Efficient Symbolic Implementation of Graph Automata with Applications to Invariant Checking[⋆]

Christoph Blume, H. J. Sander Bruggink, Dominik Engelke, and Barbara König

Universität Duisburg-Essen, Germany
{christoph.blume,sander.bruggink,barbara_koenig}@uni-due.de

Abstract. We introduce graph automata as a more automata-theoretic view on (bounded) automaton functors and we present how automaton-based techniques can be used for invariant checking in graph transformation systems. Since earlier related work on graph automata suffered from the explosion of the size of the automata and the need of approximations due to the non-determinism of the automata, we here employ symbolic BDD-based techniques and recent antichain algorithms for language inclusion to overcome these issues. We have implemented techniques for generating, manipulating and analyzing graph automata and perform an experimental evaluation.

1 Introduction

Regular languages and (word) automata are the cornerstone of several verification techniques (for example [9]). Similarly, tree automata [11] have been used in regular model-checking [8]. Challenges in the analysis of dynamic graph-like structures, such as pointer structures on the heap, object graphs or evolving networks, naturally lead to the question whether graph languages and graph automata can serve the same purpose. There is indeed an established theory of recognizable graph languages by Courcelle [12], although substantial work needs to be done before this theory can be put to good use in complex verification scenarios.

In order to close this gap, we here give a very concrete variant of graph automata accepting a subclass of the recognizable graph languages à la Courcelle. Furthermore we reformulate our own earlier work on invariant checking [7] in this setting. However, our main motivation is to fight state explosion, which is a major problem when working with graph automata. Graph automata cannot input all graphs but only graphs up to a certain width (in our case: path width), which is a restriction on the interface size of the alphabet of "building blocks" of graphs. However the size of the automaton typically grows exponentially when this bound is raised. This is a major problem that forced earlier work such as [16], based on the algorithms in [14], [20] and [13] to restrict to very small interface

[⋆] This work is supported by the DFG-project GaReV.

sizes. Recent work abstains from a representation of the automaton, but pursues a game-based approach, obtaining much better runtime results [18].

However, all these approaches have a different focus than ours, in that they concentrate on solving the membership problem: given a description of the language (often specified by a formula in monadic second-order graph logic) and a graph, check whether the graph is in the language. Courcelle's theorem shows that for a fixed formula this can be done in linear time for graphs of bounded treewidth (or path width). However the large constants involved lead to severe efficiency problems when the automata are represented directly.

Here we are less interested in solving the membership problem: with the applications that we have in mind we are interested in designing an automaton tool suite that treats automata as representatives of languages that can be suitably manipulated and analyzed. However, we have to face the same problem as the other approaches: the sheer size of the automata involved. Hence we are using symbolic BDD-based techniques to represent the set of states and the transition function, which enable us to generate non-deterministic automata for large interface sizes. To avoid determinization, our earlier work [7] used an approximation, but this approach will not be used in this paper. In order to perform useful analyses, needed for instance for invariant checking as mentioned above, we have to solve the language inclusion problem, which is PSPACE-complete. Our new approach uses recent methods based on antichains as introduced in [21, 1]. We have implemented our techniques and we perform an extensive experimental evaluation, which shows a clear improvement over earlier work.

The structure of the paper is as follows. In Sect. 2 we will introduce preliminary definitions such as cospans, hypergraphs and binary decision diagrams. In Sect. 3 we will take a look at graph automata and the connection between them and automaton functors of bounded size. Then in Sect. 4 we will show how techniques for solving the language inclusion problem can be used to perform invariant checking and in Sect. 5 we will present implementation details about the *Raven* tool suite which implements language inclusion algorithms for invariant checking. Furthermore we will present results about our case studies. Finally, we will conclude in Sect. 6.

2 Preliminaries

By $\mathbb{N}_k$ we denote the set $\{1, \ldots, k\}$. The set of finite sequences over a set A is denoted by A^*. If $f\colon A \to B$ is a function from A to B, we will implicitly extend it to subsets and sequences; for $A' \subseteq A$ and $\boldsymbol{a} = a_1 \ldots a_n \in A^*$: $f(A') = \{f(a) \mid a \in A'\}$ and $f(\boldsymbol{a}) = f(a_1) \ldots f(a_n)$. By $|\boldsymbol{a}|$ we denote the length of $\boldsymbol{a} \in A^*$. By $\wp(A)$ we denote the powerset of A.

Categories and Cospans. We presuppose a basic knowledge of category theory. For an arrow f from A to B we write $f\colon A \to B$ and define $dom(f) = A$ and $cod(f) = B$. For arrows $f\colon A \to B$ and $g\colon B \to C$, the composition of f and g is denoted $(f\ ;\ g)\colon A \to C$. The category **Rel** has sets as objects and relations as arrows.

Let $\mathbf{C}$ be a category in which all pushouts exist. A cospan in $\mathbf{C}$ is a pair $c = (c_L, c_R)$ of $\mathbf{C}$-arrows $J \xrightarrow{c_L} G \xleftarrow{c_R} K$. Two cospans c, d are isomorphic if their middle objects are isomorphic (such that the isomorphism commutes with the component morphisms of the cospan). In this case we write $c \simeq d$. Isomorphism classes of cospans are the arrows of so-called cospan categories. That is, for a category $\mathbf{C}$ with pushouts, the category $\mathit{Cospan}(\mathbf{C})$ has the same objects as $\mathbf{C}$. The isomorphism class of a cospan $c\colon J \xrightarrow{c_L} G \xleftarrow{c_R} K$ in $\mathbf{C}$ is an arrow from J to K in $\mathit{Cospan}(\mathbf{C})$ and will be denoted by $c\colon J \leftrightarrow K$. Composition of two cospans $(c_L, c_R), (d_L, d_R)$ is computed by taking the pushout of the arrows c_R and d_L. A cospan is called *output linear* if the right leg of the cospan is a monomorphism.

Graphs and Output Linear Cospans. Let Λ be a set of labels and let $ar\colon \Lambda \to \mathbb{N}$ be the function that maps each alphabet symbol to its arity.

A *hypergraph* over a set of labels Λ (in the following also simply called *graph*) is a structure $G = (V, E, att, lab)$, where V is a finite set of nodes, E is a finite set of edges, $att\colon E \to V^*$ maps each edge to a finite sequence of nodes attached to it, such that $|att(e)| = ar(lab(e))$, and $lab\colon E \to \Lambda$ assigns a label to each edge. A *discrete graph* is a graph without edges; the discrete graph with node set $\mathbb{N}_k$ is denoted by D_k. We denote the *empty graph* by $\emptyset$ instead of D_0.

A graph morphism is a structure preserving map between two graphs. The category of graphs and graph morphisms is denoted by **Graph**. Recall, that the *monomorphisms* (monos) and *epimorphisms* (epis) of the category **Graph** are the injective and surjective graph morphisms, respectively.

A cospan $J \xrightarrow{c_L} G \xleftarrow{c_R} K$ (over a set of labels Λ) in **Graph** can be viewed as a graph (G over Λ) with two interfaces (J and K), called the *inner interface* and *outer interface* respectively. Informally said, only elements of G which are in the image of one of the interfaces can be "touched". By $[G]$ we denote the trivial cospan $\emptyset \to G \leftarrow \emptyset$, the graph G with two empty interfaces.

The *category of output linear cospans* $\mathbf{OLCG}_n$ has discrete graphs (of size at most n) as objects and output linear cospans of graphs with discrete interfaces (of size at most n) as arrows. Note that the middle objects of the cospans of the category $\mathbf{OLCG}_n$ can still be arbitrary graphs. The idea for using this category is that we want to be able to fuse nodes via cospan composition, but we want to avoid that nodes of the middle graph are shared in the outer interface.

Binary Decision Diagrams. A *binary decision diagram* (BDD) is a rooted, directed, acyclic graph which serves as a representation of a boolean function. Every BDD has two distinguished terminal nodes, called *one* and *zero*, representing the logical constants *true* and *false*. The inner nodes are labeled by the variables of the boolean formula represented by the BDD, such that on each path from the root to the terminal nodes, every variable of the boolean formula occurs at most once. Each inner node has exactly two distinguished outgoing edges, called *high* and *low*, which represent the case that the variable of the inner node has been set to *true* or *false* respectively. A boolean formula $f(x_1, \ldots, x_n)$ can be evaluated by following the path from the root node to a terminal node.

We will use a special class of BDDs, called *reduced and ordered* BDDs (ROBDDs), in which the order of the variables occuring in the BDD is fixed and redundancy is avoided, i.e. if both child nodes of a parent node are identical, the parent node is dropped from the BDD and isomorphic parts of the BDD are merged. The great advantage of ROBDDs is that each boolean formula can be uniquely represented by an ROBDD (if the order of the variables is fixed). For a detailed introduction of these BDDs see [2].

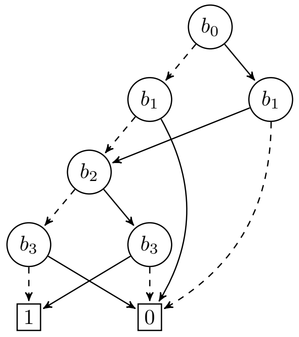


Fig. 1. BDD for the set $\{0000, 0011, 1100, 1111\}$

As an example, we consider the following set of 4-bit vectors: $\{0000, 0011, 1100, 1111\}$. We assume that the bits of the bit vectors are numbered from b_0 to b_3 with b_0 the least significant bit. The ROBDD representing this set of bit vectors is shown in Fig. 1. Variables are depicted as rounded nodes, terminals as rectangular nodes. The *high* and *low* edges are depicted as solid and dashed lines respectively.

3 Bounded Graph Automata

Recognizable graph languages are a generalization of regular (word) languages to graph languages which were first investigated by Courcelle [3, 12]. In this section we define bounded graph automata, which accept a subclass of the recognizable graph languages due to the bound. Similar to word languages we define graph languages based on an alphabet. Each letter of the alphabet represents an output linear cospan such that the concatenation of these letters (or cospans respectively) yields a graph (seen as a cospan with empty interfaces).

Let $n \in \mathbb{N}$ and a *doubly-ranked alphabet* $\Sigma = (\Sigma_{i,j})_{i,j \leq n}$ be given. The set of *(doubly-ranked) sequences* $S_\Sigma = (S_{i,j})_{i,j \leq n}$ over a doubly-ranked alphabet Σ is defined inductively:

- for every $i \leq n$ the *empty sequence* ε_i is in $S_{i,i}$
- for every $i, j \leq n$ every letter $\sigma \in \Sigma_{i,j}$ is in $S_{i,j}$
- for every $i, j, k \leq n$ and for every $\boldsymbol{\sigma} \in S_{i,j}$, $\boldsymbol{\sigma'} \in S_{j,k}$ the *concatenation* $\boldsymbol{\sigma} ; \boldsymbol{\sigma'}$ of $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma'}$ is in $S_{i,k}$

The *width* of a sequence is the maximum rank of its letters. We will also write S instead of S_Σ if the underlying alphabet is clear from the context.

Let Λ be a set of labels. By $\Gamma(\Lambda)$ we denote the doubly-ranked alphabet containing the following letters:

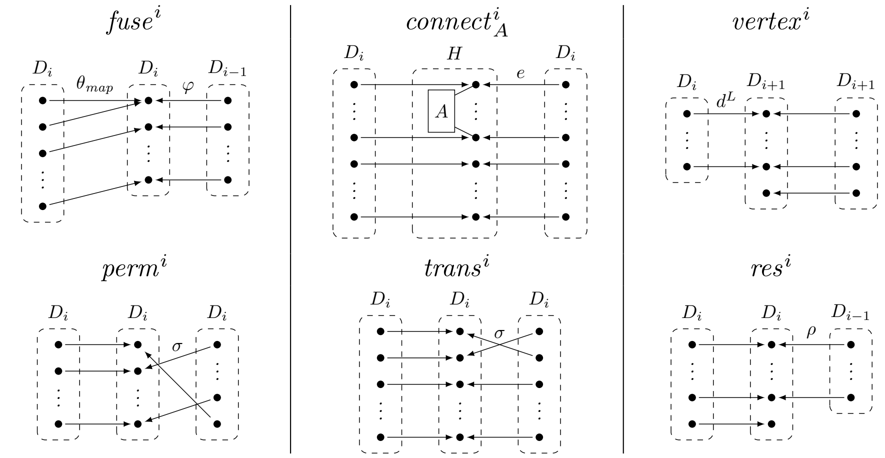
Letter:	$connect^i_A$	$fuse^i$	$perm^i$	res^i	$trans^i$	$vertex^i$
Type:	(i,i)	$(i,i-1)$	(i,i)	$(i,i-1)$	(i,i)	$(i,i+1)$
Constraint:	$A \in \Lambda$, $ar(A) \leq i$	$i \geq 2$	$i \geq 3$	$i \geq 1$	$i \geq 2$	$-$

The meaning of these letters is given by the evaluation function defined below. Note that *res* is a restriction of the interface, *perm* permutes the interface and *trans* transposes the first two interface nodes. Due to the fact that for two elements permutation and transposition are identical operations the constraint of the letter $perm^n$ is $n \geq 3$.

Now we define an evaluation function which maps each letter of the alphabet $\Gamma(\Lambda)$ to an output linear cospan.

Definition 1 (Evaluation function). *Let Λ be a set of labels.*

(i) The evaluation function $\eta \colon \Gamma(\Lambda) \to \mathbf{OLCG}_i$ *maps each letter to an output linear cospan as shown below:*



(ii) The extended evaluation function $\hat{\eta} \colon S_{\Gamma(\Lambda)} \to \mathbf{OLCG}_i$ *is defined as*

$$\hat{\eta}(\boldsymbol{\sigma}) = \begin{cases} D_j \to D_j \leftarrow D_j, & \textit{if } \boldsymbol{\sigma} = \varepsilon_j \in S_{j,j} \\ \eta(\sigma), & \textit{if } \boldsymbol{\sigma} = \sigma \in \Gamma(\Lambda) \\ \hat{\eta}(\boldsymbol{\sigma}_1) \mathbin{;} \hat{\eta}(\boldsymbol{\sigma}_2), & \textit{if } \boldsymbol{\sigma} = \boldsymbol{\sigma}_1 \mathbin{;} \boldsymbol{\sigma}_2 \end{cases}$$

We call the cospans which correspond to the six letters above *atomic cospans.*

Let c be an output-linear cospan. The *width* of c is the minimal width of all $\boldsymbol{\sigma}$ such that $\hat{\eta}(\boldsymbol{\sigma}) = c$.

The following lemma shows that every graph (seen as an output linear cospan with two empty interfaces) can be constructed by the alphabet $\Gamma(\Lambda)$. Hence, we will restrict ourselves to this alphabet in the following:

Lemma 1 ([7]). *Let c be an output linear cospan over Λ. Then it holds that:*

1. *c can be constructed by a sequence $c_1, \ldots, c_m$ of atomic cospans, i.e. c can be obtained as the decomposition $c = c_1 \mathbin{;} \ldots \mathbin{;} c_m$.*
2. *There exists a sequence $\boldsymbol{\sigma} \in S_{\Gamma(\Lambda)}$ such that $\hat{\eta}(\boldsymbol{\sigma}) = c$.*

In the following we are considering graphs with an arbitrary inner interface and an empty outer interface. We need the arbitrary inner interface in order to state Theorem 2 below. We could, without major problems, also parametrize over the outer interface, but this is not necessary for the theory.

Definition 2 (Bounded graph automaton). *Let $n \in \mathbb{N}$ and $k \leq n$ be given. An n-*bounded graph automaton *$\mathcal{A} = (Q, \Sigma, \delta, I, F)$* from k*, where $\Sigma = \Gamma(\Lambda)$, consists of*

- $Q = (Q_i)_{i \leq n}$ *the* family of finite state sets*,*
- $\Sigma = (\Sigma_{i,j})_{i,j \leq n}$ *the* doubly-ranked input alphabet*,*
- $\delta = (\delta_{i,j})_{i,j \leq n}$ *is a* family of transition functions*, where $\delta_{i,j} \colon Q_i \times \Sigma_{i,j} \to \wp(Q_j)$*
- $I \subseteq Q_k$ *the set of* initial states *and*
- $F \subseteq Q_0$ *the set of* final states

such that the following condition holds for all $q \in Q$ and $\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2 \in S_{i,j}$:

$$\text{if } \hat{\eta}(\boldsymbol{\sigma}_1) \simeq \hat{\eta}(\boldsymbol{\sigma}_2) \text{ then } \hat{\delta}_{i,j}(\{q\}, \boldsymbol{\sigma}_1) = \hat{\delta}_{i,j}(\{q\}, \boldsymbol{\sigma}_2), \qquad (\star)$$

where $\hat{\delta}_{i,j} \colon \wp(Q_i) \times S_{i,j} \to \wp(Q_j)$ is defined as follows:

$$\hat{\delta}_{i,j}(R, \boldsymbol{\sigma}) := \begin{cases} R & \text{if } \boldsymbol{\sigma} = \epsilon_i \in \Sigma_{i,i} \text{ and } i = j \\ \delta(R, \sigma) & \text{if } \boldsymbol{\sigma} = \sigma \in \Sigma_{i,j} \\ \hat{\delta}_{k,j}(\delta_{i,k}(R, \boldsymbol{\sigma}_1), \boldsymbol{\sigma}_2) & \text{if } \boldsymbol{\sigma} = (\boldsymbol{\sigma}_1 \,;\, \boldsymbol{\sigma}_2), \boldsymbol{\sigma}_1 \in S_{i,k}, \boldsymbol{\sigma}_2 \in S_{k,j} \end{cases}.$$

A sequence $\boldsymbol{\sigma} \in S_{k,0}$ over Σ is accepted *by $\mathcal{A}$ if and only if $\hat{\delta}_{k,0}(I, \boldsymbol{\sigma}) \cap F \neq \emptyset$.*

The idea behind a graph automaton is to get a decomposition of an input graph and to process it "piece by piece". The condition $(\star)$ guarantees that the graph automaton accepts an input graph independently of the decomposition of the graph. Showing that this condition holds for some prospective graph automaton is not trivial in general. A solution would be to automatically translate formulas of monadic second-order logic to correct graph automata.

Definition 3 (Accepted language). *Let an n-bounded graph automaton $\mathcal{A}$ from k be given. The* language *accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, contains exactly the sequences accepted by $\mathcal{A}$. The* cospan language accepted by $\mathcal{A}$ *is*

$$\mathcal{G}(\mathcal{A}) = \{c \mid \hat{\eta}(\boldsymbol{\sigma}) = c \text{ for some } \boldsymbol{\sigma} \in \mathcal{L}(\mathcal{A})\}.$$

The cospan language of a bounded graph automaton contains cospans. When we want to accept graphs, we can interpret the cospan $[G]$ as the graph G.

Since a graph automaton is bounded, it is a kind of non-deterministic finite automaton (NFA). Therefore, we can apply standard algorithms from formal language theory, such as the subset construction and constructing the cross product of two automata. It can be shown that these constructions preserve the condition $(\star)$ of graph automata. Thus, the languages accepted by n-bounded graph automata are closed under boolean operations, and many important decision problems (such as the membership, emptiness and language inclusion problems) are decidable. Note that the language inclusion algorithm for NFA is PSPACE-complete, and thus no efficient algorithms for the problem exist yet.

Example 1. First we consider the language L_U of all graphs which contain a fixed subgraph U. The bounded graph automaton $\mathcal{A}_U$ accepting this language works as follows: Every state in each of the state sets Q_i contains two pieces of information. The first piece of information says which parts of the subgraph have already been recognized. The second piece of information is a function which maps every outer node to a node which has already been recognized or to some "bottom element" to indicate that the interface node is not mapped to a node of the wanted subgraph U. The transition function "updates" this information according to the letter which is currently processed. Since the input graph might contain several parts which are isomorphic to the wanted subgraph U, the bounded graph automaton is highly non-deterministic. More details about the construction of this graph automaton can be found in [5].

Example 2. Now we consider the language $C_{(k)}$ of all k-colorable graphs (for some $k \in \mathbb{N}$). A k-coloring of a graph G is a function $f\colon V_G \to \mathbb{N}_k$ such that for all edges $e \in E_G$ and for all nodes $v_1, v_2 \in att_G(e)$ it holds that $f(v_1) \neq f(v_2)$ if $v_1 \neq v_2$. The question whether a graph is k-colorable is essential in many applications, for example in scheduling. The idea of the graph automaton $\mathcal{A}_{(k)}$ accepting all k-colorable graphs (as defined in [10]) is as follows: Every state is a valid k-coloring of D_i, that is $Q_i = \{f\colon V_{D_i} \to \mathbb{N}_k \mid f \text{ is a valid } k\text{-coloring of } D_i\}$. The transition function $\delta_{i,j}$ maps a coloring $f \in Q_i$ and a letter $\sigma \in \Sigma$ to a coloring $f' \in Q_j$ if and only if the coloring of the inner nodes of $\eta(\sigma)$ according to f and the coloring of the outer nodes of $\eta(\sigma)$ according to f' leads to a valid coloring of $\eta(\sigma)$. More details on graph automata for coloring can be found in Sect. 5.1.

In the rest of the section we compare bounded graph automata to automaton functors, which were introduced in [10], in particular to automaton functors for the category $\mathbf{OLCG}_i$ (bounded automaton functors). We show that they accept the same class of language. The main difference between the two is that bounded automaton functors are defined on *all* cospans of bounded size (of which there are infinitely many), while graph automata are only defined for the letters of the input alphabet, which correspond to only the atomic cospans (of which there are finitely many).

Definition 4 (Bounded Automaton Functor). *Let $n \in \mathbb{N}$. An n-*bounded automaton functor from k *is a structure $\mathcal{A} = (\mathcal{A}_0, I, F)$, where*

- *$\mathcal{A}_0\colon \mathbf{OLCG}_n \to \mathbf{Rel}$ is a functor which maps every discrete graph D_i to a finite set $\mathcal{A}_0(D_i)$ (the* state set *of D_i) and every output linear cospan $c\colon D_i \nrightarrow D_j$ to a relation $\mathcal{A}_0(c) \subseteq \mathcal{A}_0(D_i) \times \mathcal{A}_0(D_j)$ (the* transition relation *of c),*
- *$I \subseteq \mathcal{A}_0(D_k)$ is the* set of initial *states and*
- *$F \subseteq \mathcal{A}_0(\emptyset)$ is the* set of final states*.*

For a discrete graph G or a output linear cospan c we will, in the following, usually write $\mathcal{A}(G)$ and $\mathcal{A}(c)$ instead of $\mathcal{A}_0(G)$ and $\mathcal{A}_0(c)$, respectively. A cospan $c\colon D_k \nrightarrow \emptyset$ is accepted by $\mathcal{A}$, if $(q, q') \in \mathcal{A}(c)$ for some $q \in I$ and $q' \in F$.

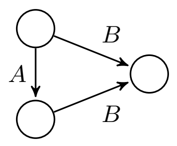


Fig. 2. Wanted subgraph D

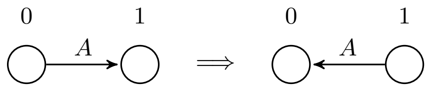


Fig. 3. Transformation rule ρ_A

Definition 5 (Accepted language). *Let $\mathcal{A}$ be an n-bounded automaton functor. The language accepted by $\mathcal{A}$, denoted by $\mathcal{G}(\mathcal{A})$, contains exactly the cospans accepted by $\mathcal{A}$.*

Theorem 1. *Let L be a language of cospans from D_k to $\emptyset$. Then L is the cospan language of an n-bounded graph automaton from k if and only if it is the language of an n-bounded automaton functor from k.*

4 Invariant Checking and Language Inclusion

One of the applications of our approach is to automatically check invariants of graph transformation systems (GTSs). The following definition of graph transformation is equivalent to the well-known double-pushout approach [19], where we have injective rule spans and not necessarily injective matches.

Definition 6 (Graph transformation).

(i) *Let $\ell \colon \emptyset \leftrightarrow D_i$ and $r \colon \emptyset \leftrightarrow D_i$ be two output linear cospans (called left-hand and right-hand side). The pair $\rho = (\ell, r)$ is called a* (graph) transformation rule. *A* graph transformation system *is a finite set of transformation rules.*

(ii) *Let $\rho = (\ell, r)$ be a transformation rule. The rule ρ is* applicable *to a graph G if and only if $[G] = \ell \,;\, c$ for some output linear cospan $D_i \leftrightarrow \emptyset$. In this case we write $G \Rightarrow_{\rho,c} H$, where H is the graph obtained from $[H] = r \,;\, c$.*

A language L is an invariant according to a graph transformation rule ρ if it holds for all graphs G and H with $G \Rightarrow_\rho H$ that $[G] \in L$ implies $[H] \in L$.

Example 3. As an example we take the graph D (which is depicted in Fig. 2) as wanted subgraph. The language L_D of all graphs containing D as a subgraph is an invariant for the rule ρ_A (shown in Fig. 3) which "switches" an A-labeled edge. Obviously, every graph which contains D as subgraph before the application of ρ_A does contain D also after the rule application.

Example 4. The next example we consider is the language $C_{(2)}$ of all 2-colorable graphs (see Ex. 2 for details about $C_{(2)}$). This language is an invariant for the transformation rule α_n depicted in Fig. 4 which adds two new nodes between two adjacent nodes on a path. That the language $C_{(2)}$ is an invariant for this rule is clear since every path with an even number of nodes is 2-colorable.

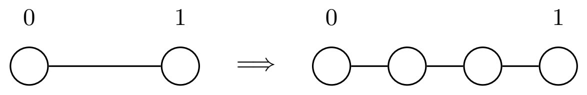


Fig. 4. Transformation rule α_n

For an output linear cospan $c\colon D_k \nrightarrow D_m$ and a n-bounded graph automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ from k we obtain a new n-bounded graph automaton $\mathcal{A}[c] = (Q, \Sigma, \delta, I', F)$ from m with $I' = \hat{\delta}_{k,m}(I, \boldsymbol{\sigma})$, where $\boldsymbol{\sigma}$ is some word from S_Σ such that $\hat{\eta}(\boldsymbol{\sigma}) = c$. (If the width of c is larger than n, such a σ does not exist, and we take $I' = \emptyset$, such that $\mathcal{L}(\mathcal{A}[c]) = \emptyset$.) The new automaton has as new initial states all states reachable from the original initial states by processing c. Note that I' is independent of the specific decomposition of c into a sequence $\boldsymbol{\sigma}$.

The following theorem easily follows from the observation that $\boldsymbol{\sigma}_\ell \,;\, \boldsymbol{\sigma}_c \in \mathcal{L}(\mathcal{A})$ if and only if $\boldsymbol{\sigma}_c \in \mathcal{L}(\mathcal{A}[\ell]))$, where $\boldsymbol{\sigma}_\ell$ and $\boldsymbol{\sigma}_c$ are sequences such that $\hat{\eta}(\boldsymbol{\sigma}_\ell) = \ell$ and $\hat{\eta}(\boldsymbol{\sigma}_c) = c$.

Theorem 2 (Invariant checking). *Let $\mathcal{A}$ be an n-bounded graph automaton (from 0) accepting the cospan language L, and let $\rho = (\ell, r)$ be a transformation rule. The cospan language L is an invariant of ρ if and only if $\mathcal{L}(\mathcal{A}[\ell]) \subseteq \mathcal{L}(\mathcal{A}[r])$.*

5 Implementation and Results

We implemented a language inclusion algorithm and invariant checking in the Java-based tool *Raven*. In this section we examine some implementation details of the tool and present results of case studies.

5.1 Representation of Automata with BDDs

Graph automata are represented by means of BDDs. First, states of the automaton are represented by a bit string, and secondly the transition relations for the various atomic cospans (or letters respectively) are stored as a BDD which encodes a relation on these bit strings.

As an example, we look at the encoding for the automaton which accepts all k-colorable graphs (see Ex. 2). The state encoding has to take care of the following information: the interface size (of the outer interface of the graph seen so far) and the color of each node currently occurring in the outer interface.

A good ordering of the bits holding the information is essential to construct compact BDDs. We have experimented with different orderings, and found the following to be the best. Let n be the maximum interface size and k the number of colors. Furthermore, let $m = \lceil \log_2 n \rceil$ be the number of bits required to store the interface size, and $\ell = \lceil \log_2(k+1) \rceil$ the number of bits to store one color (we need an extra value to represent uncolored or unused nodes). A state is encoded by the bit sequence $\boldsymbol{b}\,\boldsymbol{c_1} \ldots \boldsymbol{c_n} = b_1 \ldots b_m (c_{1,1} \ldots c_{1,\ell}) \ldots (c_{n,1} \ldots c_{n,\ell})$, where $\boldsymbol{b} = b_1 \dots b_m$ encodes the current interface size as a binary number and $\boldsymbol{c_i} = (c_{i,1} \dots c_{i,\ell})$ (for $1 \le i \le n$) represents the color of the i-th interface node.

For each of the letters of $\Gamma(\Lambda)$ we define a propositional formula describing the transition relations – for all permitted interfaces – of the graph automaton. These formulas can then be easily transformed into BDDs which describe the transition functions. (As usual with BDD representations of relations, the bits of the domain and codomain states are interleaved.)

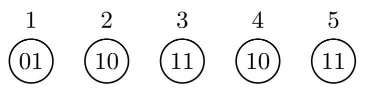
We present the formula $f_{connect^i_A}$ as an example. To distinguish between the bits for the current state and the bits for the successor state we indicate the successor state encoding by $\boldsymbol{b'c'_1} \dots \boldsymbol{c'_n}$. The formula consists of four parts (where $p = i - ar(A) + 1$ is the index of the first node attached to the new edge):

$$f_1 := (ar(A) \le i) \wedge (\boldsymbol{b} = i) \wedge (\boldsymbol{b} = \boldsymbol{b'}) \qquad f_3 := \bigwedge_{j=1}^{n} (\boldsymbol{c_j} = \boldsymbol{c'_j})$$

$$f_2 := \bigwedge_{j=i+1}^{n} (\boldsymbol{c_j} = \boldsymbol{0}) \qquad f_4 := \bigwedge_{j=p}^{i} \bigwedge_{j'=p}^{i} (j \neq j') \rightarrow (\boldsymbol{c_j} \neq \boldsymbol{c_{j'}})$$

The subformula f_1 expresses that the arity of the added edge is less than or equal to the current interface and that the interface size of both the current state and the successor state is i. The subformula f_2 expresses that the nodes of the encoding which do not belong to the current interface, that is the last $n - i + 1$ nodes in the encoding, have not been colored. Next, f_3 expresses that all nodes have the same color in the source and the target state. Finally, f_4 expresses that the nodes which are connected by the new edge have different colors. Now, we take $f_{connect_A} := f_1 \wedge f_2 \wedge f_3 \wedge f_4$, that is, a transition $q \xrightarrow{connect^i_A} q'$ is allowed if and only if the above four conditions hold.

Example 5. We consider the 3-colorability automaton with a maximum interface size of 5. The size of the state encoding is $3 + (2 \cdot 5) = 13$ bits. Consider the state q depicted in Fig. 5 (on the left): we have five nodes in the current interface, colored with color 1, 2, 3, 2 and 3, respectively. The bit string which encodes this state is given in Fig. 5 on the right.



b_1	b_2	b_3	$\boldsymbol{c_1}$	$\boldsymbol{c_2}$	$\boldsymbol{c_3}$	$\boldsymbol{c_4}$	$\boldsymbol{c_5}$
1	0	1	01	10	11	10	11

Fig. 5. State q and its representation as bit string

Suppose that the graph automaton is currently in state q, and that the next letter is $connect^5_A$, where A is a label with arity 2. Since the last two nodes of the interface are colored differently, none of the nodes connected by the A-edge have the same color. Hence the transition $q \xrightarrow{connect^5_A} q$ is in the transition relation. Suppose on the other hand that the graph automaton is in state q and the next operation is $connect_B^5$, where the arity of B is 3. Since the third and fifth node of q's interface have the same color, no state can be reached from q.

Apart from a graph automaton which accepts k-colorable graphs and one which accepts graphs with a specific subgraph (see Ex. 1), we also implemented graph automata for vertex cover and dominating set. A *vertex cover* of graph G is a set C of nodes of G such that each edge is incident to at least one node of C. A *dominating set* of a graph G is a set D of nodes of G such that each node of G is either in D or adjacent to a node in D. The states of automata checking if the input graph has a vertex cover of size k or if the input graph has a dominating set of size k respectively need to encode the following pieces of information:

- *Vertex cover:* the interface size of the outer interface of the graph seen so far, which nodes of the current interface are part of the vertex cover and the size of the vertex cover (where nodes in the vertex cover are counted when they are removed from the interface).
- *Dominating set:* the interface size of the outer interface of the graph seen so far, which nodes of the current interface are part of the dominating set, which nodes of the current interface are dominated by some node of the dominating set and the size of the dominating set (where nodes in the dominating set are counted when they are removed from the interface).

Note that we use BDDs in a different way than other tools. In our case, the alphabet is small and the state set is huge, and we use BDDs to encode a transition relation for each symbol. In other tools, such as MONA [17], the state set is relatively small and the alphabet is huge. Thus MONA uses BDDs not to encode the transition relation for each symbol, but to encode the possible transitions of each single state, that is for each state there is a BDD encoding all transitions for each alphabet symbol starting at that specific state.

5.2 Checking Language Inclusion

In [7] we presented a technique for checking invariants based on the Myhill-Nerode quasi-order. The main disadvantage of this approach is that the algorithm for computing the Myhill-Nerode quasi-order applies only to deterministic (graph) automata, whereas in general our graph automata are highly non-deterministic. Determinization is not an option because it would lead to an exponential blow-up of already huge automata. Therefore we had to settle for an approximation.

To overcome this problem, here we use the antichain-based algorithm from [21] to check for language inclusion, which can be used to check invariants via Theorem 2. In the worst case this approach can still need exponential time, but in practise one can often achieve very good runtimes.

An antichain is a set of elements which are uncomparable with respect to some ordering. What the elements look like and what ordering is used depends on the application; here we present an antichain-based algorithm to decide language inclusion. In this subsection, we forget typing information of the states and consider bounded automata as regular finite automata.

Let $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$ be n-bounded graph automata. Let $\overline{F_\mathcal{B}} = Q_\mathcal{B} \setminus F_\mathcal{B}$, that is, the set of $\mathcal{B}$'s non-accepting states. We want to decide whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. In particular, we are trying to falsify that claim by finding a state $q \in I_\mathcal{A}$ and a set of states $S \subseteq I_\mathcal{B}$ such that $\hat{\delta}_\mathcal{A}(\{q\}, \boldsymbol{\sigma}) \cap F_\mathcal{A} \neq \emptyset$ and $\hat{\delta}_\mathcal{B}(S, \boldsymbol{\sigma}) \subseteq \overline{F_\mathcal{B}}$, for some word $\boldsymbol{\sigma}$.

Let $\mathcal{U} = Q_A \times \wp(Q_B)$. For $(q_1, S_1), (q_2, S_2) \in \mathcal{U}$, we define $(q_1, S_1) \leq (q_2, S_2)$ if $q_1 = q_2$ and $S_1 \subseteq S_2$. Now, an *antichain* (for language inclusion) is a set $K \subseteq \mathcal{U}$ such that for all $p_1, p_2 \in K$ with $p_1 \neq p_2$, it holds that neither $p_1 \leq p_2$ nor $p_2 \leq p_1$. A pair $p \in K$ is called maximal, if there is no $p' \in K$ such that $p \leq p'$; by $\lceil K \rceil$ we denote the set of maximal elements of K. Minimal elements and the set $\lfloor K \rfloor$ of minimal elements are defined symmetrically.

The algorithm searches through the automaton backwards. We define:

$$\mathsf{Pre}_{\mathcal{A},\mathcal{B}}(K) = \big\{(q, S) \;\big|\; \exists \sigma \in \Sigma : \exists (q', S') \in K : q' \in \delta_\mathcal{A}(q, \sigma) \wedge \hat{\delta}_\mathcal{B}(S, \sigma) \subseteq S'\big\}.$$

The function does the following: For each $(q', S') \in K$, we take the pairs (q, S) such that, for some symbol σ, q is an σ-predecessor of q' and S is the set of states, from which a state in S' is surely reached when reading σ.

Formally, the basic version of the algorithm, which returns *true* if and only if $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$, works as follows:

```
**input:** $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$
$K \leftarrow F_\mathcal{A} \times \{\overline{F_\mathcal{B}}\}$
**repeat**
    $K' \leftarrow K$
    $K \leftarrow \lceil K \cup \mathsf{Pre}_{\mathcal{A},\mathcal{B}}(K) \rceil$
**until** $K = K'$
**return** there exist $q \in I_\mathcal{A}$ and $S \supseteq I_\mathcal{B}$ such that $(q, S) \in K$
```

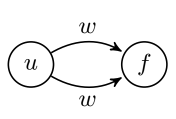
The line $K \leftarrow \lceil K \cup \mathsf{Pre}(K) \rceil$ adds new elements to the current antichain and removes all but the maximal ones. At all times it holds that for all $(q, S) \in K$ there is a word $\boldsymbol{\sigma}$ such that $\hat{\delta}_\mathcal{A}(\{q\}, \boldsymbol{\sigma}) \cap F_\mathcal{A} \neq \emptyset$ and $\hat{\delta}_\mathcal{B}(S), \boldsymbol{\sigma}) \subseteq \overline{F_\mathcal{B}}$.

The basic algorithm can be optimized in various ways. First, only new elements need to be processed in each step instead of all the elements in K. Second, since the function is monotone, the algorithm can return *true* as soon as the final condition is satisfied (meaning that $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$). For a correctness proof of the algorithm, we refer to [21].

Note that in the implementation that we used in the tool, both the automata and the pairs in the antichains are represented symbolically as BDDs. We also tested a forward search variant of the algorithm, but do not include it here due to poor runtimes.

5.3 Results

In this section we present results for several case studies. All tests were performed on a 64-bit Linux machine with a Xeon Dualcore 5150 processor and 8 GB of available main memory.



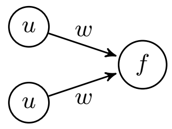


Fig. 6. Forbidden subgraphs "Double Access" and "Two Users"

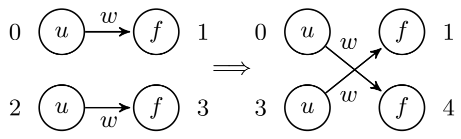


Fig. 7. Operation "Switch Write Access" as transformation rule

In the following, we briefly describe the several examples for which we have computed results for different interface sizes using our tool suite. For each of these examples we used the backwards language inclusion algorithm to compute our results.

3-Colorability and 4-Colorability. We checked $C_{(3)} \subseteq C_{(4)}$ and $C_{(4)} \not\subseteq C_{(3)}$ (in the case of non-inclusion a counter example is generated).

Triangle subgraph and *2-Colorability with path extension.* These are the invariants from Ex. 3 and Ex. 4, respectively.

Multi-user file system. We validate the file system example from [7]. In this example, a system state is modelled as a graph: users and files are nodes, access permissions (either "read" or "write") are labelled, directed edges. The system behaviour (add new user, change access permissions, . . .) is modelled as transformation rules. The problem is to check whether the file system can reach at least one of two forbidden states. These forbidden states are modelled as the subgraphs depicted in Fig. 6: "Double Access" models the situation where a user obtains double write access to a system resource and "Two Users" models the situation where two users both have write access to the same system resource.

To validate this system we perform a "backwards invariant check": we swap the left- and right-hand sides of the rules and check whether the language of all graphs which contain certain "forbidden subgraphs" is an invariant of this reversed system. The idea is that a forbidden state is reachable (in the original system) only if the system already started in a forbidden state.

Because in [5] a simulation relation was used to approximate the Myhill-Nerode quasi-order, validating the operation "Switch Write Access" (see Fig. 7), which switches the write access of two users, was unsuccessful, although the language is an invariant w.r.t. this operation. Now we succesfully verified it.

Dominating Set and Vertex Cover. We computed results for the inclusion of the language $\mathit{NonIso} \cap D_{(k)}$ of all graphs without isolated nodes which have a dominating set of size at most k in the language $V_{(k)}$ of all graphs which have a vertex cover of size at most k and the non-inclusion of the opposite direction.

In Table 1 the runtime results for the case studies are presented. We can handle some non-trivial examples up to relatively large interface size (note that in practical applications the width, and thus the interface size of graphs, is in general relatively small). For example, the "triangle subgraph automaton" has 37 440 states in case of maximum interface 3 and 19 173 952 states in case of

Table 1. Case study runtimes (in seconds); TO: timed out, OM: out of memory, n.a.: not applicable

	Maximum Interface Size							
Case study	3	4	5	6	7	8	9	10
$C_{(3)} \subseteq C_{(4)}$	< 1	3	14	410	28 713	TO	–	–
$C_{(4)} \not\subseteq C_{(3)}$	n.a.	9	270	63 065	TO	–	–	–
Triangle subgraph	4	15	123	1 978	OM	–	–	–
$C_{(2)}$ and path extension	2	2	3	5	13	53	385	4 193
Multi-user file system	n.a.	19	217	OM	–	–	–	–
$\mathit{NonIso} \cap D_{(2)} \subseteq V_{(2)}$	n.a.	432	26 337	TO	–	–	–	–
$V_{(2)} \not\subseteq \mathit{NonIso} \cap D_{(2)}$	n.a.	2	12	14	154	4 701	TO	–

interface size 6. From the first two and last two case studies, it is also apparent that the runtimes are better when the first automaton is small (the automaton for $C_{(3)}$ and $V_{(2)}$, respectively). This is unsurprising, because the states of the first automaton are explicitly represented (more formally, as a BDD representing a singleton set), whereas the (sets of) states of the second automaton are collectively represented by a BDD.

6 Conclusion

We gave a concrete variant of graph automata accepting recognizable languages. The languages such graph automata can accept contain cospans which have a bounded width, which means that we can only accept graphs with a bounded path width [6]. We applied the approach to automatically checking whether the language of one automaton is included in the language of the other and whether a language is an invariant of a graph transformation system. Case studies show that we can handle non-trivial examples in a relatively short time. However, it seems that the size of the generated automata and the running times grow exponentially with the interface size of the automaton.

Note that our approach differs from the approach in MONA [17], another tool based on recognizable languages. MONA is suitable when the alphabet is large (since BDDs are used to encode the alphabet), whereas in our case the state space is huge.

Another related work [4] considers graph patterns consisting of negative and positive components and shows that they are invariants via an exhaustive search.

For further research, we would like to try more algorithms; in particular we want to implement the simulation-based algorithm of [1] in our tool to see if better results can be obtained. Also, an algorithm that can translate formulas of monadic second-order logic into automata would be helpful. Finally, it is ongoing research to see whether graph automata can help in proving termination of graph transformation systems, much like in the case of string rewrite systems [15].

References

1. Abdulla, P.A., Chen, Y.F., Holík, L., Vojnar, T.: When simulation meets antichains (on checking language inclusion of NFAs). In: Proc. of TACAS '10. pp. 158–174. LNCS 6015, Springer (2010)
2. Andersen, H.R.: An introduction to binary decision diagrams. Course Notes (1997), http://www.configit.com/fileadmin/Configit/Documents/bdd-eap.pdf
3. Bauderon, M., Courcelle, B.: Graph expressions and graph rewritings. Mathematical Systems Theory 20(2-3), 83–127 (1987)
4. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic invariant verification for systems with dynamic structural adaptation. In: Proc. of ICSE '06 (International Conference on Software Engineering). pp. 72–81. ACM (2006)
5. Blume, C.: Graphsprachen für die Spezifikation von Invarianten bei verteilten und dynamischen Systemen. Master's thesis, Universität Duisburg-Essen (2008)
6. Blume, C., Bruggink, S., Friedrich, M., König, B.: Treewidth, pathwidth and cospan decompositions. In: Proc. of GT-VMT 2011 (2011)
7. Blume, C., Bruggink, S., König, B.: Recognizable graph languages for checking invariants. In: Proc. of GT-VMT 2010. ECEASST 29 (2010)
8. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Proc. of SAS '06. pp. 52–70. LNCS 4134, Springer (2006)
9. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Proc. of CAV '00. pp. 403–418. LNCS 1855, Springer (2000)
10. Bruggink, S., König, B.: On the recognizability of arrow and graph languages. In: Proceedings of ICGT '08. pp. 336–350. LNCS 5214, Springer (2008)
11. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007), http://www.grappa.univ-lille3.fr/tata, 12 October 2007
12. Courcelle, B.: The monadic second-order logic of graphs I. recognizable sets of finite graphs. Inf. Comput. 85(1), 12–75 (1990)
13. Courcelle, B., Durand, I.: Verifying monadic second order graph properties with tree automata. In: European Lisp Symposium (May 2010)
14. Flum, J., Frick, M., Grohe, M.: Query evaluation via tree-decompositions. Journal of the ACM 49(6), 716–752 (2002)
15. Geser, A., Hofbauer, D., Waldmann, J.: Match-bounded string rewriting. Applicable Algebra in Engineering, Communication and Computing 15(3–4), 149–171 (2004)
16. Gottlob, G., Pichler, R., Wei, F.: Bounded treewidth as a key to tractability of knowledge representation and reasoning. Journal of Artificial Intelligence 174(1), 105–132 (2010)
17. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. International Journal of Foundations of Computer Science 13(4), 571–586 (2002)
18. Kneis, J., Langer, A., Rossmanith, P.: Courcelle's theorem – a game-theoretic approach. Discrete Optimization (2011)
19. Sassone, V., Sobociński, P.: Reactive systems over cospans. In: Proc. of LICS '05. pp. 311–320. IEEE (2005)
20. Soguet, D.: Génération automatique d'algorithmes linéaires. Ph.D. thesis, Université Paris-Sud (2008)
21. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: Proceedings of CAV 2006. pp. 17–30. LNCS 4144, Springer (2006)