into collectors nodes of the shape (since the multiplicity bounds are one, any natural greater than one is mapped to $\mathbf{2}^+$ in the abstract

[2]The definition of a shape is “generic” in the sense that any binary relation on nodes can be used as the similarity relation $\simeq$. In this paper, however, we consider only shapes where the relation $\simeq$ is taken as the neighbourhood equivalence $\equiv_0$.

domain[3]). An important point to note is that the shape in Figure 3 serves as an abstract representative not only for the graph in Figure 1 but also for any graph with *two or more* packets of the correct kind at the corresponding locations of the shape. Given a shape S, we write $\mathsf{concr}(S)$ to indicate the (possibly infinite) set of *concretisations* of S, *i.e.*, the set of graphs that can be abstracted to S.

3 Subsumption for State Space Reduction

In this section we first present the general algorithm for abstract state space traversal and we show the current duplicate detection mechanism used in GROOVE, based on graph certificates and graph isomorphism checks. We then proceed to explain the new method of subsumption collapsing for abstract states and we give a modified version for the traversal algorithm. This subsumption relation, along with the experimental results given in Section 4, constitute the new contributions of this paper.

3.1 Abstract State Space Traversal

Listing 1 gives the pseudo-code for exploring the abstract state space. Q is the set of all shapes and F the set of fresh, yet to be explored shapes; R is the set of rules, G the start graph, and P is the set of rule applications that were computed during exploration.

Listing 1: Algorithm for abstract state space exploration.

```
let $S := \mathsf{abstract}(G),\ Q := \emptyset,\ P := \emptyset,\ F := \{S\}$
while $F \neq \emptyset$
do choose $S \in F$      // which S is selected depends on the exploration strategy
   let $F := F \setminus \{S\}$
   for $r \in R,\ m \in \mathsf{prematch}(r,S),\ S' \in \mathsf{materialise}(m,S)$
   do let $T := \mathsf{normalise}(\mathsf{apply}(r,m,S'))$
      if $\mathsf{isFresh}(T,Q,F)$      // if $T \notin Q$
      then let $Q := Q \cup \{T\},\ F := F \cup \{T\}$
      fi
      let $P := P \cup \{S \xrightarrow{r,m} T\}$
   od
od
```

The main phases in this algorithm are:

- abstract computes the shape of a graph, as explained in the previous section.
- prematch computes non-injective morphisms of a rule r into a shape S. Such a morphism is not yet a match, because the images of r's LHS may be collector elements; in this case they have to be materialised.
- materialise creates concrete nodes and edges for the image of r in S. This is a non-deterministic step, as there may be options for choosing multiplicities for the materialised elements.
- 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.

[3]This loss of precision is intended, otherwise the abstraction would not be finite.

Each of these phases are explained in detail in [18]. For the purposes of this paper it suffices to focus on the operations at lines 3 and 7 of the algorithm given in Listing 1.

Line 3 deals with the policy for selecting a shape S from the set F of shapes to be explored. We consider two policies, namely Breadth-First Search (BFS) and Depth-First Search (DFS). When using BFS, F is implemented as a queue, whereas in DFS F is a stack. We use the term *exploration strategies* to refer to these search policies. Section 4 gives an experimental comparison on the performance of these two strategies in abstract state space exploration.

Line 7 handles the duplicate state detection mechanism. Procedure $\mathsf{isFresh}(T,Q,F)$ is responsible for checking if shape T (or an equivalent canonical representative) is already in the set Q of all explored shapes. Set Q can be quite large, so this check has to be implemented with care, since it can greatly impact performance. This is even more important when working with graph grammars; in particular, in GROOVE, states are collapsed under an isomorphic representative. Since graph isomorphism can be a rather expensive check, it cannot be performed over all elements of Q.

Listing 2 gives the algorithm for procedure isFresh, as originally described in [16]. This algorithm is based on *graph certificates*, which basically correspond to a hashing method tuned for graphs. In addition to being relatively inexpensive to compute, certificates are built in such a way that graph isomorphism implies certificate equality. The converse, however, is not true, since the certificate function may produce false positives. In the algorithm of Listing 2, certificates are used to filter elements of set Q, thus producing a much smaller set $\bar{Q} \subseteq Q$, composed only of graphs with the same certificate of T. We then proceed to check if there exists $U \in \bar{Q}$ such that T and U are isomorphic (denoted $T \sim U$). If no such U is found then we can conclude that T is fresh.

Listing 2: Algorithm for procedure $\mathsf{isFresh}(T, Q, F)$.

```
let $C := \mathsf{certificate}(T)$,  $\bar{Q} := \{U \in Q \mid C = \mathsf{certificate}(U)\}$
for $U \in \bar{Q}$
do if $T \sim U$    // if $T$ and $U$ are isomorphic
   then return false
   fi
od
return true    // we checked all candidates but none are isomorphic to $T$, so $T$ is fresh
```

The method just described works very well in practice for concrete state space exploration and since shapes also have a graph structure, we can immediately reuse the same algorithm of Listing 2 for abstract exploration. However, shapes carry additional information that is not taken into account when using only isomorphism checks. To use this additional information, the notion of *shape subsumption* was developed.

3.2 Shape Subsumption

The key insight behind the shape subsumption relation (denoted by the same symbol $\sqsubseteq$ used for multiplicity subsumption) lies in the comparison between the concretisations of shapes. Let S and T be two *isomorphic* shapes, with $\mathsf{concr}(S) \subseteq \mathsf{concr}(T)$. Since T has more concretisations than S, rule applications on T capture more behaviour than rule applications on S. In fact, all behaviour of S is subsumed by the behaviour of T, and therefore, in an abstract exploration we can discard S and only explore T. More formally, if S is subsumed by T then for all $S \xrightarrow{r} U \in P$ there exists $T \xrightarrow{r} U' \in P$ such that $U \sqsubseteq U'$, for any $r \in R$.

Subsumption is an asymmetric relation built upon isomorphism. Shape S is subsumed by shape T if: (i) S and T are isomorphic, (ii) they have the same node similarity relation, and (iii) all multiplicities in

S are subsumed by the multiplicities in T. Formally, we have the following definition, where, for $v \in N$, we write $[v]_{\simeq}$ to denote the equivalence class of v induced by $\simeq$, *i.e.*, $[v]_{\simeq} = \{w \in N \mid v \simeq w\}$.

Definition 6 (Shape subsumption) *Given two shapes S and T, S is subsumed by T, denoted $S \sqsubseteq T$, if:*

- *there exists an isomorphism $\varphi : G_S \sim G_T$ between the graph structures of the shapes;*
- *for any $\langle v, w\rangle \in \simeq_S$, $\langle \varphi(v), \varphi(w)\rangle \in \simeq_T$;*
- $\mathsf{mult}^{\mathsf{n}}_S(v) \sqsubseteq \mathsf{mult}^{\mathsf{n}}_T(\varphi(v))$*, for all $v \in N_S$; and*
- $\mathsf{mult}^{\mathsf{o}}_S(v,l,[w]_{\simeq_S}) \sqsubseteq \mathsf{mult}^{\mathsf{o}}_T(\varphi(v),l,[\varphi(w)]_{\simeq_T})$ *and* $\mathsf{mult}^{\mathsf{i}}_S(v,l,[w]_{\simeq_S}) \sqsubseteq \mathsf{mult}^{\mathsf{i}}_T(\varphi(v),l,[\varphi(w)]_{\simeq_T})$, *for all $\langle v,l,w\rangle \in E_S$.* ◀

Note that $S \sqsubseteq T$ implies $\mathsf{concr}(S) \subseteq \mathsf{concr}(T)$. As a simple example, take S as a shape containing only one node of multiplicity $\mathbf{2}^+$ and no edges. Then take T also as a shape with a single node, but with multiplicity $\mathbf{1}^+$. From the definition, we have that $S \sqsubseteq T$. Set $\mathsf{concr}(S)$ contains graphs with two or more nodes, whereas set $\mathsf{concr}(T)$ has one more element, namely the graph with just one node. Hence, $\mathsf{concr}(S) \subseteq \mathsf{concr}(T)$.

Shapes that subsume one another, *i.e.*, shapes that are isomorphic and have the same multiplicities for all elements, are called *strictly isomorphic*. From the above it follows that strictly isomorphic shapes have the same concretisations (see the accompanying technical report of [3] for the proof).

Listing 3 shows the modified algorithm for procedure isFresh, with shape subsumption checks incorporated. Given a new shape T that must be tested for freshness, we begin as before, by constructing set $\bar{Q}$, consisting of the shapes in Q with the same certificate as T. In addition, we initialise an auxiliary set B, to store shapes from $\bar{Q}$ that were identified as subsumed by T. Since subsumption is an asymmetric relation, we must check it in both directions (lines 3 and 5 of the algorithm). Note, however, that these two subsumption checks do not require two isomorphism checks, since we can first look for an isomorphism between T and U (the potentially most expensive operation) and then proceed to check both subsumptions using the same isomorphism.

Listing 3: Algorithm for procedure $\mathsf{isFresh}(T, Q, F)$ with subsumption check.

```
1  let $C := \mathsf{certificate}(T)$,  $\bar{Q} := \{U \in Q \mid C = \mathsf{certificate}(U)\}$,  $B := \emptyset$
2  for $U \in \bar{Q}$
3  do if $T \sqsubseteq U$     // if $T$ is subsumed by $U$
4     then return false
5     else if $U \sqsubseteq T$     // if $U$ is subsumed by $T$
6          then let $B := B \cup \{U\}$     // mark $U$ as subsumed
7          fi
8     fi
9  od
10 $F := F \setminus B$     // remove the states marked as subsumed from the set of states to be explored
11 return true
```

An interesting aspect of the new isFresh procedure is that it can now modify the set F of shapes to be explored. If there exists $U \in \bar{Q}$ such that $T \sqsubseteq U$, then T is not fresh and can be discarded as before (lines 3 and 4 of Listing 3). However, if we discover that $U \sqsubseteq T$, then not only we know that T is fresh, but also that U should not be explored, since all its behaviour is subsumed by T. We then mark U as subsumed by adding it to set B (line 6) and continue looking for other subsumed shapes in $\bar{Q}$. At the end of the procedure we remove all shapes marked as subsumed from F (line 10), thus trimming the search space.

4 Experiments and Results

The theory of neighbourhood abstraction ensures that the number of shapes for any graph grammar is finite, and therefore that the abstract state space is also finite [3]. However, the theoretical upper bound of the abstract state space size is still quite large, meaning that in practice we have to optimise the state space traversal in order to implement an efficient tool. In [18] we described the major aspects of the implementation of neighbourhood abstraction in GROOVE (though without shape subsumption), and reported a few experiments. In this section we present more of such experiments, that illustrate: (i) the upper bounds on abstract state space sizes that are reachable in practice, (ii) the performance gains that are obtained with the shape subsumption technique presented in Section 3, and (iii) how different exploration strategies perform in the abstract setting.

For our experiments we collected 8 graph grammars, from various problem domains. Some of these problems can also be solved with other abstractions, but they are usually tuned with certain characteristics from the domain at hand[4]. Our abstraction extension for GROOVE, on the other hand, is generic, in the sense that the concept of neighbourhood equivalence is always applicable, but perhaps with varying performance. Here is a list with a short description of the grammars used.

- **linked-list**: a grammar modelling operations on a single-linked list structure. Elements can always be appended at the end of the list, which can thus grow unbounded.
- **circ-buf-0**: a grammar modelling a circular buffer structure with an unbounded number of cells. Cell usage is marked with special labels, without reference to stored objects.
- **circ-buf-1**: a variant of the circular buffer where the stored objects are explicitly represented.
- **euler-0**: a grammar that can construct Euler's walks of arbitrary size. Adapted from the classical Königsberg bridges problem from graph theory.
- **euler-1**: a variant of the Euler grammar without explicitly representing connecting bridges.
- **firewall-[2-6]**: the grammar of our firewall example. Network structure is fixed, while packages are collapsed by the abstraction. Instances vary on the number of locations: from 2 to 6.
- **firewall-6-F**: variant of the firewall grammar with a network of six fully connected locations.
- **car-platoon**: grammar simulating a wireless communication protocol between cars, for establishing platoons in highways. Cars can enter and leave a platoon at any time, which leads to an exponential growth on the number of possible configurations.

Table 1 gives all the figures on state space sizes for the grammars listed above. Numbers for the BFS and DFS exploration strategies are grouped per grammar, to ease the comparison between the two. State space sizes are divided in two groups of values: the number of explored states, *i.e.*, the number of shapes produced, and the number of transitions between states, *i.e.*, the count of rule applications. State and transition counts in Table 1 are broken down in five and three types, respectively.

The following five types of state count are given in columns 3-7 of Table 1.

- **Maximum** is the upper bound on the number of abstract states of the grammar. This number is obtained by exploring the state space without shape subsumption, *i.e.*, by running the exploration algorithm of Listing 1 with the original isFresh procedure of Listing 2. Thus, these are the figures that would have been reported by the prior implementation of [18]. Empty entries in this column

[4]Shape analysis, for example, is designed to work on heap pointer structures, which are deterministic graphs. In our setting, this correspond to having shapes with all outgoing edge multiplicities limited to **1**.

Table 1: State space sizes for the explorations performed with different graph grammars.

Grammar	Strat.	States					Transitions		
		Maximum	Generated	Subsumed	Relevant	Discarded	Maximum	Generated	Relevant
linked-list	BFS	9	9	3	6	0	17	17	11
	DFS		8	2	6	1		14	11
circ-buf-0	BFS	54	31	5	26	3	130	65	59
	DFS		39	13	26	2		88	59
circ-buf-1	BFS	57	33	16	17	0	182	100	40
	DFS		30	13	17	2		89	40
euler-0	BFS	878	248	96	152	54	10,448	1,356	584
	DFS		213	61	152	28		1,286	618
euler-1	BFS	14	14	4	10	0	23	23	15
	DFS		13	3	10	2		19	15
firewall-2	BFS	125	98	90	8	36	875	409	37
	DFS		50	42	8	15		207	37
firewall-3	BFS	1,625	991	971	20	549	19,825	5,021	121
	DFS		232	212	20	102		1,314	121
firewall-4	BFS	4,875	2,356	2,326	30	1,487	83,850	13,577	203
	DFS		427	397	30	212		2,959	203
firewall-5	BFS		14,878	14,818	60	10,549		93,549	459
	DFS		1,201	1,141	60	654		10,421	459
firewall-6	BFS		25,251	25,171	80	18,373		187,126	643
	DFS		1,783	1,703	80	1,007		18,485	643
firewall-6-F	BFS		183,478	182,966	512	147,028		1,409,451	8,711
	DFS		5,930	5,418	512	3,003		93,087	8,711
car-platoon	DFS	Out of memory after exploring 445,439 states and 8,484,600 transitions							

for the larger cases of the firewall grammar indicate that the upper bound could not be computed: these runs timed out after several hours of execution, due to state space explosion. Note that we give only one maximum state count for both BFS and DFS strategies. The reason is that all states are explored when subsumption is off, and therefore the maximum state count is the same, regardless of the strategy used. The upper bound provided by this column gives an interesting basis of comparison when analysing the reduction obtained with subsumption.

- **Generated** is the number of states produced during exploration using shape subsumption, *i.e.*, the exploration algorithm of Listing 1 was run with the new isFresh procedure of Listing 3. Numbers in this column correspond to the size of set Q, *i.e.*, the number of states that were added to the set of all explored states (line 8 in Listing 1). When comparing the number of generated states against the maximum upper bound we can see the reduction given by subsumption. Take, for example, the *firewall-4* line, where the number of generated states using DFS with subsumption is an order of magnitude smaller than the maximum upper bound obtained without subsumption. The gain provided by subsumption can also be seen for the larger cases of the firewall grammar: runs that timed-out without subsumption can now be finished when we turn it on.
- **Subsumed** is the number of states generated that were later marked as subsumed by another state. They correspond to states that are added to set B at line 6 of Listing 3.
- **Relevant** is the number of states that were never marked as subsumed during the exploration. This column corresponds to the **Generated** column minus the **Subsumed** one. The closer the number of generated states gets to the relevant state count the better. A perfect exploration method would generate only the relevant abstract states, since they are sufficient to cover all concrete behaviour.
- **Discarded** is the number of states that were marked as subsumed and were removed from the set of states to be explored. This number corresponds to the sum of all states removed from F at line 10 of Listing 3.

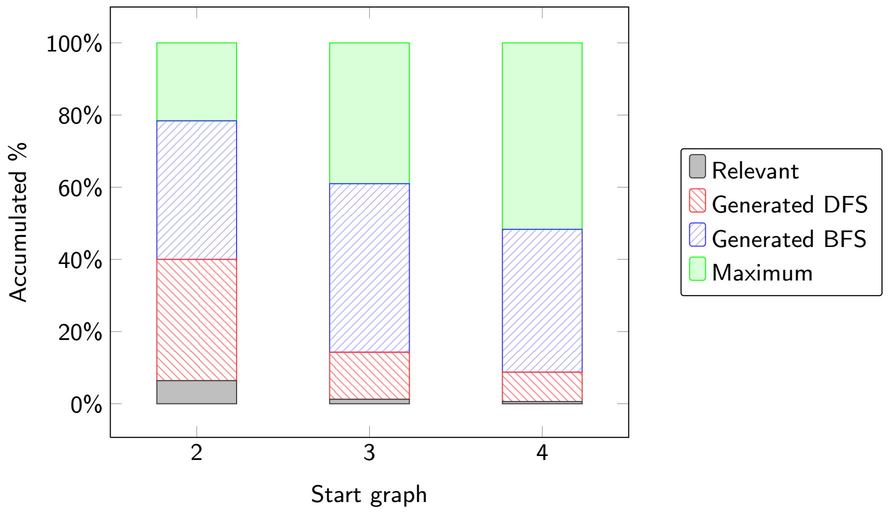


Figure 4: Accumulated percentages of the number of states relative to the maximum state space size, for the firewall grammar instances with a known maximum state count.

The remainder columns of Table 1 give the number of transitions outgoing from the states in the associated state count columns.

When comparing the figures in Table 1 for BFS and DFS exploration, it is clear that DFS gives a much better performance. The DFS generated state count is smaller than the BFS count in all but one test (*circ-buf-0*), and as we move to grammars with larger state spaces the advantage increases greatly, until reaching two orders of magnitude for the *firewall-6-F* case. The reason for this performance difference between BFS and DFS is simple. Usually, the more a shape is transformed by subsequent rule applications the more abstract it becomes, until it reaches a fix-point, *i.e.*, further rule applications yield the same shape again. These more abstract shapes capture more concrete behaviour and thus can subsume other shapes in the state space. As a rule-of-thumb, more abstract shapes are more likely to be part of the set of relevant states, and since they are only discovered after a succession of rule applications, these relevant states are deeper in the state space. Therefore, DFS is more likely to reach these states first. This fact can be seen from the numbers in the **Discarded** column: BFS generates a lot of states that are later discarded. This is wasted effort: in BFS a state is produced and added to sets Q and F but it is very likely that later it is going to be removed from F (while remaining in Q). On the other hand, since DFS already found more abstract shapes, it is more probable that the search will immediately throw a new state away, without storing it on Q, since the new state will be subsumed by some other state already in Q.

From the discussion above, one may wonder why shapes that were marked as subsumed are kept in set Q. The reason is that removing states from Q could leave "dangling" transitions in the set of generated transitions P. A possible solution could be the following. For $S \in Q$ and T fresh, if $S \sqsubseteq T$ then we should take all transitions in P with S as a source or target and replace S by T. This on-the-fly state space collapsing under subsumption is not provided by the current implementation, but the tool offers a simpler option: *reachability mode*. In this mode we are only interested in the shapes that are reachable in the abstract state space, and thus there is no need to store the transitions, *i.e.*, set P is kept empty. In this case there is no danger of having "dangling" transitions and we can remove subsumed shapes from Q, thus decreasing memory usage. Reachability mode was a late addition to the implementation and as such its experimental analysis is left as future work.

Figure 4 gives a visual aid for the comparison between BFS and DFS, for the firewall grammar

Table 2: Running time and memory consumption, with and without state subsumption checks.

Grammar	Time (s)				Memory (MB)			
	Subsump. OFF		Subsump. ON		Subsump. OFF		Subsump. ON	
	BFS	DFS	BFS	DFS	BFS	DFS	BFS	DFS
linked-list	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
circ-buf-0	< 1	< 1	< 1	< 1	2	2	2	2
circ-buf-1	< 1	< 1	< 1	< 1	2	2	2	2
euler-0	61	47	2	2	57	57	12	11
euler-1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
firewall-2	2	2	< 1	< 1	6	6	4	2
firewall-3	177	157	5	2	110	110	49	12
firewall-4	4,448	3,824	16	3	432	432	136	25
firewall-5			347	10			1,054	85
firewall-6			1,679	16			2,001	143
firewall-6-F			3,732	55			14,277	556

instances with a known maximum state count. The figure shows a stacked bar chart with the accumulated percentages of the number of states relative to the maximum state space size. From this chart we see that the percentage of states generated with DFS decreases as the start graph size increases. On the other hand, the percentage of states generated with BFS remains roughly the same, at around 40% the maximum (this can be seen from the interval sizes of the generated BFS bars in Figure 4).

Other metrics that must be analysed in a performance evaluation are running time and memory consumption. Results for these measurements are given in Table 2, for runs with and without state subsumption checks. The experiments were performed in a machine with a Intel Xeon X5365 CPU running at 3 GHz and a total 32 GB of RAM. Blank entries indicate timed-out executions. From the numbers in Table 2 we see the performance improvement given by subsumption: running times for the firewall grammar are two orders of magnitude smaller when subsumption is used, and memory consumption is also reduced. When comparing the running times for BFS and DFS, we see that both strategies have a similar performance when subsumption is not used but when it is turned on, DFS is far more efficient than BFS, both in execution time and memory consumption. This performance figures are directly related to the number of states generated by each strategy: DFS produces far fewer states than BFS, which translates to a large performance gain. This can be confirmed visually with the chart in Figure 5, where the running times for the firewall grammar are plotted against start graph sizes. Clearly, DFS has a much more tamed growth (note that the time axis is in a logarithmic scale).

5 Related Work

Abstraction is an essential ingredient in nearly all methods for system analysis and verification and as such there is a vast body of work describing the use of abstractions in different domains. In this section we give a (non-exhaustive) discussion on related work that involves state space traversal and graphs.

In [11], Holte *et al.* tackle the area of problem solving in artificial intelligence, which boils down to finding the shortest path between a start and a goal state. There is a relation of opposition between solution quality (the path length) and search effort (states traversed) and many heuristics can be used to guide the search. The authors define a so-called “explicit graph notation”, where the state space is represented by a labelled transition system (LTS), and they proceed to define abstraction algorithms that can be used to speed-up the search. One of such algorithms, called STAR, works by building state classes that are connected up to a certain abstraction radius. Despite having many similar concepts with our work, the abstractions used by Holte are not state abstractions; they operate on the LTS level and not on the state representation. Furthermore, the concrete state spaces considered in [11] are always finite.

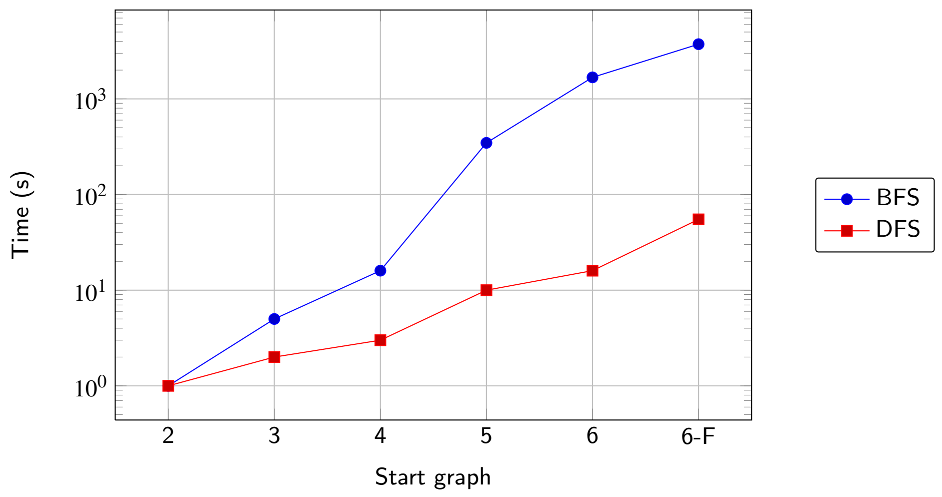


Figure 5: Running time (with subsumption on) versus start graph size for the firewall grammar.

In [6], Edelkamp *et al.* consider the problem of partial analysis/exploration of the state space of graph transformation systems. As in the work of Holte *et al.*, this amounts to a guided search over the concrete state space where abstraction can be used as an heuristic. Properties of interest for the analysis usually encompass existential checks for graph structures; *e.g.*, is a graph with a certain node and edge configuration reachable from the start state? Any abstraction that preserves reachability of the goal state in the abstract state space can be used to define an heuristic for the guided search in the concrete level. Since our neighbourhood abstraction preserves reachability, it could in principle be used as the abstraction mechanism for an heuristic search. However, performance may be an issue, since computing the transitions of an abstract state is a rather expensive operation.

Concerning the verification of infinite-state graph transformation systems, König *et al.* have an extensive corpus of work, starting with [2]. Given a graph grammar their analysis technique extracts an *approximated unfolding*; a finite structure (called *Petri graph*) that is composed of a hyper-graph and a Petri net. The Petri graph captures all structure that can occur in the reachable graphs of the system, and dependencies for rule applications are recorded by the Petri net transitions. The final Petri graph obtained is an over-approximation that can be used to check safety properties in the original system. If a spurious counter-example is introduced by the over-approximation, the abstraction can be incrementally refined [12]. These techniques are implemented in the tool AUGUR which is now in its second version [13]. An experimental comparison between this tool and our implementation is considered as future work.

6 Conclusions and Future Work

In this paper we present an abstraction technique for the exploration of graph transformation systems with infinite state spaces. We explain the main points of neighbourhood abstraction as implemented in GROOVE and we propose a new method for state collapsing, based on the concept of shape subsumption. Experimental results show that subsumption gives a significant reduction on the number of states that have to be explored, thus improving both the running time and memory consumption of the tool. Furthermore, the experiments also show that the choice of the exploration strategy has a heavy influence on performance, with DFS giving much better results.

We see the results presented in this paper as an important achievement over the original implementation of abstraction in GROOVE. As any tool developer would know, performance improvements in programs that deal with highly combinatorial problems such as state space exploration usually involve a painstaking cycle of refactorings, experimentation and fine-tuning. Our case was no different, where the original abstraction code had to be rewritten from scratch in order to accommodate shape subsumption. A further improvement over the code from [18] is that rules with NACs (negative application conditions) are now also supported, which increases rule expressivity.

There are many directions where the current research/tool can be extended. Aside from the usual points, such as additional experimentation with more test cases and comparison with other tools, we consider the following items as future work.

- **Stronger notion of subsumption.** The subsumption relation presented here depends on the existence of an isomorphism between two shapes. This dependence can be weakened by requiring only the existence of an embedding morphism between the shapes, which is not an isomorphism but instead an injective sub-graph morphism, similar to a rule match. This weakening of the subsumption pre-condition makes the relation stronger, and thus should lead to further reductions of the state space. This new relation, however, requires additional refactoring of the code, since we can no longer re-use the isomorphism checking package from GROOVE.
- **More expressive notions of abstraction.** While neighbourhood abstraction can be used for many different classes of problems, it does not fare very well when some structural properties should be preserved by the abstraction. It cannot, for example, represent connectivity information between nodes. When the abstraction does not limit the possible concrete structures that can be generated, all cases have to be considered and this leads to a blow-up in the abstract state space size that can cripple performance. We can see this from the results for the *car-platoon* grammar in Table 1: the number of states in the state space is too large, and execution was aborted due to an out-of-memory error. To tackle these problems, other notions of abstraction are thus in order. We are currently working on the theory for a pattern based abstraction, a method that will allow certain graph structures of interest to be preserved in the abstract domain.

Availability. The current abstraction extension described in this paper is implemented in GROOVE version 4.4.6, available at `http://groove.cs.utwente.nl`. The grammars for the experiments described in Section 4 along with the results obtained can also be downloaded at the same address.

References

[1] C. Baier & J. P. Katoen (2008): *Principles of Model Checking*. MIT Press, New York.

[2] P. Baldan, A. Corradini & B. König (2001): *A Static Analysis Technique for Graph Transformation Systems*. In: International Conference on Concurrency Theory (CONCUR), *LNCS* 2154, Springer, pp. 381–395. Available at `http://dx.doi.org/10.1007/3-540-44685-0_26`.

[3] J. Bauer, I. B. Boneva, M. E. Kurban & A. Rensink (2008): *A Modal-Logic Based Graph Abstraction*. In Ehrig et al. [8], pp. 321–335. Available at `http://dx.doi.org/10.1007/978-3-540-87405-8_22`.

[4] J. Bauer & R. Wilhelm (2007): *Static Analysis of Dynamic Communication Systems by Partner Abstraction*. In: Static Analysis Symposium (SAS), *LNCS* 4634, Springer, pp. 249–264. Available at `http://dx.doi.org/10.1007/978-3-540-74061-2_16`.

[5] P. Cousot & R. Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: Principles of Programming Languages (POPL), ACM, pp. 238–252. Available at `http://doi.acm.org/10.1145/512950.512973`.

[6] S. Edelkamp, S. Jabbar & A. Lluch-Lafuente (2006): *Heuristic Search for the Analysis of Graph Transition Systems*. In: *International Conference on Graph Transformations (ICGT)*, *LNCS* 4178, Springer, pp. 414–429. Available at `http://dx.doi.org/10.1007/11841883_29`.

[7] H. Ehrig, G. Engels, H.-J. Kreowski & G. Rozenberg, editors (1999): *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*. World Scientific Publishing Co.

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

[9] A. Ghamarian, M. de Mol, A. Rensink, E. Zambon & M. Zimakova (2012): *Modelling and Analysis Using* GROOVE. *International Journal on Software Tools for Technology Transfer (STTT)* 14(1), pp. 15–40. Available at `http://dx.doi.org/10.1007/s10009-011-0186-x`.

[10] P. Godefroid (1996): *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer Verlag, New York.

[11] R. Holte, T. Mkadmi, R. Zimmer & A. MacDonald (1996): *Speeding up Problem Solving by Abstraction: A Graph Oriented Approach*. *Artificial Intelligence* 85(1-2), pp. 321–361. Available at `http://dx.doi.org/10.1016/0004-3702(95)00111-5`.

[12] B. König & V. Kozioura (2006): *Counterexample-Guided Abstraction Refinement for the Analysis of Graph Transformation Systems*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, *LNCS* 3920, Springer, pp. 197–211. Available at `http://dx.doi.org/10.1007/11691372_13`.

[13] B. König & V. Kozioura (2008): *Augur 2 - A New Version of a Tool for the Analysis of Graph Transformation Systems*. *Electronic Notes in Theoretical Computer Science (ENTCS)* 211, pp. 201–210. Available at `http://dx.doi.org/10.1016/j.entcs.2008.04.042`.

[14] A. Rensink (2004): *The* GROOVE *Simulator: A Tool for State Space Generation*. In: *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, *LNCS* 3062, Springer, pp. 479–485. Available at `http://dx.doi.org/10.1007/978-3-540-25959-6_40`.

[15] A. Rensink (2004): *Canonical Graph Shapes*. In: *European Symposium on Programming (ESOP)*, *LNCS* 2986, Springer, pp. 401–415. Available at `http://dx.doi.org/10.1007/978-3-540-24725-8_28`.

[16] A. Rensink (2006): *Isomorphism Checking in* GROOVE. In: *International Workshop on Graph-Based Tools (GraBaTs)*, *Electronic Communications of the EASST* 1, European Association of Software Science and Technology. Available at `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/77`.

[17] A. Rensink & D. Distefano (2006): *Abstract Graph Transformation*. In: *Workshop on Software Verification and Validation (SVV)*, *Electronic Notes in Theoretical Computer Science (ENTCS)* 157, pp. 39–59. Available at `http://dx.doi.org/10.1016/j.entcs.2006.01.022`.

[18] A. Rensink & E. Zambon (2010): *Neighbourhood Abstraction in* GROOVE. In: *International Workshop on Graph-Based Tools (GraBaTs)*, *Electronic Communications of the EASST* 32, European Association of Software Science and Technology. Available at `http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/501`.

[19] S. Rieger & T. Noll (2008): *Abstracting Complex Data Structures by Hyperedge Replacement*. In Ehrig et al. [8], pp. 69–83. Available at `http://dx.doi.org/10.1007/978-3-540-87405-8_6`.

[20] S. Sagiv, T. W. Reps & R. Wilhelm (2002): *Parametric Shape Analysis via 3-valued Logic*. *Transactions on Programming Languages and Systems (ToPLaS)* 24(3), pp. 217–298. Available at `http://doi.acm.org/10.1145/514188.514190`.

[21] M. Saksena, O. Wibling & B. Jonsson (2008): *Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, *LNCS* 4963, Springer, pp. 18–32. Available at `http://dx.doi.org/10.1007/978-3-540-78800-3_3`.