View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Communications of the EASST (European Association of Software Science and...

Electronic Communications of the EASST
Volume 38 (2010)

Proceedings of the Fifth International Conference on Graph Transformation - Doctoral Symposium (ICGT-DS 2010)

Efficient Implementation of Automaton Functors for the Verification of Graph Transformation Systems

Christoph Blume

15 pages

Guest Editor: Andrea Corradini
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Efficient Implementation of Automaton Functors for the Verification of Graph Transformation Systems

Christoph Blume*

Abteilung für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany
christoph.blume@uni-due.de

Abstract: In this paper we show new applications for recognizable graph languages to invariant checking. Furthermore we present details about techniques we used for an implementation of a tool suite for (finite) automaton functors which generalize finite automata to the setting of recognizable (graph) languages. In order to develop an efficient implementation we take advantage of Binary Decision Diagrams (BDDs).

Keywords: graph transformation, recognizable graph languages, invariants, implementation of automaton functors, binary decision diagrams

1 Introduction

The theory of regular languages is the basis of a number of analysis techniques, such as regular model checking [6], termination analysis [14] and reachability analysis [13]. The notion of regularity has been straightforwardly generalized to regular graph languages – which are also called *recognizable graph languages* – in different ways [2, 10, 7, 8], but all leading to the same notion of recognizability. Very roughly, one can say that a property (or language) of graphs is recognizable whenever it can be derived inductively via an arbitrary decomposition of the graph. In addition the size of the information "transported" over an interface in the decomposition must be bounded by a function which is dependent only on the size of the interface. Alternatively recognizability can be defined via a family of Myhill-Nerode style congruences of finite index, i.e., congruences with finitely many equivalence classes.

In this paper, we use the notion of recognizability by Bruggink and König [8] which is based on a categorical definition of recognizability in terms of so-called *automaton functors*, which are a generalization of non-deterministic finite automata. An advantage of this automaton-oriented notion of recognizability is that many familiar constructions on finite automata, such as the determinization, minimization as well as closure properties under boolean operations, can be straightforwardly generalized to automaton functors.

The paper is structured as follows: In Section 2 we briefly define recognizable graph languages, automaton functors, and the category-theoretic notions at the heart thereof. In Section 3 we show some new examples of our invariant checking technique developed in [4]. In Section 4 we present our implementation techniques we have used for starting to develop an automaton functor tool suite. In Section 5 we give a conclusion and point out new links for further research topics.

* Research partially supported by the DFG project GaReV.

2 Preliminaries

In this section we briefly recall some concepts of category theory and recognizable graph languages. We presuppose a basic knowledge of category theory and order theory.

2.1 Category Theory and Recognizable Graph Languages

The category which has sets as objects and relations as arrows is denoted by **Rel**. The subcategory which has total functions as arrows is denoted by **Set**. The composition of two composable arrows f and g is denoted by $f\,;g = g \circ f$.

Let **C** be a category with pushouts. A cospan $c\colon J \xrightarrow{c^L} C \xleftarrow{c^R} K$ is a pair of **C**-arrows with the same codomain. Here, J and K are the domain (or *inner interface*) and codomain (or *outer interface*) of the cospan c, respectively. The identity cospan for an object E is the cospan consisting of twice the identity arrow of E. Let $c\colon J \xrightarrow{c^L} C \xleftarrow{c^R} K$ and $d\colon K \xrightarrow{d^L} D \xleftarrow{d^R} M$ be cospans (where the codomain of c equals the domain of d). The composition of c and d is obtained by taking the pushout of c^R and d^L. A *semi-abstract cospan* is an equivalence class of cospans, where we take the middle object of the cospan up to isomorphism. Now, the cospan category *Cospan*(**C**) is defined as the category which has the objects of **C** as objects, and semi-abstract cospans as arrows.

Let a set Σ of labels be given. A *hypergraph* G, later also simply called *graph*, is a four-tuple $\langle V_G, E_G, \mathsf{att}_G, \mathsf{lab}_G\rangle$, where V_G is a finite set of *vertices* (or *nodes*) of G, E_G is a finite set of *edges* of G, $\mathsf{att}_G\colon E_G \to V_G^*$ is the *attachment function* and $\mathsf{lab}_G\colon E_G \to \Sigma$ is the *labeling function*. Here, V_G^* denotes the set of finite sequences of elements of V_G. A *hypergraph morphism* f is a structure-preserving map between two hypergraphs. A discrete graph is a graph which does not contain any edges. The discrete graph with n nodes is denoted by D_n. The *empty graph* is denoted by $\emptyset$ instead of D_0. The category of graphs and graph morphisms is denoted by **HGraph**.

A cospan of graphs (an arrow in the category *Cospan*(**HGraph**)) can be seen as a graph with an inner (left) and an outer (right) interface. Intuitively, the interfaces designate the parts of the graph which can be "touched" from the outside. With $[G]\colon \emptyset \to G \leftarrow \emptyset$ we denote the cospan consisting of a graph G with empty inner and outer interfaces.

Cospans of graphs are closely related to graph transformation systems, in particular to the double-pushout (DPO) approach to graph rewriting [19]. A DPO rewrite rule $\rho\colon L \xleftarrow{\rho_L} I \xrightarrow{\rho_R} R$ can be considered as a pair of cospans $\ell\colon \emptyset \to L \xleftarrow{\rho_L} I$ and $r\colon \emptyset \to R \xleftarrow{\rho_R} I$, which will in the following be called left- and right-hand side, respectively. Then it holds that $G \Rightarrow_\rho H$ if and only if $[G] = \ell\,;c$ and $[H] = r\,;c$, for some cospan c.

We define recognizable graph languages by using automaton functors on the category of cospans of graphs, as in [8].

Definition 1 (Automaton functor, recognizability) Let a category **C** with initial object $\emptyset$ be given. An automaton functor is a functor $\mathscr{A}\colon \mathbf{C} \to \mathbf{Rel}$, which maps every object X of **C** to a finite set $\mathscr{A}(X)$ of *states* of X and every arrow $f\colon X \to Y$ to a relation $\mathscr{A}(f) \subseteq \mathscr{A}(X) \times \mathscr{A}(Y)$, together with two distinguished sets $\mathrm{I}^{\mathscr{A}} \subseteq \mathscr{A}(\emptyset)$ and $\mathrm{F}^{\mathscr{A}} \subseteq \mathscr{A}(\emptyset)$ of *initial* and *final states*, respectively.

An automaton functor is *deterministic* if every relation $\mathscr{A}(f)$ is a function and every $\mathrm{I}^{\mathscr{A}}$ contains exactly one element.

An arrow $f\colon \emptyset \to \emptyset$ is accepted by an automaton functor $\mathscr{A}$, if $\langle s,t\rangle \in \mathscr{A}(f)$, for some $s \in \mathrm{I}^{\mathscr{A}}$ and $t \in \mathrm{F}^{\mathscr{A}}$. The language $L(\mathscr{A})$ of an automaton functor contains exactly those arrows which are accepted by it. A language L of arrows from $\emptyset$ to $\emptyset$ is a *recognizable language* if $L = L(\mathscr{A})$, for some automaton functor $\mathscr{A}$.

The intuition behind the definition is to have a mapping into a (locally) finite domain. The functor property guarantees that decomposing an object in different ways does not affect acceptance in any way. This is different from word languages, where there is essentially one way to decompose an object into subobjects.

Familiar constructions on finite automata, such as the determinization construction, can be easily generalized to automaton functors. Also, it was shown in [8], that restricting to discrete interfaces does not affect the expressiveness of the formalism. Due to the latter result, we shall restrict to discrete interfaces in the rest of this paper.

The above definition can easily be generalized to accept languages between arbitrary objects. However, in our setting we require only languages from the initial object to the initial object.

A characterization of recognizable graph languages can be obtained in terms of recognizable languages in $Cospan(\mathbf{HGraph})$:

Definition 2 (Recognizable graph language) A set L of graphs is a *recognizable graph language*, if $[L] = \{[G]\colon \emptyset \to G \leftarrow \emptyset \mid G \in L\}$ is a recognizable language in $Cospan(\mathbf{HGraph})$.

In the following we do not distinguish between L, a language of graphs, and $[L]$, a language of (cospans of) graphs with empty interfaces.

2.2 Atomic Cospans

We assume that the set of nodes of each discrete graph D_n is $V_{D_n} = \{v_0, \dots v_{n-1}\}$. We set $\mathbb{N}_n = \{0, \dots, n-1\}$ and we denote the *disjoint union of two graphs* G_1 *and* G_2 by $G_1 \oplus G_2$. We assume that G_1 and G_2 are disjoint. Furthermore we define the *disjoint union* $f \oplus g\colon G_1 \oplus G_2 \to H_1 \oplus H_2$ *of two graph morphisms* $f\colon G_1 \to H_1$ *and* $g\colon G_2 \to H_2$ where H_1 and H_2 are disjoint as follows:

$$(f \oplus g)(v) = \begin{cases} f(v), & \text{if } v \in V_{G_1} \\ g(v), & \text{if } v \in V_{G_2} \end{cases} \quad \text{and} \quad (f \oplus g)(e) = \begin{cases} f(e), & \text{if } e \in E_{G_1} \\ g(e), & \text{if } e \in E_{G_2} \end{cases}.$$

Definition 3 (Atomic graph operations) *Restriction of the outer interface:* Let $\rho\colon D_{n-1} \to D_n$ with $\rho(v_i) = v_i$ be an arrow between two discrete graphs. We define the cospan res_n as follows: $\mathsf{res}_n\colon D_n \xrightarrow{\mathsf{id}_{D_n}} D_n \xleftarrow{\rho} D_{n-1}$.

Permutation of the outer interface: Let a permutation $\pi\colon \mathbb{N}_n \to \mathbb{N}_n$ with $\pi(i) = i+1$ for $0 \le i < n-1$ and $\pi(n-1) = 0$ and an arrow $\sigma\colon D_n \to D_n$ with $v_i \mapsto v_{\pi(i)}$ between two discrete graphs be given. We define the cospan perm_n as follows: $\mathsf{perm}_n\colon D_n \xrightarrow{\mathsf{id}_{D_n}} D_n \xleftarrow{\sigma} D_n$.

Transposition of the outer interface: Let a transposition $\tau\colon \mathbb{N}_n \to \mathbb{N}_n$ with $\tau(0) = 1$, $\tau(1) = 0$ and $\tau(i) = i$ for $2 \le i \le n-1$ and an arrow $\sigma\colon V_n \to V_n$ with $v_i \mapsto v_{\tau(i)}$ between two discrete graphs be given. We define the cospan trans_n as follows: $\mathsf{trans}_n\colon D_n \xrightarrow{\mathsf{id}_{D_n}} D_n \xleftarrow{\sigma} D_n$.

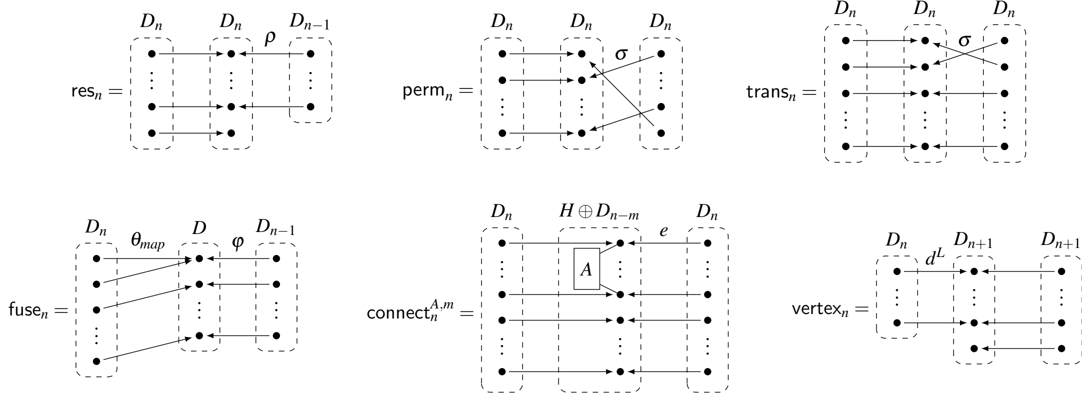


Figure 1: Graph operations

Fusion of two nodes of the outer interface: Let $n > 1$ and an equivalence relation $\theta = \text{id}_{V_n} \cup \{(v_0, v_1), (v_1, v_0)\}$, an arrow θ_{map} which maps every node of D_n to its θ-equivalence class, and an arrow $\varphi\colon D_{n-1} \to D$ with $v_i \mapsto [\![v_{i+1}]\!]_\theta$, where D is the discrete graph with node set $\{[\![v]\!]_\theta \mid v \in V_n\}$, be given. We define the cospan fuse_n as follows: $\text{fuse}_n\colon D_n -\theta_{map}\rightarrow D \leftarrow\varphi- D_{n-1}$.

Connection of a single hyperedge: Let an edge label $A \in \Sigma$, $m \in \mathbb{N}$ with $0 \leq m \leq n$ and a hypergraph H which consists of a single hyperedge h with arity m and labeled with A be given. We define the cospan $\text{connect}_n^{A,m}$ as follows: $\text{connect}_n^{A,m}\colon D_n -e\rightarrow H \oplus D_{n-m} \leftarrow e- D_n$ with $e(v_i) = \text{att}_i(h)$ for $0 \leq i < m$ and $e(v_i) = v_{i-m}$ otherwise.

Disjoint union with a single node: We define the cospan vertex_n as follows: $\text{vertex}_n\colon D_n -d^L\rightarrow D_{n+1} \leftarrow\text{id}_{D_{n+1}}- D_{n+1}$ with $d^L = \text{id}_{D_n} \oplus i$ and $i\colon \emptyset \to D_1$.

The following proposition, which is proven in [4], shows that every graph (viewed as cospan with empty inner and outer interface) can be decomposed in a sequence of atomic cospans:

Proposition 1 *Every cospan of the form $c\colon D_m -\varphi^L\rightarrow G \leftarrow\varphi^R- D_n$ where the right leg φ^R is injective can be constructed by a sequence $\text{op}_1, \ldots, \text{op}_k$ of atomic graph operations, i.e. c can be obtained as the composition $c = \text{op}_1 ; \ldots ; \text{op}_k$.*

Due to this result, we can restrict our attention to atomic cospans instead of considering arbitrary cospans in the following sections.

3 Recognizability and Invariant Checking

In this section we give a short introduction to a straightforward approach to verification which is based on recognizable graph languages. The main idea is to provide an invariant and to check that it is preserved by all transformation rules. This technique was first presented in [3, 4]. In this section we want to present new examples how to use technique for verification. An implementation which provides an invariant check among other things will be discussed in section 4.

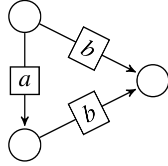


Figure 2: Subgraph T

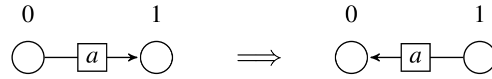


Figure 3: Transformation rule σ_a

In the case of words, a language is an invariant for a rule $\ell \to r$ if it holds for all words u and v that $u\ell v \in L$ implies $urv \in L$. If we consider regular word languages the rule $\ell \to r$ preserves the language L if and only if ℓ and r are ordered with respect to a monotone well-quasi-order such that L is upward-closed w.r.t this well-quasi-order [12, 18]. The coarsest such order is the Myhill-Nerode quasi-order of a language L which relates arbitrary words v and w if and only if it holds for all words u and x that $uvx \in L$ implies $uwx \in L$. This is the coarsest monotone quasi-order such that L is upward-closed with respect to this quasi-order and it can be computed by a fixed-point iteration similar to the computation of the minimal finite automaton [3].

The notion of the Myhill-Nerode quasi-order and the result that a rule $\ell \to r$ preserves a languages if and only if ℓ and r are ordered with respect to the Myhill-Nerode quasi-order can be lifted to recognizable graph languages (based on $Cospan(\mathbf{Graph})$) [4]. The algorithm for computing the Myhill-Nerode quasi-order can also be adapted to the more general setting and there exists a prototype implementation to check whether the language of all graphs containing a given subgraph is an invariant according to a given graph transformation rule [3]. In the following we present some instances for recognizable graph languages which are invariants for different transformation rules.

First we consider the language L_U of all graphs which contain a fixed subgraph U. The automaton functor $\mathscr{A}$ accepting this language works as follows: For every discrete graph D_i, $i \in \mathbb{N}$, the automaton functors contains a state set $\mathscr{A}_U(D_i)$. Every state in each of the state sets has to hold two informations. The first information represents which parts of the subgraph have already been recognized. The second information is a function which maps every node of the interface D_i 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. For every cospan $c\colon D_m \xrightarrow{c^L} G \xleftarrow{c^R} D_n$ two states $(U', f') \in \mathscr{A}_U(D_m)$ and $(U' \cup U'', f'') \in \mathscr{A}_U(D_n)$ are related by $\mathscr{A}_U(c)$ if and only if U'' is a graph containing those nodes and edges of U which lie in the graph G and by "updating" the function f' to a function f'' according to c. Since the graph G might contain several graphs U'' as subgraph, the automaton functor is highly non-deterministic. More details about the construction of this automaton functor can be found in [3].

Example 1 *As an example we want to take the graph T (see Figure 2) as the wanted subgraph. It can be shown that the language L_T of all graphs containing T as a subgraph is an invariant for the rule σ_a (see Figure 3). The general idea is now to perform the following three steps. The first step is to compute the automaton functor $\mathscr{A}_T$ for L_T. The next step is to compute the Myhill-Nerode quasi-order for L_T using the automaton functor $\mathscr{A}_T$ and the algorithm presented in*

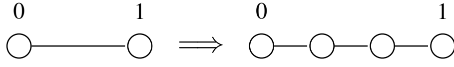
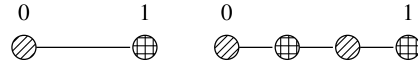


Figure 4: Transformation rule α_n



(a) Left-hand side (b) Right-hand side

Figure 5: Colorings for the rule α_n

[3]. The last step is to check whether the left-hand side of σ_a (viewed as a cospan) is related to the right-hand side of σ_a (viewed as a cospan) according to the Myhill-Nerode quasi-order. Note that in practice we use a slightly different algorithm which computes the simulation relation instead of the Myhill-Nerode quasi-order due to the fact that the automaton functor is non-deterministic and the algorithm computing the Myhill-Nerode quasi-order is only applicable to deterministic automaton functors.

Another recognizable graph language we want to consider is the language $L_{(k)}$ 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 $v_1, v_2 \in \mathtt{att}_G(e)$ 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 theory to find a solution for allocations of limited (hardware) resources or for assignments of limited bandwidth in networks. The idea of the automaton functor $\mathscr{A}_{(k)}\colon Cospan(\mathbf{HGraph}) \to \mathbf{Rel}$ accepting all k-colorable graphs (as defined in [8]) is as follows: Every discrete graph D_i, $i \in \mathbb{N}$, is mapped to the state set $\mathscr{A}_{(k)}(D_i)$ containing all valid k-colorings of D_i, i.e.

$$\mathscr{A}_{(k)}(D_i) = \{f\colon V_{D_i} \to N_k \mid f \text{ is a valid } k\text{-coloring of } D_i\}.$$

For every cospan $c\colon D_m \xrightarrow{c^L} G \xleftarrow{c^R} D_n$ two states $f_m \in \mathscr{A}_{(k)}(D_m)$ and $f_n \in \mathscr{A}_{(k)}(D_n)$ are related by $\mathscr{A}_{(k)}(c)$ if and only if $f(c^L(v)) = f_m(v)$ for every node $v \in V_{D_m}$ and $f(c^R(v)) = f_n(v)$ for every node $v \in V_{D_n}$.

Example 2 *Now we want to consider two examples. The first example is the transformation rule α_n (see Figure 4) for which the language $L_{(2)}$ is an invariant. The rule α_n simply adds two new nodes to an existing path. The second example is the transformation rule α_r (see Figure 6) for which the language language $L_{(3)}$ is an invariant. The second rule replaces the middle node of a rectangular graph by a new rectangle. Since the path of the left-hand side is 2-colorable (see Figure 5) and since the two nodes in the image of the interface do not need to be re-colored, it is obvious that the language $L_{(2)}$ is an invariant for the rule α_n. Due to the fact that the outer rectangle of the LHS of α_r is a (closed) path it is also 2-colorable (see Figure 7) and since the inner node of the left-hand side of α_r in each case forms a triangle with two of the outer rectangle nodes and every triangle is 3-colorable it is clear, that the left-hand side is 3-colorable. The 3-colorability of the right-hand side of the rule α_r can be obtained in a similar way, therefore $L_{(3)}$ is an invariant for the rule α_n.*

Please note that, although it is guaranteed that the colorability for a graph is preserved by the applied transformation rule, it might be necessary to “re-color” the graph during the processing

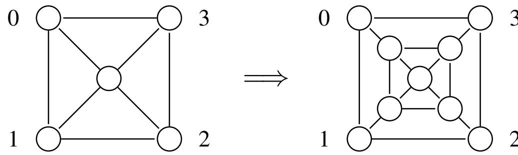


Figure 6: Transformation rule α_r

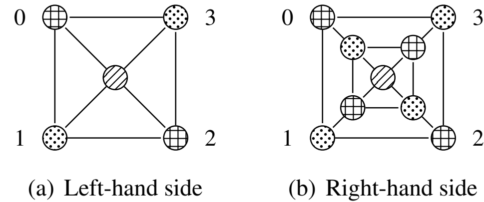


Figure 7: Colorings for the rule α_r

of (the decomposition of) the graph to obtain a valid k-coloring of the graph. The information about possible "re-colorings" are automatically hold by the automaton functor $\mathscr{A}_{(k)}$ due to its construction.

4 Efficient Implementations of Automaton Functors

In this section we present our (prototype) Java-implementation of a tool suite for computing and manipulating automaton functors. In order to implement such a tool suite we have to restrict ourselves to automaton functors of finite size, i.e. the considered automaton functors must not consist of infinitely many finite state sets.

But in general an automaton functor is not finite, since graphs with an arbitrary pathwidth have to be considered. But if only recognizable graph languages of bounded pathwidth are allowed, it is possible to use automaton functors of bounded size, since the size of the interface of every graph is bounded. In the following we only take cospans into account which have a bounded interface size.

Definition 4 (Bounded cospan) A cospan $c\colon S \xrightarrow{c^L} G \xleftarrow{c^R} T$ is called *bounded (by k)*, if there exist atomic graph operations $\mathrm{op}_1, \ldots, \mathrm{op}_j$ such that $c = \mathrm{op}_1 ; \ldots ; \mathrm{op}_j$ and for every graph operation $\mathrm{op}_i\colon D_{n_i} \xrightarrow{op_i^L} G_i \xleftarrow{op_i^R} D_{m_i}$ for $1 \leq i \leq j$ it holds that $n_i, m_i \leq k$.

However, the automaton functors might still be very large. Therefore it is not suitable to represent the automaton functors explicitly. To achieve a compact representation of the automaton functor we use Binary Decision Diagrams (BDDs) [1] to encode the transition relation of the automaton functor. The basic idea is to encode each state of the automaton functor as a bit string of length t. The transition relation can then be seen as set T of bit strings of length $2t$ where a bit string $b_1 \ldots b_{2t}$ is contained in T if and only if $b_1 \ldots b_t$ is the encoding of a state q, $b_{t+1} \ldots b_{2t}$ is the encoding of a state q' and q is related to q' by the transition relation. The set T can be characterized by a boolean function F with $2n$ atomic propositions and this formula F can be represented by a BDD.

As an example we want to 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 the least significant bit left. Then the set can be characterized by the formula $(b_0 \leftrightarrow b_1) \wedge (b_2 \leftrightarrow b_3)$. The BDD which encodes this formula can be seen in Figure 8.

Besides the compact representation of sets and relations BDDs provide some other useful

features. One of these features is that BDDs are unique (up to isomorphism) if the ordering of the atomic propositions of the represented boolean function is fixed and if the BDD is reduced. We use this property of BDDs to efficiently check the equivalence of boolean functions. Another feature of BDDs is the possibility to directly compute boolean operations or both existential and universal quantifications on BDDs instead of performing these operations on the represented boolean functions. In our implementation we use the JavaBDD[1]-package which is based on the Buddy[2]-BDD-package written in C as implementation of BDDs.

In the following we present the state encoding and the propositional formulas we used for the implementation of the automaton functor accepting all graphs containing a specific subgraph (see Section 3). The state encoding has to take care of the following informations:

- the interface size (of the outer interface of the cospan seen so far)
- the parts of the subgraph which have been recognized so far
- the overlap of the parts (of the subgraph) with the current interface

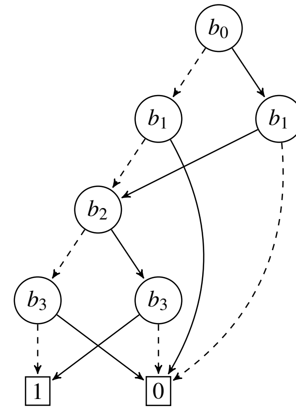


Figure 8: BDD for the formula $(b_0 \leftrightarrow b_1) \wedge (b_2 \leftrightarrow b_3)$

Since a good ordering of the bits holding these informations is essential to construct BDDs which are very compact, we have done some experiments to find the best possible ordering. The resulting encoding of a state is as follows if we assume that the maximum interface size is k, $\ell = \lceil \log_2(k+1) \rceil$ and that the wanted subgraph has m edges and n nodes:

$$b_1 \dots b_\ell e_0 \dots e_{m-1}(v_0 f_{0,0} \dots f_{0,k-1}) \dots (v_{n-1} f_{n-1,0} \dots f_{0,k-1}).$$

The bits $b_1 \dots b_\ell$ encode the current interface size as a binary number, the bit e_i (for $0 \le i \le m-1$) and the bit v_j (for $0 \le j \le n-1$) respectively represent the (non-)existence of the i-th edge and j-th node respectively, and the bit $f_{i,j}$ (for $0 \le i \le n-1$, $0 \le j \le k-1$) encode that the j-th interface node is (not) mapped to the i-th node of the wanted subgraph. To distinguish between the bits encoding the current state and the bits encoding the successor state we indicate the successor state encoding by $b'_1 \dots b'_\ell e'_0 \dots e'_{m-1}(v'_0 f'_{0,0} \dots f'_{0,k-1}) \dots (v'_{n-1} f'_{n-1,0} \dots f'_{n-1,k-1})$. In the following we do not distinguish between the nodes and edges of the wanted subgraphs and the bits encoding the (non-)existence of these nodes and edges in states encoded by the several bit strings.

For each of the atomic graph operations ($\mathsf{connect}^{A,p}$, fuse, perm, res, trans, vertex) we define a separate propositional formula describing all transitions – for all permitted interfaces – of the automaton functor (for the particular atomic graph operation). These formulas can then be easily transformed in BDDs which describe the transition functions for the different atomic graph operations.

In the following we present the formula $f_{\mathsf{connect}^{A,p}}$ for the $\mathsf{connect}^{A,p}$-operation as an example. In order to define the formula we use six auxiliary formulas

[1] JavaBDD Project Homepage: http://javabdd.sourceforge.net
[2] Buddy Manual: http://buddy.sourceforge.net/manual/main.html

- to describe that none of the edges from e_i to e_j have been changed:

$$\mathsf{EdgesUnchanged}(i,j) \iff \bigwedge_{t=i}^{j} (e_t \leftrightarrow e'_t)$$

- to describe that the A-labeled edge e_i of arity s has been added and all other edges have not been changed:

$$\mathsf{EdgeAdded}(i,A,s) \iff \mathsf{EdgesUnchanged}(0,i-1) \wedge (\mathsf{lab}(e_i) = A \\ \wedge |\mathsf{att}(e_i)| = s \wedge \neg e_i \wedge e'_i) \wedge \mathsf{EdgesUnchanged}(i+1,m-1)$$

- to describe that none of the nodes from v_i to v_j have been changed:

$$\mathsf{NodesUnchanged}(i,j) \iff \bigwedge_{t=i}^{j} (v_t \leftrightarrow v'_t)$$

- to describe that the node v_i must have been recognized and be present in the interface if it is adjacent to the edge e_j and must not be changed otherwise:

$$\mathsf{AdjacentNodeExisting}(i,j) \iff (v_i \in \mathsf{att}(e_j) \to (v_i \wedge v'_i \wedge \bigvee_{t=0}^{k-1} f_{i,t})) \\ \wedge (v_i \notin \mathsf{att}(e_j) \to (v_i \leftrightarrow v'_i))$$

- to describe that the interface from j to k for the node v_i has not been changed:

$$\mathsf{InterfaceUnchanged}(i,j,k) \iff \bigwedge_{t=j}^{k} (f_{i,t} \leftrightarrow f'_{i,t})$$

- to describe that the interface from j to k for the node v_i is undefined:

$$\mathsf{InterfaceUndefined}(i,j,k) \iff \bigwedge_{t=j}^{k} (\neg f_{i,t} \wedge \neg f'_{i,t}).$$

Now we can define the formula $f_{\mathsf{connect}_i^{A,p}}$. A transition $q \xrightarrow{\mathsf{connect}_i^{A,p}} q'$ is allowed if and only if both states q and q' belong to the same state set $\mathscr{A}(D_i)$ (i.e. both states have the same interface size) and if either the currently added edge does not belong to the wanted subgraph (i.e. the parts of the subgraph already recognized does not change) or the currently added edge is exactly one of the edges which belong to the wanted subgraph (i.e. this edge will be added to the parts of the

subgraph already recognized):

$$f_{\mathsf{connect}_i^{A,s}} = \bigwedge_{j=1}^{\ell} \left((bin_j(i) \leftrightarrow b_j) \wedge (b_j \leftrightarrow b'_j)\right) \wedge \quad (1)$$

$$\Big(\mathsf{EdgesUnchanged}(0, m-1) \vee \quad (2)$$

$$\bigvee_{j=0}^{m-1} \left(\mathsf{EdgeAdded}(j, A, s) \rightarrow \bigwedge_{t=0}^{n-1} \mathsf{AdjacentNodeExisting}(t, j)\right)\Big) \wedge \quad (3)$$

$$\bigwedge_{j=0}^{n-1} \left(\mathsf{InterfaceUnchanged}(j, 0, i) \wedge \mathsf{InterfaceUndefined}(j, i+1, k-1)\right) \quad (4)$$

The function $bin_i(x)$ used in the formula above returns the i-th bit of the binary encoding of a natural number x. In line 1 it is required that the interface size of the current state is i and that the successor state has also the interface size i. In line 2 the case is described that the currently added edge does not belong to subgraph. Whereas in line 3 it is described that in the case that the added edge belongs to the subgraph it has to be checked that the label and the arity of the new edge matches to an edge of the subgraph, that exactly one edge which has not be recognized before has been added and that all nodes which are adjacent to the new edge has been recognized before and are still present in the interface. In line 4 it is required that the interface nodes which represent the current interface, i.e. the first i interface bits, have not been changed and the other interface bits, i.e. the bits from $i+1$ to $k-1$ have not been set. Note that in the case that the wanted subgraph does not contain an edge with the specified label and the specified arity it can only occur that no edge of the subgraph has been changed.

Example 3 *We consider that the graph S (see Figure 9) is the subgraph we are looking for and that the maximum interface of the corresponding automaton functor is 3. The length of the state encoding then is 16, since we need two bits for the interface size, two bits for the edges, three bits for the nodes and nine bits for the interface function.*

The encoding of the state q (see Figure 10) of this automaton functor in which the nodes v_0 and v_1 have already been recognized, the node v_2 as well as both edges e_0 and e_1 are still missing and the second interface node is mapped to v_1 and the third is mapped to v_0 would be:

b_1	b_2	e_0	e_1	v_0	$f_{0,0}$	$f_{0,1}$	$f_{0,2}$	v_1	$f_{1,0}$	$f_{1,1}$	$f_{1,2}$	v_2	$f_{2,0}$	$f_{2,1}$	$f_{2,2}$
1	1	0	0	1	0	0	1	1	0	1	0	0	0	0	0

If we assume that the automaton functor is in state q and the next atomic graph operation it processes is the $\mathsf{connect}_3^{a,2}$-operation, two transitions are possible. By the first transition the automaton functor decides non-deterministically that the new binary a-labeled edge is none of the edges belonging to S and therefore the automaton functors remains in state q. By the second possible transition the automaton functor decides that the new edge is one of the edges of S and moves to the state q' (see Figure 10) in which the newly added edge has been recognized. Since all adjacent nodes must have been recognized before an edge can be added, there is only the possibility to add the edge e_0 in this situation. The encoding of the state q' is as follows:

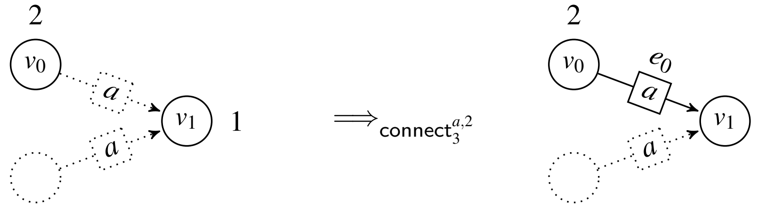


Figure 10: Transition from state q to state q'

$b_1\; b_2\; e_0\; e_1\; v_0\; f_{0,0}\; f_{0,1}\; f_{0,2}\; v_1\; f_{1,0}\; f_{1,1}\; f_{1,2}\; v_2\; f_{2,0}\; f_{2,1}\; f_{2,2}$
1 1 1 0 1 0 0 1 1 0 1 0 0 0 0 0

The encoding of relations is usually done in an "interleaving fashion", more precisely the bits encoding the first element of the pair are alternated with the bits encoding the second element. The great advantage of this interleaving encoding is that the BDD representing the relation gets rather small, since the bits encoding the same piece of information are near each other. For example the following bit vector encodes the pair (q,q') for the $\mathsf{connect}_3^{a,2}$*-operation:*

$b_1\; b_1'\; b_2\; b_2'\; e_0\; e_0'\; e_1\; e_1'\; v_0\; v_0'\; f_{0,0}\; f_{0,0}'\; f_{0,1}\; f_{0,1}'\; f_{0,2}\; f_{0,2}'\; v_1\; v_1'\; f_{1,0}\; f_{1,0}'\; f_{1,1}\; f_{1,1}'\; f_{1,2}\; f_{1,2}'\; v_2\; v_2'\; f_{2,0}\; f_{2,0}'\; f_{2,1}\; f_{2,1}'\; f_{2,2}\; f_{2,2}'$
1 1 1 1 0 1 0 0 1 1 0 0 0 0 1 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0

In order to optimize the computation and size of the BDDs needed to represent the transition relations for the different atomic graph operations, we permitted all bit vectors which encode an interface which has a size less than or equal to the maximum interface size of the automaton functor. The idea behind this is that the BDDs spend much information to the check whether a state (bit vector) is valid, i.e. it has to check four consistency conditions:

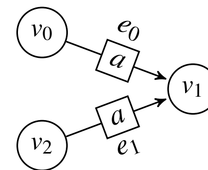


Figure 9: Subgraph S

- the parts of the subgraph already recognized do not contain dangling edges (for example if e_0 is set also v_0 and v_1 have to be set),
- interface nodes are only mapped to subgraph nodes which have already been recognized (for example if one of $f_{0,i}$, $0 \le i \le 2$, is set also v_0 has to be set),
- an interface node is only mapped to exactly one subgraph node (for example if $f_{1,1}$ is set, then $f_{0,1}$ and $f_{2,1}$ must not be set),
- the bits of interface nodes which do not belong to the current interface must not be set (for example if the current interface size is 1, then $f_{i,1}$ and $f_{i,2}$, $0 \le i \le 2$, must not be set).

path-width	number of valid states	number of BDD nodes	time for BDD construction	time for explicit computation[3]
5	7475	2049	0.37 sec	17 sec
10	6041421	7782	0.42 sec	–[4]
20	$5.9 \cdot 10^{12}$	30044	0.67 sec	–[4]
50	$6.8 \cdot 10^{30}$	183038	7.85 sec	–[4]
100	$8.6 \cdot 10^{60}$	726156	1 min 10 sec	–[4]
200	$1.4 \cdot 10^{121}$	2892392	60 min 44 sec	–[4]

Table 1: Performance statistics for the automaton functor $\mathscr{A}_S$

Since the formulas for the several transitions guarantee that these consistency conditions are fulfilled by the transitions, it is not possible to reach an "invalid" state by starting in a "valid" state. In Table 1 we present some results for the automaton functor from example 3 for different interface sizes. The first column indicates the maximum interface size, the second column shows the number of states – without "invalid" bit vectors – of the automaton functor recognizing the subgraph S. In the third column the number of BDD nodes is given which are needed to represent the BDDs for all atomic graph operations. The last two columns show the time for constructing the automaton functor recognizing the subgraph S by computing the BDD-based representation and by computing an explicit-state representation[3]. The table shows that it is not efficiently possible to represent the automaton functor in an explicit way. Due to the fact that the number of states grows exponential the explicit representation leads to an exhaustive consumption of resources. Therefore it is only possible to compute the automaton functor (accepting all graphs containing S as subgraph) for a maximum interface size of 7 on a machine with 2 GB main memory. Despite of that we have successfully tested the implicit BDD-based representation for maximum interface sizes of 1000 and more.

In Table 2 the results for another automaton functor recognizing the subgraph R which can be seen in Figure 11 are shown. It turns out that the computation of automaton functors also scales for subgraphs with higher number of nodes and edges. However, the computation of the simulation relation (which is needed to perform the invariant check) does not benefit from the usage of BDDs in the same dimension as the automaton functor computation. In Table 3 we present the time needed for the BDD-based computation of the simulation relation. Currently we were able to compute the simulation relation up to a maximum interface size of 7. For comparison, if we use an explicit representation of the simulation relation we were only able to compute the relation up to a maximum interface size of 4 [3]. The problem of the computation is that the "intermediate relations" which occur during the fixed-point computation are not efficiently representable by a BDD, i.e. with a small number of BDD nodes.

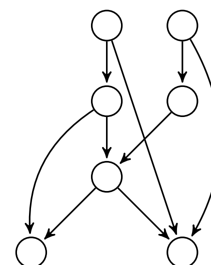


Figure 11: Subgraph R

[3] Further informations about the explicit-state implementation can be found in [3].

[4] Computation impossible due to exhaustive resource consumption.

path-width	number of valid states	number of BDD nodes	time for BDD construction
5	32748051	7521	0.47 sec
10	$7.7 \cdot 10^{11}$	28014	0.77 sec
20	$6.5 \cdot 10^{20}$	106646	3.37 sec
50	$8.4 \cdot 10^{47}$	643550	1 min 7 sec
100	$1.2 \cdot 10^{93}$	2542518	23 min 18 sec

Table 2: Performance statistics for the automaton functor $\mathscr{A}_R$

pathwidth	time
1	0.49 sec
2	0.53 sec
3	0.83 sec
4	2.51 sec
5	19.32 sec
6	4 min 17 sec
7	53 min 12 sec

Table 3: Performance statistics for the simulation relation

5 Conclusions and Future Work

We have presented techniques for an efficient implementation of a tool suite for computing and manipulating (bounded) automaton functors. Apart from the invariant check which we have shown in this paper we have already implemented methods for computing the union and the intersection of automaton functors. In addition we are working on both a universality and a language inclusion check for recognizable graph languages which are both based on an antichain construction introduced by Henzinger et. al. [20]. Since our invariant check suffers from an under-approximation – due to the non-determinism of the automaton functor – we suffer from a one-sided error. This lack could be eliminated if we were able to solve the language inclusion problem on non-deterministic automaton functors (efficiently).

Furthermore we are tackling another problem: Now we have to construct our automaton functors very directly, but this is hard because of the functor property which has to be enforced. In [9] a category-based logic (called a *logic on subobjects*) which has the same expressive power as monadic second-order logic is presented that can be employed for all kinds of (graph-like) structures. This logic can be used for generating automaton functors from formulas due to a result by Courcelle [10, 11] which states that every monadic second-order definable language is recognizable. Currently, we are working on a project to implement this formula-based generation.

Finally, we are studying another approach to the verification of distributed and infinite-state systems: regular model checking [6]. The main idea is to describe (possibly infinite) sets of states as regular languages and transitions of the system as regular relations represented by finite-state transducers transforming a regular language into another regular language. For this purpose we are developing the notion of so-called *transduction functors*, i.e. the counterpart of finite-state transducers in the case of word languages. Since this approach has been extended to the setting of regular tree languages and tree transducers [5] it is a logical step to generalize regular model checking to graph transformation systems where recognizable graph languages play the role of regular languages and transduction functor play the role of finite-state transducers.

There already exists the notion of MSO-definable transductions invented by Courcelle [11], but this notion does not seem to be that useful, since these transductions are very complex and do not guarantee to transform a recognizable graph language into another recognizable graph language in general which is required for a forward analysis. It has to be investigated how the notion of

finite-state transducer can be generalized to transduction functors similar to the generalization of finite-state automata to automaton functors. The goal is to have a notion of transduction functor which is equivalent to finite-state transducers when restricted to word languages.

In addition to our application, BDDs have also been used in other automaton-oriented tools such as MONA [16, 15]. In contrast to our approach in which we represent sets and relations on states implicitly by BDDs, in MONA BDDs are used in a different way, since in MONA the input labels of the automaton are encoded as bit vectors. The idea is that the root node and the leaf nodes of the BDDs indicate states of the automaton which are represented explicitly. Starting in the root node and following some path in the BDD to a leaf node one reaches the successor state according to the input letter encoded by the chosen path. This idea is quite different to our approach since we use a different BDD for each input letter, i.e. for each atomic cospan, to represent the whole transition relation induced by this input letter.

Another related work by Kneis and Langer [17] is based on dynamic programming on tree decompositions to check whether a given graph has some property. For their approach Kneis and Langer take advantage of game-theoretic formalisms to check after each step of the processing of the tree decomposition whether the property still holds. We have not yet compared our results with this approach in detail and it has also to be investigated further, whether this technique could be applied to our setting.

Acknowledgements: The author would like to thank Barbara König, Sander Bruggink and Mathias Hülsbusch for their suggestions and their valuable discussions on this research topic.

Bibliography

[1] Andersen, H.R.: An introduction to binary decision diagrams. Course Notes (1997), http://www.configit.com/fileadmin/Configit/Documents/bdd-eap.pdf

[2] Bauderon, M., Courcelle, B.: Graph expressions and graph rewritings. Mathematical Systems Theory 20(2-3), 83–127 (1987)

[3] Blume, C.: Graphsprachen für die Spezifikation von Invarianten bei verteilten und dynamischen Systemen. Master's thesis, Universität Duisburg-Essen (2008)

[4] Blume, C., Bruggink, S., König, B.: Recognizable graph languages for checking invariants. In: Proceedings of GT-VMT '10. Electronic Communications of the EASST, vol. 29 (2010)

[5] Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Proceedings of SAS '06. pp. 52–70. Springer (2006)

[6] Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Proceedings of CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification. pp. 403–418. Springer (2000), LNCS 1855

[7] Bozapalidis, S., Kalampakas, A.: Graph automata. Theoretical Computer Science 393, 147–165 (2008)

[8] Bruggink, H.J.S., König, B.: On the recognizability of arrow and graph languages. In: Proceedings of ICGT '08. pp. 336–350. Springer (2008), LNCS 5214

[9] Bruggink, H.S., König, B.: A logic on subobjects and recognizability. In: Calude, C., Sassone, V. (eds.) Theoretical Computer Science. IFIP Advances in Information and Communication Technology, vol. 323, pp. 197–212. Springer Boston (2010)

[10] Courcelle, B.: The monadic second-order logic of graphs I. Recognizable sets of finite graphs. Inf. Comput. 85(1), 12–75 (1990)

[11] Courcelle, B.: The expression of graph properties and graph transformations in monadic second-order logic. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations, chap. 5. World Scientific (1997)

[12] Ehrenfeucht, A., Haussler, D., Rozenberg, G.: On regularity of context-free languages. Theoretical Computer Science 27, 311–332 (1983)

[13] Fribourg, L., Olsén, H.: Reachability sets of parameterized rings as regular languages. In: Proceedings of Infinity '97. Electronic Notes in Theoretical Computer Science, vol. 9. Elsevier (1997)

[14] Geser, A., Hofbauer, D., Waldmann, J.: Match-bounded string rewriting systems. Applicable Algebra in Engineering, Communication and Computing 15(3–4), 149–171 (2004)

[15] Gottlob, G., Pichler, R., Wei, F.: Bounded treewidth as a key to tractability of knowledge representation and reasoning. Artif. Intell. 174(1), 105–132 (2010)

[16] Klarlund, N., Møller, A.: MONA Version 1.4 User Manual. BRICS, Department of Computer Science, Aarhus University (January 2001)

[17] Kneis, J., Langer, A.: A practical approach to Courcelle's theorem. In: Proceedings of the International Doctoral Workshop on MEMICS '08. Electronic Notes in Theoretical Computer Science, vol. 251, pp. 65–81. Elsevier (2009)

[18] de Luca, A., Varricchio, S.: Well quasi-orders and regular languages. Acta Inf. 31(6), 539–557 (1994)

[19] Sassone, V., Sobociński, P.: Reactive systems over cospans. In: Proceedings of LICS '05. pp. 311–320. IEEE Computer Society (2005)

[20] 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 '06. pp. 17–30 (2006)